Reverse Engineering per Principianti

Dennis Yurichev

Reverse Engineering per Principianti

(Capire il linguaggio Assembly)

Perchè due titoli? Leggi qua: on page viii.

Dennis Yurichev
my emails

Teston (23 ottobre 2023).

L'ultima versione (e l'edizione Russa) del testo sono accessibili al seguente sito: https://beginners.re/.

Cerchiamo traduttori!

Puoi aiutare a tradurre questo progetto in linguaggi diversi dall'Inglese ed il Russo. Basta inviarmi un qualsiasi pezzo di testo tradotto (non importa quanto è lungo) e lo aggiungerò al mio codice sorgente scritto in LaTeX.

Leggi qui.

La velocità non è importante, perchè è un progetto Open Source, dopo tutto. Il tuo nome sarà menzionato come Contributore del progetto. Coreano, Cinese e Persiano sono linguaggi reservati ai publisher. Le versioni in Inglese e Russo le traduco da solo, ma il mio Inglese è ancora terribile, quindi vi sono grato ad ogni correzione sulla mia grammatica, ecc... Anche il mio Russo non è ancora perfetto, quindi sono felice di ricevere correzioni anche per il Russo!

Non esitate a contattarmi: my emails.

Sommario

Indice

[1] Instruction Set Architecture

Prefazione

Da cosa derivano i due titoli?

Il libro era chiamato “Reverse Engineering for Beginners” nel periodo 2014-2018, ma ho sempre sospettato che questo restringesse troppo i potenziali lettori.

Nel campo Infosec le persone conoscono il “reverse engineering”, ma raramente ho sentito la parola “assembler” da parte loro.

Similmente, il termine “reverse engineering” è in qualche modo criptico per il resto dei programmatori, ma sanno cos’è l’“assembler”.

A luglio 2018, per esperimento, ho cambiato il titolo in “Assembly Language for Beginners” e postato il link sul sito Hacker News [2], ed il libro ha avuto un buon successo.

Quindi è così, il libro adesso ha due titoli.

Tuttavia, ho modificato il secondo titolo in “Understanding Assembly Language”, perchè qualcuno aveva già scritto un libro “Assembly Language for Beginners”. Inoltre la gente dice che “for Beginners” sembra un po’ sarcastico per un libro di ~1000 pagine.

I due libri sono differenti solo per il titolo, il nome del file (UAL-XX.pdf versus RE4B-XX.pdf), l’URL ed un paio di pagine iniziali.

Sul reverse engineering

Esistono diversi significati per il termine «ingegneria inversa»:

1) Il reverse engineering del software; riguardo la ricerca su programmi compilati

2) La scansione di strutture 3D e la successiva manipolazione digitale necessaria alla loro riproduzione

3) Ricreare strutture in DBMS[3]

Questo libro riguarda il primo significato.

Prerequisiti

Conoscenza di base del C PL[4]. Letture raccomandate: 8.1.3 on page 308.

Esercizi e compiti

...possono essere trovati su: http://challenges.re.

[2] https://news.ycombinator.com/item?id=17549050
[3] Database Management Systems
[4] Linguaggio di programmazione (Programming Language)

Elogi per questo libro

https://beginners.re/#praise.

Ringraziamenti

Per aver pazientemente risposto a tutte le mie domande: SkullC0DEr.

Per avermi inviato note riguardo i miei errori e le inaccuratezze: Alexander Lysenko, Federico Ramondino, Mark Wilson, Razikhova Meiramgul Kayratovna, Anatoly Prokofiev, Kostya Begunets, Valentin “netch” Nechayev, Aleksandr Plakhov, Artem Metla, Alexander Yastrebov, Vlad Golovkin[5], Evgeny Proshin, Alexander Myasnikov, Alexey Tretiakov, Oleg Peskov, Pavel Shakhov, Zhu Ruijin, Changmin Heo, Vitor Vidal, Stijn Crevits, Jean-Gregoire Foulon[6], Ben L., Etienne Khan, Norbert Szetei[7], Marc Remy, Michael Hansen, Derk Barten, The Renaissance[8], Hugo Chan, Emil Mursalimov, Tanner Hoke, Tan90909090@GitHub, Ole Petter Orhagen, Sourav Punoriyar, Vitor Oliveira, Alexis Ehret, Maxim Shlochiski, Greg Paton, Pierrick Lebourgeois, Abdullah Alomair, Bobby Battista, Ashod Nakashian..

Per avermi aiutato in altri modi: Andrew Zubinski, Arnaud Patard (rtp on #debian-arm IRC), noshadow on #gcc IRC, Aliaksandr Autayeu, Mohsen Mostafa Jokar, Peter Sovietov, Misha “tiphareth” Verbitsky.

Per aver tradotto il libro in Cinese Semplificato: Antiy Labs (antiy.cn), Archer.

Per la traduzione Coreana: Byungho Min.

Per la traduzione in Olandese: Cedric Sambre (AKA Midas).

Per la traduzione in Spagnolo: Diego Boy, Luis Alberto Espinosa Calvo, Fernando Guida, Diogo Mussi, Patricio Galdames, Emiliano Estevarena.

Per la traduzione in Portoghese: Thales Stevan de A. Gois, Diogo Mussi, Luiz Filipe, Primo David Santini.

Per la traduzione Italiana: Federico Ramondino[9], Paolo Stivanin[10], twyK, Fabrizio Bertone, Matteo Sticco, Marco Negro[11], bluepulsar.

Per la traduzione in Francese: Florent Besnard[12], Marc Remy[13], Baudouin Landais, Téo Dacquet[14], BlueSkeye@GitHub[15].

[5] goto-vlad@github
[6] https://github.com/pixjuan
[7] https://github.com/73696e65
[8] https://github.com/TheRenaissance
[9] https://github.com/pinkrab
[10] https://github.com/paolostivanin
[11] https://github.com/Internaut401
[12] https://github.com/besnardf
[13] https://github.com/mremy
[14] https://github.com/T30rix
[15] https://github.com/BlueSkeye

Per la traduzione in Tedesco: Dennis Siekmeier[16], Julius Angres[17], Dirk Loser[18], Clemens Tamme, Philipp Schweinzer, Tobias Deiminger.

Per la traduzione in Polacco: Kateryna Rozanova, Aleksander Mistewicz, Wiktoria Lewicka, Marcin Sokołowski.

Per la traduzione in Giapponese: shmz@github[19],4ryuJP@github[20].

Per la revisione: Vladimir Botov, Andrei Brazhuk, Mark “Logxen” Cooper, Yuan Jochen Kang, Mal Malakov, Lewis Porter, Jarle Thorsen, Hong Xie.

Vasil Kolev[21] ha speso una notevole quantità di tempo per la revisione e la correzione di molti errori.

Grazie inoltre a tutti quelli su github.com che hanno contribuito a note e correzioni.

Sono stati usati molti pacchetti LaTeX : vorrei ringraziare tutti gli autori di tali moduli.

Donatori

Tutti quelli che mi hanno supportato durante il tempo in cui ho scritto la parte più significativa del libro:

2 * Oleg Vygovsky (50+100 UAH), Daniel Bilar ($50), James Truscott ($4.5), Luis Rocha ($63), Joris van de Vis ($127), Richard S Shultz ($20), Jang Minchang ($20), Shade Atlas (5 AUD), Yao Xiao ($10), Pawel Szczur (40 CHF), Justin Simms ($20), Shawn the R0ck ($27), Ki Chan Ahn ($50), Triop AB (100 SEK), Ange Albertini (€10+50), Sergey Lukianov (300 RUR), Ludvig Gislason (200 SEK), Gérard Labadie (€40), Sergey Volchkov (10 AUD), Vankayala Vigneswararao ($50), Philippe Teuwen ($4), Martin Haeberli ($10), Victor Cazacov (€5), Tobias Sturzenegger (10 CHF), Sonny Thai ($15), Bayna AlZaabi ($75), Redfive B.V. (€25), Joona Oskari Heikkilä (€5), Marshall Bishop ($50), Nicolas Werner (€12), Jeremy Brown ($100), Alexandre Borges ($25), Vladimir Dikovski (€50), Jiarui Hong (100.00 SEK), Jim Di (500 RUR), Tan Vincent ($30), Sri Harsha Kandrakota (10 AUD), Pillay Harish (10 SGD), Timur Valiev (230 RUR), Carlos Garcia Prado (€10), Salikov Alexander (500 RUR), Oliver Whitehouse (30 GBP), Katy Moe ($14), Maxim Dyakonov ($3), Sebastian Aguilera (€20), Hans-Martin Münch (€15), Jarle Thorsen (100 NOK), Vitaly Osipov ($100), Yuri Romanov (1000 RUR), Aliaksandr Autayeu (€10), Tudor Azoitei ($40), Z0vsky (€10), Yu Dai ($10), Anonymous ($15), Vladislav Chelnokov ($25), Nenad Noveljic ($50), Ryan Smith ($25), Andreas Schommer (€5), Nikolay Gavrilov ($300), Ernesto Bonev Reynoso ($30).

Grazie di cuore a tutti i donatori!

mini-FAQ

Q: Quali sono i prerequisiti per leggere questo libro?

[16] https://github.com/DSiekmeier
[17] https://github.com/JAngres
[18] https://github.com/PolymathMonkey
[19] https://github.com/shmz
[20] https://github.com/4ryuJP
[21] https://vasil.ludost.net/

A: È consigliato avere almeno una conoscenza base di C/C++.

Q: Dovrei veramente imparare x86/x64/ARM e MIPS allo stesso tempo? Non è troppo?

A: Chi inizia può leggere semplicemente su x86/x64, e saltare o sfogliare velocemente le parti su ARM e MIPS.

Q: Posso acquistare la versione in Russo/Inglese del libro?

A: Sfortunatamente no, nessun editore (al momento) è interessato nel pubblicare questo libro. Nel frattempo puoi chiedere alla tua copisteria di fiducia di stamparlo. https://yurichev.com/news/20200222_printed_RE4B/.

Q: C'è una versione EPUB/MOBI?

A: Il libro dipende fortemente da alcuni hacks in TeX/LaTeX, quindi convertire il tutto in HTML (EPUB/MOBI è un set di HTMLs) non sarebbe facile.

Q: Perchè qualcuno dovrebbe studiare assembly al giorno d'oggi?

A: A meno che tu non sia uno sviluppatore di **sistemi operativi!**[22], probabilmente non avrai mai bisogno di scrivere codice assembly —i compilatori moderni sono migliori dell'uomo nell'effettuare ottimizzazioni [23].

Inoltre, le CPU[24] moderne sono dispositivi molto complessi e la semplice conoscenza di assembly non basta per capire il loro funzionamento interno.

Ci sono però almeno due aree in cui una buona conoscenza di assembly può tornare utile: analisi malware/ricercatore in ambito sicurezza e per avere una miglior comprensione del codice compilato durante il debugging di un programma. Questo libro è perciò pensato per quelle persone che vogliono capire il linguaggio assembly piuttosto che imparare a programmare con esso.

Q: Ho cliccato su un link all'interno del PDF, come torno indietro?

A: In Adobe Acrobat Reader clicca Alt+FrecciaSinistra. In Evince clicca il pulsante "<".

Q: Posso stampare questo libro / usarlo per insegnare?

A: Certamente! Il libro è rilasciato sotto licenza Creative Commons (CC BY-SA 4.0).

Q: Perchè questo libro è gratis? Hai svolto un ottimo lavoro. È sospetto come molte altre cose gratis.

A: Per mia esperienza, gli autori di libri tecnici fanno queste cose per auto-pubblicizzarsi. Non è possibile ottenere un buon ricavato da un lavoro così oneroso.

Q: Come si fa ad ottenere un lavoro nel campo del reverse engineering?

A: Ci sono threads di assunzione che appaiono di tanto in tanto su reddit RE[25]. Prova a guardare lì.

Qualcosa di simile si può anche trovare nel subreddit «netsec».

[22] **sistemi operativi!**

[23] Un testo consigliato relativamente a questo argomento: [Agner Fog, *The microarchitecture of Intel, AMD and VIA CPUs*, (2016)]

[24] Central Processing Unit

[25] reddit.com/r/ReverseEngineering/

Q: Avrei una domanda...

A: Inviamela tramite e-mail (my emails).

La traduzione in Coreano

A gennaio 2015, la Acorn publishing company (www.acornpub.co.kr) in Corea del Sud ha compiuto un enorme lavoro traducendo e pubblicando questo libro (aggiornato ad agosto 2014) in Coreano.

Adesso è disponibile sul loro sito.

Il traduttore è Byungho Min (twitter/tais9). La copertina è stata creata dall'artistico Andy Nechaevsky, un amico dell'autore: facebook/andydinka. Acorn detiene inoltre i diritti della traduzione in Coreano.

Quindi se vuoi un *vero* libro in Coreano nella tua libreria e vuoi supportare questo lavoro, è disponibile per l'acquisto.

La traduzione in Persiano/Farsi

Nel 2016 il libro è stato tradotto da Mohsen Mostafa Jokar (che è anche conosciuto nell comunità iraniana per la traduzione del manuale di Radare [26]). Potete trovarlo sul sito dell'editore[27] (Pendare Pars).

Qua c'è il link ad un estratto di 40 pagine: https://beginners.re/farsi.pdf.

Informazioni nella Libreria Nazionale dell'Iran: http://opac.nlai.ir/opac-prod/bibliographic/4473995.

La traduzione Cinese

Ad aprile 2017, la traduzione in Cinese è stata completata da Chinese PTPress. Possiedono inoltre i diritti della traduzione in Cinese.

La versione cinese può essere ordinata a questo indirizzo: http://www.epubit.com.cn/book/details/4174. Una recensione parziale, con informazioni sulla traduzione è disponibile qua: http://www.cptoday.cn/news/detail/3155.

Il traduttore principale è Archer, al quale l'autore deve molto. E' stato estremamente meticoloso (in senso buono) ed ha segnalato buona parte degli errori e bug, il che è molto importante in un libro come questo. L'autore raccomanderebbe i suoi servizi a qualsiasi altro autore!

I ragazzi di Antiy Labs hanno inoltre aiutato nella traduzione. Qua c'è la prefazione scritta da loro.

[26] http://rada.re/get/radare2book-persian.pdf
[27] http://goo.gl/2Tzx0H

Capitolo 1

Pattern di codice

1.1 Il metodo

Quando l'autore di questo libro ha cominciato ad imparare il C e, successivamente, C++, era solito scrivere piccoli pezzi di codice, compilarli e guardare l'output prodotto in linguaggio assembly. Questo procedimento ha facilitato la comprensione del comportamento del codice che aveva scritto. [1]. Lo ha fatto talmente tante volte che la relazione tra il codice C/C++ e ciò che viene prodotto dal compilatore si è impressa profondamente nella sua mente. E' facile immaginare a colpo d'occhio un contorno della forma e della funzione di un dato codice C. Magari questa tecnica può rivelarsi utile anche per gli altri.

Ad ogni modo, esiste un utile sito in cui puoi fare lo stesso, con diversi compilatori, invece di installarli sul tuo PC. Puoi usarlo a questo indirizzo: https://godbolt.org/.

Esercizi

Quando l'autore di questo libro studiava il linguaggio assembly, era solito anche compilare piccola funzioni C e riscriverle gradualmente in assembly tentando di restringere il codice il più possibile. Questa pratica è oggi probabilmente inutile in uno scenario reale, in quanto è molto difficile competere, in termini di efficienza, con i moderni compilatori. Rappresenta comunque un ottimo modo di acquisire una migliore conoscenza dell'assembly. Sentitevi quindi liberi di prendere qualunque pezzo di codice assembly da questo libro e cercare di renderlo più piccolo. Tuttavia non dimenticate di testare il vostro risultato.

[1] In effetti lo fa ancora oggi quando non riesce a capire cosa fa un particolare pezzo di codice.

Livelli di ottimizzazione e informazioni di debug

Il codice sorgente può essere compilato da compilatori diversi e con vari livelli di ottimizzazione. Un compilatore tipico ne prevede solitamente tre, dei quali il livello zero corrisponde a nessuna ottimizzazione (ottimizzazione disabilitata).

L'ottimizzazione può essere orientata verso la dimensione del codice o la sua velocità di esecuzione. Un compilatore non ottimizzante è più veloce e produce codice più comprensibile (sebbene prolisso), mentre un compilatore ottimizzante è più lento e cerca di produrre codice più veloce in termini di performance (ma non necesariamente più compatto). Oltre ai livelli di ottimizzazione, un compilatore può includere informazioni di debug nel file risultante, producendo quindi codice che può essere debuggato più facilmente. Una delle caratteristiche più importanti del codice di ´debug' è che può contenere collegamenti tra ogni riga del codice sorgente e l'indirizzo del corrispondente codice macchina. I compilatori ottimizzanti tendono invece a produrre output in cui intere righe di codice sorgente possono essere ottimizzate a tal punto da non essere neanche presenti nel codice macchina risultante. I reverse engineers possono incontrare entrambe le versioni, semplicemente perchè alcuni sviluppatori utilizzano le opzioni di ottimizzazione dei compilatori ed altri no. A causa di ciò negli esempi proveremo, quando possibile, a lavorare sia sulle versioni di debug che su quelle di release del codice illustrato in questo libro.

A volte in questo libro vengono utilizzate delle versioni di compilatori particolarmente vecchie, in modo da ottenere il più corto (o semplice) blocco di codice.

1.2 Alcune basi teoriche

1.2.1 Una breve introduzione alla CPU

La CPU è il dispositivo che esegue il codice macchina di cui è fatto un programma.

Un breve glossario:

Istruzione : Una primitiva per la CPU. L'esempio più semplice include: spostare dati da un registro all'altro, lavorare con la memoria, effettuare operazioni aritmetiche primitive. Di norma ogni CPU ha il suo insieme di istruzioni, detto instruction set architecture o (ISA).

Codice macchina : Codice che la CPU è in grado di processare direttamente. Ciascuna istruzione è solitamente codificata da diversi byte.

Linguaggio Assembly : Codice mnemonico ed alcune estensioni come le macro introdotti per facilitare la vita del programmatore.

Registro CPU : Ogni CPU ha un certo numero definito di registri generici (GPR[2]). ≈ 8 in x86, ≈ 16 in x86-64, ≈ 16 in ARM. Il modo più semplice per capire con'è un registro è quello di pensare ad esso come una variabile temporanea senza tipo. Immagina se stessi lavorando con un linguaggio di programmazione di alto livello PL e potessi usare solo otto variabili a 32 (o 64) bit.

[2] General Purpose Registers

Anche solo con quelle potresti fare molte cose!

Qualcuno potrebbe chiedersi perchè ci debba essere una differenza tra il codice macchina ed un PL. La risposta risiede nel fatto che gli umani e i processori (CPU) non sono uguali—per un umano è molto più facille utilizzare un linguaggio di programmazione ad alto livello come C/C++, Java, Python, etc., mentre per una CPU è più semplice utilizzare un livello di astrazione più basso. Potrebbe essere forse possibile inventare una CPU in grado di eseguire codice di un PL ad alto livello, ma sarebbe molto più complessa dei processori che conosciamo oggi. Allo stesso modo sarebbe del tutto sconveniente per gli umani scrivere in linguaggio assembly, a causa della sua natura di basso livello e la difficoltà nello scrivere senza commettere un enorme numero di fastidiosi errori. Il programma che converte il codice di un PL di alto livello in assembly è detto un *compilatore*. [3].

1.2.2 Qualche parola sulle diverse ISA

La ISA dell'architettura x86 ha sempre avuto istruzioni di lunghezza variabile, e all'arrivo dell'era 64-bit l'estensione x64 non ha avuto impatti significativi sulla ISA. Di fatto l'architettura x86 contiene molte istruzioni apparse per la prima volta nelle CPU a 16-bit 8086 e che si trovano ancora nei processori odierni. ARM è una CPU RISC[4] progettata per avere istruzioni di lunghezza costante, che in passato avevano alcuni vantaggi. Originariamente tutte le istruzioni ARM erano codificate in 4 byte[5]. Oggi questa modalità è nota come «ARM mode». Successivamente si scoprì che non era poi tanto economico come si immaginava inizialmente. Infatti, le istruzioni CPU più usate [6] nelle applicazioni reali, possono essere codificate usando meno informazioni. Venne quindi aggiunta un'altra ISA, detta Thumb, in cui ogni istruzione era codificata in solo 2 byte. Oggi detto «Thumb mode». Tuttavia non *tutte* le istruzioni ARM possono essere codificate in solo 2 byte, quindi il set di istruzioni Thumb è quindi in qualche modo limitato. Vale la pena notare che il codice compilato in ARM mode e Thumb mode può tranquillamente coesistere all'interno dello stesso programma. I creatori di ARM pensarono di estendere Thumb, dando vita a Thumb-2, apparso per la prima volta in ARMv7. Thumb-2 utilizza ancora istruzioni da 2 byte, ma ha alcune nuove istruzioni da 4 byte. Esiste la convinzione errata che Thumb-2 sia un mix di ARM e Thumb. Questo non è vero. Piuttosto Thumb-2 è stato esteso per supportare tutte le caratteristiche del processore così da competere con l'ARM mode— un obiettivo che è stato chiaramente raggiunto, visto che la maggior parte delle applicazioni per iPod/iPhone/iPad sono compilate per il set di istruzioni Thumb-2 (in effetti, molto probabilmente dovuto anche al fatto che Xcode lo fa di default). Successivamente fu la volta di ARM a 64-bit. Questa ISA ha opcode di 4-byte, e non necessita di alcun Thumb mode aggiuntivo. Tuttavia i requisiti dei 64-bit hanno avuto un impatto sulla ISA, motivo per cui abbiamo oggi tre set di istruzioni ARM: ARM mode, Thumb mode (incluso Thumb-2) e ARM64. Queste ISA sono parzialmente simili, ma possiamo dire dire che si tratta di ISA differenti invece che varianti della stessa. Per questo motivo in questo libro cercheremo di aggiungere frammenti di codice in tutte e tre

[3] La vecchia letteratura russa in materia utilizza il termine «traduttore».

[4] Reduced Instruction Set Computing

[5] Le istruzioni a lunghezza fissa sono utili perchè è possibile calcolare senza sforzo l'indirizzo della prossiama (o della precedente) istruzione. Questa caratteristica sarà discussa nella sezione dedicata all'operatore switch() operator (1.21.2 on page 221) .

[6] Che sono MOV/PUSH/CALL/Jcc

le ISA ARM. A proposito, esistono anche molte altre ISAs di tipo RISC che utilizzano istruzioni con lunghezza fissa di 32-bit, ad esempio: MIPS, PowerPC e Alpha AXP.

1.2.3 Sistemi di numerazione

> Nowadays octal numbers seem to be used for exactly one purpose—file permissions on POSIX systems—but hexadecimal numbers are widely used to emphasize the bit pattern of a number over its numeric value.
>
> Alan A. A. Donovan, Brian W. Kernighan — The Go Programming Language

Le persone si sono abituate ad usare il sistema numerico decimale probabilmente perchè quasi tutti hanno 10 dita. Ciononostante, il numero «10» non ha alcun significato rilevante nelle scienze e nella matematica. Il sistema di numerazione naturale nell'elettronica digitale è quello binario: 0 per l'assenza di corrente nel filo e 1 per la sua presenza. 10 in binario è 2 in decimale, 100 in binary è 4 in decimale, e così via.

Se il sistema numerico ha 10 cifre, ha una *radice* (o *base*) di 10. Il sistema numerico binario ha *radice* 2.

Cose importanti da ricordare:

1) Un *numero* è un numero, mentre una *cifra* è un termine che deriva dai sistemi di scrittura, ed è solitamente un carattere

2) Il valore di un numero non cambia quando viene convertito ad un'altra radice; cambia solo la forma di scrittura del suo valore (e quindi il modo in cui viene rappresentato in RAM[7]).

1.2.4 Convertire da una radice ad un'altra

La notazione posizionale è usata in praticamente tutti i sistemi numerici. Questo significa che la cifra ha un "peso" diverso in base alla posizone in cui si trova all'interno del numero più grande. Se 2 si trova nella parte più a destra del numero, è 2, ma se si trova nella penultima posizione a destra è 20.

Per cosa sta 1234?

$10^3 \cdot 1 + 10^2 \cdot 2 + 10^1 \cdot 3 + 1 \cdot 4 = 1234$ o anche $1000 \cdot 1 + 100 \cdot 2 + 10 \cdot 3 + 4 = 1234$

Lo stesso vale per i numeri binari, ma la base è 2 invece di 10. Per cosa sta 0b101011?

$2^5 \cdot 1 + 2^4 \cdot 0 + 2^3 \cdot 1 + 2^2 \cdot 0 + 2^1 \cdot 1 + 2^0 \cdot 1 = 43$ o anche $32 \cdot 1 + 16 \cdot 0 + 8 \cdot 1 + 4 \cdot 0 + 2 \cdot 1 + 1 = 43$

Esiste anche una notazione non-posizionale, come ad esempio il sistema numerico Romano. [8]. Forse l'umanità ha deciso di passare alla notazione posizionale perchè è più facile effettuare operazioni di base (addizione, moltiplicazione, etc.) a mano su carta.

[7] Random-Access Memory

[8] Riguardo l'evoluzione dei sistemi numerici, vedi [Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997), 195–213.]

I numeri binari possono essere addizionati, sottratti e così via, nello stesso modo in cui ci è stato insegnato a scuola, con l'unica differenza che si sono solo 2 cifre a disposizione.

I numeri binari possono risultare ingombranti quando usati in codici sorgenti e dump, ed in questi casi può tornare utile il sistema esadecimale.

Il sistema esadecimale usa le cifre 0..9 ed in aggiunta 6 caratteri latini: A..F. Ogni cifra esadecimale occupa 4 bit o 4 cifre binarie, ed è quindi molto facile convertire da binario a esadecimale e viceversa, anche a mente.

hexadecimal	binary	decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Come identificare quale radice si sta usando in un certo caso?

I numeri decimali sono solitamente scritti così come sono, es. 1234. Alcuni assembler consentono di utilizzare un identificatore per i numeri in radice decimale, ed il numero può essere scritto con il suffisso "d": 1234d.

I numeri binari sono a volte preceduti dal prefisso "0b": 0b100110111 (GCC[9] ha un'estensione non standard del linguaggio per questo[10]). C'è anche un altro modo: utilizzando il suffisso "b", ad esempio: 100110111b. Il libro cerca di usare in modo coerente il prefisso "0b" per identificare i numeri binari.

I numeri esadecimali sono preceduti dal prefisso "0x" in C/C++ e altri PLs: 0x1234ABCD. In alternativa viene utilizzato il suffisso "h": 1234ABCDh. Questo è il modo in cui vengono comunemente rappresentati negli assembler e nei debugger. In questa convenzione, se il numero inizia con una lettera A..F, uno "0" viene aggiunto all'inizio: 0ABCDEFh. C'era inoltre una convenzione popolare durante l'era dei PC ad 8-bit, utilizzando il prefisso $, ad esempio $ABCD. Nel corso del libro si cercherà di usare in modo costante il prefisso "0x" per identificare i numeri esadecimali.

[9] GNU Compiler Collection

[10] https://gcc.gnu.org/onlinedocs/gcc/Binary-constants.html

Si dovrebbe imparare a convertire i numeri a mente? Una tabella di numeri decimali ad una cifra può essere memorizzata facilmente. Per numeri più grandi, probabilmente, non è il caso di tormentarsi.

Probabilmente i numeri esadecimali più visibili sono quelli all'interno degli URL[11]s. Questo è il modo con cui vengono codificati i caratteri non latini. Ad esempio: `https://en.wiktionary.org/wiki/na%C3%AFvet%C3%A9` è l'URL dell'articolo di Wiktionary sulla parola «naïveté».

Sistema di numerazione ottale

Un altro sistema di numerazione molto usato in passato in programmazione è quello ottale. In questo sistema ci sono 8 cifre (0..7) e ciascuna è associata a 3 bit, quindi è facile convertire numeri da una radice all'altra. Praticamente ovunque è stato rimpiazzato dal sistema esadecimale, ma sorprendentemente esiste una utility *NIX usata spesso e da molte persone che ha per argomento un numero ottale: `chmod`.

Come molti utenti *NIX sanno, l'argomento di `chmod` può essere un numero di 3 cifre. La prima rappresenta i diritti del proprietario del file (lettura, scrittura ed esecuzione), la seconda i diritti del gruppo a cui il file appartiene, la terza i diritti per chiunque altro. Ogni cifra che `chmod` riceve può essere rappresentata in forma binaria:

decimale	binario	significato
7	111	**rwx**
6	110	**rw-**
5	101	**r-x**
4	100	**r--**
3	011	**-wx**
2	010	**-w-**
1	001	**--x**
0	000	**---**

Quindi ogni bit corrisponde ad un flag: read/write/execute.

La ragione per cui sto parlando di `chmod` è che l'intero numero dell'argomento può essere rappresentato in ottale. Prendiamo per esempio 644. Quando si esegue `chmod 644 file`, si impostano i permessi di lettura/scrittura per il proprietario, il permesso di lettura per il gruppo, ed il permesso di lettura per tutti gli altri. Convertiamo il numero ottale 644 in binario, sara' `110100100`, o (in gruppi di 3 bit) `110 100 100`.

Possiamo notare che ogni tripletta descrive i permessi per proprietario/gruppo/altri (owner/group/others): il primo è `rw-`, il secondo è `r--` ed il terzo è `r--`.

Il sistema ottale era anche molto diffuso nei vecchi computer come PDP-8, perchè una word poteva essere di 12, 24 o 36 bit, e questi numeri sono tutti divisibili per 3, quindi il sistema ottale era naturale in quell'ambiente. Oggi tutti i computer comuni utilizano word/indirizzi lunghi 16, 32 o 64 bit, e questi numeri sono tutti divisibili per 4, di conseguenza l'esadecimale risulta più naturale.

[11] Uniform Resource Locator

Il sistema ottale è supportato da tutti i compilatori C/C++ standard. Talvolta ciò è fonte di confusione, perchè i numeri ottali sono codificati preponendo uno zero, per esempio 0377 è 255. A volte si potrebbe scrivere per errore ”09” invece di 9, e il compilatore segnalerebbe un errore. GCC potrebbe presentare un errore del genere: `error: invalid digit "9" in octal constant`.

Inoltre, il sistema ottale è in qualche modo popolare in Java. Quando IDA mostra delle stringhe Java contenenti dei caratteri non visualizzabili, li codifica nel sistema ottale invece che in quello esadecimale. Il decompilatore per Java JAD si comporta allo stesso modo.

Divisibilità

Quando si vede un numero decimale come 120, si può velocemente dedurre che è divisibile per 10, perchè l’ultima cifra è uno zero. Allo stesso modo, 123400 è divisibile per 100 perchè le ultime due cifre sono zeri.

In maniera simile, il numero esadecimale 0x1230 è divisible per 0x10 (ovvero 16), 0x123000 è divisibile per 0x1000 (ovvero 4096), etc.

Il numero binario 0b1000101000 è divisibile per 0b1000 (8), etc.

Questa proprietà può essere spesso usata per capire velocemente se un indirizzo o la dimensione di un dato blocco di memoria sono stati ”allungati” (padded) per raggiungere un certo allineamento. Per esempio, le sezioni in un file PE[12] iniziano quasi sempre ad indirizzi che terminano con 3 zeri esadecimali: 0x41000, 0x10001000, etc. La ragione per cui questo accade risiede nel fatto che quasi tutte le sezioni di un PE sono allinate per raggiungere blocchi di dimensioni multiple di 0x1000 (4096) byte.

Aritmetica e radici a precisione multipla

L’aritmetica a precisione multipla può utilizzare numeri enormi, e ognuno può venire memorizzato in più byte. Ad esempio, le chiavi RSA, sia pubbliche che private, arrivano fino a 4096 bit e più.

In [Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997), 265] possiamo trovare questa idea: quando memorizzi un numero a precisione multipla su più byte, l’intero numero può essere rappresentato utilizzando una radice di $2^8 = 256$, e ogni cifra va nel byte corrispondente. Allo stesso modo, se memorizzi un numero a precisione multipla in diversi valori interi da 32 bit, ogni cifra corrisponde a ciascuno slot da 32 bit, e puoi pensare a questo numero come rappresentato in radice 2^{32}.

Come Pronunciare Numeri non Decimali

I numeri in base non decimale sono tipicamente pronunciati cifra per cifra: “uno-zero-zero-uno-uno-...”. Parole come “dieci” e “mille” generalmente non vengono pronunciate, per evitare confusione con il sistema a base decimale.

[12] Portable Executable

Numeri a virgola mobile (Floating point)

Per distinguere i numeri floating point dai numeri interi, vengono generalmente scritti con un ".0" alla fine, ad esempio 0.0, 123.0, etc.

1.3 Una funzione vuota

La funzione più semplice è sicuramente quella che non fa niente:

Listing 1.1: Italian text placeholder

```
void f()
{
        return;
};
```

Compiliamola!

1.3.1 x86

Questo è quello che i compilatori GCC e MSVC producono per una piattaforma x86:

Listing 1.2: Con ottimizzazione GCC/MSVC (risultato dell'assembly)

```
f:
        ret
```

C'è solo un'istruzione: RET, la quale ritorna l'esecuzione al chiamante.

1.3.2 ARM

Listing 1.3: Con ottimizzazione Keil 6/2013 (Modalità ARM) risultato dell'assembly

```
f       PROC
        BX       lr
        ENDP
```

L'indirizzo di ritorno non viene salvato nello stack locale nella ISA ARM, ma invece nel registro link, quindi l'istruzione BX LR causa l'esecuzione di un salto (jump) a quell'indirizzo—effettivamente ritornando l'esecuzione al chiamante.

1.3.3 MIPS

Esistono due convenzioni utilizzate nel mondo MIPS quando vengono chiamati i registri: per numero (da $0 a $31) o per pseudonimo ($V0, $A0, etc.).

L'output in assembly di GCC qua sotto elenca i registri per numero:

Listing 1.4: Con ottimizzazione GCC 4.4.5 (risultato dell'assembly)

```
        j       $31
        nop
```

...mentre IDA[13] utilizza gli pseudonimi:

Listing 1.5: Con ottimizzazione GCC 4.4.5 (IDA)

```
        j       $ra
        nop
```

La prima istruzione è un salto (J or JR) che ritorna il flusso di esecuzione al chiamante, saltando all'indirizzo contenuto nel registro $31 (o $RA).

Questo è il registro analogo a LR[15] in ARM.

La seconda istruzione è NOP[16], che non fa niente. Per il momento possiamo ignorarla.

Una nota sulle istruzioni MIPS ed i nomi dei registri

I nomi dei registri e delle istruzioni nel mondo MIPS sono tradizionalmente scritti in minuscolo. Tuttavia, per uniformità, questo libro manterrà l'utilizzo del maiuscolo, che è la convenzione utilizzata in tutti gli altri ISA mostrati in questo libro.

1.3.4 Le funzioni vuote in pratica

Anche se le funzioni vuote sembrano inutili, sono abbastanza utilizzate nel codice a basso livello.

Prima di tutto, sono abbastanza popolari nelle funzioni per debug, come questa:

Listing 1.6: C/C++ Code

```
void dbg_print (const char *fmt, ...)
{
#ifdef _DEBUG
        // apri file di log
        // scrivi nel file di log
        // chiudi il file di log
#endif
};

void some_function()
{
        ...

        dbg_print ("we did something\n");

        ...
};
```

In una build non di debug (come in una "release"), `_DEBUG` non è definito, quindi la funzione `dbg_print()`, nonostante venga ancora chiamata durante l'esecuzione, sarà vuota.

[13] TBT[14] by Hex-Rays
[15] Link Register
[16] No Operation

Similmente, un metodo popolare per la protezione del software è di creare una build per gli acquirenti regolari, ed una build di demo. In una build di demo possono mancare alcune funzionalità importanti, come in questo esempio:

Listing 1.7: C/C++ Code

```
void save_file ()
{
#ifndef DEMO
        // un vero codice di salvataggio
#endif
};
```

La funzione `save_file()` può essere chiamata quando l'utente fa click sul menu `File->Salva`. La versione demo può essere distribuita con questa voce di menu disattivata, ma anche se un cracker la riattiva, verrà chiamata semplicemente una funzione vuota che non contiene del codice utile.

IDA segnala queste funzioni con dei nomi come `nullsub_00`, `nullsub_01`, etc.

1.4 Ritornare valori

Un'altra funzione semplice è quella che ritorna un valore costante:

Listing 1.8: Italian text placeholder

```
int f()
{
        return 123;
};
```

Compiliamola.

1.4.1 x86

Questo è quello che i compilatori GCC e MSVC producono (con ottimizzazione) per x86:

Listing 1.9: Con ottimizzazione GCC/MSVC (risultato dell'assembly)

```
f:
        mov     eax, 123
        ret
```

Ci sono solo due istruzioni: la prima inserisce il valore 123 nel registro EAX, che per convenzione viene utilizzato per memorizzare i valori di ritorno, e la seconda è RET, che ritorna l'esecuzione al chiamante.

La funzione chiamante troverà quindi il valore di ritorno nel registro EAX.

1.4.2 ARM

Ci sono alcune differenze nella piattaforma ARM:

Listing 1.10: Con ottimizzazione Keil 6/2013 (Modalità ARM) ASM Output

```
f       PROC
        MOV       r0,#0x7b ; 123
        BX        lr
        ENDP
```

ARM utilizza il registro R0 per ritornare i risultati delle funzioni, quindi 123 viene copiato in R0.

Occorre notare che MOV è un nome di funzione fuorviante sia nella ISA x86 che ARM.

I dati non vengono infatti *spostati*, ma *copiati*.

1.4.3 MIPS

L'output in assembly di GCC assembly qua sotto chiama i registri per numero:

Listing 1.11: Con ottimizzazione GCC 4.4.5 (risultato dell'assembly)

```
        j       $31
        li      $2,123                  # 0x7b
```

...mentre IDA utilizza gli pseudonimi:

Listing 1.12: Con ottimizzazione GCC 4.4.5 (IDA)

```
                jr      $ra
                li      $v0, 0x7B
```

Il registro $2 (o $V0) viene utilizzato per memorizzare il valore di ritorno della funzione. LI sta per "Load Immediate" ed è l'equivalente MIPS di MOV.

L'altra istruzione è il salto (J or JR) che ritorna il flusso di esecuzione al chiamante.

Potresti domandarti perchè le posizioni delle istruzioni Load Immediate (LI) ed il jump (J or JR) siano invertite. Questo è dovuto ad una funzionalità di RISC chiamata"branch delay slot".

Il motivo per cui accade è dovuto ad un problema nell'architettura di alcune ISA RISC e non è importante per i nostri scopi—dobbiamo semplicemente tenere a mente che in MIPS, l'istruzione che segue un jump o un'istruzione condizionale viene eseguita *prima* del salto/ramificazione stessi.

Come conseguenza, le istruzioni di ramificazione vengono sempre scambiate di posto con l'istruzione immediatamente precedente.

In pratica, le funzioni che ritornano semplicemente 1 (*true*) o 0 (*false*) sono molto frequenti.

Le più piccole utility UNIX in assoluto, */bin/true* e */bin/false* ritornano 0 ed 1 rispettivamente, come codice di uscita. (Zero come codice di uscita generalmente indica successo, valori diversi da zero indicano errori.)

1.5 Hello, world!

Utilizziamo il famoso esempio dal libro [Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2ed, (1988)]:

Listing 1.13: C/C++ Code

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
}
```

1.5.1 x86

MSVC

Compiliamolo in MSVC 2010:

```
cl 1.cpp /Fa1.asm
```

(l'opzione /Fa indica al compilatore di generare un file con il listato assembly)

Listing 1.14: MSVC 2010

```
CONST   SEGMENT
$SG3830 DB      'hello, world', 0AH, 00H
CONST   ENDS
PUBLIC  _main
EXTRN   _printf:PROC
; Function compile flags: /Odtp
_TEXT   SEGMENT
_main   PROC
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG3830
        call    _printf
        add     esp, 4
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
_TEXT   ENDS
```

MSVC produce codice assembly in sintassi Intel. La differenza tra le sintassi Intel e AT&T-syntax sarà discussa al 1.5.1 on page 15.

Il compilatore ha generato il file, `1.obj`, che deve essere linkato in `1.exe`. Nel nostro caso, il file contiene due segmenti: CONST (per i dati constanti) e _TEXT (per il codice).

La stringa `hello, world` in C/C++ ha tipo `const char[]`[Bjarne Stroustrup, *The C++ Programming Language, 4th Edition*, (2013)p176, 7.3.2], ma non ha un nome

assegnato. Il compilatore deve in qualche modo aver a che fare con la stringa, e la definisce quindi con il nome interno $SG3830.

Questo è il motivo per cui l'esempio potrebbe essere riscritto nel modo seguente:

```
#include <stdio.h>

const char $SG3830[]="hello, world\n";

int main()
{
    printf($SG3830);
    return 0;
}
```

Torniamo al listato assembly. Come possiamo vedere, la stringa è terminata con un byte zero, che è lo standard per la terminazione delle stringhe C/C++. Più informazioni sulle stringhe in C/C++: **??** on page ??.

Nel code segment, _TEXT, esiste fino ad ora solo una funzione: `main()`. La funzione `main()` inizia con il codice di prologo (prologue code) e termina con il codice di epilogo (epilogue code) (come quasi qualunque funzione) [17].

Dopo il prologo della funzione, notiamo la chiamata alla funzione `printf()` : `CALL _printf`. Prima della chiamata, l'indirizzo della stringa (o un puntatore ad essa) contenente il saluto viene messo sullo stack con l'aiuto dell'istruzione PUSH.

Quando la funzione `printf()` restituisce il controllo alla funzione `main()`, l'indirizzo della stringa (o il puntatore) si trova ancora sullo stack. Poichè non ne abbiamo più bisogno, lo stack pointer (il registro ESP) deve essere corretto.

`ADD ESP, 4` significa aggiungi 4 al valore del registro ESP.

Perchè 4? Essendo questo un programma a 32-bit, abbiamo esattamente bisogno di 4 bytes per passare un indirizzo attraverso lo stack. Se fosse stato codice x64 ne sarebbero serviti 8. `ADD ESP, 4` è a tutti gli effetti equivalente a `POP register` ma senza usare alcun registro[18].

Per lo stesso scopo, alcuni compilatori (come l'Intel C++ Compiler) potrebbero emettere l'istruzione `POP ECX` invece di ADD (ad esempio, questo tipo di pattern può essere osservato nel codice di Oracle RDBMS che è compilato con il compilatore Intel C++). Questa istruzione ha pressoché lo stesso effetto ma il contenuto del registro ECX sarà sovrascritto. Il compilatore Intel C++ usa probabilmente `POP ECX` poichè l'opcode di questa istruzione è più corto di `ADD ESP, x` (1 byte per POP contro 3 per ADD).

Ecco un esempio dell'uso di POP al posto di ADD da Oracle RDBMS:

Listing 1.15: Oracle RDBMS 10.2 Linux (Italian text placeholder)

```
.text:0800029A                push    ebx
.text:0800029B                call    qksfroChild
.text:080002A0                pop     ecx
```

[17] Maggiori informazioni si trovano nella sezione su prologo ed epilogo delle funzioni (1.6 on page 40).
[18] i flag CPU vengono comunque modificati

Tuttavia, anche MSVC può farlo.

Listing 1.16: MineSweeper da Windows 7 32-bit

```
.text:0102106F                push    0
.text:01021071                call    ds:time
.text:01021077                pop     ecx
```

Dopo la chiamata a `printf()`, il codice C/C++ originale contiene la direttiva `return 0` —restituisci 0 come risultato dalla funzione `main()`.

Nel codice generato, questa è implementata dall'istruzione `XOR EAX, EAX`.

`XOR` è infatti semplicemente «eXclusive OR, ovvero OR esclusivo»[19] ma i compilatori lo usano spesso al posto di `MOV EAX, 0`—ancora una volta poichè è un opcode leggermente più corto (2 byte per `XOR` contro 5 per `MOV`).

Alcuni compilatori emettono l'istruzione `SUB EAX, EAX`, che significa *sottrai (SUBtract) il valore nel registro* EAX *dal valore nel registro* EAX, che, in ogni caso, risulta uguale a zero.

L'ultima istruzione RET restituisce il controllo al chiamante (chiamante). Solitamente, questo è codice C/C++ CRT[20] , che, a sua volta, restituisce il controllo all' OS[21].

GCC

Proviamo adesso a compilare lo stesso codice C/C++ con il compilatore GCC 4.4.1 su Linux: `gcc 1.c -o 1`. Successivamente, con l'aiuto del disassembler IDA, vediamo come è stata creata la funzione `main()`. IDA, come MSVC, utilizza la sintassi Intel[22].

Listing 1.17: codice in IDA

```
main            proc near

var_10          = dword ptr -10h

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     eax, offset aHelloWorld ; "hello, world\n"
                mov     [esp+10h+var_10], eax
                call    _printf
                mov     eax, 0
                leave
                retn
main            endp
```

[19] wikipedia
[20] C Runtime library
[21] Sistema Operativo (Operating System)
[22] Possiamo anche fare in modo che GCC produca un listato assembly con la sintassi Intel tramite l'opzione `-S -masm=intel`.

Il risultato è pressoché lo stesso. L'indirizzo della stringa `hello, world` (memorizzato nel data segment) è caricato prima nel registro EAX e successivamente salvato sullo stack. Inoltre, il prologo della funzione contiene `AND ESP, 0FFFFFFF0h` —questa istruzione allinea il valore del registro ESP a 16-byte. Ciò fa sì che tutti i valori sullo stack siano allineati allo stesso modo (la CPU è più efficiente se i valori che tratta sono collocati in memoria ad indirizzi allineati a multipli di 4 o 16 byte).

`SUB ESP, 10h` alloca 16 byte sullo stack. Tuttavia, come vedremo a breve, solo 4 sono necessari in questo caso.

Ciò è dovuto al fatto che la dimensione dello stack allocato è anch'essa allineata a 16 byte.

L'indirizzo della stringa (o un puntatore alla stringa) è quindi memorizzato direttamente sullo stack senza utilizzare l'istruzione PUSH. *var_10* — è una variabile locale ed è anche un argomento di `printf()`. Maggiori dettagli in seguito.

Infine viene chiamata la funzione `printf()`.

Diversamente da MSVC, quando GCC compila senza ottimizzazione emette `MOV EAX, 0` invece di un opcode più breve.

L'ultima istruzione, LEAVE —è l'equivalente della coppia di istruzioni `MOV ESP, EBP` e `POP EBP` —in altre parole, questa istruzione riporta indietro lo stack pointer (ESP) e ripristina il registro EBP al suo stato iniziale. Ciò è necessario poiché abbiamo modificato i valori di questi registri (ESP and EBP) all'inizio della funzione (eseguendo `MOV EBP, ESP` / `AND ESP, …`).

GCC: Sintassi AT&T

Vediamo come tutto questo può essere rappresentato nella sintassi assembly AT&T. Questa sintassi è molto più popolare nel mondo UNIX.

Listing 1.18: compiliamo in GCC 4.7.3

```
gcc -S 1_1.c
```

Otteniamo questo:

Listing 1.19: GCC 4.7.3

```
        .file   "1_1.c"
        .section        .rodata
.LC0:
        .string "hello, world\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
```

```
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3"
        .section        .note.GNU-stack,"",@progbits
```

Il listato contiene molte macro (iniziano con il punto). Attualmente non ci interessano.

Per il momento, è solo per una questione di semplificazione, possiamo ignorarle (fatta eccezione per la macro *.string* che codifica una sequenza di caratteri che termina con il null-byte (zero) proprio come una stringa C). Consideriamo soltanto questo [23]:

Listing 1.20: GCC 4.7.3

```
.LC0:
        .string "hello, world\n"
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
```

Alcune delle differenze maggiori tra la sintassi Intel e quella AT&T sono:

- Gli operandi sorgente e destinazione sono scritti in ordine opposto.

  Sintassi Intel: <istruzione> <operando di destinazione> <operando di origine>.

  Sintassi AT&T: <istruzione> <operando di origine> <operando di destinazione>.

  Ecco un modo facile per memorizzare la differenza: quando si tratta di sintassi Intel immagina che ci sia un segno di uguaglianza (=) tra i due operandi, quando si tratta di sintassi AT&T immagina una freccia da sinistra a destra (→) [24].

[23] Questa opzione di GCC può essere usata per eliminare le macro «superflue»: *-fno-asynchronous-unwind-tables*

[24] A proposito, in alcune funzioni standard C(es., memcpy(), strcpy()) gli argomenti sono elencati nello stesso modo della sintassi Intel: prima il puntatore al blocco di memoria di destinazione, e poi il puntatore al blocco di memoria di origine.

- AT&T: Il simbolo di percentuale (%) deve essere scritto prima del nome di un registro, e il dollaro ($) prima dei numeri. Vengono utilizzate le parentesi tonde invece di quelle quadre.
- AT&T: All'istruzione si aggiunge un suffisso che definisce le dimensioni dell'operando:
  - q — quad (64 bit)
  - l — long (32 bit)
  - w — word (16 bit)
  - b — byte (8 bit)

Torniamo al risultato compilato: è identico a quello che abbiamo visto in IDA. Con una piccola differenza: `0FFFFFFF0h` è presentato come `$-16`. E' la stessa cosa: 16 nel sistema decimale è `0x10` in esadecimale. `-0x10` è uguale a `0xFFFFFFF0` (per un tipo di dato a 32-bit).

Ancora una cosa: il valore di ritorno viene settato a 0 usando `MOV`, non `XOR`. `MOV` semplicemente carica un valore in un registro. Il suo nome è fuorviante (il dato non viene spostato, bensì copiato). In altre architetture questa istruzione è chiamata «LOAD» o «STORE» o qualcosa di simile.

String patching (Win32)

Possiamo facilmente trovare la stringa "hello, world" all'interno del file eseguibile utilizzando Hiew:

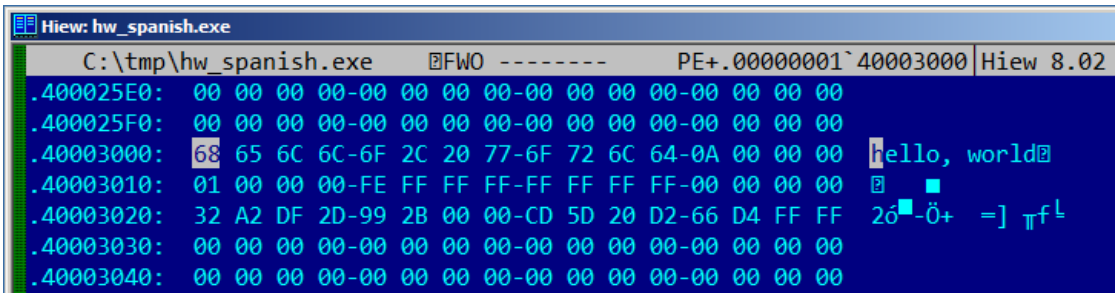


Figura 1.1: Hiew

E possiamo cercare di tradurre il messaggio in spagnolo:

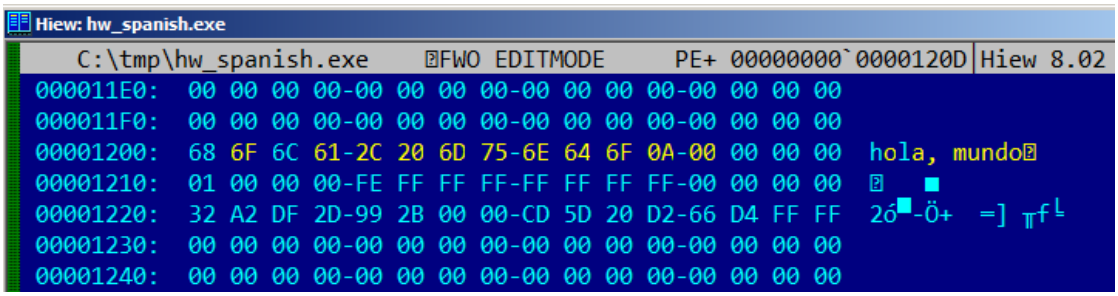


Figura 1.2: Hiew

Il testo in spagnolo è più corto di un byte rispetto a quello inglese, quindi abbiamo aggiunto anche il byte 0x0A al fondo (\n) con un byte zero.

Funziona.

E se volessimo inserire un messaggio più lungo? Ci sono alcuni byte a zero dopo il testo in inglese. E' difficile stabilire se possono essere sovrascritti: potrebbero essere utilizzati da qualche parte all'interno del codice CRT, oppure no. Ad ogni modo, sovrascrivili solo se sai esattamente cosa stai facendo.

String patching (Linux x64)

Proviamo a modificare un eseguibile Linux x64 utilizzando rada.re:

Listing 1.21: rada.re session

```
dennis@bigbox ~/tmp % gcc hw.c

dennis@bigbox ~/tmp % radare2 a.out
 -- SHALL WE PLAY A GAME?
[0x00400430]> / hello
Searching 5 bytes from 0x00400000 to 0x00601040: 68 65 6c 6c 6f
Searching 5 bytes in [0x400000-0x601040]
hits: 1
0x004005c4 hit0_0 .HHhello, world;0.

[0x00400430]> s 0x004005c4

[0x004005c4]> px
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x004005c4  6865 6c6c 6f2c 2077 6f72 6c64 0000 0000  hello, world....
0x004005d4  011b 033b 3000 0000 0500 0000 1cfe ffff  ...;0...........
0x004005e4  7c00 0000 5cfe ffff 4c00 0000 52ff ffff  |...\...L...R...
0x004005f4  a400 0000 6cff ffff c400 0000 dcff ffff  ....l...........
0x00400604  0c01 0000 1400 0000 0000 0000 017a 5200  .............zR.
0x00400614  0178 1001 1b0c 0708 9001 0710 1400 0000  .x..............
0x00400624  1c00 0000 08fe ffff 2a00 0000 0000 0000  ........*.......
0x00400634  0000 0000 1400 0000 0000 0000 017a 5200  .............zR.
0x00400644  0178 1001 1b0c 0708 9001 0000 2400 0000  .x..........$...
```

```
0x00400654  1c00 0000 98fd ffff 3000 0000 000e 1046  ........0......F
0x00400664  0e18 4a0f 0b77 0880 003f 1a3b 2a33 2422  ..J..w...?.;*3$"
0x00400674  0000 0000 1c00 0000 4400 0000 a6fe ffff  ........D.......
0x00400684  1500 0000 0041 0e10 8602 430d 0650 0c07  .....A....C..P..
0x00400694  0800 0000 4400 0000 6400 0000 a0fe ffff  ....D...d.......
0x004006a4  6500 0000 0042 0e10 8f02 420e 188e 0345  e....B....B....E
0x004006b4  0e20 8d04 420e 288c 0548 0e30 8606 480e  . ..B.(..H.0..H.

[0x004005c4]> oo+
File a.out reopened in read-write mode

[0x004005c4]> w hola, mundo\x00

[0x004005c4]> q

dennis@bigbox ~/tmp % ./a.out
hola, mundo
```

Questo è il procedimento: ho cercato la stringa «hello» utilizzando il comando `/`, poi ho impostato il *cursore* (*seek*, in rada.re) a quell'indirizzo. Poi voglio essere sicuro di essere veramente nel posto giusto: `px` mostra un dump dei dati locali. `oo+` imposta rada.re in modalità *read-write*. `w` scrive una stringa ASCII nel *seek* corrente. Nota il `\00` al fondo—è un byte zero. `q` esce (quit).

Questa è una vera storia di cracking di software

Un software di processamento immagini, quando non registrato, aggiungeva trame, come "Questa immagine è stata processata da una versione di prova di [nome del software]", sopra l'immagine. Provando a caso: trovammo la stringa nel file eseguibile e mettemmo degli spazi al suo posto. Le trame scomparirono. Tecnicamente parlando, continuavamo ad essere presenti. Tramite le funzioni Qt, le trame continuavano ad essere aggiunte all'immagine risultante. Ma aggiungendo spazi l'immagine non era alterata...

La *traduzione* del software all'epoca del MS-DOS

Questo era un metodo comune per tradurre i software per MS-DOS durante gli anni '80 e '90. A volte le parole e le frasi sono leggermente più lunghe rispetto ai corrispettivi in inglese, per questo motivo i software *adattati* hanno molti acronimi strani ed abbreviazioni difficili da comprendere.

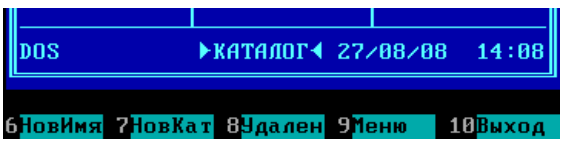


Figura 1.3: Italian text placeholder

Probabilmente questo è successo in molti Paesi durante quel periodo.

1.5.2 x86-64

MSVC: x86-64

Proviamo anche con MSVC a 64-bit:

Listing 1.22: MSVC 2012 x64

```
$SG2989 DB      'hello, world', 0AH, 00H

main    PROC
        sub     rsp, 40
        lea     rcx, OFFSET FLAT:$SG2989
        call    printf
        xor     eax, eax
        add     rsp, 40
        ret     0
main    ENDP
```

In x86-64, tutti i registri sono stati estesi a 64-bit ed il loro nome ha il prefisso R-. Per usare lo stack meno spesso (in altre parole, per accedere meno spesso alla memoria esterna/cache), esiste un metodo molto diffuso per passare gli argomenti delle funzioni tramite i registri (*fastcall*) **??** on page ??. Ovvero, una parte degli argomenti viene passata attraverso i registri, il resto —attraverso lo stack. In Win64, 4 argomenti di funzione sono passati nei registri RCX, RDX, R8, R9. Questo è ciò che vediamo qui: un puntatore alla stringa per `printf()` è adesso passato nel registro RCX anziché tramite lo stack. I puntatori adesso sono a 64-bit, quindi vengono passati nei registri a 64-bit (aventi il prefisso R-). E' comunque possibile, per retrocompatibilità, accedere alle parti a 32-bit, usando il prefisso E-. I registri RAX/EAX/AX/AL in x86-64 appaiono così:

<table>
<tr><td colspan="8">Numero byte</td></tr>
<tr><td>7°</td><td>6°</td><td>5°</td><td>4°</td><td>3°</td><td>2°</td><td>1°</td><td>0</td></tr>
<tr><td colspan="8">RAX^{x64}</td></tr>
<tr><td colspan="4"></td><td colspan="4">EAX</td></tr>
<tr><td colspan="6"></td><td colspan="2">AX</td></tr>
<tr><td colspan="6"></td><td>AH</td><td>AL</td></tr>
</table>

La funzione `main()` restituisce un valore di tipo *int*, che in C/C++, per migliore retrocompatibilità e portabilità, resta ancora a 32-bit, motivo per cui il registro EAX (quindi la parte a 32 bit del registro) viene svuotato invece di RAX alla fine della funzione. Ci sono anche 40 byte allocati nello stack locale. Questo spazio è detto «shadow space», e ne parleremo più avanti: 1.14.2 on page 133.

GCC: x86-64

Proviamo anche GCC in Linux a 64-bit:

Listing 1.23: GCC 4.4.6 x64

```
.string "hello, world\n"
main:
        sub     rsp, 8
```

```
        mov     edi, OFFSET FLAT:.LC0 ; "hello, world\n"
        xor     eax, eax  ; numbero dei registri vettore passati
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

Linux, *BSD e Mac OS X usano anche un metodo per passare argomenti di funzione nei registri. [Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] [25].

I primi 6 argomenti sono passati nei registri RDI, RSI, RDX, RCX, R8, R9, ed il resto—tramite lo stack.

Quindi il puntatore alla stringa viene passato in EDI (la parte a 32-bit del registro). Ma perchè non utilizza la parte a 64-bit, RDI?

E' importante ricordare che tutte le istruzioni MOV in modalità 64-bit che scrivono qualcosa nella parte dei 32-bit bassa di un registro, azzera anche la parte a 32-bit alta (come indicato nei manuali Intel: 8.1.4 on page 308).
Ad esempio, MOV EAX, 011223344h scrive correttamente un valore in RAX, poichè i bit della parte alta verranno azzerati.

Se apriamo il file oggetto compilato (.o), possiamo anche vedere gli opcode di tutte le istruzioni [26]:

Listing 1.24: GCC 4.4.6 x64

```
.text:00000000004004D0                  main  proc near
.text:00000000004004D0 48 83 EC 08      sub     rsp, 8
.text:00000000004004D4 BF E8 05 40 00   mov     edi, offset format ; "hello,
    world\n"
.text:00000000004004D9 31 C0            xor     eax, eax
.text:00000000004004DB E8 D8 FE FF FF   call    _printf
.text:00000000004004E0 31 C0            xor     eax, eax
.text:00000000004004E2 48 83 C4 08      add     rsp, 8
.text:00000000004004E6 C3               retn
.text:00000000004004E6                  main  endp
```

Come possiamo notare, l'istruzione che scrive in EDI all'indirizzo 0x4004D4 occupa 5 byte. La stessa istruzione che scrive un valore a 64-bit in RDI occupa 7 bytes. Apparentemente, GCC sta cercando di risparmiare un po' di spazio. Inoltre, può essere sicuro che il segmento dati contenente la stringa non sarà allocato ad indirizzi maggiori di 4GiB.

Notiamo anche che il registro EAX è stato azzerato prima della chiamata alla funzione printf(). Questo viene fatto perché, in base allo standard ABI[27] citato in precedenza, il numero dei registri vettore usati deve essere passato in EAX nei sistemi *NIX su x86-64.

[25] Italian text placeholderhttps://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

[26] Questo deve essere abilitato in **Options** → **Disassembly** → **Number of opcode bytes**

[27] Application Binary Interface

Address patching (Win64)

Se il nostro esempio venisse compilato in MSVC 2013 utilizzando lo switch `/MD` (che significa un eseguibile più piccolo a causa del link dei file `MSVCR*.DLL`), la funzione `main()` verrebbe prima, e può essere trovata facilmente:

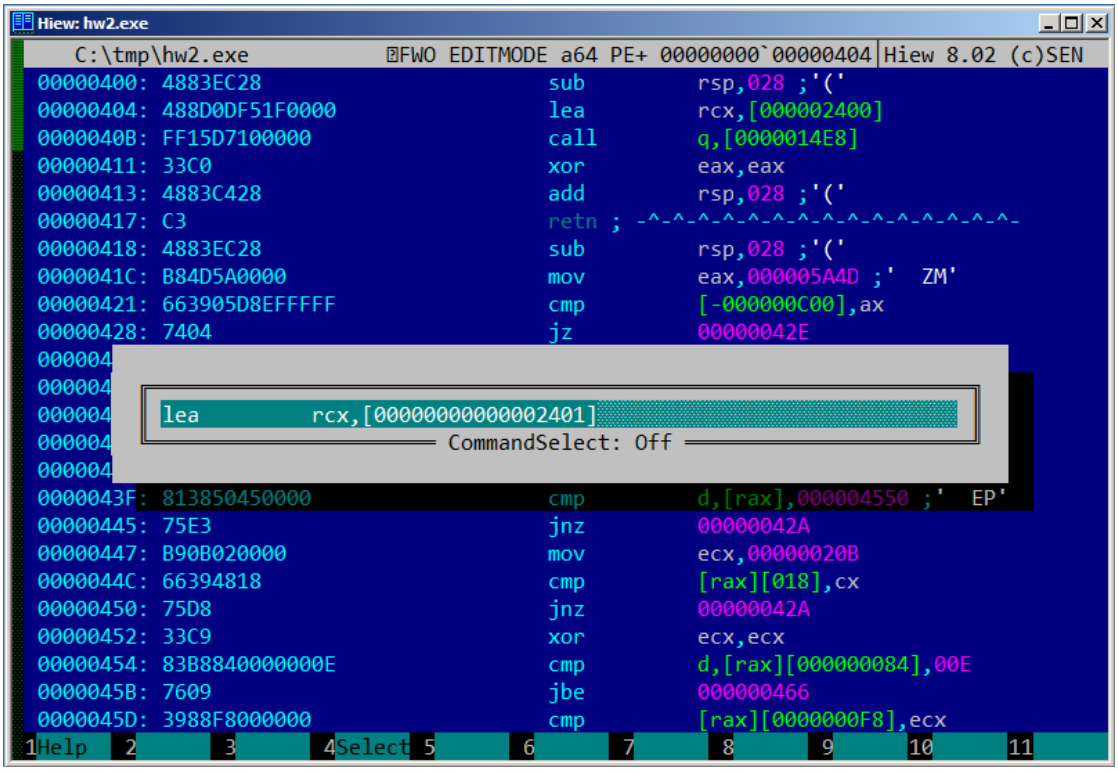


Figura 1.4: Hiew

Come esperimento, possiamo incrementare l'indirizzo di 1:

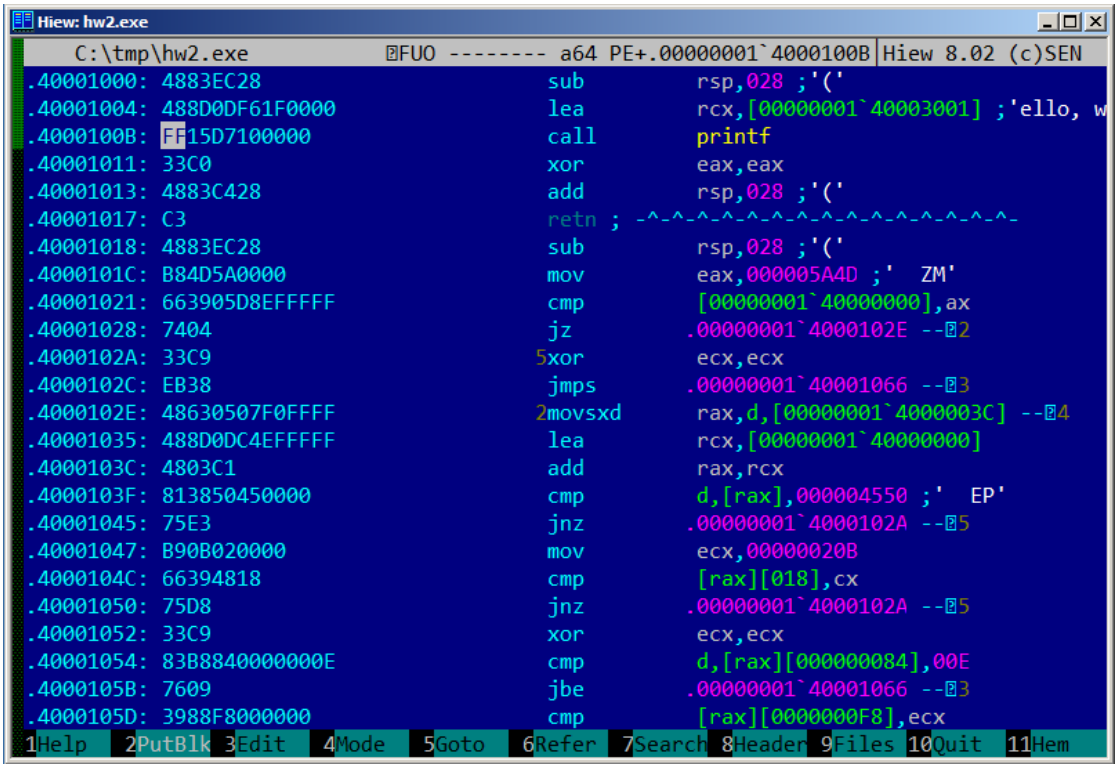


Figura 1.5: Hiew

Hiew mostra «ello, world». E quando lanciamo l'eseguibile modificato, viene stampata proprio questa stringa.

Scegliere un'altra stringa dall'immagine binaria (Linux x64)

Il file binario che si ottiene compilando il nostro esempio tramite GCC 5.4.0 su Linux x64 contiene molte altre stringhe di testo. Si tratta principalmente di nomi di funzioni e librerie importate.

Esegui objdump per ottenere il contenuto di tutte le sezioni del file compilato:

```
$ objdump -s a.out

a.out:     file format elf64-x86-64

Contents of section .interp:
 400238 2f6c6962 36342f6c 642d6c69 6e75782d  /lib64/ld-linux-
 400248 7838362d 36342e73 6f2e3200           x86-64.so.2.
Contents of section .note.ABI-tag:
 400254 04000000 10000000 01000000 474e5500  ............GNU.
 400264 00000000 02000000 06000000 20000000  ............ ...
Contents of section .note.gnu.build-id:
```

```
 400274 04000000 14000000 03000000 474e5500  ............GNU.
 400284 fe461178 5bb710b4 bbf2aca8 5ec1ec10  .F.x[.......^...
 400294 cf3f7ae4                             .?z.

...
```

Non è un problema passare l'indirizzo della stringa di test «/lib64/ld-linux-x86-64.so.2» a `printf()`:

```
#include <stdio.h>

int main()
{
    printf(0x400238);
    return 0;
}
```

E' difficile da credere, ma questo codice stampa la stringa citata prima.

Se cambiassi l'indirizzo a `0x400260`, verrebbe stampata la stringa «GNU». Questo indirizzo è corretto per la mia specifica versione di GCC, GNU toolset, etc. Sul tuo sistema, l'eseguibile potrebbe essere leggermente differente, e anche tutti gli indirizzi sarebbero differenti. Inoltre, aggiungendo o rimuovendo del codice in/da questo codice sorgente probabilmente sposterebbe tutti gli indirizzi in avanti o indietro.

1.5.3 ARM

Per gli esperimenti con i processori ARM, sono stati utilizzati diversi compilatori:

- Diffuso nel settore embedded: Keil Release 6/2013.
- Apple Xcode 4.6.3 IDE con il compilatore LLVM-GCC 4.2 [28].
- GCC 4.9 (Linaro) (per ARM64), disponibile per win32 su http://www.linaro.org/projects/armv8/.

Il codice ARM a 32 bit è utilizzato (incluse le modalità Thumb e Thumb-2) in tutti i casi in questo libro, se non specificato differentemente. Quando parliamo di ARM a 64 bit, lo chiamiamo ARM64.

Senza ottimizzazione Keil 6/2013 (Modalità ARM)

Iniziamo a compilare il nostro esempio in Keil:

```
armcc.exe --arm --c90 -O0 1.c
```

Il compilatore *armcc* produce un listato assembly con sintassi Intel, e utilizza macro di alto livello legate al processore ARM [29], tuttavia è più importante per noi vedere le istruzioni «così come sono», quindi guardiamo in IDA il risultato compilato.

[28] Apple Xcode 4.6.3 utilizza il compilatore open-source GCC come compilatore front-end ed il generatore di codice LLVM

[29] ad esempio, la modalità ARM è priva delle istruzioni PUSH/POP

Listing 1.25: Senza ottimizzazione Keil 6/2013 (Modalità ARM) IDA

```
.text:00000000             main
.text:00000000 10 40 2D E9    STMFD   SP!, {R4,LR}
.text:00000004 1E 0E 8F E2    ADR     R0, aHelloWorld ; "hello, world"
.text:00000008 15 19 00 EB    BL      __2printf
.text:0000000C 00 00 A0 E3    MOV     R0, #0
.text:00000010 10 80 BD E8    LDMFD   SP!, {R4,PC}

.text:000001EC 68 65 6C 6C+aHelloWorld  DCB "hello, world",0    ; DATA XREF:
    main+4
```

Nell'esempio possiamo facilmente vedere che ogni istruzione ha lunghezza pari a 4 byte. Difatti abbiamo compilato il codice per la modalità ARM e non Thumb.

La prima istruzione, `STMFD SP!, {R4,LR}`[30], funziona come l'istruzione PUSH in x86, scrivendo i valori di due registri (R4 e LR) nello stack.

Infatti il listato di output prodotto dal compilatore *armcc*, per semplificazione, mostra l'istruzione `PUSH {r4,lr}`. Ma questo non è del tutto esatto. L'istruzione PUSH esiste solo in modalità Thumb. Utilizziamo quindi IDA per non fare confusione.

Questa istruzione dapprima decrementa il valore di **SP!**[32] così da farlo puntare alla porzione dello stack che è libera di ospitare nuovi dati, quindi salva il valore dei registri R4 e LR all'indirizzo memorizzato nel registro **SP!** appena modificato.

Questa istruzione (esattamente come PUSH in Thumb mode) è in grado di salvare il valore di più registri contemporaneamente, cosa che può risultare molto utile. A proposito, non esiste un equivalente in x86. Si può notare anche che l'istruzione STMFD è una generalizzazione dell'istruzione PUSH (estendendone le sue funzionalità), poichè può funzionare con qualunque registro, e non solo **SP!**. In altre parole, STMFD può essere usata per memorizzare un insieme di registri all'indirizzo di memoria specificato.

L'istruzione `ADR R0, aHelloWorld` aggiunge o sottrae il valore del registro **PC!**[33] all'offset in cui è memorizzata la stringa `hello, world`. Ci si potrebbe chiedere, come è utilizzato in questo caso il registro PC? Questo viene detto «codice indipendente dalla posizione»[34].

Questo tipo di codice può essere eseguito ad un indirizzo non fisso in memoria. In altre parole, si tratta di un indirizzamento relativo a **PC!** (**PC!**-relative addressing). L'istruzione ADR tiene conto della differenza tra l'indirizzo di questa istruzione e l'indirizzo dove si trova la stringa. Questa differenza (offset) dovrà sempre essere la stessa, a prescindere dall'indirizzo in cui nostro codice sarà caricato dall'OS. Ciò spiega perchè bisogna soltanto aggiungere l'indirizzo dell'istruzione corrente (tramite **PC!**) per ottenere l'indirizzo assoluto in memoria della nostra stringa C.

L'istruzione `BL __2printf`[35] chiama la funzione `printf()`. Questa istruzione funziona così:

[30] STMFD[31]
[32] **SP!**
[33] **PC!**
[34] Maggiori informazioni sono fornite nella relativa sezione (**??** on page ??)
[35] Branch with Link

- memorizza l'indirizzo successivo all'istruzione BL (0xC) nel registro LR;
- quindi passa il controllo a `printf()` scrivendo il suo indirizzo nel registro **PC!**.

Quando la funzione `printf()` termina la sua esecuzione, deve sapere a chi restituire il controllo (dove ritornare). Per questo motivo ogni funzione passa il controllo all'indirizzo memorizzato nel registro LR.

Questa è una differenza tra processori RISC «puri» come ARM e processori CISC[36] come x86, nei quali il return address viene solitamente memorizzato nello stack Maggiori informazioni si trovano nella prossima sezione (1.9 on page 41).

A proposito, un indirizzo assoluto o un offset a 32-bit non può essere codificato nell'istruzione a 32-bit BL poichè ha solo spazio per 24 bit. Come potremmo ricordare, tutte le istruzioni in ARM-mode hanno dimensione fissa di 4 byte (32 bit). Dunque possono essere collocate solo su indirizzi allineati su 4-byte. Ciò implica che gli ultimi 2 bit dell'indirizzo dell'istruzione (che sono sempre zero) possono essere omessi. Abbiamo in definitiva 26 bit per la codifica dell'offset (offset encoding). E cio è sufficiente per codificare $current\_PC \pm \approx 32M$.

L'istruzione successiva, `MOV R0, #0`[37] scrive semplicemente 0 nel registro R0. Questo perchè la nostra funzione C restituisce 0, ed il valore di ritorno deve essere memorizzato nel registro R0.

L'ultima istruzione `LDMFD SP!, R4,PC`[38]. Carica valori dallo stack (o qualunque altra zona di memoria) per salvarli nei registri R4 e **PC!**, e incrementa lo stack pointer **SP!**. In questo caso funziona come POP.
N.B. La prima istruzione STMFD aveva salvato la coppia di registri R4 e LR sullo stack, ma R4 e **PC!** vengono *ripristinati* durante l'esecuzione di LDMFD.

Come già sappiamo, l'indirizzo del posto a cui ogni funzione devere restituire il controllo è solitamente memorizzato nel registro LR. La prima istruzione salva il suo valore nello stack perchè lo stesso registro sarà utilizzato dalla nostra funzione `main()` per la chiamata a `printf()`. Al termine della funzione, questo valore può essere scritto direttamente nel registro **PC!**, passando di fatto il controllo al punto in cui la nostra funzione era stata chiamata.

Dal momento che `main()` è solitamente la funzione principale in C/C++, il controllo verrà restituito al loader dell' OS oppure ad un punto in una CRT, o qualcosa del genere.

Tutto questo consente di omettere l'istruzione `BX LR` alla fine della funzione.

`DCB` è una direttiva assembly che definisce un array di byte o una stringa ASCII, analoga alla direttiva DB nel linguaggio assembly x86.

Senza ottimizzazione Keil 6/2013 (Modalità Thumb)

Compiliamo lo stesso esempio usando Keil in Thumb mode:

```
armcc.exe --thumb --c90 -O0 1.c
```

[36] Complex Instruction Set Computing
[37] abbreviazione di MOVe
[38] LDMFD[39] è l'istruzione inversa rispetto a STMFD

Otteniamo (in IDA):

Listing 1.26: Senza ottimizzazione Keil 6/2013 (Modalità Thumb) + IDA

```
.text:00000000             main
.text:00000000 10 B5          PUSH    {R4,LR}
.text:00000002 C0 A0          ADR     R0, aHelloWorld ; "hello, world"
.text:00000004 06 F0 2E F9    BL      __2printf
.text:00000008 00 20          MOVS    R0, #0
.text:0000000A 10 BD          POP     {R4,PC}

.text:00000304 68 65 6C 6C+aHelloWorld  DCB "hello, world",0    ; DATA XREF:
    main+2
```

Possiamo facilmente individuare gli opcode a 2-byte (16-bit). Questo è, come già detto, Thumb. L'istruzione BL , tuttavia, è composta da due istruzioni a 16-bit. Ciò accade perchè è impossibile caricare un offset per la funzione `printf()` usando il poco spazio a disposizione in un opcode a 16-bit. Pertanto la prima istruzione a 16-bit carica i 10 bit alti dell'offset e la seconda istruzione carica gli 11 bit più bassi dell'offset.

Come già detto, tutte le istruzione in Thumb mode hanno dimensione pari a 2 bytes (o 16 bit). Ciò implica che è impossibile trovare un'istruzione Thumb ad un indirizzo dispari. Per questo motivo, l'ultimo bit dell'indirizzo può essere omesso nell'encoding delle istruzioni.

Per riassumere, l'istruzione Thumb BL può codificare un indirizzo in $current\_PC \pm \approx 2M$.

Riguardo le altre istruzioni nella funzione: PUSH e POP qui funzionano come STMFD/LDMFD con l'unica differenza che il registro **SP!** in questo caso non viene menzionato esplicitamente. ADR funziona esattamente come nell'esempio precedente. MOVS scrive 0 nel registro R0 per restituire zero.

Con ottimizzazione Xcode 4.6.3 (LLVM) (Modalità ARM)

Xcode 4.6.3 senza ottimizzazioni produce un sacco di codice ridondante, perciò studieremo l'output ottimizzato in cui le le istruzioni sono il meno possibile, settando lo switch del compilatore `-03`.

Listing 1.27: Con ottimizzazione Xcode 4.6.3 (LLVM) (Modalità ARM)

```
__text:000028C4             _hello_world
__text:000028C4 80 40 2D E9   STMFD           SP!, {R7,LR}
__text:000028C8 86 06 01 E3   MOV             R0, #0x1686
__text:000028CC 0D 70 A0 E1   MOV             R7, SP
__text:000028D0 00 00 40 E3   MOVT            R0, #0
__text:000028D4 00 00 8F E0   ADD             R0, PC, R0
__text:000028D8 C3 05 00 EB   BL              _puts
__text:000028DC 00 00 A0 E3   MOV             R0, #0
__text:000028E0 80 80 BD E8   LDMFD           SP!, {R7,PC}

__cstring:00003F62 48 65 6C 6C+aHelloWorld_0  DCB "Hello world!",0
```

Le istruzioni STMFD e LDMFD ci sono già familiari.

L'istruzione MOV scrive il numero 0x1686 nel registro R0 . Questo è l'offset che punta alla stringa «Hello world!» .

Il registro R7 (per come standardizzato in [*iOS ABI Function Call Guide*, (2010)][40]) è un puntatore di frame (frame pointer). Maggiori informazioni in basso.

L'istruzione MOVT R0, #0 (MOVe Top) scrive 0 nei 16 bit alti (higher 16 bits) del registro. Il problema qui è che l'istruzione generica MOV in ARM mode potrebbe scrivere solo i 16 bit bassi del registro.

Ricorda che tutti gli opcode delle istruzioni in ARM mode sono limitati ad una lunghezza di 32 bit. Ovviamente questa limitazione non riguarda lo spostamento dei dati tra registri. Per questo motivo esiste l'istruzione aggiuntiva MOVT per scrivere nelle parti alte dei registri (nei bit da 16 a 31, inclusi). Il suo uso qui è comunque ridondante, perchè l'istruzione MOV R0, #0x1686 di sopra ha azzerato la parte alta del registro. Si tratta probabilmente di un difetto/svista del compilatore.

L'istruzione ADD R0, PC, R0 aggiunge il valore in **PC!** al valore in R0, per calcolare l'indirizzo assoluto della stringa «Hello world!». Come sappiamo, si tratta di «codice indipendente dalla posizione» e quindi questa correzione risulta essenziale in questo caso.

L'istruzione BL chiama la funzione puts() ivece di printf().

LLVM ha sostituito la prima chiamata a printf() con puts(). Infatti: printf() con un solo argomento è quasi analoga a puts().

Quasi, perchè le due funzioni producono lo stesso risultato solo nel caso in cui la stringa non contiene identificatori di formato (format identifiers) che iniziano con %. In caso contrario l'effetto di queste due funzioni sarebbe diverso [41].

Perchè il compilatore ha sostituito printf() con puts()? Probabilmente perchè puts() è più veloce [42].

Poichè passa direttamente i caratteri a stdout senza confrontare ciascuno di essi con il simbolo %.

Andando avanti, vediamo la familiare istruzione MOV R0, #0 che imposta il registro R0 a 0.

Con ottimizzazione Xcode 4.6.3 (LLVM) (Modalità Thumb-2)

Di default Xcode 4.6.3 genera codice per Thumb-2 in questo modo:

Listing 1.28: Con ottimizzazione Xcode 4.6.3 (LLVM) (Modalità Thumb-2)

```
__text:00002B6C                     _hello_world
__text:00002B6C 80 B5           PUSH            {R7,LR}
__text:00002B6E 41 F2 D8 30     MOVW            R0, #0x13D8
```

[40] Italian text placeholderhttp://developer.apple.com/library/ios/documentation/Xcode/Conceptual/iPhoneOSABIReference/iPhoneOSABIReference.pdf

[41] Bisogna anche notare che puts() non richiede un simbolo new line '\n' alla fine della stringa, per questo non lo vediamo qui.

[42] ciselant.de/projects/gcc_printf/gcc_printf.html

```
__text:00002B72 6F 46          MOV             R7, SP
__text:00002B74 C0 F2 00 00    MOVT.W          R0, #0
__text:00002B78 78 44          ADD             R0, PC
__text:00002B7A 01 F0 38 EA    BLX             _puts
__text:00002B7E 00 20          MOVS            R0, #0
__text:00002B80 80 BD          POP             {R7,PC}

...

__cstring:00003E70 48 65 6C 6C 6F 20+aHelloWorld  DCB "Hello world!",0xA,0
```

Le istruzioni BL e BLX in Thumb mode, come ricordiamo, sono codificate con una coppia di istruzioni 16-bit. In Thumb-2 questi opcode *surrogati* sono estesi in modo tale che le nuove istruzioni possano essere codificate in istruzioni a 32-bit.

Ciò appare ovvio considerando che che gli opcodes delle istruzioni Thumb-2 iniziano sempre con `0xFx` o `0xEx`.

Ma nel listato IDA i byte degli opcode sono invertiti poichè per i processori ARM le istruzioni sono codificate secondo il seguente principio: l'ultimo byte viene prima ed è seguito dal primo byte (per le modalità Thumb e Thumb-2) oppure, per istruzioni in ARM mode il quarto byte viene prima, seguito dal terzo, dal secondo ed infine dal primo (a causa della diversa endianness).

Quindi i byte nei listati IDA sono collocati cosi':

- per ARM and ARM64 modes: 4-3-2-1;
- per Thumb mode: 2-1;
- per coppie di istruzioni a 16-bit in Thumb-2 mode: 2-1-4-3.

Come possiamo vedere, le istruzioni `MOVW`, `MOVT.W` e `BLX` iniziano con `0xFx`.

Ona delle istruzioni Thumb-2 è `MOVW R0, #0x13D8` —memorizza un valore a 16-bit nella parte bassa del registro `R0` , azzerando i bit più alti.

Allo stesso modo, `MOVT.W R0, #0` funziona come `MOVT` nel precedente esempio, ma in Thumb-2.

Tra le altre differenze, l'istruzione BLX in questo caso è usata al posto di BL.

La differenza sta nel fatto che, oltre a salvare RA[43] nel registro LR e passare il controllo alla funzione `puts()`, il processore passa dalla modalità Thumb/Thumb-2 alla modalità ARM mode (o viceversa).

Questa istruzione è posta qui poichè l'istruzione a cui il controllo viene passato appare così (è codificata in ARM mode):

```
__symbolstub1:00003FEC _puts           ; CODE XREF: _hello_world+E
__symbolstub1:00003FEC 44 F0 9F E5     LDR  PC, =__imp__puts
```

Si tratta essenzialmente di un jump alla zona dove è scritto l'indirizzo di `puts()` nella imports section.

[43] Indirizzo di Ritorno

Il lettore attento potrebbe chiedere: perchè non chiamare `puts()` proprio nel punto del codice dove serve effettivamente?

Perchè non è efficiente in termini di spazio.

Quasi tutti i programmi utilizzano librerie esterne dinamiche (come le DLL in Windows, .so in *NIX o .dylib in Mac OS X). Le librerie dinamiche contengono funzioni usate di frequente, inclusa la funzione C standard `puts()`.

In un file eseguibile (Windows PE .exe, ELF o Mach-O) è presente una sezione di import (import section). Si tratta di una lista di simboli (symbols - funzioni o variabili globali) importata da moduli esterni insieme ai nomi dei moduli stessi.

Il loader dell' OS carica tutti i moduli necessari e, mentre enumera gli import symbols nel modulo primario, determina gli indirizzi corretti per ciascun simbolo.

Nel nostro caso, *__imp__puts* è una variabile a 32-bit usata dal loader dell'OS per memorizzare l'indirizzo corretto della funzione in una libreria esterna. Successivamente l'istruzione LDR legge semplicemente il valore a 32-bit da questa variabile e lo scrive nel registro **PC!**, passando il controllo ad esso.

Quindi, per ridurre il tempo necessario al loader dell'OS per completare questa procedura, è una buona idea scrivere l'indirizzo di ogni simbolo solo una volta, in un punto dedicato.

Inoltre, come abbiamo già capito, è impossibile caricare un valore a 32-bit in un registro utilizzando solo una istruzione senza accedere alla memoria.

Pertanto, la soluzione ottimale è quella di allocare una funzione separata, che funziona in ARM mode, con il solo scopo di passare il controllo alla libreria dinamica e quindi saltare dal codice Thumb a questa piccola funzione di una sola istruzione (la cosiddetta Funzione thunk).

A proposito, nel precedente esempio (compilato per ARM mode) il controllo viene passato da BL alla stessa Funzione thunk. La modalità del processore però non viene cambiata (da cui l'assenza di una «X» nella instruction mnemonic).

Altre informazioni sulle funzioni thunk

Le thunk-functions sono difficili da comprendere, apparentemente, a causa di una denominazione impropria. Il modo migliore per capirle è pensarle come adattatori o convertitori da un tipo di jack ad un altro. Ad esempio, un adattatore che consente l'inserimento di una spina elettrica Inglese in una presa Americana, o viceversa. Le thunk functions sono a volte anche dette *wrappers*.

Seguono altre descrizioni di queste funzioni:

> "Un pezzo di codice che fornisce un indirizzo:", secondo to P. Z. Ingerman, che ha inventato le thunk nel 1961 come un modo per legare i parameters alle loro definizioni formali nelle chiamate a procedura in Algol-60. Se una procedura viene chiamata con un'espressione al posto di un parametro formale, il compilatore genera una thunk che

> calcola l'espressione e lascia l'indirizzo del risultato in una posizione standard.
>
> ...
>
> Microsoft e IBM hanno definito, nei loro sistemi basati su Intel, un "ambiente a 16-bit" (con orrendi registri di segmento e limitazioni di indirizzi a 64K) e un "ambiente a 32-bit" (con indirizzamento piatto (flat) e gestione della memoria semi reale). I due ambienti possono girare contemporaneamente sullo stesso computer e OS (grazie a quello che, nel mondo Microsoft, è chiamato WOW, acronimo per Windows On Windows). MS e IBM hanno entrambi deciso che il processo di passare da 16 a 32 bit e viceversa è detto un "thunk"; in Windows 95, esiste anche un tool, THUNK.EXE, detto "thunk compiler".

(The Jargon File)

Un ulteriore esempio possiamo trovarlo all'interno della libreria LAPACK—un "Linear Algebra PACKage" scritto in FORTRAN. Anche gli sviluppatori C/C++ vogliono utilizzare LAPACK, ma non è pensabile riscriverla in C/C++ e mantenere diverse versioni. Esistono quindi delle piccole funzioni C chiamabili da un ambiente C/C++, che a loro volta chiamano le funzioni FORTRAN, e non fanno quasi nient'altro:

```
double Blas_Dot_Prod(const LaVectorDouble &dx, const LaVectorDouble &dy)
{
    assert(dx.size()==dy.size());
    integer n = dx.size();
    integer incx = dx.inc(), incy = dy.inc();

    return F77NAME(ddot)(&n, &dx(0), &incx, &dy(0), &incy);
}
```

Anche questo tipo di funzioni vengono chiamate "wrapper".

ARM64

GCC

Compiliamo l'esempio con GCC 4.8.1 per ARM64:

Listing 1.29: Senza ottimizzazione GCC 4.8.1 + objdump

```
0000000000400590 <main>:
  400590:       a9bf7bfd        stp     x29, x30, [sp,#-16]!
  400594:       910003fd        mov     x29, sp
  400598:       90000000        adrp    x0, 400000 <_init-0x3b8>
  40059c:       91192000        add     x0, x0, #0x648
  4005a0:       97ffffa0        bl      400420 <puts@plt>
  4005a4:       52800000        mov     w0, #0x0  // #0
  4005a8:       a8c17bfd        ldp     x29, x30, [sp],#16
  4005ac:       d65f03c0        ret

```

```
11 ...
12
13 Contents of section .rodata:
14  400640 01000200 00000000 48656c6c 6f210a00  ........Hello!..
```

In ARM64 non ci sono le modalità Thumb e Thumb-2, ma solo ARM, quindi esistono soltanto istruzioni a 32-bit. Il numero di registri è raddoppiato: **??** on page ??. I registri a 64-bit hanno il prefisso `X-` prefixes, mentre le loro parti a 32-bit hanno il prefisso — `W-`.

L'istruzione STP (*Store Pair*) salva simultaneamente due registri nello stack: `X29` e `X30`.

Questa istruzione può ovviamente salvare la coppia di valori in una posizione arbitraria in memoria, tuttavia in questo caso è specificato il registro **SP!**, e di conseguenza la coppia viene salvata nello stack.

I registri ARM64 sono a 64-bit, ognuno di essi ha dimensione pari a 8 byte, quindi sono necessari 16 byte per salvare i due registri.

Il punto esclamativo ("!") dopo l'operando sta a significare che 16 deve esse prima sottratto da **SP!**, e solo successivamente i valori devono essere scritti nello stack.

Questo è anche detto *pre-index*. Per le differenze tra *post-index* e *pre-index* leggere qui: **??** on page ??.

Quindi, in termini del più familiare x86, la prima istruzione è semplicamente l'analogo della coppia `PUSH X29` e `PUSH X30`. `X29` in ARM64 è usato come FP[44], e `X30` come LR, e questo spiega perchè sono salvati nel prologo della funzione e ripristinati nell'epilogo.

La seconda istruzione copia **SP!** in `X29` (o FP). Ciò viene fatto per impostare lo stack frame della funzione.

Le istruzioni `ADRP` e `ADD` sono usate per inserire l'indirizzo della stringa «Hello!» nel registro `X0`, poichè il primo argomento della funzione viene passato in questo registro.

Non esiste alcun tipo di istruzione in ARM in grado di salvare un numero molto grande in un registro (perchè la lunghezza delle istruzioni è limitata a 4 byte, maggiori informazioni qui: **??** on page ??). Perciò devono essere utilizzate più istruzioni. La prima (`ADRP`), scrive l'indirizzo della pagina di 4KiB (4KiB page) in cui si trova la stringa, nel registro `X0`, e la seconda (`ADD`) aggiunge semplicemente il resto dell'indirizzo. Maggiori informazioni su questo tema: **??** on page ??.

`0x400000 + 0x648 = 0x400648`, e vediamo la nostra C-string «Hello!» nel `.rodata` data segment a questo indirizzo.

`puts()` viene chiamata subito dopo usando l'istruzione `BL`. Questo è già stato discusso: 1.5.3 on page 28.

`MOV` scrive 0 in `W0`. `W0` è la parte bassa a 32 bits del registro a 64-bit `X0`:

[44] Frame Pointer

Parte alta dei 32 bit	Parte bassa dei 32 bit
X0	
	W0

Il risultato della funzione viene restituito tramite `X0` e `main()` restituisce 0, quindi è in questo modo che viene preparato il valore da restituire. Ma perchè usare la parte a 32-bit?

Perchè il tipo *int* in ARM64, esattamente come in x86-64, è ancora a 32 bit, per maggiore compatibilità. Quindi se una funzione restituisce un *int* a 32 bit, solo la parte più bassa a 32 bit del registro `X0` verrà utilizzata.

Per verificare quanto detto, cambiamo leggermente l'esempio e ricompiliamolo. Adesso `main()` restituisce un valore a 64-bit:

Listing 1.30: `main()` che ritorna un valore di tipo `uint64_t`

```
#include <stdio.h>
#include <stdint.h>

uint64_t main()
{
        printf ("Hello!\n");
        return 0;
}
```

Il risultato è lo stesso, ma quell'istruzione MOV adesso appare così:

Listing 1.31: Senza ottimizzazione GCC 4.8.1 + objdump

```
  4005a4:       d2800000        mov     x0, #0x0        // #0
```

LDP (*Load Pair*) infine riprisina i registri X29 e X30.

Non c'è il punto esclamativo dopo l'istruzione: ciò implica che il valore viene prima caricato dallo stack, e solo successivamente **SP!** è incrementato di 16. Questo viene detto *post-index*.

Una nuova istruzione è apparsa in ARM64: RET. Funziona esattamente come `BX LR`, con l'aggiunta di uno speciale *hint* bit, che informa la CPU del fatto che si tratta di un ritorno da una funzione, e non soltanto una normale istruzione jump, in questo modo può venire eseguita in modo più ottimale.

A causa della semplicità della funzione, GCC con le opzioni di ottimizzazione genera esattamente lo stesso codice.

1.5.4 MIPS

Qualche parola sul «global pointer»

Un importante concetto MIPS è il «global pointer». Come potremmo già sapere, ogni istruzione MIPS ha lunghezza pari a 32 bit, quindi è impossibile inserire un indirizzo a 32-bit in una sola istruzione: occorre utilizzarne una coppia (come ha fatto GCC nell'esempio per il caricamento dell'indirizzo della stringa). E' comunque possibile

caricare dati da un indirizzo nell'intervallo $register - 32768 ... register + 32767$ utilizzando una singola istruzione (perchè 16 bit di un signed offset possono essere codificati in una singola istruzione). Possiamo quindi allocare un registro per questo scopo e allocare anche un'area di 64KiB per i dati più utilizzati. Questo registro dedicato è detto «global pointer» e punta in mezzo all'area di 64KiB. Questa area solitamente contiene variabili globali e indirizzi di funzioni importate come `printf()`, perchè gli sviluppatori di GCC hanno deciso che il recupero dell'indirizzo di una funzione deve essere veloce tanto quanto l'esecuzione di una singola istruzione invece di due. In un file ELF questa area di 64KiB è collocata parzialmente nelle sezioni .sbss («small BSS[45]») per dati non inizializzati e .sdata («small data») per dati inizializzati. Ciò implica che il programmatore può scegliere a quale dati si possa accedere più velocemente e piazzarli nelle sezioni .sdata/.sbss. Alcuni programmatori old-school potrebbero ricordarsi del memory model MS-DOS **??** on page ?? o dei memory manger MS-DOS come XMS/EMS, in cui tutta la memoria era divisa in blocchi da 64KiB.

Questo concetto non è unicamente di MIPS. Anche PowerPC usa la stessa tecnica.

Con ottimizzazione GCC

Consideriamo il seguente esempio che illustra il concetto di «global pointer».

Listing 1.32: Con ottimizzazione GCC 4.4.5 (risultato dell'assembly)

```
1  $LC0:
2  ; \000 è zero byte in base ottale:
3          .ascii  "Hello, world!\012\000"
4  main:
5  ; prologo funzione.
6  ; imposta il GP:
7          lui     $28,%hi(__gnu_local_gp)
8          addiu   $sp,$sp,-32
9          addiu   $28,$28,%lo(__gnu_local_gp)
10 ; salva il RA nello stack locale:
11         sw      $31,28($sp)
12 ; carica l'indirizzo della funzione puts() dal GP a $25:
13         lw      $25,%call16(puts)($28)
14 ; carica l'indirizzo della sringa di testo in $4 ($a0):
15         lui     $4,%hi($LC0)
16 ; salta a puts(), salvando l'indirizzo di ritorno nel link register:
17         jalr    $25
18         addiu   $4,$4,%lo($LC0) ; branch delay slot
19 ; ripristina il RA:
20         lw      $31,28($sp)
21 ; copia 0 da $zero a $v0:
22         move    $2,$0
23 ; ritorna saltando al RA:
24         j       $31
25 ; epilogo della funzione:
26         addiu   $sp,$sp,32 ; branch delay slot + liberazione dello stack
       locale
```

[45] Block Started by Symbol

Come possiamo vedere, il registro $GP è settato nel prologo della funzione affinchè punti nel mezzo di questa area. Il registro RA viene anche salvato sullo stack locale. `puts()` anche qui viene usata al posto di `printf()`. L'indirizzo della funzione `puts()` è caricato in $25 usando LW, l'istruzione («Load Word»). Successivamente l'indirizzo della stringa viene caricato in $4 usando la coppia di istruzioni LUI («Load Upper Immediate») e ADDIU («Add Immediate Unsigned Word»). LUI setta i 16 bit alti del registro (da cui la parola «upper» nel nome dell'istruzione) e ADDIU aggiunge i 16 bit più bassi dell'indirizzo.

ADDIU segue JALR (ti ricordi dei *branch delay slots*?). Il registro $4 è anche detto $A0, viene usato per passare il primo argomento di una funzione [46].

JALR («Jump and Link Register») salta all'indirizzo memorizzato nel registro $25 register (indirizzo di `puts()`) salvando l'indirizzo della prossima istruzione (LW) in RA. Questo è molto simile ad ARM. Oh, e una cosa importate è che l'indirizzo salvato in RA non è l'indirizzo della prossima istruzione (perchè è in un *delay slot* e viene eseguito prima prima dell'istruzione jump), ma l'indirizzo dell'istruzione dopo la prossima (dopo il *delay slot*). Quindi, $PC + 8$ viene scritto in RA durante l'esecuzione di JALR, nel nostro caso, questo è l'indirizzo dell'istruzione LW successiva a ADDIU.

LW («Load Word») alla riga 20 ripristina RA dallo stack locale (questa istruzione è in effetti partedell'epilogo della funzione).

MOVE alla riga 22 copia il valore dal registro $0 ($ZERO) al $2 ($V0).

MIPS ha un registro *costante*, il cui valore è sempre zero. Apparentemente, gli sviluppatori MIPS hanno pensato che zero è la costante più usata in programmazione, quindi usiamo il registro $0 ogni volta che serve il valore zero.

Un altro fatto interessante è che in MIPS non c'è un'istruzione che trasferisce dati tra registri. Infatti, `MOVE DST, SRC` è `ADD DST, SRC, $ZERO` ($DST = SRC + 0$), che fa la stessa cosa. Apparentemente gli sviluppatori MIPS desideravano avere una tabella di opcode compatta. Questo non significa che un'addizione si verifichi per ogni istruzione MOVE. Molto probabilmente, la CPU ottimizza queste pseudoistruzioni e la ALU[47] non viene mai usata.

J a riga 24 salta all'indirizzo in RA, effettuando di fatti il ritorno dalla funzione. ADDIU dopo J viene in effetti eseguita prima di J (ricordi i *branch delay slots*?) e fa parte dell'epilogo della funzione. Ecco anche il listato generato da IDA. Ogni registro qui ha il suo pseudonimo:

Listing 1.33: Con ottimizzazione GCC 4.4.5 (IDA)

```
1  .text:00000000 main:
2  .text:00000000
3  .text:00000000 var_10          = -0x10
4  .text:00000000 var_4           = -4
5  .text:00000000
6  ; prologo della funzione.
7  ; imposta il GP:
8  .text:00000000                 lui     $gp, (__gnu_local_gp >> 16)
9  .text:00000004                 addiu   $sp, -0x20
```

[46] La tabella dei registri MIPS è riportata in appendice **??** on page ??

[47] Unità aritmetica e logica (Arithmetic Logic Unit)

```
10 .text:00000008                la      $gp, (__gnu_local_gp & 0xFFFF)
11 ; salva il RA nello stack locale:
12 .text:0000000C                sw      $ra, 0x20+var_4($sp)
13 ; salva il GP nello stack locale:
14 ; per qualche ragione, questa istruzione non è presente nell' output assembly
       di GCC:
15 .text:00000010                sw      $gp, 0x20+var_10($sp)
16 ; carica l'indirizzo della funzione puts() dal GP al $t9:
17 .text:00000014                lw      $t9, (puts & 0xFFFF)($gp)
18 ; forma l'indirizzo della stringa di testo in $a0:
19 .text:00000018                lui     $a0, ($LC0 >> 16)  # "Hello, world!"
20 ; salta a puts(), salvando l'indirizzo di ritorno nel link register:
21 .text:0000001C                jalr    $t9
22 .text:00000020                la      $a0, ($LC0 & 0xFFFF)  # "Hello,
       world!"
23 ; ripristina il RA:
24 .text:00000024                lw      $ra, 0x20+var_4($sp)
25 ; copia 0 da $zero a $v0:
26 .text:00000028                move    $v0, $zero
27 ; ritorna saltando al RA:
28 .text:0000002C                jr      $ra
29 ; epilogo funzione:
30 .text:00000030                addiu   $sp, 0x20
```

L'istruzione alla riga 15 salva il valore di GP sullo stack locale, e questa istruzione manca misteriosamente dal listato prodotto da GCC, forse per un errore di GCC [48]. Il valore di GP deve essere infatti salvato, perchè ogni funzione può usare la sua finestra dati da 64KiB. Il registro contenente l'indirizzo di `puts()` è chiamato $T9, perchè i registri con il prefisso T- sono detti «temporaries» ed il loro contenuto può non essere preservato.

Senza ottimizzazione GCC

Senza ottimizzazione GCC è più verboso.

Listing 1.34: Senza ottimizzazione GCC 4.4.5 (risultato dell'assembly)

```
1  $LC0:
2          .ascii  "Hello, world!\012\000"
3  main:
4  ; prologo funzione.
5  ; Salva il RA ($31) e FP nello stack:
6          addiu   $sp,$sp,-32
7          sw      $31,28($sp)
8          sw      $fp,24($sp)
9  ; imposta il FP (stack frame pointer):
10         move    $fp,$sp
11 ; imposta il GP:
12         lui     $28,%hi(__gnu_local_gp)
13         addiu   $28,$28,%lo(__gnu_local_gp)
14 ; carica l'indirizzo della stringa di testo:
```

[48] Apparentemente, le funzioni che generano i listati non sono fondamentali per gli utenti GCC, quindi può esserci qualche errore non ancora corretto.

```
15          lui     $2,%hi($LC0)
16          addiu   $4,$2,%lo($LC0)
17  ; carica l'indirizzo di puts() usando il GP:
18          lw      $2,%call16(puts)($28)
19          nop
20  ; chiama puts():
21          move    $25,$2
22          jalr    $25
23          nop  ; branch delay slot
24
25  ; ripristina il GP dallo stack locale:
26          lw      $28,16($fp)
27  ; imposta il registro $2 ($V0) a zero:
28          move    $2,$0
29  ; epilogo funione.
30  ; ripristina il SP:
31          move    $sp,$fp
32  ; ripristina il RA:
33          lw      $31,28($sp)
34  ; ripristina il FP:
35          lw      $fp,24($sp)
36          addiu   $sp,$sp,32
37  ; salta al RA:
38          j       $31
39          nop  ; branch delay slot
```

Qui vediamo che il registro FP è usato come un puntatore allo stack frame. Vediamo anche 3 NOP. Di cui il secondo e terzo seguono all'istruzione branch. Forse GCC aggiunge sempre dei NOP (a causa dei *branch delay slots*) dopo le istruzioni branch e successivamente, se le ottimizzazioni sono attivate, forse li elimina. Quindi in questo caso sono rimasti.

Ecco anche il listato IDA:

Listing 1.35: Senza ottimizzazione GCC 4.4.5 (IDA)

```
1  .text:00000000 main:
2  .text:00000000
3  .text:00000000 var_10          = -0x10
4  .text:00000000 var_8           = -8
5  .text:00000000 var_4           = -4
6  .text:00000000
7  ; prologo funzione.
8  ; salva il RA e FP nello stack:
9  .text:00000000                 addiu   $sp, -0x20
10 .text:00000004                 sw      $ra, 0x20+var_4($sp)
11 .text:00000008                 sw      $fp, 0x20+var_8($sp)
12 ; imposta il FP (stack frame pointer):
13 .text:0000000C                 move    $fp, $sp
14 ; imposta il GP:
15 .text:00000010                 la      $gp, __gnu_local_gp
16 .text:00000018                 sw      $gp, 0x20+var_10($sp)
17 ; carica l'indirizzo della stringa di testo:
```

```
18 .text:0000001C                lui     $v0, (aHelloWorld >> 16)  # "Hello,
       world!"
19 .text:00000020                addiu   $a0, $v0, (aHelloWorld & 0xFFFF)  #
       "Hello, world!"
20 ; carica l'indirizzo di puts() usando il GP:
21 .text:00000024                lw      $v0, (puts & 0xFFFF)($gp)
22 .text:00000028                or      $at, $zero ; NOP
23 ; chiama puts():
24 .text:0000002C                move    $t9, $v0
25 .text:00000030                jalr    $t9
26 .text:00000034                or      $at, $zero ; NOP
27 ; ripristina il GP dallo stack locale:
28 .text:00000038                lw      $gp, 0x20+var_10($fp)
29 ; imposta il registro $2 ($V0) a zero:
30 .text:0000003C                move    $v0, $zero
31 ; epilogo funzione.
32 ; ripristina lo SP:
33 .text:00000040                move    $sp, $fp
34 ; ripristina il RA:
35 .text:00000044                lw      $ra, 0x20+var_4($sp)
36 ; ripristina il FP:
37 .text:00000048                lw      $fp, 0x20+var_8($sp)
38 .text:0000004C                addiu   $sp, 0x20
39 ; salta al RA:
40 .text:00000050                jr      $ra
41 .text:00000054                or      $at, $zero ; NOP
```

E' interessante notare che IDA ha riconosciuto la coppia di istruzioni LUI/ADDIU e le ha fuse in un'unica pseudoistruzione LA («Load Address») alla riga 15. Possiamo anche vedere che questa pseudoistruzione è lunga 8 byte! Questa è una pseudoistruzione (o *macro*) in quanto non è una vera istruzione MIPS, ma soltanto un nome comodo per una coppia di istruzioni.

Un'altra cosa è che IDA non ha riconosciuto le istruzioni NOP che sono alle righe 22, 26 e 41. E' OR $AT, $ZERO. Essenzialmente, questa istruzione applica l'operazione OR al contenuto del registro $AT con zero, che è, ovviamente, un'istruzione inutile. MIPS, come molte altre ISA, non ha un'istruzione NOP propria.

Ruolo dello the stack frame in questo esempio

L'indirizzo della stringa è passato nel registro. Perchè allora impostare ugualmente uno stack locale? La ragione sta nel fatto che i valori dei registri RA e GP devono essere salvati da qualche parte (poichè viene chiamata `printf()`), e lo stack locale è usato proprio per questo scopo. Se fosse stata una funzione foglia, sarebbe stato possibile fare a meno del prologo e dell'epilogo, ad esempio: 1.4.3 on page 11.

Con ottimizzazione GCC: carichiamolo in GDB

Listing 1.36: sample GDB session

```
root@debian-mips:~# gcc hw.c -O3 -o hw
```

```
root@debian-mips:~# gdb hw
GNU gdb (GDB) 7.0.1-debian
...
Reading symbols from /root/hw...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x400654
(gdb) run
Starting program: /root/hw

Breakpoint 1, 0x00400654 in main ()
(gdb) set step-mode on
(gdb) disas
Dump of assembler code for function main:
0x00400640 <main+0>:    lui     gp,0x42
0x00400644 <main+4>:    addiu   sp,sp,-32
0x00400648 <main+8>:    addiu   gp,gp,-30624
0x0040064c <main+12>:   sw      ra,28(sp)
0x00400650 <main+16>:   sw      gp,16(sp)
0x00400654 <main+20>:   lw      t9,-32716(gp)
0x00400658 <main+24>:   lui     a0,0x40
0x0040065c <main+28>:   jalr    t9
0x00400660 <main+32>:   addiu   a0,a0,2080
0x00400664 <main+36>:   lw      ra,28(sp)
0x00400668 <main+40>:   move    v0,zero
0x0040066c <main+44>:   jr      ra
0x00400670 <main+48>:   addiu   sp,sp,32
End of assembler dump.
(gdb) s
0x00400658 in main ()
(gdb) s
0x0040065c in main ()
(gdb) s
0x2ab2de60 in printf () from /lib/libc.so.6
(gdb) x/s $a0
0x400820:        "hello, world"
(gdb)
```

1.5.5 Conclusione

La differenza principale tra il codice x86/ARM e x64/ARM64 è che il puntatore alla stringa è adesso lungo 64 bit. Infatti, le moderne CPU sono ora a 64-bit grazie ai costi ridotti della memoria e alla sua grande richiesta da parte delle applicazioni moderne. Possiamo aggiungere ai nostri computer più memoria di quanto i puntatori a 32-bit siano in grado di indirizzare. Di conseguenza, tutti i puntatori sono adesso a 64-bit.

1.5.6 Esercizi

- http://challenges.re/48
- http://challenges.re/49

1.6 Prologo ed epilogo delle funzioni

Il prologo (o preambolo) di una funzione è una sequenza di istruzioni all'inizio della funzione stessa. Spesso ha una forma simile al seguente frammento di codice:

```
    push    ebp
    mov     ebp, esp
    sub     esp, X
```

Cosa fanno queste istruzioni: salvano il valore del registro EBP, impostano il valore del registro EBP con il valore di ESP e allocano spazio sullo stack per le variabili locali.

Il valore di EBP resta costante durante il periodo di esecuzione della funzione, ed è usato per accede a variabili locali e argomenti. Per lo stesso scopo si può usare ESP, ma siccome questo cambia nel tempo, si tratta di approccio non molto conveniente.

L'epilogo della funzione libera lo spazio allocato nello stack, ripristina il valore nel registro EBP al suo stato iniziale e restituisce il controllo al chiamante:

```
    mov     esp, ebp
    pop     ebp
    ret     0
```

Prologo ed epilogo di funzioni sono solitamente identificati nei disassemblatori per delimitare le funzioni.

1.6.1 Ricorsione

Epiloghi e prologhi possono avere un effetto negativo sulla performance in caso di ricorsione.

Maggiori informazioni sulla ricorsione in questo libro: **??** on page ??.

1.7 Una Funzione Vuota: redux

Torniamo all'esempio della funzione vuota 1.3 on page 8. Ora che conosciamo il prologo e l'epilogo delle funzioni, questa è una funzione vuota 1.1 on page 8 compilata con GCC non ottimizzato:

Listing 1.37: Senza ottimizzazione GCC 8.2 x64 (risultato dell'assembly)

```
f:
        push    rbp
        mov     rbp, rsp
        nop
        pop     rbp
        ret
```

E' RET, ma il prologo e l'epilogo della funzione, probabilmente, non sono state ottimizzate e sono state lasciate cosi. NOP Sembrerebbe un'altro artefatto del compilatore. In ogni caso, l'unica istruzione effettiva qui è RET. Tutte le altre istruzioni possono essere rimosse (oppure ottimizzate).

1.8 Valori di Ritorno: redux

Di nuovo, ora che conosciamo prologo ed epilogo di una funzione, ricompiliamo un esempio che ritorna un valore (1.4 on page 10, 1.8 on page 10) usando GCC non ottimizzato:

Listing 1.38: Senza ottimizzazione GCC 8.2 x64 (risultato dell'assembly)

```
f:
        push    rbp
        mov     rbp, rsp
        mov     eax, 123
        pop     rbp
        ret
```

Qui le istruzioni effettive sono MOV e RET, le altre sono – prologo e epilogo.

1.9 Stack

Lo stack è una delle strutture dati più importanti in informatica [49]. AKA[50] LIFO[51].

Tecnicamente, è soltanto un blocco di memoria nella memoria di un processo insieme al registro ESP o RSP in x86 o x64, o il registro **SP!** in ARM, come puntatore all'interno di quel blocco.

Le istruzioni di accesso allo stack più usate sono PUSH e POP (sia in x86 che in ARM Thumb-mode). PUSH sottrae da ESP/RSP/**SP!** 4 in modalità 32-bit (oppure 8 in modalità 64-bit) e scrive successivamente il contenuto del suo unico operando nell'indirizzo di memoria puntato da ESP/RSP/**SP!**.

POP è l'operazione inversa: recupera il dato dalla memoria a cui punta **SP!**, lo carica nell'operando dell'istruzione (di solito un registro) e successivamente aggiunge 4 (o 8) allo stack pointer.

A seguito dell'allocazione dello stack, lo stack pointer punta alla base (fondo) dello stack. PUSH decrementa lo stack pointer e POP lo incrementa. La base dello stack è in realtà all'inizio del blocco di memoria allocato per lo stack. Sembra strano, ma è così.

ARM supporta sia stack decrescenti che crescenti.

Ad esempio le istruzioni STMFD/LDMFD, STMED[52]/LDMED[53] sono fatte per operare con uno stack decrescente (che cresce verso il basso, inizia con un indirizzo alto e prosegue verso il basso). Le istruzioni STMFA[54]/LDMFA[55], STMEA[56]/LDMEA[57] sono

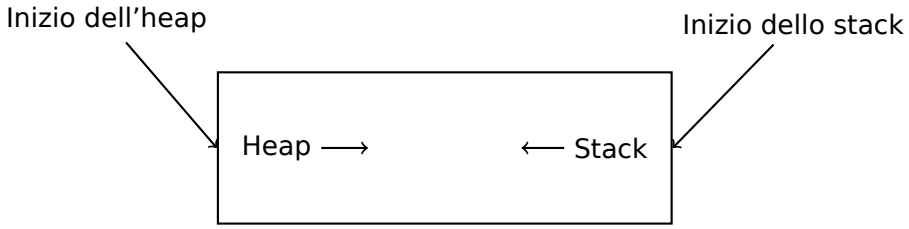
[49] wikipedia.org/wiki/Call_stack
[50] Also Known As — anche conosciuto come
[51] Ultimo arrivato primo ad uscire (Last In First Out)
[52] Store Multiple Empty Descending ()
[53] Load Multiple Empty Descending ()
[54] Store Multiple Full Ascending ()
[55] Load Multiple Full Ascending ()
[56] Store Multiple Empty Ascending ()
[57] Load Multiple Empty Ascending ()

fatte per operare con uno stack crescente (che cresce verso l'alto, da un indirizzo basso verso uno più alto).

1.9.1 Perchè lo stack cresce al contrario?

Intuitivamente potremmo pensare che lo stack cresca verso l'alto, ovvero verso indirizzi più alti, come qualunque altra struttura dati.

La ragione per cui lo stack cresce verso il basso è probabilmente di natura storica. Quando i computer erano talmente grandi da occupare un'intera stanza, era facile dividere la memoria in due parti, una per lo heap e l'altra per lo stack. Ovviamente non era possibile sapere a priori quanto sarebbero stati grandi lo stack e lo heap durante l'esecuzione di un programma, e questa soluzione era la più semplice.



In [D. M. Ritchie and K. Thompson, *The UNIX Time Sharing System*, (1974)][58]possiamo leggere:

> Il nucleo utente di una immagine è diviso in tre segmenti logici. Il segmento text del programma inizia in posizione 0 nel virtual address space. Durante l'esecuzione questo segmento viene protetto da scrittura, ed una sua singola copia viene condivisa tra i processi che eseguono lo stesso programma. Al primo limite di 8K byte sopra il segmento text del programma, nel virtual address space comincia un segmento dati scrivibile, non condiviso, le cui dimensioni possono essere estese da una chiamata di sistema.A partire dall'indirizzo più alto nel virtual address space c'è lo stack segment, che automaticammente cresce verso il basso al variare dello stack pointer hardware.

Questo ricorda molto come alcuni studenti utilizzino lo stesso quaderno per prendere appunti di due diverse materie: gli appunti per la prima materia sono scritti normalmente, e quelli della seconda materia sono scritti a partire dalla fine del quaderno, capovolgendolo. Le note si potrebbero "incontrare" da qualche parte in mezzo al quaderno, nel caso in cui non ci sia abbastanza spazio libero.

1.9.2 Per cosa viene usato lo stack?

Salvare l'indirizzo di ritorno della funzione

x86

[58] Italian text placeholderURL

Quando si chiama una funzione con l'istruzione CALL, l'indirizzo del punto esattamente dopo la CALL viene salvato nello stack, e successivamente viene eseguito un jump non condizionale all'indirizzo dell'operando di CALL.

L'istruzione CALL è equivalente alla coppia di istruzioni `PUSH indirizzo_dopo_call / JMP operando`.

RET preleva un valore dallo stack ed effettua un jump ad esso — ciò equivale alla coppia di istruzioni `POP tmp / JMP tmp`.

Riempire lo stack fino allo straripamento è semplicissimo. Basta ricorrere alla ricorsione eterna:

```
void f()
{
        f();
};
```

MSVC 2008 riporta il problema:

```
c:\tmp6>cl ss.cpp /Fass.asm
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 15.00.21022.08 for ↙
    ↳ 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

ss.cpp
c:\tmp6\ss.cpp(4) : warning C4717: 'f' : recursive on all control paths, ↙
    ↳ function will cause runtime stack overflow
```

...ma genera in ogni caso il codice correttamente:

```
?f@@YAXXZ PROC                          ; f
; Line 2
        push    ebp
        mov     ebp, esp
; Line 3
        call    ?f@@YAXXZ               ; f
; Line 4
        pop     ebp
        ret     0
?f@@YAXXZ ENDP                          ; f
```

...Se attiviamo le ottimizzazioni del compilatore (/Ox option) il codice ottimizzato non causerà overflow dello stack e funzionerà invece *correttamente*[59]:

```
?f@@YAXXZ PROC                          ; f
; Line 2
$LL3@f:
; Line 3
        jmp     SHORT $LL3@f
?f@@YAXXZ ENDP                          ; f
```

GCC 4.4.1 genera codice simile in antrambi i casi, senza avvertire del problema.

[59] sarcasmo, si fa per dire

Anche i programmi ARM usano lo stack per salvare gli indirizzi di ritorno, ma lo fanno in maniera diversa. Come detto in «Hello, world!» (1.5.3 on page 24), il RA viene salvato nel LR (registro link). Se si presenta comunque la necessità di chiamare un'altra funzione ed usare il registro LR ancora una volta, il suo valore deve essere salvato. Solitamente questo valore viene salvato nel preambolo della funzione.

Spesso vediamo istruzioni come `PUSH R4-R7,LR` insieme ad istruzioni nell'epilogo come `POP R4-R7,PC`—perciò i valori dei registri che saranno usati nella funzione vengono salvati nello stack, incluso LR.

Ciononostante, se una funzione non chiama al suo interno nessun'altra funzione, in terminologia RISC è detta *funzione foglia*, o funzione foglia.[60]. Di conseguenza, le leaf functions non salvano il registro LR register (perchè difatti non lo modificano). Se una simile funzione è molto breve e usa un piccolo numero di registri, potrebbe non usare del tutto lo stack. E' quindi possible chiamare le leaf functions senza usare lo stack, cosa che può essere più veloce rispetto alle vecchie macchine x86 perchè la RAM esterna non viene usata per lo stack [61]. Lo stesso principio può tornare utile quando la memoria per lo stack non è stata ancora allocata o non è disponibile.

Alcuni esempi di funzioni foglia: 1.14.3 on page 136, 1.14.3 on page 137, **??** on page ??, **??** on page ??, **??** on page ??, 1.192 on page 268, 1.190 on page 265, **??** on page ??.

Passaggio di argomenti alle funzioni

Il modo più diffuso per passare parametri in x86 è detto «cdecl»:

```
push arg3
push arg2
push arg1
call f
add esp, 12 ; 4*3=12
```

La funzioni chiamate, Chiamata, ricevono i propri argomenti tramite lo stack pointer.

Quindi è così che i valori degli argomenti sono posizionati nello stack prima dell'esecuzione della prima istruzione della funzione `f()`:

ESP	return address
ESP+4	argomento#1, marcato in IDA come `arg_0`
ESP+8	argomento#2, marcato in IDA come `arg_4`
ESP+0xC	argomento#3, marcato in IDA come `arg_8`
...	...

[60] infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13785.html

[61] Tempo fa, su PDP-11 e VAX, l'istruzione CALL (usata per chiamare altre funzioni) era costosa; poteva richiedere fino al 50% del tempo di esecuzione, ed era quindi consuetudine pensare che avere un grande numero di piccole funzioni fosse un anti-pattern [Eric S. Raymond, *The Art of UNIX Programming*, (2003)Chapter 4, Part II].

Per ulteriori informazioni su altri tipi di convenzioni di chiamata (calling conventions), fare riferimento alla sezione (**??** on page ??).

A proposito, la funzione chiamatachiamata non possiede alcuna informazione su quanti argomenti sono stati passati. Le funzioni C con un numero variabile di argomenti (come `printf()`) determinano il loro numero attraverso specificatori di formato stringa (che iniziano con il simbolo %).

Se scriviamo qualcosa come:

```
printf("%d %d %d", 1234);
```

`printf()` scriverà 1234, e successivamente due numeri casuali[62], che si trovavano lì vicino nello stack.

Per questo motivo non è molto importante come dichiariamo la funzione `main()`: come `main()`,
`main(int argc, char *argv[])` oppure `main(int argc, char *argv[], char *envp[])`.

Infatti, il codice CRT sta chiamando `main()` circa in questo modo:

```
push envp
push argv
push argc
call main
...
```

Se dichiari `main()` come `main()` senza argomenti, questi sono, in ogni caso, ancora presenti nello stack, ma non vengono utilizzati. Se dichiari `main()` come `main(int argc, char *argv[])`, sarai in grado di utilizzare i primi due argomenti, ed il terzo rimarrà «invisibile» per la tua funzione. In più, è possibile dichiarare `main(int argc)`, e continuerà a funzionare.

Metodi alternativi per passare argomenti

Vale la pena notare che non c'è nulla che obbliga il programmatore a passare gli argomenti attraverso lo stack. Non è un requisito necessario. Si potrebbe implementare un qualunque altro metodo anche senza usare per niente lo stack.

Un metodo abbastanza popolare tra chi inizia a programmare in linguaggio assembly language è di passare argomenti attraverso variabili globali, in questo modo:

Listing 1.39: Assembly code

```
        ...

        mov     X, 123
        mov     Y, 456
        call    do_something

        ...
```

[62] Non casuali in senso stretto, ma piuttosto non predicibili: 1.9.4 on page 51

```
X       dd      ?
Y       dd      ?

do_something proc near
        ; take X
        ; take Y
        ; do something
        retn
do_something endp
```

Tuttavia questo metodo ha un limite evidente: la funzione *do_something()* non può richiamare sè stessa in modo ricorsivo (o attraverso un'altra funzione), perchè deve cancellare i suoi stessi argomenti. Lo stesso accade con le variabili locali: se le tieni in variabili globali, la funzione non può chiamare se stessa. Inoltre questo non sarebbe thread-safe [63]. Il metodo di memorizzare queste informazioni nello stack rende il tutto più semplice—può mantenere quanti argomenti di funzione e/o valori, quanto spazio è disponibile.

[Donald E. Knuth, *The Art of Computer Programming*, Volume 1, 3rd ed., (1997), 189] menziona alcuni schemi ancora più strani e particolarmente convenienti su IBM System/360.

MS-DOS utilizzava un modo per passare tutti gli argomenti di funzione via registri, ad esempio, in questo pezzo di codice per MS-DOS a 16 bit scrive "Hello, world!":

```
mov  dx, msg      ; indirizzo del messaggio
mov  ah, 9        ; 9 indica la funzione "print string"
int  21h          ; "syscall" (chiamata di sistema) DOS

mov  ah, 4ch      ; funzione "termina il programma"
int  21h          ; "syscall" DOS

msg  db 'Hello, World!\$'
```

Questo è abbastanza simile al metodo **??** on page ??. Ed è inoltre molto simile alle chiamate syscalls in Linux (**??** on page ??) e Windows.

Se una funzione MS-DOS restituisce un valore di tipo boolean (cioè, un singolo bit, di solito per indicare uno stato di errore), il flag CF era spesso utilizzato.

Ad esempio:

```
mov ah, 3ch       ; crea file
lea dx, filename
mov cl, 1
int 21h
jc  error
mov file_handle, ax
...
error:
```

[63] Implementato correttamente, ciascun thread avrebbe il suo proprio stack con i suoi argomenti/variabili.

```
...
```

In caso di errore, il flag CF viene innalzato. Altrimenti, l'handle ad un nuovo file creato viene restituito attraverso AX.

Questo metodo viene ancora utilizzato dai programmatori assembly. Nel codice sorgente del Windows Research Kernel (che è abbastanza simile a Windows 2003) possiamo trovare qualcosa tipo: (file *base/ntos/ke/i386/cpu.asm*):

```
        public  Get386Stepping
Get386Stepping  proc

        call    MultiplyTest            ; Esegue test di moltiplicazione
        jnc     short G3s00             ; se nc, muttest è ok
        mov     ax, 0
        ret
G3s00:
        call    Check386B0              ; Verifica B0 stepping
        jnc     short G3s05             ; se nc, è B1/later
        mov     ax, 100h                ; è B0/earlier stepping
        ret

G3s05:
        call    Check386D1              ; Verifica D1 stepping
        jc      short G3s10             ; se c, non è D1
        mov     ax, 301h                ; è D1/later stepping
        ret

G3s10:
        mov     ax, 101h                ; suppone che sia B1 stepping
        ret

        ...

MultiplyTest    proc

        xor     cx,cx                   ; 64K volte è un bel numero tondo
mlt00:  push    cx
        call    Multiply                ; la moltiplicazione funziona in
    questo chip?
        pop     cx
        jc      short mltx              ; se c, No, esci
        loop    mlt00                   ; se nc, Si, cicla per riprovare
        clc
mltx:
        ret

MultiplyTest    endp
```

Memorizzazione di variabili locali

Una funzione può allocare spazio nello stack per le sue variabili locali, semplicemente decrementando lo stack pointer verso il basso dello stack.

Pertanto l'operazione risulta molto veloce, a prescinedere dal numero di variabili locali definite. Anche in questo caso utilizzare lo stack per memorizzare variabili locali non è un requisito necessario. Si possono memorizzare le variabili locali dove si vuole, ma tradizionalmente si fa in questo modo.

x86: la funzione alloca()

Vale la pena esaminare la funzione `alloca()` [64]. Questa funzione opera come `malloc()`, ma alloca memoria direttamente nello stack. Il pezzo di memoria allocato non necessita di essere liberato tramite una chiamata alla funzione `free()` function call, poichè l'epilogo della funzione (1.6 on page 40) ripristina ESP al suo valore iniziale e la memoria allocata viene semplicemente *abbandonata*. Vale anche la pena notare come è implementata la funzione `alloca()`. In termini semplici, questa funzione sposta ESP verso il basso, verso la base dello stack, per il numero di byte necessari e setta ESP per puntare al blocco *allocato*.

Proviamo:

```
#ifdef __GNUC__
#include <alloca.h> // GCC
#else
#include <malloc.h> // MSVC
#endif
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
#else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
#endif

    puts (buf);
};
```

La funzione `_snprintf()` opera come `printf()`, ma invece di inviare il risultato a stdout (es. al terminale o console), lo scrive nel buffer `buf`. La funzione `puts()` copia il contenuto di `buf` in stdout. Ovviamente queste due chiamate potrebbero essere rimpiazzate da una sola chiamata a `printf()`, ma questo è solo un esempio per illustrare l'uso di un piccolo buffer.

MSVC

Compiliamo (MSVC 2010):

Listing 1.40: MSVC 2010

[64] In MSVC, l'implementazione della funzione si trova in `alloca16.asm` e `chkstk.asm` in `C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\crt\src\intel`

```
...

    mov    eax, 600  ; 00000258H
    call   __alloca_probe_16
    mov    esi, esp

    push   3
    push   2
    push   1
    push   OFFSET $SG2672
    push   600       ; 00000258H
    push   esi
    call   __snprintf

    push   esi
    call   _puts
    add    esp, 28

...
```

L'unico argomento di `alloca()` viene passato tramite il registro EAX (anzichè inserirlo nello stack) [65].

GCC + Sintassi Intel

GCC 4.4.1 fa lo stesso senza chiamare funzioni esterne:

Listing 1.41: GCC 4.7.3

```
.LC0:
        .string "hi! %d, %d, %d\n"
f:
        push    ebp
        mov     ebp, esp
        push    ebx
        sub     esp, 660
        lea     ebx, [esp+39]
        and     ebx, -16                ; allinea puntatore con un bordo di
    16-byte
        mov     DWORD PTR [esp], ebx    ; s
        mov     DWORD PTR [esp+20], 3
        mov     DWORD PTR [esp+16], 2
        mov     DWORD PTR [esp+12], 1
        mov     DWORD PTR [esp+8], OFFSET FLAT:.LC0 ; "hi! %d, %d, %d\n"
        mov     DWORD PTR [esp+4], 600  ; maxlen
        call    _snprintf
        mov     DWORD PTR [esp], ebx    ; s
```

[65] Questo perchè alloca() è più una "compiler intrinsic" (**??** on page ??) che una funzione normale. Una delle ragioni per cui abbiamo bisogno di una funzione separata, invece di utilizzare semplicemente un paio di istruzioni nel codice, è che l'implementazione di alloca() usata da MSVC[66] include anche del codice che legge dalla memoria appena allocata, per far si che l'OS effettui il mapping della memoria fisica in questa regione della VM[67]. Dopo la chiamata a `alloca()`, ESP punta al blocco di 600 byte, ed è possibile utilizzarlo come memoria per l'array `buf`.

```
        call    puts
        mov     ebx, DWORD PTR [ebp-4]
        leave
        ret
```

GCC + Sintassi AT&T

Esaminiamo lo stesso codice, ma in sintassi AT&T:

Listing 1.42: GCC 4.7.3

```
.LC0:
        .string "hi! %d, %d, %d\n"
f:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $660, %esp
        leal    39(%esp), %ebx
        andl    $-16, %ebx
        movl    %ebx, (%esp)
        movl    $3, 20(%esp)
        movl    $2, 16(%esp)
        movl    $1, 12(%esp)
        movl    $.LC0, 8(%esp)
        movl    $600, 4(%esp)
        call    _snprintf
        movl    %ebx, (%esp)
        call    puts
        movl    -4(%ebp), %ebx
        leave
        ret
```

Il codice è uguale a quello del listato precedente.

A proposito, `movl $3, 20(%esp)` corrisponde a `mov DWORD PTR [esp+20], 3` in sintassi Intel. In sintassi AT&T, il formato registro+offset per indirizzare memoria appare come `offset(%register)`.

(Windows) SEH

I record SEH[68], se presenti, sono anch'essi memorizzati nello stack. Maggiori informazioni qui: (5.2.1 on page 300).

Protezione contro buffer overflow

Maggiori informazioni qui (1.25.2 on page 280).

[68] Structured Exception Handling

Deallocazione automatica dei dati nello stack

Probabilmente la ragione per cui si memorizzano nello stack le variabili locali e i record SEH deriva dal fatto che questi dati vengono ”liberati” automaticamente all’uscita dalla funzione, usando soltanto un’istruzione per correggere lo stack pointer (spesso è ADD). Si può dire che anche gli argomenti delle funzioni sono deallocati automaticamente alla fine della funzione. Invece, qualunque altra cosa memorizzata nello *heap* deve essere deallocata esplicitamente.

1.9.3 Una tipico layout dello stack

Una disposizione tipica dello stack in un ambiente a 32-bit all’inizio di una funzione, prima dell’esecuzione della sua prima istruzione, appare così:

...	...
ESP-0xC	variabile locale#2, marcato in IDA come var_8
ESP-8	variabile locale#1, marcato in IDA come var_4
ESP-4	valore memorizzato diEBP
ESP	Indirizzo di Ritorno
ESP+4	argomento#1, marcato in IDA come arg_0
ESP+8	argomento#2, marcato in IDA come arg_4
ESP+0xC	argomento#3, marcato in IDA come arg_8
...	...

1.9.4 Rumore nello stack

> Quando qualcuno afferma che qualcosa sembra casuale, solitamente intende dire che è l’unico a non vederne la regolarità.
>
> Stephen Wolfram, A New Kind of Science.

In questo libro si fa spesso riferimento a «rumore» o «spazzatura» (garbage) nello stack o in memoria. Da dove arrivano? Si tratta di ciò che resta dopo l’esecuzione di altre funzioni. Un piccolo esempio:

```
#include <stdio.h>

void f1()
{
        int a=1, b=2, c=3;
};

void f2()
{
        int a, b, c;
        printf ("%d, %d, %d\n", a, b, c);
};

int main()
{
```

```
        f1();
        f2();
};
```

Compilando si ottiene:

Listing 1.43: Senza ottimizzazione MSVC 2010

```
$SG2752 DB      '%d, %d, %d', 0aH, 00H

_c$ = -12       ; size = 4
_b$ = -8        ; size = 4
_a$ = -4        ; size = 4
_f1     PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 12
        mov     DWORD PTR _a$[ebp], 1
        mov     DWORD PTR _b$[ebp], 2
        mov     DWORD PTR _c$[ebp], 3
        mov     esp, ebp
        pop     ebp
        ret     0
_f1     ENDP

_c$ = -12       ; size = 4
_b$ = -8        ; size = 4
_a$ = -4        ; size = 4
_f2     PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 12
        mov     eax, DWORD PTR _c$[ebp]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp]
        push    edx
        push    OFFSET $SG2752 ; '%d, %d, %d'
        call    DWORD PTR __imp__printf
        add     esp, 16
        mov     esp, ebp
        pop     ebp
        ret     0
_f2     ENDP

_main   PROC
        push    ebp
        mov     ebp, esp
        call    _f1
        call    _f2
        xor     eax, eax
        pop     ebp
        ret     0
```

```
_main   ENDP
```

Il compilatore si lamenterà un pochino...

```
c:\Polygon\c>cl st.c /Fast.asm /MD
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.40219.01 for ⇙
    ↳ 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

st.c
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'c' ⇙
    ↳ used
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'b' ⇙
    ↳ used
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'a' ⇙
    ↳ used
Microsoft (R) Incremental Linker Version 10.00.40219.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:st.exe
st.obj
```

Ma quando avvieremo il programma ...

```
c:\Polygon\c>st
1, 2, 3
```

Oh, che cosa strana! Non abbiamo impostato il valore di alcuna variabile in `f2()`. Si tratta di valori «fantasma», che si trovano ancora nello stack.

Carichiamo l'esempio in OllyDbg:

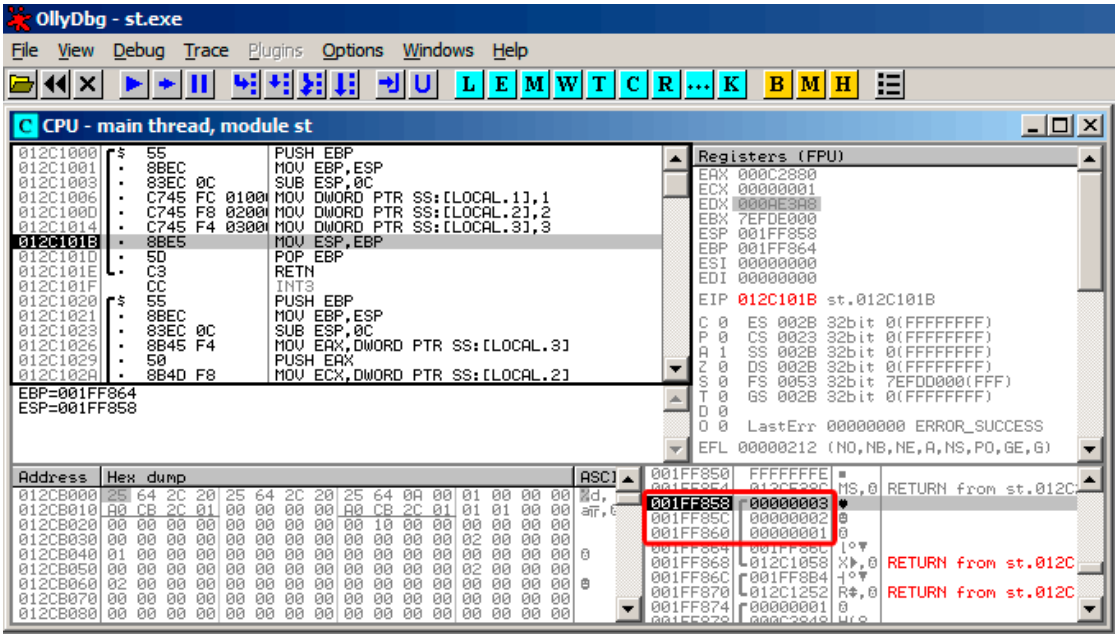


Figura 1.6: OllyDbg: `f1()`

Quando `f1()` assegna le variabili a, b e c, i loro valori sono memorizzati all'indirizzo `0x1FF860` e seguenti.

E quando viene eseguita `f2()`:

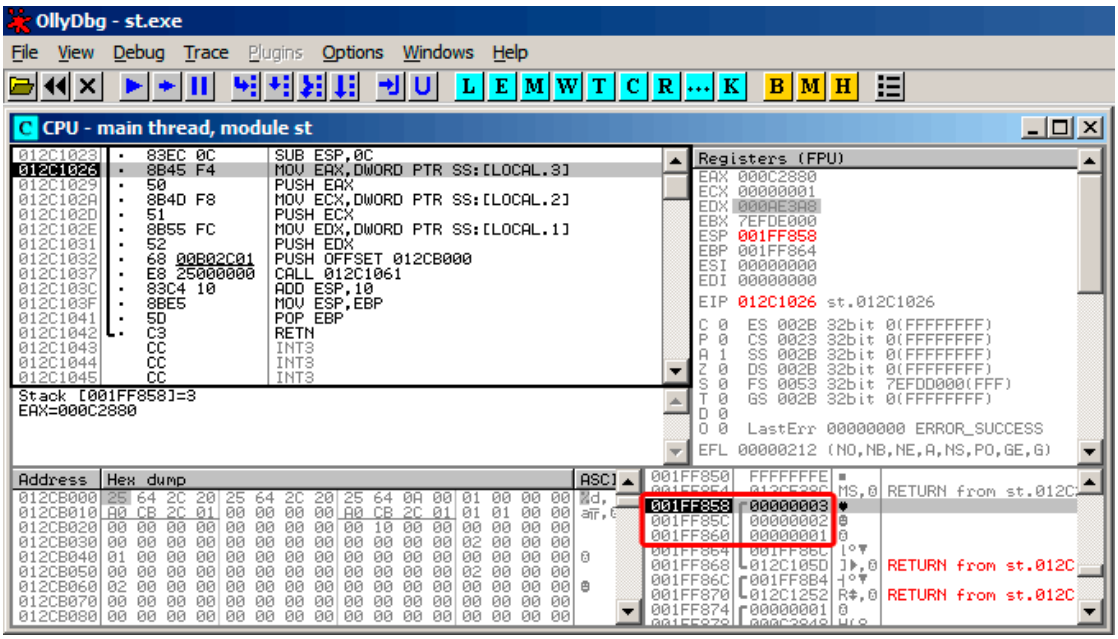


Figura 1.7: OllyDbg: `f2()`

... a, b e c di `f2()` si trovano agli stessi indirizzi! Nessuno ha ancora sovrascritto quei valori, quindi in quel punto sono ancora intatti. Quindi, affinchè questa strana situazione si verifichi, più funzioni devono essere chiamate una dopo l'altra e **SP!** deve essere uguale ad ogni ingresso nella funzione (ovvero le funzioni devono avere lo stesso numero di argomenti). A quel punto le variabili locali si troveranno nelle stesse posizioni nello stack. Per riassumere, tutti i valori nello stack (e nelle celle di memoria in generale) hanno valori lasciati lì dall'esecuzione di funzioni precedenti. Non sono letteralmente casuali, piuttosto hanno valori non predicibili. C'è un'altra opzione? Sarebbe possibile ripulire porzioni dello stack prima di ogni esecuzione di una funzione, ma sarebbe un lavoro extra probabilmente inutile.

MSVC 2013

L'esempio è stato compilato con MSVC 2010. Un lettore di questo libro ha provato a compilare l'esempio con MSVC 2013, lo ha eseguito, ed ha ottenuto i 3 numeri in ordine inverso:

```
c:\Polygon\c>st
3, 2, 1
```

Perchè? Ho compilato anche io l'esempio in MSVC 2013 ed ho visto questo:

Listing 1.44: MSVC 2013

```
_a$ = -12       ; dimensione = 4
_b$ = -8        ; dimensione = 4
_c$ = -4        ; dimensione = 4
_f2     PROC

...

_f2     ENDP

_c$ = -12       ; dimensione = 4
_b$ = -8        ; dimensione = 4
_a$ = -4        ; dimensione = 4
_f1     PROC

...

_f1     ENDP
```

Contrariamente a MSVC 2010, MSVC 2013 ha allocato le variabili a/b/c nella funzione `f2()` in ordine inverso.E ciò è del tutto corretto, perchè lo standard C/C++ non ha una regola che definisce in quale ordine le variabili locali devono essere allocate nello stack. La ragione per cui si presenta questa differenza è che MSVC 2010 lo fa in un certo modo, mentre MSVC 2013 ha probabilmente subito modifiche all'interno del compilatore, e si comporta quindi in modo leggermente diverso.

1.9.5 Esercizi

- http://challenges.re/51
- http://challenges.re/52

1.10 Una funzione quasi vuota

Questo è un vero pezzo di codice che ho trovato in Boolector[69]:

```
// forward declaration. the function is residing in some other module:
int boolector_main (int argc, char **argv);

// executable
int main (int argc, char **argv)
{
        return boolector_main (argc, argv);
}
```

Perchè qualcuno dovrebbe farlo? Non lo so, ma la mia ipotesi migliore è che `boolector_main()` potrebbe essere compilata in qualche sorta di DLL o libreria dinamica, ed essere chiamata da una suite di test. Sicuramente, una suite di test può preparare variabili `argc/argv` come farebbe CRT.

[69] https://boolector.github.io/

E' abbastanza interessante come viene compilato:

Listing 1.45: Senza ottimizzazione GCC 8.2 x64 (risultato dell'assembly)

```
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR -4[rbp], edi
        mov     QWORD PTR -16[rbp], rsi
        mov     rdx, QWORD PTR -16[rbp]
        mov     eax, DWORD PTR -4[rbp]
        mov     rsi, rdx
        mov     edi, eax
        call    boolector_main
        leave
        ret
```

Questo è OK, prologo, non necessario (non ottimizzato) scambio dei due argomenti CALL, epilogo, RET. Ma guardiamo la versione ottimizzata:

Listing 1.46: Con ottimizzazione GCC 8.2 x64 (risultato dell'assembly)

```
main:
        jmp     boolector_main
```

Così semplice che: stack/registri non vengono toccati e `boolector_main()` ha gli stessi argomenti settati. Quindi tutto ciò che dobbiamo fare è passare l'esecuzione ad un altro indirizzo.

Questo è molto simile alla Funzione thunk.

Vedremo qualcosa di più avanzato dopo: 1.11.2 on page 73, 1.21.1 on page 201.

1.11 printf() con più argomenti

Estendiamo l'esempio *Hello, world!* (1.5 on page 12) sostituendo la chiamata a `printf()` nella funzione `main()` con quanto segue:

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d", 1, 2, 3);
        return 0;
};
```

1.11.1 x86

x86: 3 argomenti

MSVC

Quando compiliamo l'esempio con MSVC 2010 Express otteniamo:

```
$SG3830 DB      'a=%d; b=%d; c=%d', 00H

...

        push    3
        push    2
        push    1
        push    OFFSET $SG3830
        call    _printf
        add     esp, 16                                 ; 00000010H
```

Notiamo che gli argomenti di `printf()` sono messi sullo stack in ordine inverso. Il primo argomento è quello inserito per ultimo.

A proposito, le variabili di tipo *int* in ambienti a 32-bit hanno dimensione pari a 32-bit, ovvero 4 byte.

Abbiamo 4 argomenti, quindi $4 * 4 = 16$ —occupano esattamente 16 byte nello stack: un puntatore da 32 bit alla stringa, e 3 numeri di tipo *int*.

Quando lo stack pointer (registro ESP) viene ripristinato dall'istruzione `ADD ESP, X` dopo una chiamata a funzione, in molti casi, il numero di argomenti della funzione può essere dedotto semplicemente dividendo X per 4.

Questa caratteristica è propria solo della calling convention *cdecl* ed in ambienti a 32 bit.

Si veda anche la sezione sulle calling conventions (**??** on page ??).

In certi casi, quando diverse funzioni ritornano una dopo l'altra, il compilatore potrebbe accorpare più istruzioni «ADD ESP, X» dopo l'ultima chiamata, emettendo una sola istruzione:

```
push a1
push a2
call ...
...
push a1
call ...
...
push a1
push a2
push a3
call ...
add esp, 24
```

Ecco un esempio reale:

Listing 1.47: x86

```
.text:100113E7    push    3
.text:100113E9    call    sub_100018B0 ; prende un argomento (3)
.text:100113EE    call    sub_100019D0 ; non prende nessun argomento
.text:100113F3    call    sub_10006A90 ; non prende nessun argomento
```

```
.text:100113F8    push    1
.text:100113FA    call    sub_100018B0 ; prende un argomento (1)
.text:100113FF    add     esp, 8       ; rilascia due argomenti dallo stack
    in una volta sola
```

MSVC e OllyDbg

Proviamo ora ad esaminare l'esempio con OllyDbg. Si tratta di uno dei più popolari debugger per win32 in user-land. Possiamo compilare l'esempio in MSVC 2012 con l'opzione `/MD`, che indica al compilatore di linkare con `MSVCR*.DLL`, in maniera tale da poter vedere in modo chiaro, nel debugger, le funzioni importate.

Carichiamo quindi l'eseguibile in OllyDbg. Il primo breakpoint avviene in `ntdll.dll`, premiamo F9 (run) per continuare l'esecuzione. Il secondo breakpoint è nel codice CRT. Ora dobbiamo trovare la funzione `main()`.

Per farlo scorriamo il codice verso l'alto (MSVC alloca la funzione `main()` proprio all'inizio della sezione code):

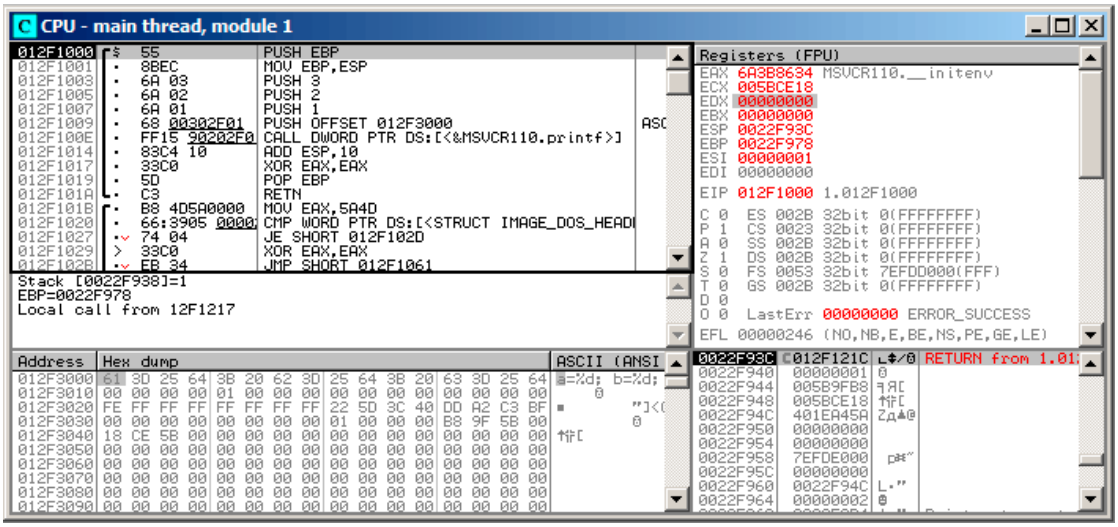


Figura 1.8: OllyDbg: l'inizio della funzione `main()`

Clickiamo sull'istruzione `PUSH EBP`, premiamo F2 (set breakpoint) e quindi F9 (run). Queste azioni ci consentono di saltare tutta la parte legata al codice CRT, in quanto per il momento non ci interessa.

Premiamo F8 (step over) per 6 volte, ovvero saltiamo (avanziamo di) 6 istruzioni:

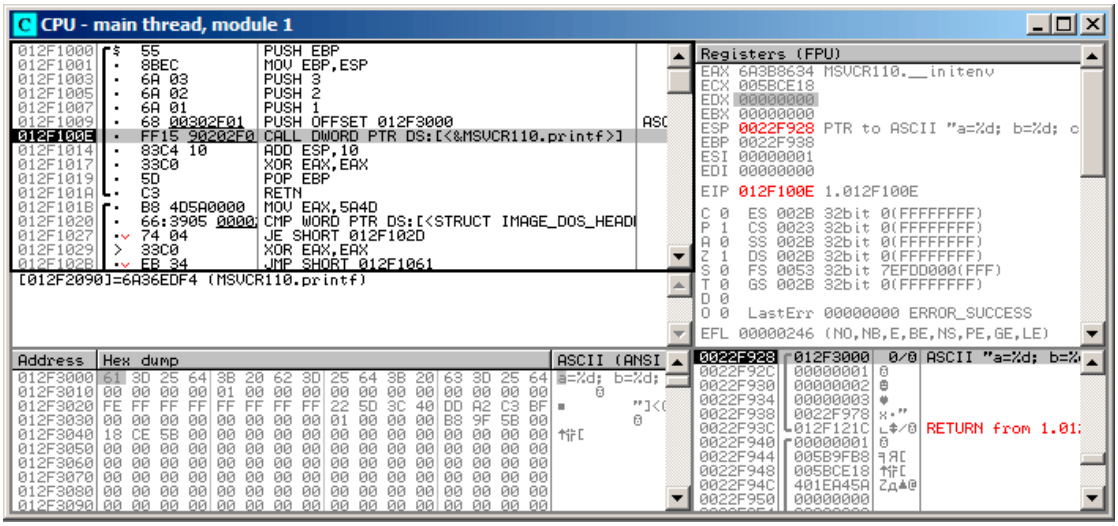


Figura 1.9: OllyDbg: prima dell'esecuzione di `printf()`

Adesso il **PC!** punta all'istruzione CALL `printf`. OllyDbg, come altri debugger, evidenzia il valore dei registri che sono stati modificati. Quindi ogni volta che si preme F8, EIP cambia ed il suo valore è mostrato in rosso. Anche ESP cambia, poichè i valori degli argomenti vengono messi sullo stack.

Dove sono i valori messi nello stack? Diamo un'occhiata alla finestra del debugger in basso a destra:

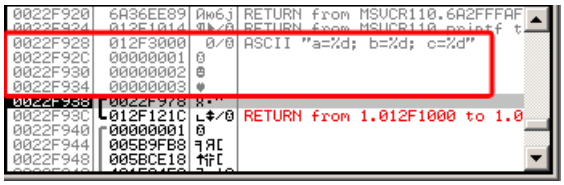


Figura 1.10: OllyDbg: stato dello stack dopo il push dei valori degli argomenti (il rettangolo rosso è stato aggiunto dall'autore per evidenziare la finestra)

Notiamo 3 colonne: indirizzo nello stack, valore nello stack ed alcuni commenti aggiuntivi di OllyDbg. OllyDbg riconosce le stringhe `printf()`-like, e riporta quindi la stringa insieme ai 3 valori *associati*.

Facendo click destro sulla stringa di formato, quindi click su «Follow in dump», è possibile vedere la format string nella finestra del debugger in basso a sinistra, che mostra una zona di memoria. I valori mostrati possono anche essere modificati. E' ad esempio possibile cambiare la format string, che renderebbe diverso il risultato

dell’esempio. Non è molto utile in questo particolare caso, ma può comunque essere un esercizio utile per iniziare a prendere dimestichezza con lo strumento.

Premiamo F8 (step over).

Il seguente output viene riportato in console:

```
a=1; b=2; c=3
```

Vediamo come sono cambiati i registri e lo stack:

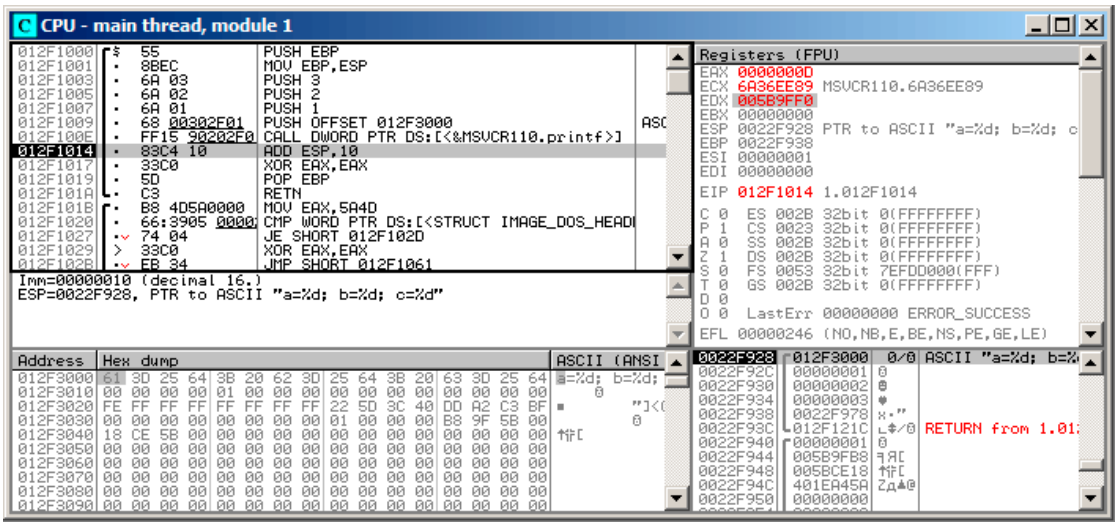


Figura 1.11: OllyDbg dopo dell'esecuzione di `printf()`

Il registro EAX adesso contiene `0xD` (13). Questo valore è corretto, poichè `printf()` restituisce il numero di caratteri stampati. Il valore di EIP è cambiato: adesso contiene infatti l'indirizzo dell'istruzione che viene dopo `CALL printf`. Anche i valori di ECX e EDX sono cambiati. Apparentemente, i meccanismi interni alla funzione `printf()` hanno usato quei registri durante l'esecuzione di printf, per le sue necessità.

Un fatto molto importante è che nè il valore di ESP nè lo stack sono stati modificati! Vediamo chiaramente che la format string ed i suoi 3 valori si trovano ancora lì. Questo è infatti il comportamento della calling convention *cdecl*: il chiamata (la funzione chiamata) non ripristina ESP al suo valore precedente. Farlo è una responsabilità del chiamante (chiamante).

Premiamo F8 nuovamente per eseguire l'istruzione `ADD ESP, 10`:

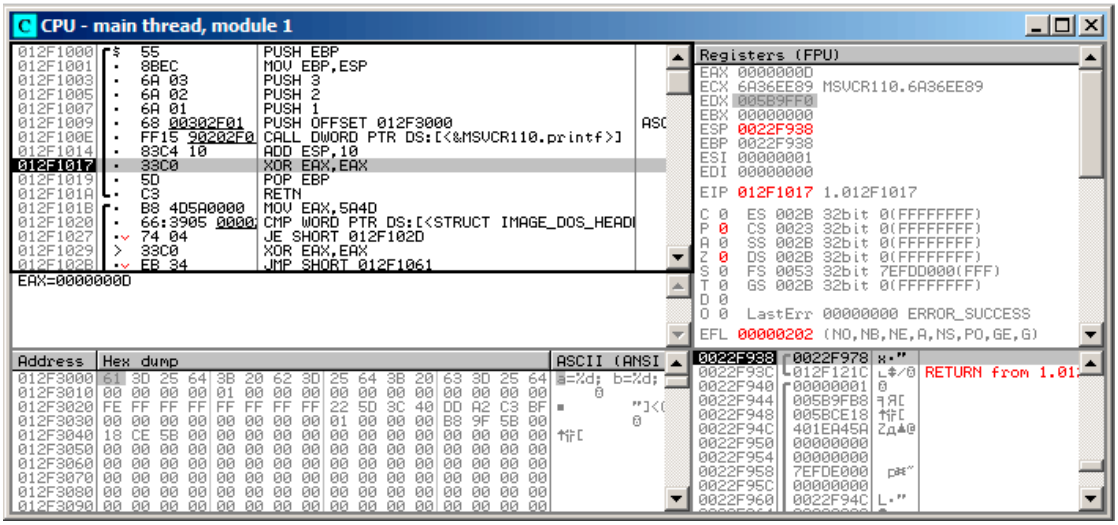


Figura 1.12: OllyDbg: dopo l'esecuzione dell'istruzione `ADD ESP, 10`

ESP è cambiato, ma i valori si trovano ancora sullo stack! Ovviamente si: non c'è necessità di azzerare i valori o effettuare altre simili operazioni di "pulizia". Qualunque cosa si trovi sopra lo stack pointer (**SP!**) è *rumore* o *garbage* e non ha alcun significato. Pulire i valori non utilizzati nello stack sarebbe una perdita di tempo inutile, e non vi è alcuna necessità di farlo.

GCC

Compiliamo lo stesso programma su Linux usando GCC 4.4.1, e diamo un'occhiata al risultato con IDA:

```
main            proc near

var_10          = dword ptr -10h
var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     eax, offset aADBDCD ; "a=%d; b=%d; c=%d"
                mov     [esp+10h+var_4], 3
                mov     [esp+10h+var_8], 2
                mov     [esp+10h+var_C], 1
                mov     [esp+10h+var_10], eax
```

```
                call    _printf
                mov     eax, 0
                leave
                retn
main            endp
```

Si nota che la differenza tra il codice prodotto da MSVC e GCC risiede soltanto nel modo in cui gli argomenti sono memorizzati sullo stack. In questo caso GCC lavora diversamente con lo stack, senza l'uso di PUSH/POP.

GCC e GDB

Proviamo l'esempio anche con GDB[70] su Linux.

L'opzione `-g` indica al compilatore di includere le informazioni di debug nel file eseguibile.

```
$ gcc 1.c -g -o 1
```

```
$ gdb 1
GNU gdb (GDB) 7.6.1-ubuntu
...
Reading symbols from /home/dennis/polygon/1...done.
```

Listing 1.48: impostiamo un breakpoint su `printf()`

```
(gdb) b printf
Breakpoint 1 at 0x80482f0
```

Avviamolo. Non abbiamo il sorgente della funzione `printf()` qui, quindi GDB non può mostrarlo, anche se ne avrebbe la capacità.

```
(gdb) run
Starting program: /home/dennis/polygon/1

Breakpoint 1, __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at printf.c:29
29      printf.c: No such file or directory.
```

Stampiamo 10 elementi dello stack. La colonna più a sinistra contiene gli indirizzi nello stack.

```
(gdb) x/10w $esp
0xbffff11c:     0x0804844a      0x080484f0      0x00000001      0x00000002
0xbffff12c:     0x00000003      0x08048460      0x00000000      0x00000000
0xbffff13c:     0xb7e29905      0x00000001
```

Il primo elemento è il RA (`0x0804844a`). Possiamo verificarlo disassemblando la memoria a questo indirizzo:

[70] GNU Debugger

```
(gdb) x/5i 0x0804844a
   0x804844a <main+45>: mov    $0x0,%eax
   0x804844f <main+50>: leave
   0x8048450 <main+51>: ret
   0x8048451:   xchg   %ax,%ax
   0x8048453:   xchg   %ax,%ax
```

Le due istruzioni XCHG sono istruzioni «inutili» (idle), analoghe a dei NOP.

Il secondo elemento (0x080484f0) è l'indirizzo della format string:

```
(gdb) x/s 0x080484f0
0x80484f0:      "a=%d; b=%d; c=%d"
```

I successivi 3 elementi (1, 2, 3) sono gli argomenti di `printf()`. Il resto degli elementi potrebbe essere «immondizia» nello stack, oppure valori provenienti da altre funzioni, come le loro variabili locali, etc. Per il momento li possiamo ignorare.

Eseguiamo il comando «finish». Questo comando dice a GDB di «eseguire tutte le istruzioni fino alla fine della funzione». In questo caso: esegui fino alla fine di `printf()`.

```
(gdb) finish
Run till exit from #0  __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at ↙
    ↳ printf.c:29
main () at 1.c:6
6               return 0;
Value returned is $2 = 13
```

GDB mostra il risultato di `printf()` restituito in EAX (13). Questo è il numero di caratteri stampati, proprio come nell'esempio in OllyDbg.

Vediamo anche «return 0;» e l'informazione che questa espressione si trova nel file `1.c` alla riga 6. Infatti il file `1.c` si trova nella directory corrente, e GDB trova la stringa lì. Come fa GDB a sapere quale riga del codice C viene eseguita? Ciò è dovuto al fatto che il compilatore, quando genera le informazioni di debug, salva anche una tabella di relazioni tra le righe del codice sorgente e gli indirizzi delle istruzioni. Dopotutto GDB è un «source-level debugger».

Esaminiamo i registri. 13 in EAX:

```
(gdb) info registers
eax            0xd      13
ecx            0x0      0
edx            0x0      0
ebx            0xb7fc0000       -1208221696
esp            0xbffff120       0xbffff120
ebp            0xbffff138       0xbffff138
esi            0x0      0
edi            0x0      0
eip            0x804844a        0x804844a <main+45>
...
```

Disassembliamo le istruzioni correnti. La freccia punta alla prossima istruzione da eseguire.

```
(gdb) disas
Dump of assembler code for function main:
   0x0804841d <+0>:     push   %ebp
   0x0804841e <+1>:     mov    %esp,%ebp
   0x08048420 <+3>:     and    $0xfffffff0,%esp
   0x08048423 <+6>:     sub    $0x10,%esp
   0x08048426 <+9>:     movl   $0x3,0xc(%esp)
   0x0804842e <+17>:    movl   $0x2,0x8(%esp)
   0x08048436 <+25>:    movl   $0x1,0x4(%esp)
   0x0804843e <+33>:    movl   $0x80484f0,(%esp)
   0x08048445 <+40>:    call   0x80482f0 <printf@plt>
=> 0x0804844a <+45>:    mov    $0x0,%eax
   0x0804844f <+50>:    leave
   0x08048450 <+51>:    ret
End of assembler dump.
```

GDB usa la sintassi AT&T di default. Ma è anche possibile passare alla sintassi Intel:

```
(gdb) set disassembly-flavor intel
(gdb) disas
Dump of assembler code for function main:
   0x0804841d <+0>:     push   ebp
   0x0804841e <+1>:     mov    ebp,esp
   0x08048420 <+3>:     and    esp,0xfffffff0
   0x08048423 <+6>:     sub    esp,0x10
   0x08048426 <+9>:     mov    DWORD PTR [esp+0xc],0x3
   0x0804842e <+17>:    mov    DWORD PTR [esp+0x8],0x2
   0x08048436 <+25>:    mov    DWORD PTR [esp+0x4],0x1
   0x0804843e <+33>:    mov    DWORD PTR [esp],0x80484f0
   0x08048445 <+40>:    call   0x80482f0 <printf@plt>
=> 0x0804844a <+45>:    mov    eax,0x0
   0x0804844f <+50>:    leave
   0x08048450 <+51>:    ret
End of assembler dump.
```

Eseguiamo la prossima istruzione. GDB mostra la parentesi graffa chiusa, che sta a significare la fine del blocco di codice.

```
(gdb) step
7       };
```

Esaminiamo i registri dopo l'esecuzione dell'istruzione `MOV EAX, 0`. EAX è zero a questo punto.

```
(gdb) info registers
eax            0x0      0
ecx            0x0      0
edx            0x0      0
ebx            0xb7fc0000       -1208221696
esp            0xbffff120       0xbffff120
```

```
ebp            0xbffff138       0xbffff138
esi            0x0      0
edi            0x0      0
eip            0x804844f        0x804844f <main+50>
...
```

x64: 8 argomenti

Per vedere come altri argomenti sono passati tramite lo stack, cambiamo nuovamente l'esempio per aumentare il numero degli argomenti a 9 (format string di `printf()` + 8 variabili *int*):

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3,↙
    ↳  4, 5, 6, 7, 8);
        return 0;
};
```

MSVC

Come già detto in precedenza, i primi 4 argomenti devono essere passati tramite i registri RCX, RDX, R8, R9 in Win64, mentre tutto il resto —tramite lo stack. E' esattamente quello che vediamo qui. Tuttavia, l'istruzione MOV instruction è usata al posto di PUSH per preparare lo stack, in modo tale che i valori siano memorizzati nello stack in maniera diretta.

Listing 1.49: MSVC 2012 x64

```
$SG2923 DB      'a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d', 0aH, 00H

main    PROC
        sub     rsp, 88

        mov     DWORD PTR [rsp+64], 8
        mov     DWORD PTR [rsp+56], 7
        mov     DWORD PTR [rsp+48], 6
        mov     DWORD PTR [rsp+40], 5
        mov     DWORD PTR [rsp+32], 4
        mov     r9d, 3
        mov     r8d, 2
        mov     edx, 1
        lea     rcx, OFFSET FLAT:$SG2923
        call    printf

        ; ritorna 0
        xor     eax, eax

        add     rsp, 88
```

```
        ret     0
main    ENDP
_TEXT   ENDS
END
```

Il lettore attento potrebbe chiedere perchè per i valori *int* sono allocati 8 byte quando ne bastano 4? Bisogna ricordare che per ogni tipo di dato più piccolo di 64 bit, sono allocati 8 byte. Questo è stabilito per convenienza: rende più facile calcolare l'indirizzo di argomenti arbitrari. E inoltre fa si che tutti siano allocati ad indirizzi di memoria allineati. Succede lo stesso in ambienti a 32-bit environments: sono riservati 4 byte per ogni tipo di dato.

GCC

L'immagine è simile per i sistemi operativi x86-64 e *NIX, con l'eccezzione che i primi 6 argomenti sono passati attraverso i registri `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9` registers. Tutto il resto —tramite lo stack. GCC genera il codice memorizzando il puntatore alla stringa in `EDI` invece che `RDI`—lo abbiamo visto in precedenza: 1.5.2 on page 21.

Abbiamo anche notato prima che il registro EAX è stato azzerato prima di una chiamata a `printf()`: 1.5.2 on page 21.

Listing 1.50: Con ottimizzazione GCC 4.4.6 x64

```
.LC0:
        .string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"

main:
        sub     rsp, 40

        mov     r9d, 5
        mov     r8d, 4
        mov     ecx, 3
        mov     edx, 2
        mov     esi, 1
        mov     edi, OFFSET FLAT:.LC0
        xor     eax, eax ; numero dei registri vettore passati
        mov     DWORD PTR [rsp+16], 8
        mov     DWORD PTR [rsp+8], 7
        mov     DWORD PTR [rsp], 6
        call    printf

        ; ritorna 0

        xor     eax, eax
        add     rsp, 40
        ret
```

GCC + GDB

Proviamo l'esempio in GDB.

```
$ gcc -g 2.c -o 2
```

```
$ gdb 2
GNU gdb (GDB) 7.6.1-ubuntu
...
Reading symbols from /home/dennis/polygon/2...done.
```

Listing 1.51: impostiamo il breakpoint su `printf()`, e avviamo

```
(gdb) b printf
Breakpoint 1 at 0x400410
(gdb) run
Starting program: /home/dennis/polygon/2

Breakpoint 1, __printf (format=0x400628 "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d↙
    ↳ ; g=%d; h=%d\n") at printf.c:29
29      printf.c: No such file or directory.
```

I registri RSI/RDX/RCX/R8/R9 hanno i valori previsti. RIP ha l'indirizzo della prima istruzione della funzione `printf()`.

```
(gdb) info registers
rax            0x0      0
rbx            0x0      0
rcx            0x3      3
rdx            0x2      2
rsi            0x1      1
rdi            0x400628 4195880
rbp            0x7fffffffdf60   0x7fffffffdf60
rsp            0x7fffffffdf38   0x7fffffffdf38
r8             0x4      4
r9             0x5      5
r10            0x7fffffffdce0   140737488346336
r11            0x7ffff7a65f60   140737348263776
r12            0x400440 4195392
r13            0x7fffffffe040   140737488347200
r14            0x0      0
r15            0x0      0
rip            0x7ffff7a65f60   0x7ffff7a65f60 <__printf>
...
```

Listing 1.52: ispezioniamo la format string

```
(gdb) x/s $rdi
0x400628:       "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
```

Effettuiamo un dump dello stack, questa volta con il comando x/g —*g* sta per *giant words*, ovvero 64-bit words.

```
(gdb) x/10g $rsp
0x7fffffffdf38: 0x0000000000400576      0x0000000000000006
```

```
0x7fffffffdf48: 0x0000000000000007      0x00007fff00000008
0x7fffffffdf58: 0x0000000000000000      0x0000000000000000
0x7fffffffdf68: 0x00007ffff7a33de5      0x0000000000000000
0x7fffffffdf78: 0x00007fffffffe048      0x0000000100000000
```

Il primo elemento dello stack, proprio come nel caso precedente, è il RA. 3 valori vengono passatri trvamite lo stack : 6, 7, 8. Notiamo anche che 8 è passato nei 32-bit alti non azzerati: `0x00007fff00000008`. Ciò va bene, perchè i valori hanno tipo *int*, che è a 32-bit. Quindi, i registri alti o gli elementi alti dello stack potrebbero contenere «random garbage».

Se andiamo a vedere dove viene restituito il controllo dopo l'esecuzione di `printf()`, GDB mostrerà l'intera funzione `main()`:

```
(gdb) set disassembly-flavor intel
(gdb) disas 0x0000000000400576
Dump of assembler code for function main:
   0x000000000040052d <+0>:     push   rbp
   0x000000000040052e <+1>:     mov    rbp,rsp
   0x0000000000400531 <+4>:     sub    rsp,0x20
   0x0000000000400535 <+8>:     mov    DWORD PTR [rsp+0x10],0x8
   0x000000000040053d <+16>:    mov    DWORD PTR [rsp+0x8],0x7
   0x0000000000400545 <+24>:    mov    DWORD PTR [rsp],0x6
   0x000000000040054c <+31>:    mov    r9d,0x5
   0x0000000000400552 <+37>:    mov    r8d,0x4
   0x0000000000400558 <+43>:    mov    ecx,0x3
   0x000000000040055d <+48>:    mov    edx,0x2
   0x0000000000400562 <+53>:    mov    esi,0x1
   0x0000000000400567 <+58>:    mov    edi,0x400628
   0x000000000040056c <+63>:    mov    eax,0x0
   0x0000000000400571 <+68>:    call   0x400410 <printf@plt>
   0x0000000000400576 <+73>:    mov    eax,0x0
   0x000000000040057b <+78>:    leave
   0x000000000040057c <+79>:    ret
End of assembler dump.
```

Finiamo di eseguire `printf()`, eseguiamo l'istruzione che azzera EAX, e notiamo che il registro EAX register ha esattamente il valore zero. RIP adesso punta all'istruzione LEAVE , ovvero la penultima nella funzione `main()`.

```
(gdb) finish
Run till exit from #0  __printf (format=0x400628 "a=%d; b=%d; c=%d; d=%d; e↙
    ↳ =%d; f=%d; g=%d; h=%d\n") at printf.c:29
a=1; b=2; c=3; d=4; e=5; f=6; g=7; h=8
main () at 2.c:6
6               return 0;
Value returned is $1 = 39
(gdb) next
7       };
(gdb) info registers
rax            0x0      0
rbx            0x0      0
rcx            0x26     38
```

```
rdx            0x7ffff7dd59f0   140737351866864
rsi            0x7fffffd9       2147483609
rdi            0x0      0
rbp            0x7fffffffdf60   0x7fffffffdf60
rsp            0x7fffffffdf40   0x7fffffffdf40
r8             0x7ffff7dd26a0   140737351853728
r9             0x7ffff7a60134   140737348239668
r10            0x7fffffffd5b0   140737488344496
r11            0x7ffff7a95900   140737348458752
r12            0x400440 4195392
r13            0x7fffffffe040   140737488347200
r14            0x0      0
r15            0x0      0
rip            0x40057b 0x40057b <main+78>
...
```

1.11.2 ARM

ARM: 3 argomenti

Lo schema tradizionale per il passaggio di argomenti (calling convention) di ARM si comporta in questo modo: i primi 4 argomenti vengono passati attraverso i registri R0-R3 , i restanti attraverso lo stack. Ciò ricorda molto il metodo per il passaggio di argomenti in fastcall (**??** on page ??) o win64 (**??** on page ??).

32-bit ARM

Senza ottimizzazione Keil 6/2013 (Modalità ARM)

Listing 1.53: Senza ottimizzazione Keil 6/2013 (Modalità ARM)

```
.text:00000000 main
.text:00000000 10 40 2D E9   STMFD   SP!, {R4,LR}
.text:00000004 03 30 A0 E3   MOV     R3, #3
.text:00000008 02 20 A0 E3   MOV     R2, #2
.text:0000000C 01 10 A0 E3   MOV     R1, #1
.text:00000010 08 00 8F E2   ADR     R0, aADBDCD     ; "a=%d; b=%d; c=%d"
.text:00000014 06 00 00 EB   BL      __2printf
.text:00000018 00 00 A0 E3   MOV     R0, #0          ; ritorna 0
.text:0000001C 10 80 BD E8   LDMFD   SP!, {R4,PC}
```

I primi 4 argomenti sono quindi passati attraverso i registri R0-R3 nel seguente ordine: un puntatore alla format string di `printf()` in R0, 1 in R1, 2 in R2 e 3 in R3. L'istruzione a 0x18 scrive 0 in R0—questo equivale allo statement C *return 0*. Niente di insolito fino a qui.

Con ottimizzazione Keil 6/2013 genera lo stesso codice.

Con ottimizzazione Keil 6/2013 (Modalità Thumb)

Listing 1.54: Con ottimizzazione Keil 6/2013 (Modalità Thumb)

```
.text:00000000 main
.text:00000000 10 B5        PUSH    {R4,LR}
.text:00000002 03 23        MOVS    R3, #3
.text:00000004 02 22        MOVS    R2, #2
.text:00000006 01 21        MOVS    R1, #1
.text:00000008 02 A0        ADR     R0, aADBDCD     ; "a=%d; b=%d; c=%d"
.text:0000000A 00 F0 0D F8  BL      __2printf
.text:0000000E 00 20        MOVS    R0, #0
.text:00000010 10 BD        POP     {R4,PC}
```

Non c'è nessuna differenza significativa nel codice non ottimizzato per modo ARM.

Con ottimizzazione Keil 6/2013 (Modalità ARM) + rimozione di return

Modifichiamo leggermente l'esempio rimuovendo *return 0*:

```
#include <stdio.h>

void main()
{
        printf("a=%d; b=%d; c=%d", 1, 2, 3);
};
```

Il risultato è alquanto insolito:

Listing 1.55: Con ottimizzazione Keil 6/2013 (Modalità ARM)

```
.text:00000014 main
.text:00000014 03 30 A0 E3   MOV     R3, #3
.text:00000018 02 20 A0 E3   MOV     R2, #2
.text:0000001C 01 10 A0 E3   MOV     R1, #1
.text:00000020 1E 0E 8F E2   ADR     R0, aADBDCD     ; "a=%d; b=%d; c=%d\n"
.text:00000024 CB 18 00 EA   B       __2printf
```

Questa è la versione ottimizzata (`-03`) per ARM mode e stavolta notiamo `B` come ultima istruzione, al posto della familiare `BL`. Un'altra differenza tra questa versione ottimizzata e la precedente (compilata senza ottimizzazione) è la mancanza di prologo ed epilogo della funzione (le istruzioni che preservano i valori dei registri `R0` e LR). L'istruzione `B` salta semplicemente ad un altro indirizzo, senza alcuna manipolazione del registro LR, in modo simile a `JMP` in x86. Perchè funziona? Perchè questo codice è infatti equivalente al precedente. Principalmente per due motivi: 1) nè lo stack nè **SP!** (lo stack pointer) vengono modificati; 2) la chiamata a `printf()` è l'ultima istruzione, quindi non succede niente dopo di essa. Al completamento, `printf()` restituisce semplicemente il controllo all'indirizzo memorizzato in LR. Poichè LR attualmente contiene l'indirizzo del punto da cui la nostra funzione era stata chiamata, il controllo verrà restituito da `printf()` a quello stesso punto. Pertanto non c'è alcun

bisogno di salvare LR in quanto non abbiamo necessità di modificare LR. E non vogliamo affatto modificare LR poichè non ci sono altre chiamate a funzione ad eccezzione di `printf()`. Inoltre, dopo questa chiamata non abbiamo nient'altro da fare! Queste sono le ragioni per cui una simile ottimizzazione è possibile.

Questa ottimizzazione è spesso usata in funzioni in cui l'ultimo statement è una chiamata ad un'altra funzione. Un esempio simile è fornito di seguito: 1.21.1 on page 202.

ARM64

Senza ottimizzazione GCC (Linaro) 4.9

Listing 1.56: Senza ottimizzazione GCC (Linaro) 4.9

```
.LC1:
        .string "a=%d; b=%d; c=%d"
f2:
; salva FP e LR nello stack frame:
        stp     x29, x30, [sp, -16]!
; imposta lo stack frame (FP=SP):
        add     x29, sp, 0
        adrp    x0, .LC1
        add     x0, x0, :lo12:.LC1
        mov     w1, 1
        mov     w2, 2
        mov     w3, 3
        bl      printf
        mov     w0, 0
; ripristina FP e LR
        ldp     x29, x30, [sp], 16
        ret
```

La prima istruzione STP (*Store Pair*) salva FP (X29) e LR (X30) nello stack. La seconda istruzione `ADD X29, SP, 0` forma lo stack frame. Scrive semplicemente il valore di **SP!** in X29.

Successivamente vediamo la familiare coppia di istruzioni `ADRP/ADD`, che forma un puntatore alla stringa. *lo12* indica 12 bit bassi (low 12 bits), ovvero, il linker scriverà i 12 bit bassi dell'indirizzo di LC1 nell'opcode dell'istruzione `ADD`. `%d` nella format string di `printf()` è un *int* a 32-bit, quindi u valori 1, 2 e 3 sono caricati nelle parti a 32-bit dei registri.

Con ottimizzazione GCC (Linaro) 4.9 genera lo stesso codice.

ARM: 8 argomenti

Usiamo nuovamente l'esempio con 9 argomenti della sezione precedente: 1.11.1 on page 68.

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3,↙
    ↳  4, 5, 6, 7, 8);
        return 0;
};
```

Con ottimizzazione Keil 6/2013: Modalità ARM

```
.text:00000028             main
.text:00000028
.text:00000028             var_18 = -0x18
.text:00000028             var_14 = -0x14
.text:00000028             var_4  = -4
.text:00000028
.text:00000028 04 E0 2D E5  STR    LR, [SP,#var_4]!
.text:0000002C 14 D0 4D E2  SUB    SP, SP, #0x14
.text:00000030 08 30 A0 E3  MOV    R3, #8
.text:00000034 07 20 A0 E3  MOV    R2, #7
.text:00000038 06 10 A0 E3  MOV    R1, #6
.text:0000003C 05 00 A0 E3  MOV    R0, #5
.text:00000040 04 C0 8D E2  ADD    R12, SP, #0x18+var_14
.text:00000044 0F 00 8C E8  STMIA  R12, {R0-R3}
.text:00000048 04 00 A0 E3  MOV    R0, #4
.text:0000004C 00 00 8D E5  STR    R0, [SP,#0x18+var_18]
.text:00000050 03 30 A0 E3  MOV    R3, #3
.text:00000054 02 20 A0 E3  MOV    R2, #2
.text:00000058 01 10 A0 E3  MOV    R1, #1
.text:0000005C 6E 0F 8F E2  ADR    R0, aADBDCDDDEDFDGD ; "a=%d; b=%d; c=%d;
    d=%d; e=%d; f=%d; g=%"...
.text:00000060 BC 18 00 EB  BL     __2printf
.text:00000064 14 D0 8D E2  ADD    SP, SP, #0x14
.text:00000068 04 F0 9D E4  LDR    PC, [SP+4+var_4],#4
```

Il codice può essere diviso in più parti

- Preambolo della funzione:

  La prima istruzione `STR LR, [SP,#var_4]!` salva LR sullo stack, poichè questo registro sarà usato per la chiamata a `printf()`. Il punto esclamativo all fine indica il *pre-index*.

  Questo implica che **SP!** deve essere prima decrementato di 4, e successivamente LR sarà salvato all'indirizzo memorizzato in **SP!**. Tutto ciò è simile a PUSH in x86. Maggiori informazioni qui: **??** on page ??.

  La seconda istruzione `SUB SP, SP, #0x14` decrementa **SP!** (lo stack pointer) per allocare `0x14` (20) byte sullo stack. Infatti dobbiamo passare 5 valori a 32-bit tramite lo stack per la funzione `printf()`, e ciascuno di essi occupa 4 byte,

che è esattamente $5 * 4 = 20$. Gli altri 4 valori a 32-bit saranno passati tramite registri.

- Passaggio di 5, 6, 7 e 8 tramite lo stack: sono memorizzati nei registri R0, R1, R2 e R3, rispettivamente.
  Successivamente l'istruzione `ADD R12, SP, #0x18+var_14` scrive l'indirizzo dello stack, dove queste 4 variabili saranno memorizzate, nel registro R12. *var_14* è una macro assembly, uguale a -0x14, creata da IDA per visualizzare in maniera conveniente il codice che accede allo stack. Le macro *var_?* generate da IDA riflettono le variabili locali nello stack.

  Quindi, SP+4 sarà memorizzato nel registro R12. L'istruzione successiva `STMIA R12, R0-R3` scrive il contenuto dei registri R0-R3 alla memoria puntata da R12. STMIA è abbreviazione per *Store Multiple Increment After*. *Increment After* (incrementa dopo) implica che R12 deve essere incrementato di 4 dopo ciascuna scrittura di un valore nei registri.

- Passaggio di 4 tramite lo stack: 4 è memorizzato in R0 e questo valore, con l'aiuto dell'istruzione `STR R0, [SP,#0x18+var_18]`, viene salvato sullo stack. *var_18* è -0x18, quindi l'offset deve essere 0, da cui il valore dal registro R0 (4) sara' scritto all'indirizzo memorizzato in **SP!**.

- Passaggio di 1, 2 e 3 tramite registri: I valori dei primi 3 numeri (a, b, c) (1, 2, 3 rispettivamente) sono passatti attraverso i registri R1, R2 e R3 poco prima della chiamata a `printf()` call.

- chiamata a `printf()`.

- epilogo della funzione:

  L'istruzione `ADD SP, SP, #0x14` ripristina il puntatore **SP!** al suo valore precedente, pulendo quindi lo stack. Ovviamente quello che era stato memorizzato nello stack rimarra lì, e sarà probabilmente riscritto interamente durante l'esecuzione delle funzioni seguenti.

  L'istruzione `LDR PC, [SP+4+var_4],#4` carica il valore di LR salvato dallo stack nel nel registro **PC!**, causando quindi l'uscita dalla funzione. Non c'è il punto esclamativo —infatti **PC!** è caricato prima dall'indirizzo memorizzato in **SP!** ($4 + var\_4 = 4 + (-4) = 0$), questa istruzione è quindi analoga a `LDR PC, [SP],#4`), e successivamente **SP!** è incrementato di 4. Questo è detto *post-index*[71]. Perchè IDA mostra l'istruzione in quel modo? Perchè vuole illustrare il layout dello stack ed il fatto che var_4 è allocata per salvare il valore di LR nello stack locale. Questa istruzione è più o meno simile a `POP PC` in x86[72].

Con ottimizzazione Keil 6/2013: Modalità Thumb

```
.text:0000001C              printf_main2
.text:0000001C
.text:0000001C              var_18 = -0x18
.text:0000001C              var_14 = -0x14
```

[71] Maggiori dettagli: **??** on page ??.
[72] E' impossibile settare il valore di IP/EIP/RIP usando POP in x86, ma in ogni caso hai capito l'analogia.

```
.text:0000001C             var_8  = -8
.text:0000001C
.text:0000001C 00 B5        PUSH    {LR}
.text:0000001E 08 23        MOVS    R3, #8
.text:00000020 85 B0        SUB     SP, SP, #0x14
.text:00000022 04 93        STR     R3, [SP,#0x18+var_8]
.text:00000024 07 22        MOVS    R2, #7
.text:00000026 06 21        MOVS    R1, #6
.text:00000028 05 20        MOVS    R0, #5
.text:0000002A 01 AB        ADD     R3, SP, #0x18+var_14
.text:0000002C 07 C3        STMIA   R3!, {R0-R2}
.text:0000002E 04 20        MOVS    R0, #4
.text:00000030 00 90        STR     R0, [SP,#0x18+var_18]
.text:00000032 03 23        MOVS    R3, #3
.text:00000034 02 22        MOVS    R2, #2
.text:00000036 01 21        MOVS    R1, #1
.text:00000038 A0 A0        ADR     R0, aADBDCDDDEDFDGD ; "a=%d; b=%d; c=%d;
    d=%d; e=%d; f=%d; g=%"...
.text:0000003A 06 F0 D9 F8  BL      __2printf
.text:0000003E
.text:0000003E             loc_3E   ; CODE XREF: example13_f+16
.text:0000003E 05 B0        ADD     SP, SP, #0x14
.text:00000040 00 BD        POP     {PC}
```

L'output è quasi identico al precedente esempio. Tuttavia questo è codice Thumb e i valori sono disposti nello stack in modo differente: 8 per primo, quindi 5, 6, 7, e infine 4.

Con ottimizzazione Xcode 4.6.3 (LLVM): Modalità ARM

```
__text:0000290C             _printf_main2
__text:0000290C
__text:0000290C             var_1C = -0x1C
__text:0000290C             var_C  = -0xC
__text:0000290C
__text:0000290C 80 40 2D E9   STMFD  SP!, {R7,LR}
__text:00002910 0D 70 A0 E1   MOV    R7, SP
__text:00002914 14 D0 4D E2   SUB    SP, SP, #0x14
__text:00002918 70 05 01 E3   MOV    R0, #0x1570
__text:0000291C 07 C0 A0 E3   MOV    R12, #7
__text:00002920 00 00 40 E3   MOVT   R0, #0
__text:00002924 04 20 A0 E3   MOV    R2, #4
__text:00002928 00 00 8F E0   ADD    R0, PC, R0
__text:0000292C 06 30 A0 E3   MOV    R3, #6
__text:00002930 05 10 A0 E3   MOV    R1, #5
__text:00002934 00 20 8D E5   STR    R2, [SP,#0x1C+var_1C]
__text:00002938 0A 10 8D E9   STMFA  SP, {R1,R3,R12}
__text:0000293C 08 90 A0 E3   MOV    R9, #8
__text:00002940 01 10 A0 E3   MOV    R1, #1
__text:00002944 02 20 A0 E3   MOV    R2, #2
__text:00002948 03 30 A0 E3   MOV    R3, #3
__text:0000294C 10 90 8D E5   STR    R9, [SP,#0x1C+var_C]
```

```
__text:00002950 A4 05 00 EB   BL     _printf
__text:00002954 07 D0 A0 E1   MOV    SP, R7
__text:00002958 80 80 BD E8   LDMFD  SP!, {R7,PC}
```

Quasi lo stesso codice visto prima, ad eccezione dell'istruzione STMFA (Store Multiple Full Ascending), che è sinonimo di STMIB (Store Multiple Increment Before). Questa istruzione incrementa il valore nel registro **SP!** e solo successivamente scrive il prossimo valore del registro in memoria, invece che operare le due azioni in ordine inverso.

Un'altra cosa che salta all'occhio è che le istruzioni sono disponste in maniera apparentemente casuale. Ad esempio, il valore nel registro `R0` è manipolato in tre posti diversi agli indirizzi `0x2918`, `0x2920` e `0x2928`, quando invece sarebbe stato possibile farlo in un punto solo.

Ad ogni modo, il compilatore ottimizzante avrà avuto le sue ragioni per ordinare le istruzioni in questa maniera ed ottenere una maggiore efficacia durante l'esecuzione del codice.

Solitamente il processero prova ad eseguire simultaneamentele istruzioni vicine. Ad esempio, istruzioni come `MOVT R0, #0` e `ADD R0, PC, R0` non possono essere eseguite simultaneamente poichè entrambe modificano il registro `R0`. D'altra parte, `MOVT R0, #0` e `MOV R2, #4` possono invece essere eseguite simultaneamente poichè l'effetto della loro esecuzione non genera conflitti tra loro. Presumibilmente, il compilatore prova a generare codice in questo modo (quando possibile).

Con ottimizzazione Xcode 4.6.3 (LLVM): Modalità Thumb-2

```
__text:00002BA0                _printf_main2
__text:00002BA0
__text:00002BA0                var_1C = -0x1C
__text:00002BA0                var_18 = -0x18
__text:00002BA0                var_C  = -0xC
__text:00002BA0
__text:00002BA0 80 B5           PUSH     {R7,LR}
__text:00002BA2 6F 46           MOV      R7, SP
__text:00002BA4 85 B0           SUB      SP, SP, #0x14
__text:00002BA6 41 F2 D8 20     MOVW     R0, #0x12D8
__text:00002BAA 4F F0 07 0C     MOV.W    R12, #7
__text:00002BAE C0 F2 00 00     MOVT.W   R0, #0
__text:00002BB2 04 22           MOVS     R2, #4
__text:00002BB4 78 44           ADD      R0, PC  ; char *
__text:00002BB6 06 23           MOVS     R3, #6
__text:00002BB8 05 21           MOVS     R1, #5
__text:00002BBA 0D F1 04 0E     ADD.W    LR, SP, #0x1C+var_18
__text:00002BBE 00 92           STR      R2, [SP,#0x1C+var_1C]
__text:00002BC0 4F F0 08 09     MOV.W    R9, #8
__text:00002BC4 8E E8 0A 10     STMIA.W  LR, {R1,R3,R12}
__text:00002BC8 01 21           MOVS     R1, #1
__text:00002BCA 02 22           MOVS     R2, #2
__text:00002BCC 03 23           MOVS     R3, #3
```

```
__text:00002BCE CD F8 10 90    STR.W    R9, [SP,#0x1C+var_C]
__text:00002BD2 01 F0 0A EA    BLX      _printf
__text:00002BD6 05 B0          ADD      SP, SP, #0x14
__text:00002BD8 80 BD          POP      {R7,PC}
```

L'output è quasi lo stesso dell'esempio precedente, ad eccezione dell'uso di istruzioni Thumb/Thumb-2.

ARM64

Senza ottimizzazione GCC (Linaro) 4.9

Listing 1.57: Senza ottimizzazione GCC (Linaro) 4.9

```
.LC2:
        .string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
f3:
; prende più spazio nello stack:
        sub     sp, sp, #32
; salva FP e LR nello stack frame:
        stp     x29, x30, [sp,16]
; imposta stack frame (FP=SP+16):
        add     x29, sp, 16
        adrp    x0, .LC2 ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
        add     x0, x0, :lo12:.LC2
        mov     w1, 8           ; 9° argomento
        str     w1, [sp]        ; salva 9° argomento nello stack
        mov     w1, 1
        mov     w2, 2
        mov     w3, 3
        mov     w4, 4
        mov     w5, 5
        mov     w6, 6
        mov     w7, 7
        bl      printf
        sub     sp, x29, #16
; ripristina FP e LR
        ldp     x29, x30, [sp,16]
        add     sp, sp, 32
        ret
```

I primi 8 argomenti sono passati nei registri X- o W-: [*Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*, (2013)][73]. Un puntatore ad una string richiede un registro a 64-bit, quindi è passato in X0. Tutti gli altri valori hanno tipo *int* a 32-bit, quindi sono memorizzati nella parte a 32-bit dei registri (W-). Il nono argomento (8) è passato tramite lo stack. Infatti non è possibile passare un grande numero di argomenti tramite registri, in quanto il loro numero è limitato.

[73] Italian text placeholderhttp://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf

Con ottimizzazione GCC (Linaro) 4.9 genera lo stesso codice.

1.11.3 MIPS

3 argomenti

Con ottimizzazione GCC 4.4.5

La differenza principale con l'esempio «Hello, world!» è che in questo caso `printf()` è chiamata al posto di `puts()`, e 3 argomenti aggiuntivi sono passati attraverso i registri $5...$7 (o $A0...$A2). Questo è il motivo per cui questi registri hanno il prefisso A-, che implica il loro uso per il passaggio di argomenti di funzioni.

Listing 1.58: Con ottimizzazione GCC 4.4.5 (risultato dell'assembly)

```
$LC0:
        .ascii  "a=%d; b=%d; c=%d\000"
main:
; prologo funzione:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-32
        addiu   $28,$28,%lo(__gnu_local_gp)
        sw      $31,28($sp)
; carica l'indirizzo di printf():
        lw      $25,%call16(printf)($28)
; carica l'indirizzo della stringa di testo e imposta il 1° argomento di
    printf():
        lui     $4,%hi($LC0)
        addiu   $4,$4,%lo($LC0)
; imposta il 2° argomento di printf():
        li      $5,1                    # 0x1
; imposta il 3° argomento di printf():
        li      $6,2                    # 0x2
; chiama printf():
        jalr    $25
; imposta il 4° argomento di printf() (branch delay slot):
        li      $7,3                    # 0x3

; epilogo funzione:
        lw      $31,28($sp)
; imposta il valore di ritorno a 0:
        move    $2,$0
; ritorna
        j       $31
        addiu   $sp,$sp,32 ; branch delay slot
```

Listing 1.59: Con ottimizzazione GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_10          = -0x10
.text:00000000 var_4           = -4
.text:00000000
```

```
; prologo funzione:
.text:00000000                lui     $gp, (__gnu_local_gp >> 16)
.text:00000004                addiu   $sp, -0x20
.text:00000008                la      $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C                sw      $ra, 0x20+var_4($sp)
.text:00000010                sw      $gp, 0x20+var_10($sp)
; carica l'indirizzo di printf():
.text:00000014                lw      $t9, (printf & 0xFFFF)($gp)
; carica l'indirizzo della stringa di testo e imposta il 1° argomento di
    printf():
.text:00000018                la      $a0, $LC0        # "a=%d; b=%d; c=%d"
; imposta il 2° argomento di printf():
.text:00000020                li      $a1, 1
; imposta il 3° argomento di printf():
.text:00000024                li      $a2, 2
; chiama printf():
.text:00000028                jalr    $t9
; imposta il 4° argomento di printf() (branch delay slot):
.text:0000002C                li      $a3, 3
; prologo funzione:
.text:00000030                lw      $ra, 0x20+var_4($sp)
; imposta il valoe di ritorno a 0:
.text:00000034                move    $v0, $zero
; return
.text:00000038                jr      $ra
.text:0000003C                addiu   $sp, 0x20 ; branch delay slot
```

IDA ha fuso le coppie di istruzioni LUI e ADDIU in una unica pseudoistruzione LA. Questo è il motivo per cui non c'è nessuna istruzione all'indirizzo 0x1C: perchè LA *occupa* 8 byte.

Senza ottimizzazione GCC 4.4.5

Senza ottimizzazione GCC è più verboso:

Listing 1.60: Senza ottimizzazione GCC 4.4.5 (risultato dell'assembly)

```
$LC0:
        .ascii  "a=%d; b=%d; c=%d\000"
main:
; prologo funzione:
        addiu   $sp,$sp,-32
        sw      $31,28($sp)
        sw      $fp,24($sp)
        move    $fp,$sp
        lui     $28,%hi(__gnu_local_gp)
        addiu   $28,$28,%lo(__gnu_local_gp)
; carica l'indirizzo della stringa di testo:
        lui     $2,%hi($LC0)
        addiu   $2,$2,%lo($LC0)
; imposta il 1° argomento di printf():
        move    $4,$2
; imposta il 2° argomento di printf():
```

```
        li      $5,1                    # 0x1
; imposta il 3° argomento di printf():
        li      $6,2                    # 0x2
; imposta il 4° argomento di printf():
        li      $7,3                    # 0x3
; prendi l'indirizzo di printf():
        lw      $2,%call16(printf)($28)
        nop
; chiama printf():
        move    $25,$2
        jalr    $25
        nop

; epilogo funzione:
        lw      $28,16($fp)
; imposta il valore di ritorno a 0:
        move    $2,$0
        move    $sp,$fp
        lw      $31,28($sp)
        lw      $fp,24($sp)
        addiu   $sp,$sp,32
; ritorna
        j       $31
        nop
```

Listing 1.61: Senza ottimizzazione GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_10          = -0x10
.text:00000000 var_8           = -8
.text:00000000 var_4           = -4
.text:00000000
; prologo funzione:
.text:00000000                 addiu   $sp, -0x20
.text:00000004                 sw      $ra, 0x20+var_4($sp)
.text:00000008                 sw      $fp, 0x20+var_8($sp)
.text:0000000C                 move    $fp, $sp
.text:00000010                 la      $gp, __gnu_local_gp
.text:00000018                 sw      $gp, 0x20+var_10($sp)
; carica l'indirizzo della stringa di testo:
.text:0000001C                 la      $v0, aADBDCD     # "a=%d; b=%d; c=%d"
; imposta il 1° argomento di printf():
.text:00000024                 move    $a0, $v0
; imposta il 2° argomento di printf():
.text:00000028                 li      $a1, 1
; imposta il 3° argomento di printf():
.text:0000002C                 li      $a2, 2
; imposta il 4° argomento di printf():
.text:00000030                 li      $a3, 3
; prendi l'indirizzo di printf():
.text:00000034                 lw      $v0, (printf & 0xFFFF)($gp)
.text:00000038                 or      $at, $zero
; chiama printf():
```

```
.text:0000003C                  move    $t9, $v0
.text:00000040                  jalr    $t9
.text:00000044                  or      $at, $zero ; NOP
; epilogo funzione:
.text:00000048                  lw      $gp, 0x20+var_10($fp)
; imposta il valore di ritorno a 0:
.text:0000004C                  move    $v0, $zero
.text:00000050                  move    $sp, $fp
.text:00000054                  lw      $ra, 0x20+var_4($sp)
.text:00000058                  lw      $fp, 0x20+var_8($sp)
.text:0000005C                  addiu   $sp, 0x20
; ritorna
.text:00000060                  jr      $ra
.text:00000064                  or      $at, $zero ; NOP
```

8 argomenti

Usiamo nuovamente l'esempio con 9 argomenti dalla sezione prcedente: 1.11.1 on page 68.

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3,↙
    ↳  4, 5, 6, 7, 8);
        return 0;
};
```

Con ottimizzazione GCC 4.4.5

Solo i primi 4 argomenti sono passati nei registri $A0 ...$A3, gli altri sono passati tramite lo stack.

Questa è la calling convention O32 (che è la più comune nel mondo MIPS). Altre calling conventions (come N32) possono usare i registri per scopi diversi.

SW è l'abbreviazione di «Store Word» (da un registro alla memoria). MIPS manca di istruzioni per memorizzare un valore in memoria, è quindi necessario usare una coppia di istruzioni (LI/SW).

Listing 1.62: Con ottimizzazione GCC 4.4.5 (risultato dell'assembly)

```
$LC0:
        .ascii  "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\012\000"
main:
; prologo funzione:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-56
        addiu   $28,$28,%lo(__gnu_local_gp)
        sw      $31,52($sp)
```

```
; passa il 5° argomento nello stack:
        li      $2,4                    # 0x4
        sw      $2,16($sp)
; passa il 6° argomento nello stack:
        li      $2,5                    # 0x5
        sw      $2,20($sp)
; passa il 7° argomento nello stack:
        li      $2,6                    # 0x6
        sw      $2,24($sp)
; passa l' 8° argomento nello stack:
        li      $2,7                    # 0x7
        lw      $25,%call16(printf)($28)
        sw      $2,28($sp)
; passa il 1° argomento in $a0:
        lui     $4,%hi($LC0)
; passa il 9° argomento nello stack:
        li      $2,8                    # 0x8
        sw      $2,32($sp)
        addiu   $4,$4,%lo($LC0)
; passa il 2° argomento in $a1:
        li      $5,1                    # 0x1
; passa 3° argomento in $a2:
        li      $6,2                    # 0x2
; chiama printf():
        jalr    $25
; passa il 4° argomento in $a3 (branch delay slot):
        li      $7,3                    # 0x3

; epilogo funzione:
        lw      $31,52($sp)
; imposta il valore di ritorno a 0:
        move    $2,$0
; ritorna
        j       $31
        addiu   $sp,$sp,56 ; branch delay slot
```

Listing 1.63: Con ottimizzazione GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_28          = -0x28
.text:00000000 var_24          = -0x24
.text:00000000 var_20          = -0x20
.text:00000000 var_1C          = -0x1C
.text:00000000 var_18          = -0x18
.text:00000000 var_10          = -0x10
.text:00000000 var_4           = -4
.text:00000000
; prologo funzione:
.text:00000000                 lui     $gp, (__gnu_local_gp >> 16)
.text:00000004                 addiu   $sp, -0x38
.text:00000008                 la      $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C                 sw      $ra, 0x38+var_4($sp)
.text:00000010                 sw      $gp, 0x38+var_10($sp)
```

```
; passa il 5° argomento nello stack:
.text:00000014                li      $v0, 4
.text:00000018                sw      $v0, 0x38+var_28($sp)
; passa il 6° argomento nello stack:
.text:0000001C                li      $v0, 5
.text:00000020                sw      $v0, 0x38+var_24($sp)
; passa il 7° argomento nello stack:
.text:00000024                li      $v0, 6
.text:00000028                sw      $v0, 0x38+var_20($sp)
; passa l' 8° argomento nello stack:
.text:0000002C                li      $v0, 7
.text:00000030                lw      $t9, (printf & 0xFFFF)($gp)
.text:00000034                sw      $v0, 0x38+var_1C($sp)
; prepara il 1° argomento in $a0:
.text:00000038                lui     $a0, ($LC0 >> 16)  # "a=%d; b=%d;
    c=%d; d=%d; e=%d; f=%d; g=%"...
; passa il 9° argomento nello stack:
.text:0000003C                li      $v0, 8
.text:00000040                sw      $v0, 0x38+var_18($sp)
; passa il 1° argomento in $a0:
.text:00000044                la      $a0, ($LC0 & 0xFFFF)  # "a=%d; b=%d;
    c=%d; d=%d; e=%d; f=%d; g=%"...
; passa il 2° argomento in $a1:
.text:00000048                li      $a1, 1
; passa il 3° argomento in $a2:
.text:0000004C                li      $a2, 2
; chiama printf():
.text:00000050                jalr    $t9
; passa il 4° argomento in $a3 (branch delay slot):
.text:00000054                li      $a3, 3
; epilogo funzione:
.text:00000058                lw      $ra, 0x38+var_4($sp)
; imposta il valore di ritorno a 0:
.text:0000005C                move    $v0, $zero
; ritorna
.text:00000060                jr      $ra
.text:00000064                addiu   $sp, 0x38 ; branch delay slot
```

Senza ottimizzazione GCC 4.4.5

Senza ottimizzazione GCC è più verboso:

Listing 1.64: Senza ottimizzazione GCC 4.4.5 (risultato dell'assembly)

```
$LC0:
        .ascii  "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\012\000"
main:
; prologo funzione:
        addiu   $sp,$sp,-56
        sw      $31,52($sp)
        sw      $fp,48($sp)
        move    $fp,$sp
        lui     $28,%hi(__gnu_local_gp)
```

```
        addiu   $28,$28,%lo(__gnu_local_gp)
        lui     $2,%hi($LC0)
        addiu   $2,$2,%lo($LC0)
; passa il 5° argomento nello stack:
        li      $3,4                    # 0x4
        sw      $3,16($sp)
; passa il 6° argomento nello stack:
        li      $3,5                    # 0x5
        sw      $3,20($sp)
; passa il 7° argomento nello stack:
        li      $3,6                    # 0x6
        sw      $3,24($sp)
; passa l' 8° argomento nello stack:
        li      $3,7                    # 0x7
        sw      $3,28($sp)
; passa il 9° argomento nello stack:
        li      $3,8                    # 0x8
        sw      $3,32($sp)
; passa il 1° argomento in $a0:
        move    $4,$2
; passa il 2° argomento in $a1:
        li      $5,1                    # 0x1
; passa il 3° argomento in $a2:
        li      $6,2                    # 0x2
; passa il 4° argomento in $a3:
        li      $7,3                    # 0x3
; chiama printf():
        lw      $2,%call16(printf)($28)
        nop
        move    $25,$2
        jalr    $25
        nop
; epilogo funzione:
        lw      $28,40($fp)
; imposta il valore di ritorno a 0:
        move    $2,$0
        move    $sp,$fp
        lw      $31,52($sp)
        lw      $fp,48($sp)
        addiu   $sp,$sp,56
; ritorna
        j       $31
        nop
```

Listing 1.65: Senza ottimizzazione GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_28          = -0x28
.text:00000000 var_24          = -0x24
.text:00000000 var_20          = -0x20
.text:00000000 var_1C          = -0x1C
.text:00000000 var_18          = -0x18
.text:00000000 var_10          = -0x10
```

```
.text:00000000 var_8           = -8
.text:00000000 var_4           = -4
.text:00000000
; prologo funzione:
.text:00000000                 addiu   $sp, -0x38
.text:00000004                 sw      $ra, 0x38+var_4($sp)
.text:00000008                 sw      $fp, 0x38+var_8($sp)
.text:0000000C                 move    $fp, $sp
.text:00000010                 la      $gp, __gnu_local_gp
.text:00000018                 sw      $gp, 0x38+var_10($sp)
.text:0000001C                 la      $v0, aADBDCDDDEDFDGD  # "a=%d; b=%d;
    c=%d; d=%d; e=%d; f=%d; g=%"...
; passa il 5° argomento nello stack:
.text:00000024                 li      $v1, 4
.text:00000028                 sw      $v1, 0x38+var_28($sp)
; passa il 6° argomento nello stack:
.text:0000002C                 li      $v1, 5
.text:00000030                 sw      $v1, 0x38+var_24($sp)
; passa il 7° argomento nello stack:
.text:00000034                 li      $v1, 6
.text:00000038                 sw      $v1, 0x38+var_20($sp)
; passa l' 8° argomento nello stack:
.text:0000003C                 li      $v1, 7
.text:00000040                 sw      $v1, 0x38+var_1C($sp)
; passa il 9° argomento nello stack:
.text:00000044                 li      $v1, 8
.text:00000048                 sw      $v1, 0x38+var_18($sp)
; passa il 1° argomento in $a0:
.text:0000004C                 move    $a0, $v0
; passa il 2° argomento in $a1:
.text:00000050                 li      $a1, 1
; passa il 3ç argomento in $a2:
.text:00000054                 li      $a2, 2
; passa il 4° argomento in $a3:
.text:00000058                 li      $a3, 3
; chiama printf():
.text:0000005C                 lw      $v0, (printf & 0xFFFF)($gp)
.text:00000060                 or      $at, $zero
.text:00000064                 move    $t9, $v0
.text:00000068                 jalr    $t9
.text:0000006C                 or      $at, $zero ; NOP
; epilogo funzione:
.text:00000070                 lw      $gp, 0x38+var_10($fp)
; imposta il valore di ritorno a 0:
.text:00000074                 move    $v0, $zero
.text:00000078                 move    $sp, $fp
.text:0000007C                 lw      $ra, 0x38+var_4($sp)
.text:00000080                 lw      $fp, 0x38+var_8($sp)
.text:00000084                 addiu   $sp, 0x38
; ritorna
.text:00000088                 jr      $ra
.text:0000008C                 or      $at, $zero ; NOP
```

1.11.4 Conclusione

Qua di seguito c'è uno scheletro della chiamata alla funzione:

Listing 1.66: x86

```
...
PUSH 3° argomento
PUSH 2° argomento
PUSH 1° argomento
CALL funzione
; modifica lo stack pointer (se necessario)
```

Listing 1.67: x64 (MSVC)

```
MOV RCX, 1° argomento
MOV RDX, 2° argomento
MOV R8, 3° argomento
MOV R9, 4° argomento
...
PUSH 5°, 6° argomento, ecc. (se necessario)
CALL funzione
; modifica lo stack pointer (se necessario)
```

Listing 1.68: x64 (GCC)

```
MOV RDI, 1° argomento
MOV RSI, 2° argomento
MOV RDX, 3° argomento
MOV RCX, 4° argomento
MOV R8, 5° argomento
MOV R9, 6° argomento
...
PUSH 7°, 8° argomento, ecc. (se necessario)
CALL funzione
; modifica lo stack pointer (se necessario)
```

Listing 1.69: ARM

```
MOV R0, 1° argomento
MOV R1, 2° argomento
MOV R2, 3° argomento
MOV R3, 4° argomento
; passa il 5°, 6° argomento, ecc., nello stack (se necessario)
BL funzione
; modifica lo stack pointer (se necessario)
```

Listing 1.70: ARM64

```
MOV X0, 1° argomento
MOV X1, 2° argomento
MOV X2, 3° argomento
MOV X3, 4° argomento
MOV X4, 5° argomento
```

```
MOV X5, 6° argomento
MOV X6, 7° argomento
MOV X7, 8° argomento
; passa il 9°, 10° argomento, ecc., nello stack (se necessario)
BL funzione
; modifica lo stack pointer (se necessario)
```

Listing 1.71: MIPS (O32 calling convention)

```
LI $4, 1° argomento ; AKA $A0
LI $5, 2° argomento ; AKA $A1
LI $6, 3° argomento ; AKA $A2
LI $7, 4° argomento ; AKA $A3
; passa il 5°, 6° argomento, ecc., nello stack (se necessario)
LW temp_reg, indirizzo della funzione
JALR temp_reg
```

1.11.5 A proposito

Questa differenza negli approcci utilizzati per il passaggio di argomenti in x86, x64, fastcall, ARM e MIPS è un'ottima dimostrazione del fatto che per la CPU è indifferente come gli argomenti vengono passati alle funzioni. Sarebbe anche possibile creare un compilatore ipotetico in grado di passare gli argomenti attraverso una struttura speciale, senza usare lo stack.

I registri MIPS $A0 ...$A3 sono indicati in questo modo soltanto per convenienza (cioè nella O32 calling convention). I programmatori possono usare qualunque altro registro (tranne forse $ZERO) per passare i dati, o utilizzare qualunque altra calling convention.

La CPU non è assolutamente consapevole delle calling conventions.

Possiamo anche ricordare come i programmatori principianti in assembly passino gli argomenti alle altre funzioni: di solito tramite i registri, senza un ordine esplicito, o anche attraverso variabili globali. Questi approcci sono ovviamente validi e funzionanti.

1.12 scanf()

Ora utilizziamo `scanf()`.

1.12.1 Un semplice esempio

```
#include <stdio.h>

int main()
{
        int x;
        printf ("Enter X:\n");
```

```
        scanf ("%d", &x);

        printf ("You entered %d...\n", x);

        return 0;
};
```

Oggi non è più conveniente usare `scanf()` per interagire con l'utente. Possiamo però utilizzarla per illustrare il passaggio di un puntatore ad una variabile di tipo *int*.

Puntatori

I puntatori sono fra i concetti fondamentali in informatica. Spesso il passaggio di una struttura, array o più in generale, un oggetto molto grande, è troppo costoso in termini di memoria, mentre passare il suo indirizzo è più efficace. Inoltre se la funzione chiamata (chiamata) necessita di modificare qualcosa nella struttura ricevuta come parametro e successivamente restituirla per intero, la sitauzione si fa ancora più inefficiente. Perciò la cosa più semplice da fare è passare l'indirizzo della struttura alla funzione chiamata, e lasciare che operi le modifiche necessarie.

Un puntatore in C/C++— è semplicemente un indirizzo di una locazione di mamoria.

In x86, l'indirizzo è rappresentato con un numero a 32-bit (i.e., occupa 4 byte), mentre in x86-64 è un numero a 64-bit (8 byte). Per inciso, questo è il motivo per cui alcune persone si lamentano nel passaggio a x86-64 — tutti i puntatori in architettura x64 richiedono il doppio dello spazio, inclusa la memoria cache, che è memoria "costosa".

E' possibile lavorare soltanto con puntatori senza tipo, con un po' di sforzo. Ad esempio la funzione C standard `memcpy()`, che copia un blocco di memoria da un indirizzo ad un altro, ha come argomenti 2 puntatori di tipo `void*`, poichè è impossibile predire il tipo di dati che si vuole copiare. Il tipo di dato non è importante, conta solo la dimensione del blocco.

I puntatori sono anche moldo usati quando una funzione deve restituire più di un valore (torneremo su questo argomento più avanti (1.16 on page 145)).

la funzione *scanf()* — è uno di questi casi.

Oltre al fatto che la funzione necessita di indicare quanti valori sono stati letti con successo, deve anche restituire tutti questi valori.

In C/C++ il tipo del puntatore è necessario soltanto per i controlli sui tipi a compile-time.

Internamente, nel codice compilato, non vi è alcuna informazione sui tipi dei puntatori.

x86

MSVC

Questo è ciò che si ottiene dopo la compilazione con MSVC 2010:

```
CONST    SEGMENT
$SG3831    DB    'Enter X:', 0aH, 00H
$SG3832    DB    '%d', 00H
$SG3833    DB    'You entered %d...', 0aH, 00H
CONST    ENDS
PUBLIC    _main
EXTRN    _scanf:PROC
EXTRN    _printf:PROC
; flag di compilazione della funzione: /Odtp
_TEXT    SEGMENT
_x$ = -4                        ; size = 4
_main    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    push   OFFSET $SG3831 ; 'Enter X:'
    call   _printf
    add    esp, 4
    lea    eax, DWORD PTR _x$[ebp]
    push   eax
    push   OFFSET $SG3832 ; '%d'
    call   _scanf
    add    esp, 8
    mov    ecx, DWORD PTR _x$[ebp]
    push   ecx
    push   OFFSET $SG3833 ; 'You entered %d...'
    call   _printf
    add    esp, 8

    ; ritorna 0
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
```

x è una variabile locale.

In base allo standard C/C++ deve essere visibile soltanto in questa funzione e non in altri ambiti (esterni alla funzione). Tradizionalmente, le variabili locali sono memorizzate sullo stack. Ci sono probabilmente altri modi per allocarle, ma in x86 è così.

Lo scopo dell'istruzione che segue il prologo della funzione, PUSH ECX, non è quello di salvare lo stato di ECX (si noti infatti l'assenza della corrispondente istruzione POP ECX alla fine della funzione).

Infatti alloca 4 byte sullo stack per memorizzare la variabile x.

x sarà acceduta con l'aiuto della macro _x$ (che è uguale a -4) ed il registro EBP che punta al frame corrente.

Durante l'esecuzione delle funziona, EBP punta allo stack frame corrente rendendo possibile accedere alle variabili locali ed agli argomenti della funzione attraverso EBP+offset.

E' anche possibile usare ESP per lo stesso scopo, tuttavia non è molto conveniente poichè cambia di frequente. Il valore di EBP può essere pensato come uno *stato congelato* del valore in ESP all'inizio dell'esecuzione della funzione.

Questo è un tipico layout di uno stack frame in un ambiente a 32-bit:

...	...
EBP-8	local variable #2, marcato in IDA come var_8
EBP-4	local variable #1, marcato in IDA come var_4
EBP	saved value of EBP
EBP+4	return address
EBP+8	argomento#1, marcato in IDA come arg_0
EBP+0xC	argomento#2, marcato in IDA come arg_4
EBP+0x10	argomento#3, marcato in IDA come arg_8
...	...

La funzione scanf() nel nostro esempio ha due argomenti. Il primo è un puntatore alla stringa contenente %d e il secondo è l'indirizzo della variabile x.

Per prima cosa l'indirizzo della variabile x è caricato nel registro EAX dall'istruzione lea eax, DWORD PTR _x$[ebp].

LEA sta per *load effective address*, ed è spesso usata per formare un indirizzo (**??** on page ??).

Potremmo dire che in questo caso LEA memorizza semplicemente la somma del valore nel registro EBP e della macro _x$ nel registro EAX.

E' l'equivalente di lea eax, [ebp-4].

Quindi, 4 viene sottratto dal valore del registro EBP ed il risultato è memorizzato nel registro EAX. Successivamente il registro EAX è messo sullo stack (push) e scanf() viene chiamata.

printf() viene chiamata subito dopo con il suo primo argomento — un puntatore alla stringa: You entered %d...\n.

Il secondo argomento è preparato con: mov ecx, [ebp-4]. L'istruzione memorizza il valore della variabile x, non il suo indirizzo, nel registro ECX.

Successivamente il valore in ECX è memorizzato sullo stack e l'ultima printf() viene chiamata.

MSVC + OllyDbg

Proviamo ad analizzare l'esempio con OllyDbg. Carichiamo l'eseguibile e premiamo F8 (step over) fino a raggiungere il nostro eseguibile invece che `ntdll.dll`. Scorriamo verso l'alto finchè appare `main()`.

Clicchiamo sulla prima istruzione (PUSH EBP), premiamo F2 (*set a breakpoint*), e quindi F9 (*Run*). Il breakpoint sarà scatenato all'inizio della funzione `main()`.

Tracciamo adesso fino al punto in cui viene calcolato l'indirizzo della variabile x:

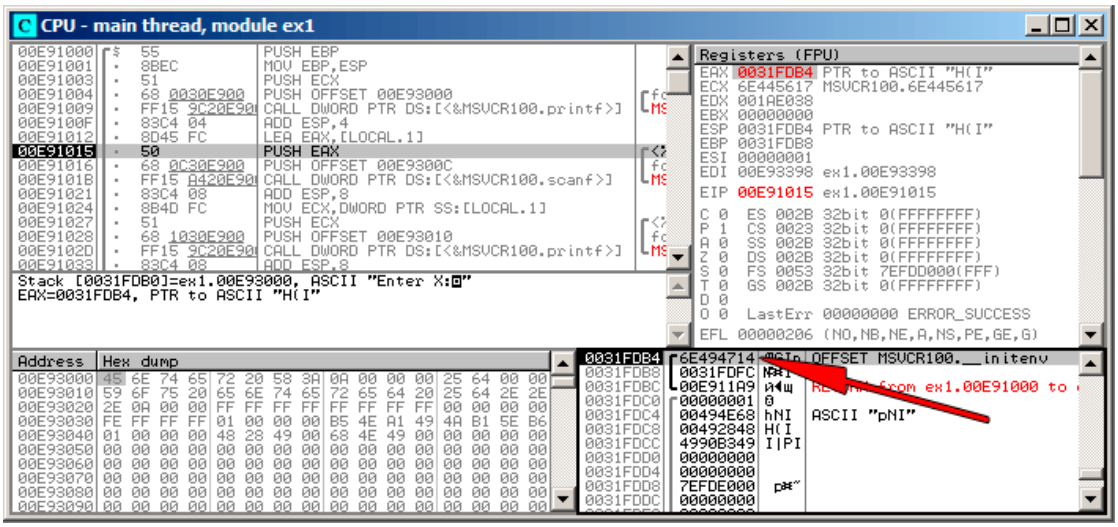


Figura 1.13: OllyDbg: The address of the local variable is calculated

Click di destra su EAX nella finestra dei registri e selezioniamo «Follow in stack».

Questo indirizzo apparirà nella finestra dello stack. La freccia rossa aggiunta punta alla variabile nello stack locale. Al momento questa locazione contiene un po' di immondizia (garbage) (0x6E494714). Con l'aiuto dell'istruzione PUSH l'indirizzo di questo elemento dello stack sarà memorizzato nello stesso stack alla posizione successiva. Tracciamo con F8 finchè non viene completata l'esecuzione della funzione `scanf()`. Durante l'esecuzione di `scanf()`, diamo in input un valore nella console. Ad esempio 123:

```
Enter X:
123
```

`scanf()` ha già completato la sua esecuzione:

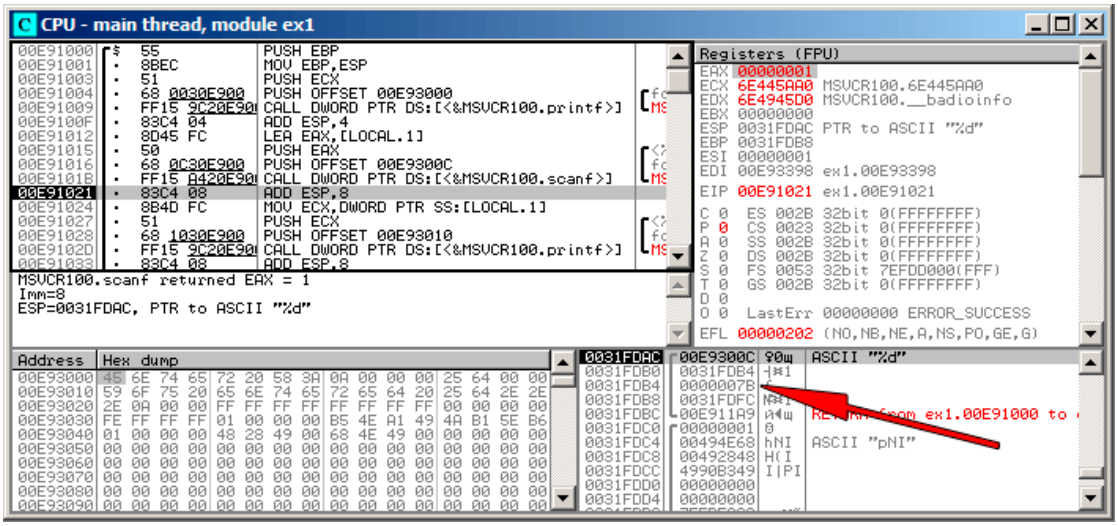


Figura 1.14: OllyDbg: `scanf()` executed

`scanf()` restituisce 1 in EAX, e ciò implica che ha letto con successo un valore. Se guardiamo nuovamente l'elemento nello stack corrispondente alla variabile locale, adesso contiene 0x7B (123).

Successivamente questo valore viene copiato dallo stack al registro ECX e passato a `printf()`:

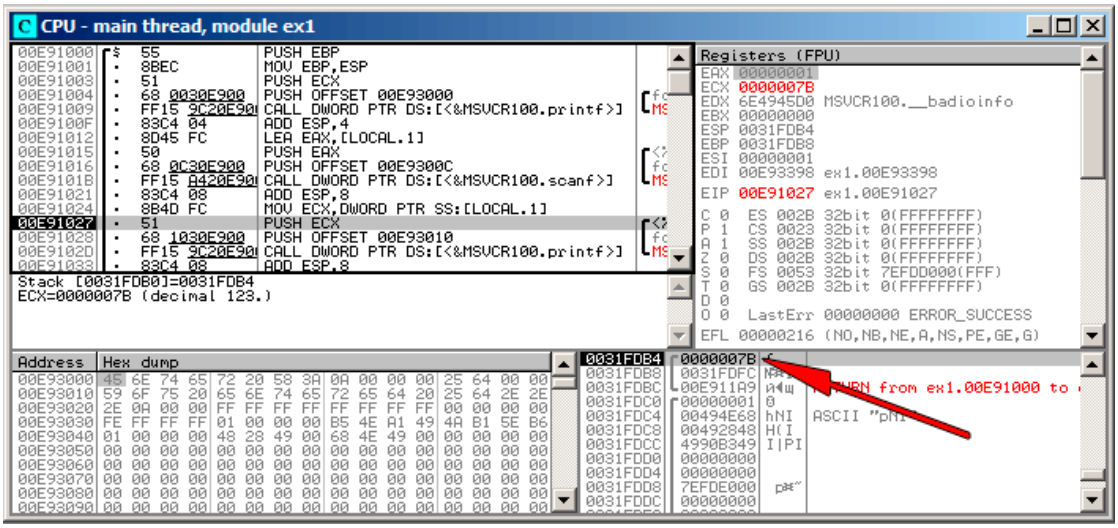


Figura 1.15: OllyDbg: preparing the value for passing to `printf()`

GCC

Proviamo a compilare questo codice con GCC 4.4.1 su Linux:

```
main            proc near

var_20          = dword ptr -20h
var_1C          = dword ptr -1Ch
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 20h
                mov     [esp+20h+var_20], offset aEnterX ; "Enter X:"
                call    _puts
                mov     eax, offset aD  ; "%d"
                lea     edx, [esp+20h+var_4]
                mov     [esp+20h+var_1C], edx
                mov     [esp+20h+var_20], eax
                call    ___isoc99_scanf
                mov     edx, [esp+20h+var_4]
                mov     eax, offset aYouEnteredD___ ; "You entered %d...\n"
                mov     [esp+20h+var_1C], edx
                mov     [esp+20h+var_20], eax
                call    _printf
                mov     eax, 0
```

```
                leave
                retn
main            endp
```

GCC ha sostituito la chiamata a `printf()` con `puts()`. La ragione per cui ciò avviene è stata spiegata in (1.5.3 on page 28).

Come nell'esempio compilato con MSVC —gli argomenti sono messi sullo stack utilizzando l'istruzione `MOV`.

A proposito

Questo semplice esempio è la dimostrazione del fatto che il compilatore traduce una lista di espressioni in blocchi C/C++ in una lista sequenziale di istruzioni. Non c'è nulla tra le espressioni in C/C++, e di conseguenza, nemmeno nel codice macchina risultate. Il flusso di controllo dunque passa da un'istruzione alla successiva.

x64

La situazione è simile, con l'unica differenza che, per il passaggio degli argomenti, i registri sono usati al posto dello stack.

MSVC

Listing 1.72: MSVC 2012 x64

```
_DATA   SEGMENT
$SG1289 DB      'Enter X:', 0aH, 00H
$SG1291 DB      '%d', 00H
$SG1292 DB      'You entered %d...', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
x$ = 32
main    PROC
$LN3:
        sub     rsp, 56
        lea     rcx, OFFSET FLAT:$SG1289 ; 'Enter X:'
        call    printf
        lea     rdx, QWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG1291 ; '%d'
        call    scanf
        mov     edx, DWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG1292 ; 'You entered %d...'
        call    printf

        ; ritorna 0
        xor     eax, eax
        add     rsp, 56
        ret     0
```

```
main    ENDP
_TEXT   ENDS
```

GCC

Listing 1.73: Con ottimizzazione GCC 4.4.6 x64

```
.LC0:
        .string "Enter X:"
.LC1:
        .string "%d"
.LC2:
        .string "You entered %d...\n"

main:
        sub     rsp, 24
        mov     edi, OFFSET FLAT:.LC0 ; "Enter X:"
        call    puts
        lea     rsi, [rsp+12]
        mov     edi, OFFSET FLAT:.LC1 ; "%d"
        xor     eax, eax
        call    __isoc99_scanf
        mov     esi, DWORD PTR [rsp+12]
        mov     edi, OFFSET FLAT:.LC2 ; "You entered %d...\n"
        xor     eax, eax
        call    printf

        ; ritorna 0
        xor     eax, eax
        add     rsp, 24
        ret
```

ARM

Con ottimizzazione Keil 6/2013 (Modalità Thumb)

```
.text:00000042             scanf_main
.text:00000042
.text:00000042             var_8           = -8
.text:00000042
.text:00000042 08 B5        PUSH    {R3,LR}
.text:00000044 A9 A0        ADR     R0, aEnterX  ; "Enter X:\n"
.text:00000046 06 F0 D3 F8  BL      __2printf
.text:0000004A 69 46        MOV     R1, SP
.text:0000004C AA A0        ADR     R0, aD  ; "%d"
.text:0000004E 06 F0 CD F8  BL      __0scanf
.text:00000052 00 99        LDR     R1, [SP,#8+var_8]
.text:00000054 A9 A0        ADR     R0, aYouEnteredD___ ; "You entered
    %d...\n"
.text:00000056 06 F0 CB F8  BL      __2printf
```

```
.text:0000005A 00 20        MOVS    R0, #0
.text:0000005C 08 BD        POP     {R3,PC}
```

Affinchè `scanf()` possa leggere l'input, necessita di un parametro —puntatore ad un *int*. *int* è 32-bit, quindi servono 4 byte per memorizzarlo da qualche parte in memoria, e entra perfettamente in un registro a 32-bit. Uno spazio per la variabile locale x è allocato nello stack e IDA lo ha chiamato *var_8*. Non è comunque necessario allocarlo in questo modo poichè **SP!** (stack pointer) punta già a quella posizione e può essere usato direttamente.

Successivamente il valore di **SP!** è copiato nel registro R1 e sono passati, insieme alla format-string, a `scanf()`.

Le istruzioni PUSH/POP si comportano diversamente in ARM rispetto a x86 (è il contrario). Sono sinonimi delle istruzioni STM/STMDB/LDM/LDMIA. E l'istruzione PUSH innanzitutto scrive un valore nello stack, *e poi* sottrae 4 allo **SP!**. POP innanzitutto aggiunge 4 allo **SP!**, *e poi* legge un valore dallo stack Quindi, dopo PUSH, lo **SP!** punta ad uno spazio inutilizzato nello stackn stack. E' usato da `scanf()`, e da `printf()` dopo.

LDMIA significa *Load Multiple Registers Increment address After each transfer*. STMDB significa *Store Multiple Registers Decrement address Before each transfer*.

Questo valore, con l'aiuto dell'istruzione LDR , viene poi spostato dallo stack al registro R1 per essere passato a `printf()`.

ARM64

Listing 1.74: Senza ottimizzazione GCC 4.9.1 ARM64

```
1  .LC0:
2          .string "Enter X:"
3  .LC1:
4          .string "%d"
5  .LC2:
6          .string "You entered %d...\n"
7  scanf_main:
8  ; sottrai 32 dallo SP, poi salva il FP ed il LR nello stack frame:
9          stp     x29, x30, [sp, -32]!
10 ; imposta lo stack frame (FP=SP)
11         add     x29, sp, 0
12 ; imposta il puntatore alla stringa "Enter X:":
13         adrp    x0, .LC0
14         add     x0, x0, :lo12:.LC0
15 ; X0=puntatore alla stringa "Enter X:"
16 ; stampalo:
17         bl      puts
18 ; imposta il puntatore alla stringa "%d":
19         adrp    x0, .LC1
20         add     x0, x0, :lo12:.LC1
21 ; trova uno spazio nello stack frame per la variabile "x" (X1=FP+28):
22         add     x1, x29, 28
```

```
23 ; X1=indirizzo della variabile "x"
24 ; passa l'indirizzo alla scanf() e chiamala:
25         bl      __isoc99_scanf
26 ; carica 32-bit dalla variabile nello stack frame:
27         ldr     w1, [x29,28]
28 ; W1=x
29 ; imposta il puntatore alla stringa "You entered %d...\n"
30 ; printf() prenderà la stringa di testo da X0 alla variabile "x" da X1 (o W1)
31         adrp    x0, .LC2
32         add     x0, x0, :lo12:.LC2
33         bl      printf
34 ; ritorna 0
35         mov     w0, 0
36 ; ripristina FP e LR, poi aggiungi 32 allo SP:
37         ldp     x29, x30, [sp], 32
38         ret
```

Ci sono 32 byte allocati per lo stack frame, che è più' grande del necessario. Forse a causa di meccanismi di allineamento della memoria? La parte più interessante è quella in cui trova spazio per la variabile x nello stack frame (riga 22). Perchè 28? Il compilatore ha in qualche modo deciso di piazzare questa variabile alla fine dello stack frame anzichè all'inizio. L'indirizzo è passato a `scanf()`, che memorizzerà il valore immesso dall'utente nella memoria a quell'indirizzo.Si tratta di un valore a 32-bit di tipo *int*.Il valore è recuperato successivamente a riga 27 e passato a `printf()`.

MIPS

Nello stack locale viene allocato spazio per la variabile x , a cui viene fatto riferimento come $\$sp + 24$.

Il suo indirizzo è passato a `scanf()`, il valore immesso dall'utente è caricato usand l'istruzione LW («Load Word») ed è infine passato a `printf()`.

Listing 1.75: Con ottimizzazione GCC 4.4.5 (risultato dell'assembly)

```
$LC0:
        .ascii  "Enter X:\000"
$LC1:
        .ascii  "%d\000"
$LC2:
        .ascii  "You entered %d...\012\000"
main:
; prologo funzione:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-40
        addiu   $28,$28,%lo(__gnu_local_gp)
        sw      $31,36($sp)
; chiama puts():
        lw      $25,%call16(puts)($28)
        lui     $4,%hi($LC0)
        jalr    $25
        addiu   $4,$4,%lo($LC0) ; branch delay slot
```

```
; chiama scanf():
        lw      $28,16($sp)
        lui     $4,%hi($LC1)
        lw      $25,%call16(__isoc99_scanf)($28)
; imposta il 2° argomento di scanf(), $a1=$sp+24:
        addiu   $5,$sp,24
        jalr    $25
        addiu   $4,$4,%lo($LC1) ; branch delay slot

; chiama printf():
        lw      $28,16($sp)
; imposta il 2°argomento di printf(),
; carica la word all'indirizzo $sp+24:
        lw      $5,24($sp)
        lw      $25,%call16(printf)($28)
        lui     $4,%hi($LC2)
        jalr    $25
        addiu   $4,$4,%lo($LC2) ; branch delay slot

; epilogo funzione:
        lw      $31,36($sp)
; imposta il valore di ritorno a 0:
        move    $2,$0
; ritorna:
        j       $31
        addiu   $sp,$sp,40      ; branch delay slot
```

IDA mostra il layout dello stack nel modo seguente:

Listing 1.76: Con ottimizzazione GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_18   = -0x18
.text:00000000 var_10   = -0x10
.text:00000000 var_4    = -4
.text:00000000
; prologo funzione:
.text:00000000          lui     $gp, (__gnu_local_gp >> 16)
.text:00000004          addiu   $sp, -0x28
.text:00000008          la      $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C          sw      $ra, 0x28+var_4($sp)
.text:00000010          sw      $gp, 0x28+var_18($sp)
; chiama puts():
.text:00000014          lw      $t9, (puts & 0xFFFF)($gp)
.text:00000018          lui     $a0, ($LC0 >> 16)  # "Enter X:"
.text:0000001C          jalr    $t9
.text:00000020          la      $a0, ($LC0 & 0xFFFF)  # "Enter X:" ; branch
    delay slot
; chiama scanf():
.text:00000024          lw      $gp, 0x28+var_18($sp)
.text:00000028          lui     $a0, ($LC1 >> 16)  # "%d"
.text:0000002C          lw      $t9, (__isoc99_scanf & 0xFFFF)($gp)
; imposta il 2° argomento di scanf(), $a1=$sp+24:
```

```
.text:00000030          addiu   $a1, $sp, 0x28+var_10
.text:00000034          jalr    $t9  ; branch delay slot
.text:00000038          la      $a0, ($LC1 & 0xFFFF)  # "%d"
; chiama printf():
.text:0000003C          lw      $gp, 0x28+var_18($sp)
; imposta il 2° argomento di printf(),
; carica la word all'indirizzo $sp+24:
.text:00000040          lw      $a1, 0x28+var_10($sp)
.text:00000044          lw      $t9, (printf & 0xFFFF)($gp)
.text:00000048          lui     $a0, ($LC2 >> 16)  # "You entered %d...\n"
.text:0000004C          jalr    $t9
.text:00000050          la      $a0, ($LC2 & 0xFFFF)  # "You entered %d...\n"
    ; branch delay slot
; epilogo funzione:
.text:00000054          lw      $ra, 0x28+var_4($sp)
; imposta il valore di ritorno a 0:
.text:00000058          move    $v0, $zero
; ritorna:
.text:0000005C          jr      $ra
.text:00000060          addiu   $sp, 0x28 ; branch delay slot
```

1.12.2 Il classico errore

E' un errore comune (e/o tipografico), passare il valore di *x* anzichè il puntatore a *x*:

```
#include <stdio.h>

int main()
{
        int x;
        printf ("Enter X:\n");

        scanf ("%d", x); // BUG

        printf ("You entered %d...\n", x);

        return 0;
};
```

Cosa succede in questo caso? *x* non è inizializzata e contiene del rumore casuale dallo stack locale. Quando `scanf()` viene chiamata, prende una stringa dall'utente, la traduce in numero e prova a scriverla in *x*, trattandola come un indirizzo di memoria. Ma c'è del rumore casuale, quindi `scanf()` proverà a scrivere ad un indirizzo casuale. Di solito, il programma si blocca.

E' abbastanza interessante il fatto che alcune librerie CRT, nella build di debug, inseriscono dei patterns distinguibili visivamente nella memoria appena allocata, come 0xCCCCCCCC o 0x0BADF00D e così via. In questo caso, *x* conterrà 0xCCCCCCCC e `scanf()` proverà a scrivere all'indirizzo 0xCCCCCCCC. Se si nota che qualcosa in un processo prova a scrivere all' indirizzo 0xCCCCCCCC, si sà che una variabile non inizializzata (o puntatore) è stato usato senza una precedente inizializzazione. Que-

sta soluzione è migliore rispetto al caso in cui la memoria appena allocata venga azzerata.

1.12.3 Varibili globali

E se la variabile x dell'esempio precedente non fosse locale ma globale? Sarebbe stata accessibile da qualunque punto del codice, non soltanto nel corpo della funzione. Le variabili glocali sono considerate anti-pattern, ma per il nostro interesse di sperimentare ci è concesso usarle.

```
#include <stdio.h>

// ora x è una variabile globale
int x;

int main()
{
        printf ("Enter X:\n");

        scanf ("%d", &x);

        printf ("You entered %d...\n", x);

        return 0;
};
```

MSVC: x86

```
_DATA    SEGMENT
COMM    _x:DWORD
$SG2456    DB    'Enter X:', 0aH, 00H
$SG2457    DB    '%d', 00H
$SG2458    DB    'You entered %d...', 0aH, 00H
_DATA    ENDS
PUBLIC    _main
EXTRN    _scanf:PROC
EXTRN    _printf:PROC
; Function compile flags: /Odtp
_TEXT    SEGMENT
_main    PROC
    push   ebp
    mov    ebp, esp
    push   OFFSET $SG2456
    call   _printf
    add    esp, 4
    push   OFFSET _x
    push   OFFSET $SG2457
    call   _scanf
    add    esp, 8
    mov    eax, DWORD PTR _x
    push   eax
    push   OFFSET $SG2458
```

```
    call   _printf
    add    esp, 8
    xor    eax, eax
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
```

In questo caso la variabile x è definita nel segmento _DATA e per essa non viene allocata alcuna memoria nello stack locale. Viene acceduta direttamente, non attraverso lo stack. Le variabili globali non inizializzate non occupano spazio nel file eseguibile (che motivo ci sarebbe di allocare spazio per varibaili inizialmente settate a zero?), ma quando qualcuno accede al loro indirizzo, l' OS allocherà un bloco di zeri al loro posto [74].

Adesso assegnamo esplicitamente un valore alla variabile:

```
int x=10; // valore di default
```

Otteniamo:

```
_DATA   SEGMENT
_x      DD      0aH

...
```

Vediamo qui un valore 0xA di tipo DWORD (DD sta per DWORD = 32 bit) per questa variabile.

Analizzando con IDA il file .exe compilato, notiamo che la variabile *x* è collocata all'inizio del segmento _DATA, e dopo di essa vediamo le stringhe testuali.

Analizzando l'eseguibile dell'esempio precedente con IDA, vedremo qualcosa del genere dove il valore di *x* non era stato impostato:

Listing 1.77: IDA

```
.data:0040FA80 _x              dd ?    ; DATA XREF: _main+10
.data:0040FA80                         ; _main+22
.data:0040FA84 dword_40FA84    dd ?    ; DATA XREF: _memset+1E
.data:0040FA84                         ; unknown_libname_1+28
.data:0040FA88 dword_40FA88    dd ?    ; DATA XREF: ___sbh_find_block+5
.data:0040FA88                         ; ___sbh_free_block+2BC
.data:0040FA8C ; LPVOID lpMem
.data:0040FA8C lpMem           dd ?    ; DATA XREF: ___sbh_find_block+B
.data:0040FA8C                         ; ___sbh_free_block+2CA
.data:0040FA90 dword_40FA90    dd ?    ; DATA XREF: _V6_HeapAlloc+13
.data:0040FA90                         ; __calloc_impl+72
.data:0040FA94 dword_40FA94    dd ?    ; DATA XREF: ___sbh_free_block+2FE
```

_x è contrassegnata con il simbolo ? insieme al resto delle variabili che non necessitano di essere inizializzate. Ciò implica che dopo il caricamento del .exe in memoria,

[74] Questo è il modo in cui vunziona una VM

verrà allocato spazio riempito di zeri per tutte queste variabili [*ISO/IEC 9899:TC3 (C C99 standard)*, (2007)6.7.8p10]. Ma nel file .exe tutte le variabili non inizializzate non occupano alcuno spazio. Questo risulta molto conveniente ad esempio nel caso di array molto grandi.

MSVC: x86 + OllyDbg

Il quadro qui è ancora più semplice:

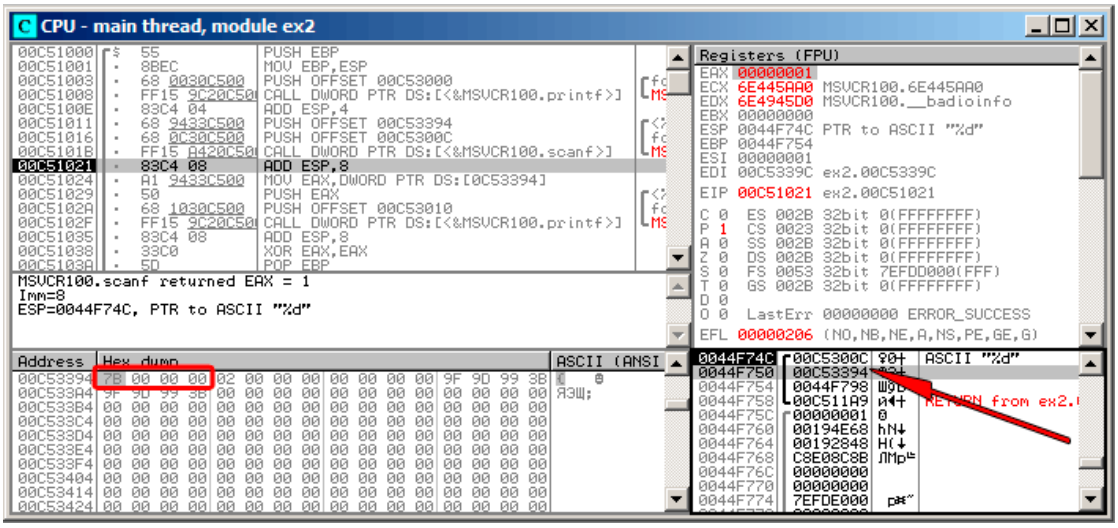


Figura 1.16: OllyDbg: dopo l'esecuzione di `scanf()`

La variabile è collocata nel data segment. Dopo che l'istruzione PUSH (che fa il push dell'indirizzo di x) viene eseguita, l'indirizzo appare nella finestra dello stack. Facciamo click destro su quella riga e selezioniamo «Follow in dump». La variabile apparirà nella finestra di memoria a sinistra. Dopo aver inserito il valore 123 in console, `0x7B` apparirà nella finestra della memoria (vedere regioni evidenziate nello screenshot).

Ma perchè il primo byte è 7B? A rigor di logica, dovremmo trovare `00 00 00 7B`. La causa per cui troviamo invece 7B è detta endianness, e x86 usa la convenzione *little-endian*. Ciò significa che il byte piu basso è scritto per primo, e quello più alto per ultimo. Maggiori informazioni sono disponibili nella sezione: 2.1 on page 281. Tornando all'esempio, il valore a 32-bit è caricato da questo indirizzo di memoria in EAX e passato a `printf()`.

L'indirizzo in memoria di x è `0x00C53394`.

In OllyDbg possiamo osservare la mappa di memoria di un processo (process memory map, Alt-M) e notare che questo indirizzo è dentro il segmento PE `.data` del nostro programma:

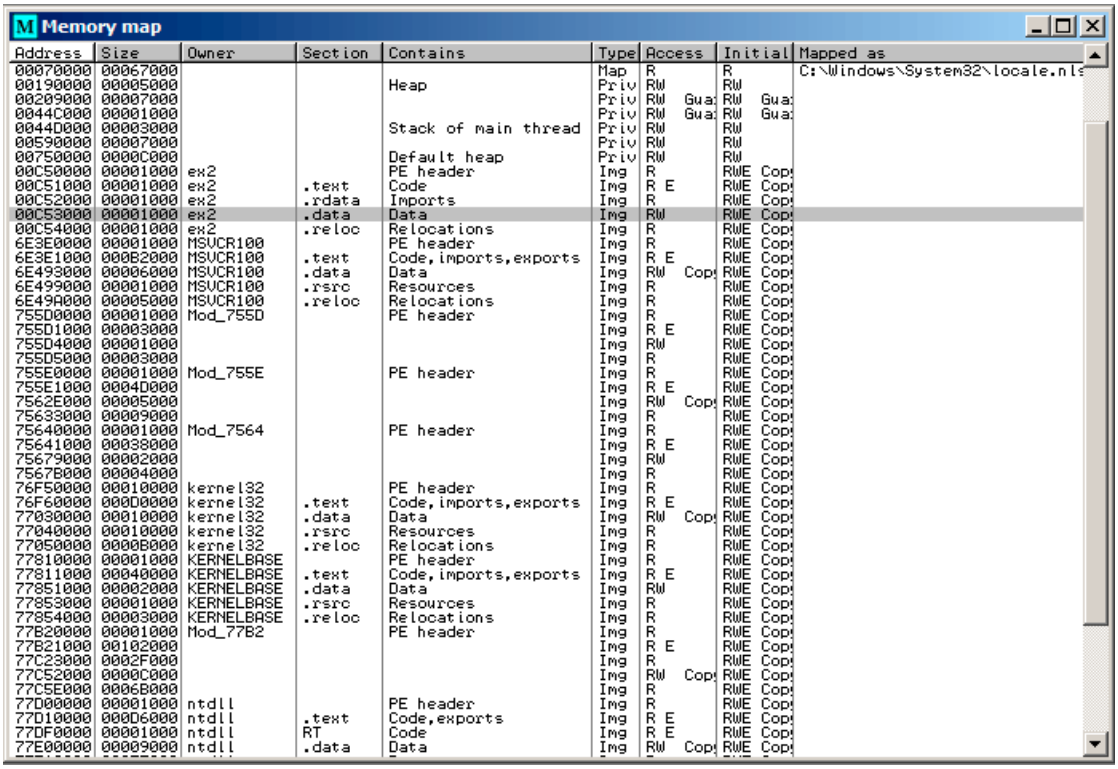


Figura 1.17: OllyDbg: process memory map

GCC: x86

La situazione in Linux è pressoché identica, con la differenza che le variabili non inizializzate sono collocate nel segmento `_bss`. In un file ELF[75] questo segmento ha i seguenti attributi:

```
; Segment type: Uninitialized
; Segment permissions: Read/Write
```

Se invece si inizializza la variabile con un qualunque valore, es. 10, sarà collocata nel segmento `_data`, che ha i seguenti attributi:

```
; Segment type: Pure data
; Segment permissions: Read/Write
```

[75] Executable and Linkable Format: Formato di file eseguibile largamente utilizzato nei sistemi *NIX, Linux incluso

MSVC: x64

Listing 1.78: MSVC 2012 x64

```
_DATA   SEGMENT
COMM    x:DWORD
$SG2924 DB      'Enter X:', 0aH, 00H
$SG2925 DB      '%d', 00H
$SG2926 DB      'You entered %d...', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
main    PROC
$LN3:
        sub     rsp, 40

        lea     rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
        call    printf
        lea     rdx, OFFSET FLAT:x
        lea     rcx, OFFSET FLAT:$SG2925 ; '%d'
        call    scanf
        mov     edx, DWORD PTR x
        lea     rcx, OFFSET FLAT:$SG2926 ; 'You entered %d...'
        call    printf

        ; ritorna 0
        xor     eax, eax

        add     rsp, 40
        ret     0
main    ENDP
_TEXT   ENDS
```

Il codice è pressoché identico a quello in x86. Si noti che l'indirizzo della variabile x è passato a `scanf()` usando un'istruzione LEA, mentre il valore della variabile è passato alla seconda `printf()` usando un'istruzione `MOV`. `DWORD PTR`— è parte del linguaggio assembly (non ha a che vedere con il codice macchina), indica che la dimensione del dato della variabile è 32-bit e l'istruzione `MOV` deve essere codificata in accordo alla dimensione.

ARM: Con ottimizzazione Keil 6/2013 (Modalità Thumb)

Listing 1.79: IDA

```
.text:00000000 ; Segment type: Pure code
.text:00000000         AREA .text, CODE
...
.text:00000000 main
.text:00000000         PUSH    {R4,LR}
.text:00000002         ADR     R0, aEnterX  ; "Enter X:\n"
.text:00000004         BL      __2printf
.text:00000008         LDR     R1, =x
.text:0000000A         ADR     R0, aD       ; "%d"
```

```
.text:0000000C          BL      __0scanf
.text:00000010          LDR     R0, =x
.text:00000012          LDR     R1, [R0]
.text:00000014          ADR     R0, aYouEnteredD___ ; "You entered %d...\n"
.text:00000016          BL      __2printf
.text:0000001A          MOVS    R0, #0
.text:0000001C          POP     {R4,PC}
...
.text:00000020 aEnterX DCB "Enter X:",0xA,0 ; DATA XREF: main+2
.text:0000002A          DCB    0
.text:0000002B          DCB    0
.text:0000002C off_2C   DCD x                ; DATA XREF: main+8
.text:0000002C                               ; main+10
.text:00000030 aD       DCB "%d",0           ; DATA XREF: main+A
.text:00000033          DCB    0
.text:00000034 aYouEnteredD___ DCB "You entered %d...",0xA,0 ; DATA XREF:
    main+14
.text:00000047          DCB 0
.text:00000047 ; .text ends
.text:00000047
...
.data:00000048 ; Segment type: Pure data
.data:00000048          AREA .data, DATA
.data:00000048          ; ORG 0x48
.data:00000048          EXPORT x
.data:00000048 x        DCD 0xA              ; DATA XREF: main+8
.data:00000048                               ; main+10
.data:00000048 ; .data ends
```

La variabile x è ora globale, e perciò è collocata in un altro segmento, ovvero il data segment (*.data*). Ci si potrebbe chiedere perchè le stringhe testuali sono collocate nel code segment (*.text*) e x nel data segment. Il motivo risiede nel fatto che x è una variabile, e per definizione il suo valore potrebbe cambiare (e anche spesso). Le stringhe testuali hanno invece tipo costante, non verranno modificate, e sono quindi collocate nel segmento *.text*.

Il code segnment può a volte trovarsi in un chip ROM[76] (ricordiamoci che oggi si ha spesso a che fare con embedded microelectronics, in cui è comune la scarsità di memoria), e le variabili mutevoli —in RAM.

Memorizzare variabili costanti in RAM non si rivela molto economico quando si ha a disposizione una ROM. Oltretutto, le variabili costanti in RAM devono essere inizializzate, in quanto dopo l'accensione la RAM, ovviamente, contiene informazioni random.

Andando avanti vediamo un puntatore alla variabile x (`off_2C`) nel code segment, e notiamo che tutte le operazioni con quella variabile avvengono attraverso questo puntatore.

Il motivo per cui ciò avviene è che la variabile x potrebbe trovarsi da qualche parte, lontano da questo particolare frammento di codice. Quindi il suo indirizzo deve essere salvato da qualche parte in prossimità del codice.

[76] Memoria di sola lettura (Read-Only Memory)

L'istruzione LDR in Thumb mode può indirizzare soltanto variabili in un intervallo di 1020 byte dalla sua posizione,

e in ARM-mode —variabili in un raggio di ± 4095 byte.

Quindi l'indirizzo della variabile x deve trovarsi nelle vicinanze, poichè non c'è garanzia che il linker sia in grado di collocare la variabile sufficientemente vicina al codice (potrebbe addirittura trovarsi in un chip di memoria esterno!).

Un'altra cosa: se una variabile è dichiarata come *const*, il compilatore Keil la colloca nel segmento `.constdata`. Forse successivamente il linker potrebbe piazzare anche questo segmento nella ROM, insieme al code segment.

ARM64

Listing 1.80: Senza ottimizzazione GCC 4.9.1 ARM64

```
1         .comm   x,4,4
2 .LC0:
3         .string "Enter X:"
4 .LC1:
5         .string "%d"
6 .LC2:
7         .string "You entered %d...\n"
8 f5:
9 ; salva FP e LR nello stack frame:
10        stp     x29, x30, [sp, -16]!
11 ; imposta lo stack frame (FP=SP)
12        add     x29, sp, 0
13 ; imposta il puntatore alla stringa "Enter X:":
14        adrp    x0, .LC0
15        add     x0, x0, :lo12:.LC0
16        bl      puts
17 ; imposta il puntatore alla stringa "%d":
18        adrp    x0, .LC1
19        add     x0, x0, :lo12:.LC1
20 ; forma l'indirizzo della variabile globale x:
21        adrp    x1, x
22        add     x1, x1, :lo12:x
23        bl      __isoc99_scanf
24 ; forma di nuovo l'indirizzo della variabile globale x:
25        adrp    x0, x
26        add     x0, x0, :lo12:x
27 ; carica il valore dalla memoria a questo indirizzo:
28        ldr     w1, [x0]
29 ; imposta il puntatore alla stringa "You entered %d...\n":
30        adrp    x0, .LC2
31        add     x0, x0, :lo12:.LC2
32        bl      printf
33 ; ritorna 0
34        mov     w0, 0
35 ; ripristina FP e LR:
36        ldp     x29, x30, [sp], 16
37        ret
```

In questo caso la variabile x è dichiarata come globale ed il suo indirizzo è calcolato utilizzando la coppia di istruzioni ADRP/ADD (righe 21 e 25).

MIPS

Variabili globali non inizializzate

La variabile x è ora globale. Compiliamo in un file eseguibile anziché oggetto e carichiamolo in IDA. IDA mostra la variabile x nella sezione ELF .sbss (ricordate il «Global Pointer»? 1.5.4 on page 33), poiché la variabile non è inizialmente inizializzata.

Listing 1.81: Con ottimizzazione GCC 4.4.5 (IDA)

```
.text:004006C0 main:
.text:004006C0
.text:004006C0 var_10  = -0x10
.text:004006C0 var_4   = -4
.text:004006C0
; prologo funzione:
.text:004006C0         lui     $gp, 0x42
.text:004006C4         addiu   $sp, -0x20
.text:004006C8         li      $gp, 0x418940
.text:004006CC         sw      $ra, 0x20+var_4($sp)
.text:004006D0         sw      $gp, 0x20+var_10($sp)
; chiama puts():
.text:004006D4         la      $t9, puts
.text:004006D8         lui     $a0, 0x40
.text:004006DC         jalr    $t9 ; puts
.text:004006E0         la      $a0, aEnterX     # "Enter X:" ; branch delay
    slot
; chiama scanf():
.text:004006E4         lw      $gp, 0x20+var_10($sp)
.text:004006E8         lui     $a0, 0x40
.text:004006EC         la      $t9, __isoc99_scanf
; prepara l'indirizzo di x:
.text:004006F0         la      $a1, x
.text:004006F4         jalr    $t9 ; __isoc99_scanf
.text:004006F8         la      $a0, aD          # "%d" ; branch delay slot
; chiama printf():
.text:004006FC         lw      $gp, 0x20+var_10($sp)
.text:00400700         lui     $a0, 0x40
; prendi l'indirizzo di x:
.text:00400704         la      $v0, x
.text:00400708         la      $t9, printf
; prendi il valore dalla variabile "x" e passala a printf() in $a1:
.text:0040070C         lw      $a1, (x - 0x41099C)($v0)
.text:00400710         jalr    $t9 ; printf
.text:00400714         la      $a0, aYouEnteredD___  # "You entered %d...\n"
    ; branch delay slot
; epilogo funzione:
.text:00400718         lw      $ra, 0x20+var_4($sp)
.text:0040071C         move    $v0, $zero
.text:00400720         jr      $ra
```

```
.text:00400724         addiu   $sp, 0x20  ; branch delay slot

...

.sbss:0041099C  # Segment type: Uninitialized
.sbss:0041099C         .sbss
.sbss:0041099C         .globl x
.sbss:0041099C x:      .space 4
.sbss:0041099C
```

IDA riduce la quantità di informazioni, quindi facciamo anche un listing usando objdump e lo commentiamo:

Listing 1.82: Con ottimizzazione GCC 4.4.5 (objdump)

```
004006c0 <main>:
; prologo funzione:
  4006c0:  3c1c0042  lui     gp,0x42
  4006c4:  27bdffe0  addiu   sp,sp,-32
  4006c8:  279c8940  addiu   gp,gp,-30400
  4006cc:  afbf001c  sw      ra,28(sp)
  4006d0:  afbc0010  sw      gp,16(sp)
; chiama puts():
  4006d4:  8f998034  lw      t9,-32716(gp)
  4006d8:  3c040040  lui     a0,0x40
  4006dc:  0320f809  jalr    t9
  4006e0:  248408f0  addiu   a0,a0,2288 ; branch delay slot
; chiama scanf():
  4006e4:  8fbc0010  lw      gp,16(sp)
  4006e8:  3c040040  lui     a0,0x40
  4006ec:  8f998038  lw      t9,-32712(gp)
; prepara l'indirizzo di x:
  4006f0:  8f858044  lw      a1,-32700(gp)
  4006f4:  0320f809  jalr    t9
  4006f8:  248408fc  addiu   a0,a0,2300 ; branch delay slot
; chiama printf():
  4006fc:  8fbc0010  lw      gp,16(sp)
  400700:  3c040040  lui     a0,0x40
; prendi l'indirizzo di x:
  400704:  8f828044  lw      v0,-32700(gp)
  400708:  8f99803c  lw      t9,-32708(gp)
; prendi il valore di "x" e passalo a printf() in $a1:
  40070c:  8c450000  lw      a1,0(v0)
  400710:  0320f809  jalr    t9
  400714:  24840900  addiu   a0,a0,2304 ; branch delay slot
; epilogo funzione:
  400718:  8fbf001c  lw      ra,28(sp)
  40071c:  00001021  move    v0,zero
  400720:  03e00008  jr      ra
  400724:  27bd0020  addiu   sp,sp,32   ; branch delay slot
; serie di NOP usata per allineare l'inizio della prossima funzione ad un
    confine di 16-byte:
  400728:  00200825  move    at,at
  40072c:  00200825  move    at,at
```

Vediamo che l'indirizzo della variabile x viene letto da un buffer dati di 64KiB usando il GP a cui viene sommato un offset negativo (riga 18). Inoltre gli indirizzi delle tre funzioni esterne usate nel programma (`puts()`, `scanf()`, `printf()`) sono anch'essi letti dal buffer dati globale di 64KiB usando il GP (righe 9, 16 e 26). GP punta a metà del buffer, e gli offset suggeriscono che gli indirizzi delle tre funzioni e della variabile sono tutti memorizzati da qualche parte vicino all'inizio di quel buffer. Ciò ha senso in quanto il nostro esempio è davvero molto piccolo.

Un'altra cosa che vale la pena notare è che la funzione finisce con due NOP (`MOVE $AT,$AT` — una "idle instruction"), per allineare l'inizio della prossima funzione ad un confine di 16-byte.

Variabile globale inizializzata

Modifichiamo il nostro esempio assegnando un valore predefinito alla variabile x:

```
int x=10; // valore di default
```

Adesso IDA mostra che la variabile x risiede nella sezione .data:

Listing 1.83: Con ottimizzazione GCC 4.4.5 (IDA)

```
.text:004006A0 main:
.text:004006A0
.text:004006A0 var_10 = -0x10
.text:004006A0 var_8  = -8
.text:004006A0 var_4  = -4
.text:004006A0
.text:004006A0         lui     $gp, 0x42
.text:004006A4         addiu   $sp, -0x20
.text:004006A8         li      $gp, 0x418930
.text:004006AC         sw      $ra, 0x20+var_4($sp)
.text:004006B0         sw      $s0, 0x20+var_8($sp)
.text:004006B4         sw      $gp, 0x20+var_10($sp)
.text:004006B8         la      $t9, puts
.text:004006BC         lui     $a0, 0x40
.text:004006C0         jalr    $t9 ; puts
.text:004006C4         la      $a0, aEnterX     # "Enter X:"
.text:004006C8         lw      $gp, 0x20+var_10($sp)
; prepara la parte alta dell'indirizzo di x:
.text:004006CC         lui     $s0, 0x41
.text:004006D0         la      $t9, __isoc99_scanf
.text:004006D4         lui     $a0, 0x40
; aggiungi la parte bassa dell'indirizzo di x:
.text:004006D8         addiu   $a1, $s0, (x - 0x410000)
; ora l'indirizzo di x è in $a1.
.text:004006DC         jalr    $t9 ; __isoc99_scanf
.text:004006E0         la      $a0, aD          # "%d"
.text:004006E4         lw      $gp, 0x20+var_10($sp)
; prendi una word dalla memoria:
.text:004006E8         lw      $a1, x
; ora il valore di x è in $a1.
```

```
.text:004006EC        la      $t9, printf
.text:004006F0        lui     $a0, 0x40
.text:004006F4        jalr    $t9 ; printf
.text:004006F8        la      $a0, aYouEnteredD___  # "You entered %d...\n"
.text:004006FC        lw      $ra, 0x20+var_4($sp)
.text:00400700        move    $v0, $zero
.text:00400704        lw      $s0, 0x20+var_8($sp)
.text:00400708        jr      $ra
.text:0040070C        addiu   $sp, 0x20

...

.data:00410920        .globl x
.data:00410920 x:     .word 0xA
```

Perchè non in .sdata? Può dipendere da qualche opzione di GCC?

Ciononostante x si trova in .data, e possiamo vedere come si lavora con variabili localizzate in questa area di memoria generica.

L'indirizzo della variabile deve essere formato utilizzando un paio di istruzioni.

Nel nostro caso sono `LUI` («Load Upper Immediate») e `ADDIU` («Add Immediate Unsigned Word»).

Vediamo anche il listato di objdump per maggiore approfondimento:

Listing 1.84: Con ottimizzazione GCC 4.4.5 (objdump)

```
004006a0 <main>:
  4006a0:  3c1c0042  lui     gp,0x42
  4006a4:  27bdffe0  addiu   sp,sp,-32
  4006a8:  279c8930  addiu   gp,gp,-30416
  4006ac:  afbf001c  sw      ra,28(sp)
  4006b0:  afb00018  sw      s0,24(sp)
  4006b4:  afbc0010  sw      gp,16(sp)
  4006b8:  8f998034  lw      t9,-32716(gp)
  4006bc:  3c040040  lui     a0,0x40
  4006c0:  0320f809  jalr    t9
  4006c4:  248408d0  addiu   a0,a0,2256
  4006c8:  8fbc0010  lw      gp,16(sp)
; prepara la parte alta dell'indirizzo di x:
  4006cc:  3c100041  lui     s0,0x41
  4006d0:  8f998038  lw      t9,-32712(gp)
  4006d4:  3c040040  lui     a0,0x40
; aggiungi la parte bassa dell'indirizzo di x:
  4006d8:  26050920  addiu   a1,s0,2336
; ora l'indirizzo di x è in $a1.
  4006dc:  0320f809  jalr    t9
  4006e0:  248408dc  addiu   a0,a0,2268
  4006e4:  8fbc0010  lw      gp,16(sp)
; la parte alta dell'indirizzo di x è ancora in $s0.
; aggiungigli la parte bassa e carica una word dalla memoria:
  4006e8:  8e050920  lw      a1,2336(s0)
; ora il valore di x è in $a1.
```

```
  4006ec:  8f99803c  lw      t9,-32708(gp)
  4006f0:  3c040040  lui     a0,0x40
  4006f4:  0320f809  jalr    t9
  4006f8:  248408e0  addiu   a0,a0,2272
  4006fc:  8fbf001c  lw      ra,28(sp)
  400700:  00001021  move    v0,zero
  400704:  8fb00018  lw      s0,24(sp)
  400708:  03e00008  jr      ra
  40070c:  27bd0020  addiu   sp,sp,32
```

Notiamo che l'indirizzo è formato usando LUI e ADDIU, ma la parte alta dell'indirizzo è ancora nel registro $S0 , ed è possibile codificare l'offset in un'istruzione LW («Load Word»), perciò una singola LW è sufficiente per caricare un valore dalla variabile e passarlo a `printf()`.

I registri che memorizzano dati temporanei hanno il prefisso T-, ma qui vediamo anche alcuni con prefisso S-, il cui contenuto deve essere preservato prima del loro utilizzo in altre funzioni (i.e., salvate altrove).

Questo è il motivo per cui il valore di $S0 era stato settato all'indirizzo 0x4006cc e usato nuovamente all'indirizzo 0x4006e8, dopo la chiamata a `scanf()`. La funzione `scanf()` non ne cambia il valore.

1.12.4 scanf()

Come già detto in precedenza, usare `scanf()` oggi è un pò antiquato. Se proprio dobbiamo, è necessario almeno controllare se `scanf()` termina correttamente senza errori.

```
#include <stdio.h>

int main()
{
        int x;
        printf ("Enter X:\n");

        if (scanf ("%d", &x)==1)
                printf ("You entered %d...\n", x);
        else
                printf ("What you entered? Huh?\n");

        return 0;
};
```

Per standard, la funzione `scanf()`[77] restituisce il numero di campi che è riuscita a leggere con successo. Nel nostro caso, se tutto va bene e l'utente inserisce un numero, `scanf()` restituisce 1, oppure 0 (o EOF[78]) in caso di errore.

Aggiungiamo un po' di codice C per controllare che `scanf()` restituisca un valore e stampi un messaggio in caso di errore.

[77] scanf, wscanf: MSDN
[78] End of File

Funziona come ci si aspetta:

```
C:\...>ex3.exe
Enter X:
123
You entered 123...

C:\...>ex3.exe
Enter X:
ouch
What you entered? Huh?
```

MSVC: x86

Questo è l'output assembly ottenuto con MSVC 2010:

```
        lea     eax, DWORD PTR _x$[ebp]
        push    eax
        push    OFFSET $SG3833 ; '%d', 00H
        call    _scanf
        add     esp, 8
        cmp     eax, 1
        jne     SHORT $LN2@main
        mov     ecx, DWORD PTR _x$[ebp]
        push    ecx
        push    OFFSET $SG3834 ; 'You entered %d...', 0aH, 00H
        call    _printf
        add     esp, 8
        jmp     SHORT $LN1@main
$LN2@main:
        push    OFFSET $SG3836 ; 'What you entered? Huh?', 0aH, 00H
        call    _printf
        add     esp, 4
$LN1@main:
        xor     eax, eax
```

La funzione chiamante (chiamante) `main()` necessita di ottenere il risultato della funzione chiamata (chiamata), e pertanto quest'ultima lo restituisce nel registro EAX register.

Il controllo viene eseguito con l'aiuto dell'istruzione `CMP EAX, 1` (*CoMPare*). In altre parole, confrontiamo il valore nel registro EAX con 1.

U jump condizionale `JNE` segue l'istruzione `CMP`. `JNE` sta per *Jump if Not Equal*.

Quindi, se il valore nel registro EAX non è uguale a 1, la CPU passerà l'esecuzione all'indirizzo specificato nell'operando di `JNE`, nel nostro caso `$LN2@main`. Passare il controllo a questo indirizzo risulta nel fatto che la CPU eseguirà la funzione `printf()` con l'argomento `What you entered? Huh?`. Ma se tutto va bene, il salto condizionale non viene effettuato, e viene eseguita un'altra chiamata a `printf()` con due argomenti: `'You entered %d...'` e il valore di x.

Poichè in questo caso la seconda `printf()` non deve essere eseguita, c'è un jump non condizionale (unconditional jump) `JMP` che la precede. Questo passa il controllo al punto dopo la seconda `printf()` e prima dell'istruzione `XOR EAX, EAX`, che implementa `return 0`.

Possiamo quindi dire che il confronto di valori è *solitamente* implementato con una coppia di istruzioni `CMP`/`Jcc`, dove *cc* è un *condition code*. `CMP` confronta due valori e imposta i flag del processore [79]. `Jcc` controlla questi flag e decide se passare o meno il controllo all'indirizzo specificato.

Può sembrare un paradosso, ma l'istruzione `CMP` è in effetti una `SUB` (subtract). Tutte le istruzioni aritmetiche settano i flag del processore, non solo `CMP`. Se confrontiamo 1 e 1, $1 - 1$ è 0 e quindi il flag `ZF` sarebbe impostato a 1 (significando che l'ultimo risultato era 0). In nessun'altra circostanza il flag `ZF` può essere impostato, eccetto il caso in cui gli operandi sono uguali. `JNE` controlla soltanto il flag `ZF` e salta se e solo se il flag non è settato. `JNE` è infatti un sinonimo di `JNZ` (*Jump if Not Zero*). L'assembler traduce entrambe le istruzioni `JNE` e `JNZ` nello stesso opcode. Quindi l'istruzione `CMP` può essere sostituita dall'istruzione `SUB` e quasi tutto funzionerà, con la differenza che `SUB` altera il valore del primo operando. `CMP` è uguale a *SUB senza salvare il risultato, ma settando i flag*.

MSVC: x86: IDA

E' arrivato il momento di avviare IDA. A proposito, per i principianti è buona norma usare l'opzione `/MD` in MSVC, che significa che tutte le funzioni standard non saranno linkate dentro il file eseguibile, ma importate dal file `MSVCR*.DLL`. In questo modo sarà più facile vedere quali funzioni standard sono usate, e dove.

Quando si analizza il codice con IDA, è sempre molto utile lasciare note per se stessi (e per gli altri, nel caso in cui si lavori in gruppo). Per esempio, analizzando questo esempio, notiamo che `JNZ` sarà innescato in caso di errore. E' possibile muovere il cursore fino alla label, premere «n» e rinominarla in «errore». Creare un'altra label —in «exit». Ecco il mio risultato:

```
.text:00401000 _main proc near
.text:00401000
.text:00401000 var_4 = dword ptr -4
.text:00401000 argc  = dword ptr  8
.text:00401000 argv  = dword ptr  0Ch
.text:00401000 envp  = dword ptr  10h
.text:00401000
.text:00401000       push    ebp
.text:00401001       mov     ebp, esp
.text:00401003       push    ecx
.text:00401004       push    offset Format ; "Enter X:\n"
.text:00401009       call    ds:printf
.text:0040100F       add     esp, 4
.text:00401012       lea     eax, [ebp+var_4]
.text:00401015       push    eax
.text:00401016       push    offset aD ; "%d"
.text:0040101B       call    ds:scanf
```

[79] x86 flags, vedere anche: wikipedia.

```
.text:00401021       add     esp, 8
.text:00401024       cmp     eax, 1
.text:00401027       jnz     short error
.text:00401029       mov     ecx, [ebp+var_4]
.text:0040102C       push    ecx
.text:0040102D       push    offset aYou ; "You entered %d...\n"
.text:00401032       call    ds:printf
.text:00401038       add     esp, 8
.text:0040103B       jmp     short exit
.text:0040103D
.text:0040103D error: ; CODE XREF: _main+27
.text:0040103D       push    offset aWhat ; "What you entered? Huh?\n"
.text:00401042       call    ds:printf
.text:00401048       add     esp, 4
.text:0040104B
.text:0040104B exit: ; CODE XREF: _main+3B
.text:0040104B       xor     eax, eax
.text:0040104D       mov     esp, ebp
.text:0040104F       pop     ebp
.text:00401050       retn
.text:00401050 _main endp
```

Adesso è leggermente più facile capire il codice. Non è comunque una buona idea commentare ogni istruzione!

Si possono anche nascondere (collapse) parti di una funzione in IDA. Per farlo, selezionare il blocco e premere Ctrl-«–» sul tastierino numerico, inserendo il testo da visualizzare al posto del blocco di codice.

Nascondiamo due blocchi e diamogli un nome:

```
.text:00401000 _text segment para public 'CODE' use32
.text:00401000       assume cs:_text
.text:00401000       ;org 401000h
.text:00401000 ; ask for X
.text:00401012 ; get X
.text:00401024       cmp  eax, 1
.text:00401027       jnz  short error
.text:00401029 ; print result
.text:0040103B       jmp  short exit
.text:0040103D
.text:0040103D error: ; CODE XREF: _main+27
.text:0040103D       push offset aWhat ; "What you entered? Huh?\n"
.text:00401042       call ds:printf
.text:00401048       add  esp, 4
.text:0040104B
.text:0040104B exit: ; CODE XREF: _main+3B
.text:0040104B       xor  eax, eax
.text:0040104D       mov  esp, ebp
.text:0040104F       pop  ebp
.text:00401050       retn
.text:00401050 _main endp
```

Per espandere dei blocchi nascosti, premere Ctrl-«+» sul tastierino numerico.

Premendo «spazio», possiamo vedere come IDA rappresenta una funzione in forma di grafo:

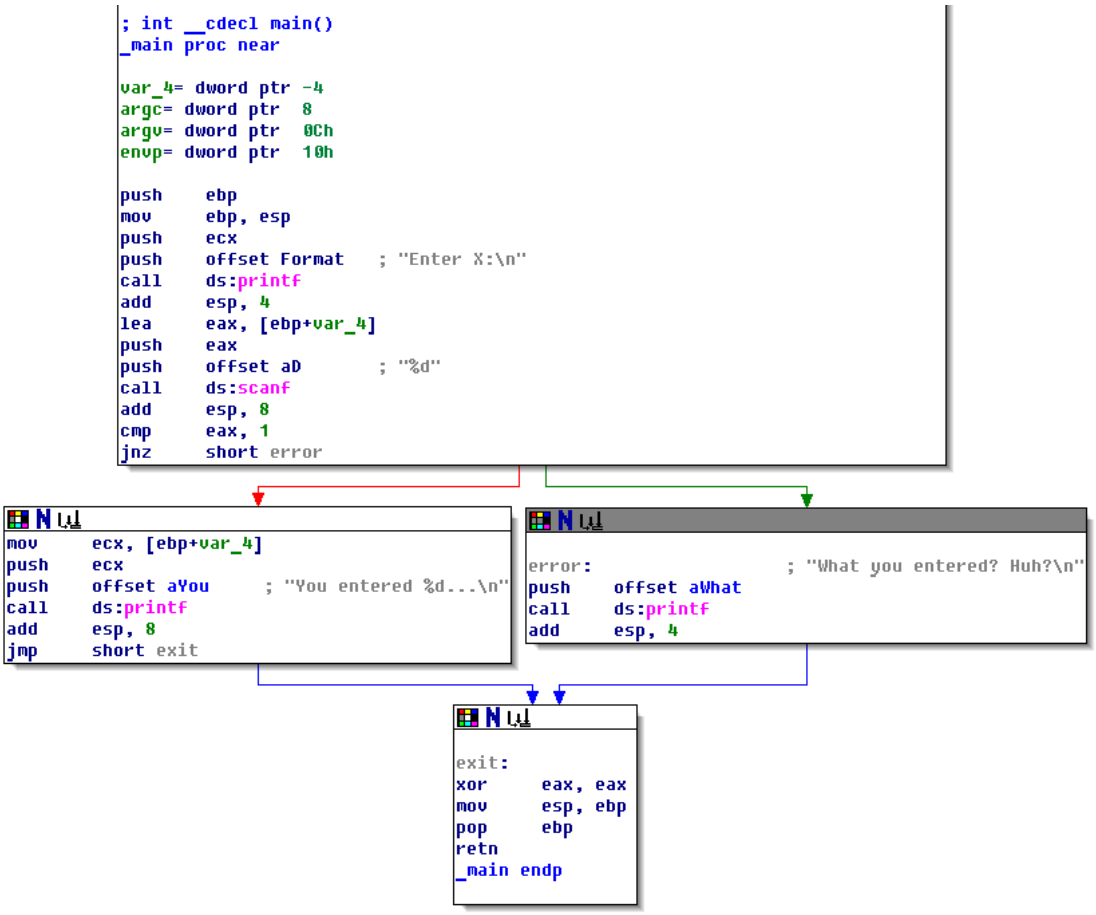


Figura 1.18: Graph mode in IDA

Ci sono due frecce dopo ogni jump condizionale: verde e rossa. La freccia verde punta al blocco che viene eseguito se il jump è innescato, la rossa nel caso opposto.

Anche in questa modalità è possibile ”chiudere” i nodi e dargli un’etichetta («group nodes»). Facciamolo per 3 blocchi:

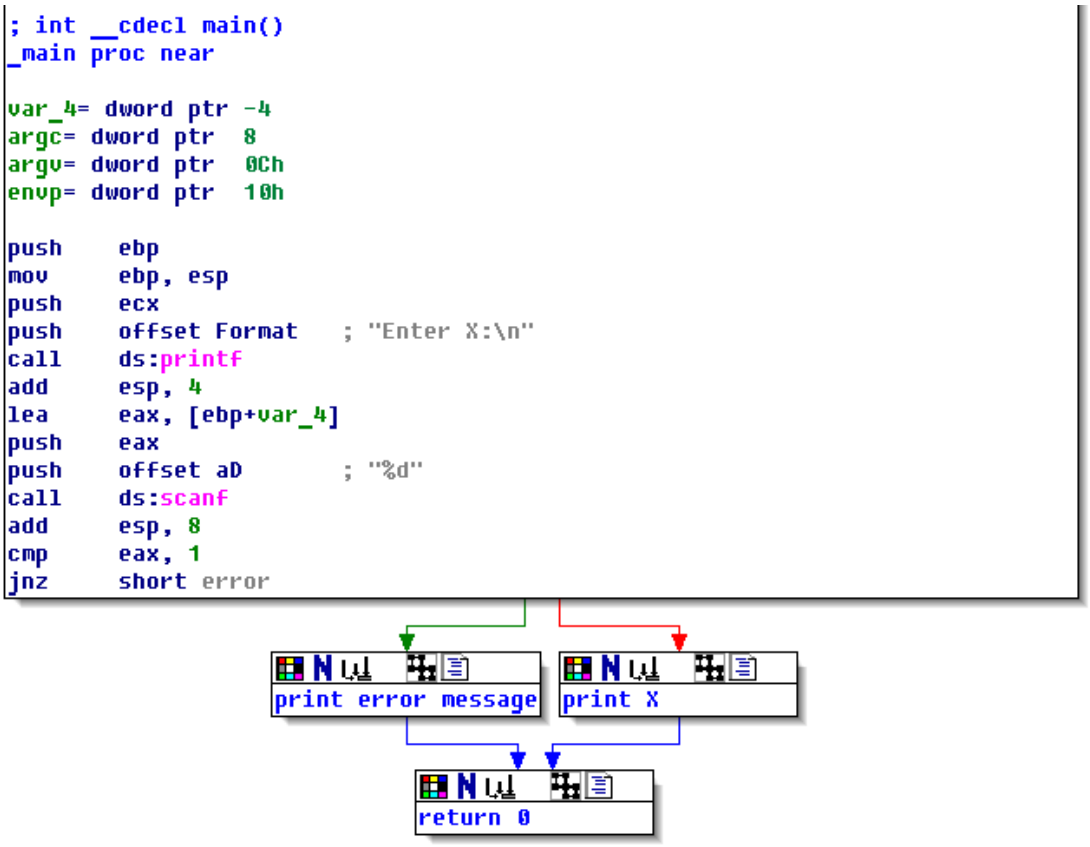


Figura 1.19: Graph mode in IDA con 3 nodi ”chiusi”

Come si può vedere questa funzione è molto utile. Si può dire che una buona parte del lavoro di un reverse engineer (così come di altri tipi di ricercatori) è rappresentata dalla riduzione della quantità di informazioni da trattare.

MSVC: x86 + OllyDbg

Proviamo ad hackerare il nostro programma in OllyDbg, forzandolo a pensare che `scanf()` funzioni sempre senza errori. Quando l'indirizzo di una variabile locale è passato a `scanf()`, la variabile inizialmente contiene un valore random inutile, in questo caso `0x6E494714`:

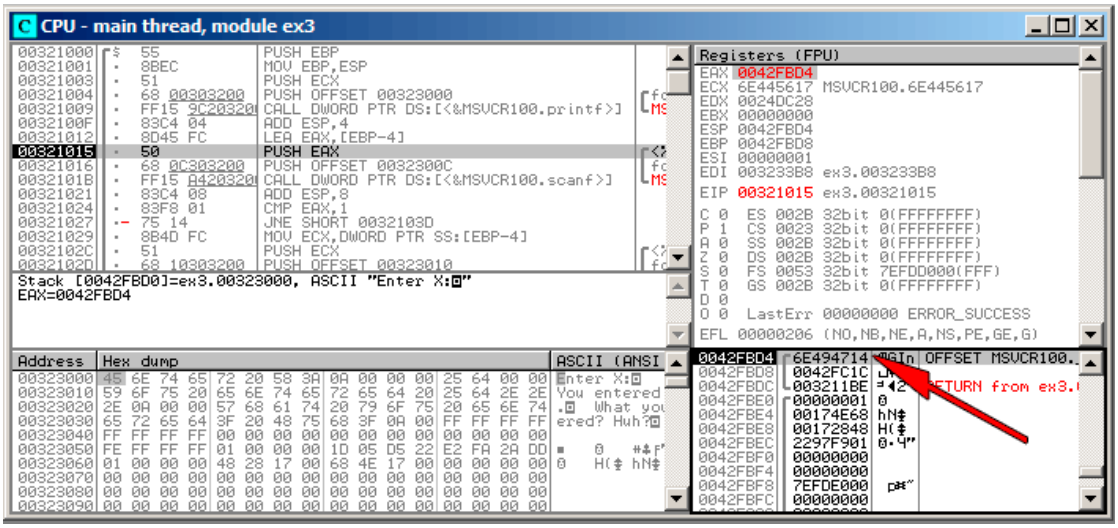


Figura 1.20: OllyDbg: passaggio dell'indirizzo della variabile a `scanf()`

Quando `scanf()` viene eseguita, immettiamo nella console qualcosa di diverso da un numero, come «asdasd». `scanf()` finisce con 0 in EAX, indicante che un errore si è verificato.

Possiamo anche controllare la variabile locale nello stack e notare che non è stata modificata. Infatti cosa avrebbe potuto scrivere `scanf()` in essa? Non ha fatto niente oltre che restituire zero.

Proviamo ad «hackerare» il nostro programma. Click destro su EAX, Tra le opzioni vediamo «Set to 1». Esattamente ciò che ci serve.

Adesso abbiamo 1 in EAX, il controllo successivo sta per essere eseguito come previsto, e `printf()` stamperà il valore della variabile nello stack.

Quando avviamo il programma (F9) vediamo il seguente output nella finestra della console:

Listing 1.85: finestra della console

```
Enter X:
asdasd
You entered 1850296084...
```

1850296084 è infatti la rappresentazione decimale del numero nello stack (`0x6E494714`)!

MSVC: x86 + Hiew

Quanto detto può essere anche usato come semplice esempio di patching di un eseguibile. Possiamo provare a modificare l'eseguibile in modo che il programma stampi sempre l'input, a prescindere da cosa si inserisce.

Assumendo che l'eseguibile sia compilato rispetto `MSVCR*.DLL` esterna (ovvero con l'opzione `/MD`) [80], vediamo la funzione `main()` all'inizio della sezione `.text`. Apriamo l'eseguibile con Hiew e troviamo l'inizio della sezione `.text` (Enter, F8, F6, Enter, Enter).

Vedremo questo:

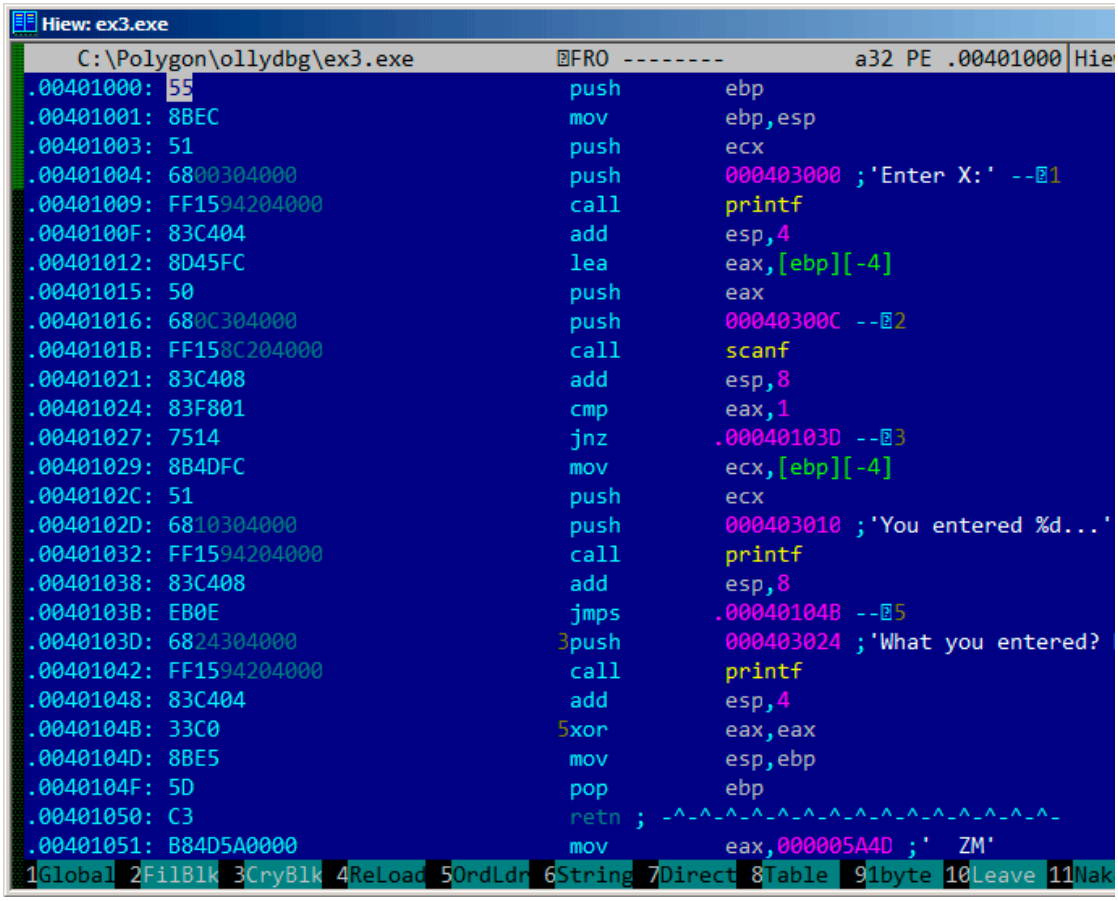


Figura 1.21: Hiew: funzione `main()`

Hiew trova le stringhe ASCIIZ[81] e le visualizza, così come i nomi delle funzioni importate.

[80] detta anche «dynamic linking»
[81] ASCII Zero ( )

Spostiamo il cursore all'indirizzo `.00401027` (dove si trova l'istruzione JNZ che vogliamo bypassare), premiamo F3, e scriviamo «9090» (cioe' due NOP):

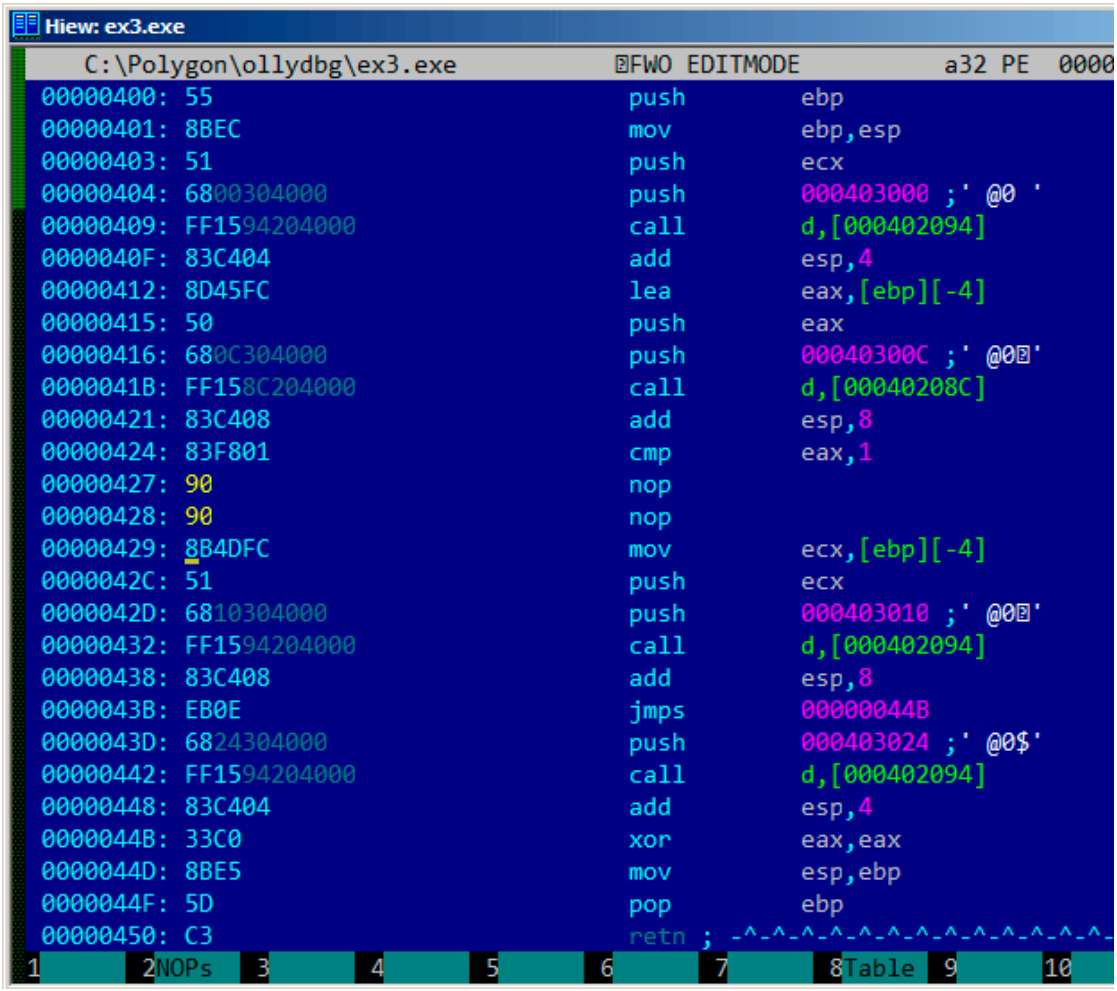


Figura 1.22: Hiew: sostituzione di JNZ con due NOP

Premiamo quindi F9 (update). L'eseguibile viene quindi salvato su disco, e si comporterà come vogliamo.

Utilizzare due NOP non rappresenta l'approccio esteticamente migliore. Un altro modo di patchare questa istruzione è scrivere 0 al secondo byte dell'opcode (offset di salto), in modo che JNZ salti sempre alla prossima istruzione.

Potremmo anche fare l'opposto: sostituire il primo byte con EB senza toccare il secondo byte (offset di salto). Otterremmo un jump non condizionale che è sempre eseguito. In questo caso il messaggio di errore sarebbe stampato sempre, a prescindere dall'input.

MSVC: x64

Poichè qui lavoriamo con variabili di tipo *int*, che sono sempre a 32-bit in x86-64, vediamo che viene usata la parte a 32-bit dei registri (con il prefisso E-). Lavorando invece con i puntatori, sono usate la parti a 64-bit dei registri (con il prefisso R-).

Listing 1.86: MSVC 2012 x64

```
_DATA   SEGMENT
$SG2924 DB      'Enter X:', 0aH, 00H
$SG2926 DB      '%d', 00H
$SG2927 DB      'You entered %d...', 0aH, 00H
$SG2929 DB      'What you entered? Huh?', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
x$ = 32
main    PROC
$LN5:
        sub     rsp, 56
        lea     rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
        call    printf
        lea     rdx, QWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG2926 ; '%d'
        call    scanf
        cmp     eax, 1
        jne     SHORT $LN2@main
        mov     edx, DWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG2927 ; 'You entered %d...'
        call    printf
        jmp     SHORT $LN1@main
$LN2@main:
        lea     rcx, OFFSET FLAT:$SG2929 ; 'What you entered? Huh?'
        call    printf
$LN1@main:
        ; ritorna 0
        xor     eax, eax
        add     rsp, 56
        ret     0
main    ENDP
_TEXT   ENDS
END
```

ARM

ARM: Con ottimizzazione Keil 6/2013 (Modalità Thumb)

Listing 1.87: Con ottimizzazione Keil 6/2013 (Modalità Thumb)

```
var_8    = -8

         PUSH    {R3,LR}
         ADR     R0, aEnterX     ; "Enter X:\n"
```

```
        BL      __2printf
        MOV     R1, SP
        ADR     R0, aD          ; "%d"
        BL      __0scanf
        CMP     R0, #1
        BEQ     loc_1E
        ADR     R0, aWhatYouEntered ; "What you entered? Huh?\n"
        BL      __2printf

loc_1A                          ; CODE XREF: main+26
        MOVS    R0, #0
        POP     {R3,PC}

loc_1E                          ; CODE XREF: main+12
        LDR     R1, [SP,#8+var_8]
        ADR     R0, aYouEnteredD___ ; "You entered %d...\n"
        BL      __2printf
        B       loc_1A
```

Le due nuove istruzioni qui sono CMP e BEQ[82].

CMP è analoga all'istruzione omonima in x86, sottrae uno degli argomenti dall'altro e aggiorna il conditional flags (se necessario).

BEQ salta ad un altro indirizzo se gli operandi sono uguali, o se il risultato dell'ultima operazione era 0, oppure ancora se il flag Z è 1. Si comporta come JZ in x86.

Tutto il resto è semplice: il flusso di esecuzione si divide in due rami, e successivamente i due rami convergono al punto in cui 0 viene scritto in R0 come valore di ritorno di una funzione, infine la funzione termina.

ARM64

Listing 1.88: Senza ottimizzazione GCC 4.9.1 ARM64

```
1  .LC0:
2          .string "Enter X:"
3  .LC1:
4          .string "%d"
5  .LC2:
6          .string "You entered %d...\n"
7  .LC3:
8          .string "What you entered? Huh?"
9  f6:
10 ; salva FP e LR nello stack frame:
11         stp     x29, x30, [sp, -32]!
12 ; imposta lo stack frame (FP=SP)
13         add     x29, sp, 0
14 ; imposta il puntatore alla stringa "Enter X:":
15         adrp    x0, .LC0
16         add     x0, x0, :lo12:.LC0
```

[82] (PowerPC, ARM) Branch if Equal

```
17          bl      puts
18  ; imposta il puntatore alla stringa "%d":
19          adrp    x0, .LC1
20          add     x0, x0, :lo12:.LC1
21  ; calcola l'indirizzo della variabile x nello stack locale
22          add     x1, x29, 28
23          bl      __isoc99_scanf
24  ; il risultato di scanf() viene messo in W0.
25  ; controlla:
26          cmp     w0, 1
27  ; BNE è Branch if Not Equal
28  ; quindi se W0<>1, avverrà il salto a L2
29          bne     .L2
30  ; in questo momento W0=1, che significa niente errore
31  ; carica il valore di x dallo stack locale
32          ldr     w1, [x29,28]
33  ; imposta il puntatore alla stringa "You entered %d...\n":
34          adrp    x0, .LC2
35          add     x0, x0, :lo12:.LC2
36          bl      printf
37  ; salta il codice, il quale stampa la stringa "What you entered? Huh?"
38          b       .L3
39  .L2:
40  ; imposta il puntatore alla stringa "What you entered? Huh?":
41          adrp    x0, .LC3
42          add     x0, x0, :lo12:.LC3
43          bl      puts
44  .L3:
45  ; ritorna 0
46          mov     w0, 0
47  ; ripristina FP e LR:
48          ldp     x29, x30, [sp], 32
49          ret
```

Il flusso di codice in questo caso si divide con l'uso della coppia di istruzioni CMP/BNE (Branch if Not Equal).

MIPS

Listing 1.89: Con ottimizzazione GCC 4.4.5 (IDA)

```
.text:004006A0 main:
.text:004006A0
.text:004006A0 var_18    = -0x18
.text:004006A0 var_10    = -0x10
.text:004006A0 var_4     = -4
.text:004006A0
.text:004006A0           lui     $gp, 0x42
.text:004006A4           addiu   $sp, -0x28
.text:004006A8           li      $gp, 0x418960
.text:004006AC           sw      $ra, 0x28+var_4($sp)
.text:004006B0           sw      $gp, 0x28+var_18($sp)
.text:004006B4           la      $t9, puts
```

```
.text:004006B8            lui     $a0, 0x40
.text:004006BC            jalr    $t9 ; puts
.text:004006C0            la      $a0, aEnterX     # "Enter X:"
.text:004006C4            lw      $gp, 0x28+var_18($sp)
.text:004006C8            lui     $a0, 0x40
.text:004006CC            la      $t9, __isoc99_scanf
.text:004006D0            la      $a0, aD          # "%d"
.text:004006D4            jalr    $t9 ; __isoc99_scanf
.text:004006D8            addiu   $a1, $sp, 0x28+var_10  # branch delay slot
.text:004006DC            li      $v1, 1
.text:004006E0            lw      $gp, 0x28+var_18($sp)
.text:004006E4            beq     $v0, $v1, loc_40070C
.text:004006E8            or      $at, $zero       # branch delay slot, NOP
.text:004006EC            la      $t9, puts
.text:004006F0            lui     $a0, 0x40
.text:004006F4            jalr    $t9 ; puts
.text:004006F8            la      $a0, aWhatYouEntered  # "What you entered?
    Huh?"
.text:004006FC            lw      $ra, 0x28+var_4($sp)
.text:00400700            move    $v0, $zero
.text:00400704            jr      $ra
.text:00400708            addiu   $sp, 0x28

.text:0040070C loc_40070C:
.text:0040070C            la      $t9, printf
.text:00400710            lw      $a1, 0x28+var_10($sp)
.text:00400714            lui     $a0, 0x40
.text:00400718            jalr    $t9 ; printf
.text:0040071C            la      $a0, aYouEnteredD___  # "You entered
    %d...\n"
.text:00400720            lw      $ra, 0x28+var_4($sp)
.text:00400724            move    $v0, $zero
.text:00400728            jr      $ra
.text:0040072C            addiu   $sp, 0x28
```

scanf() restituisce il risultato del suo lavoro nel registro $V0. Ciò viene controllato all'indirizzo 0x004006E4 confrontando il valore in $V0 con quello in $V1 (1 era stato memorizzato in $V1 precedentemente, a 0x004006DC). BEQ sta per «Branch Equal». Se i due valori sono uguali (cioè scanf() è terminata con successo), l'esecuzione salta all'indirizzo 0x0040070C.

Esercizio

Come possiamo vedere, le istruzioni JNE/JNZ possono essere scambiate con JE/JZ e viceversa. (lo stesso vale per BNE e BEQ). Ma se ciò avviene i blocchi base devono anch'essi essere scambiati. Provate a farlo in qualche esempio.

1.12.5 Esercizio

- http://challenges.re/53

1.13 Degno di nota: variabili globali vs locali

Ora sappiamo che all'inizio le variabili globali vengono riempite di zeri dall' OS (1.12.3 on page 103, [*ISO/IEC 9899:TC3 (C C99 standard)*, (2007)6.7.8p10]), ma ciò non avviene per le variabili locali (1.9.4 on page 51).

A volte, ci dimentichiamo di inizializzare una variabile globale e il nostro programma si basa sul fatto che avrà degli zeri all'inizio. Se in seguito spostiamo la variabile globale in una funzione rendendola locale, non sarà più azzerata all'inizio e potremmo avere dei bug come risultato.

1.14 Accesso agli argomenti

Abbiamo visto che la funzione chiamante (chiamante) passa gli argomenti alla funzione chiamata (chiamata) tramite lo stack. In che modo la funzione chiamata accede agli argomenti?

Listing 1.90: semplice esempio

```
#include <stdio.h>

int f (int a, int b, int c)
{
        return a*b+c;
};

int main()
{
        printf ("%d\n", f(1, 2, 3));
        return 0;
};
```

1.14.1 x86

MSVC

Ecco il risultato della compilazione ocn MSVC 2010 Express:

Listing 1.91: MSVC 2010 Express

```
_TEXT   SEGMENT
_a$ = 8         ; dimensione = 4
_b$ = 12        ; dimensione = 4
_c$ = 16        ; dimensione = 4
_f      PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        imul    eax, DWORD PTR _b$[ebp]
        add     eax, DWORD PTR _c$[ebp]
        pop     ebp
        ret     0
```

```
_f      ENDP

_main   PROC
        push    ebp
        mov     ebp, esp
        push    3 ; 3° argomento
        push    2 ; 2° argomento
        push    1 ; 1° argomento
        call    _f
        add     esp, 12
        push    eax
        push    OFFSET $SG2463 ; '%d', 0aH, 00H
        call    _printf
        add     esp, 8
        ; ritorna 0
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

Vediamo che la funzione `main()` fa il push di 3 numeri sullo stack e chiama `f(int,int,int)`.

L'accesso agli argomenti all'interno della funzione `f()` è gestito con l'aiuto di macro come: `_a$ = 8`, allo stesso modo delle variabili locali, ma con offset positivi. Si sta quindi indirizzando il lato *esterno* dello stack frame sommando la macro `_a$` al valore contenuto nel registro EBP.

Successivamente il valore di a è memorizzato in EAX. A seguito dell'esecuzione dell'istruzione `IMUL`, il valore in EAX è il prodotto del valore in EAX e del contenuto di `_b`.

Infine, `ADD` aggiunge il valore in `_c` a EAX.

Il valore EAX non necessita di essere spostato: si trova già nel posto giusto. Al termine, la funzione chiamante (chiamante) prende il valore di EAX e lo usa come argomento di `printf()`.

MSVC + OllyDbg

Illustriamo il funzionamento con OllyDbg. Quando raggiungiamo la prima istruzione in `f()` che usa uno degli argomenti (il primo) notiamo che EBP punta allo stack frame, indentificato dal riquadro rosso.

Il primo elemento dello stack frame è il valore salvato di EBP, il secondo è il RA, il terzo rappresenta il primo argomento della funzione, seguito dal secondo e terzo argomento.

Per accedere al primo argomento della funzione bisogna aggiungere esattamente 8 (2 wor a 32-bit) a EBP.

OllyDbg è in grado di distinguere gli argomenti in questo modo, ed ha aggiunto dei commenti agli elementi dello stack, ad esempio:

«RETURN from» and «Arg1 = ...», etc.

N.B.: Gli argomenti della funzione non sono membri dello stack frame della funzione chiamata, appartengono allo stack frame della funzione chiamante (chiamante).

Pertanto OllyDbg ha contrassegnato gli elementi «Arg» come membri di un altro stack frame.

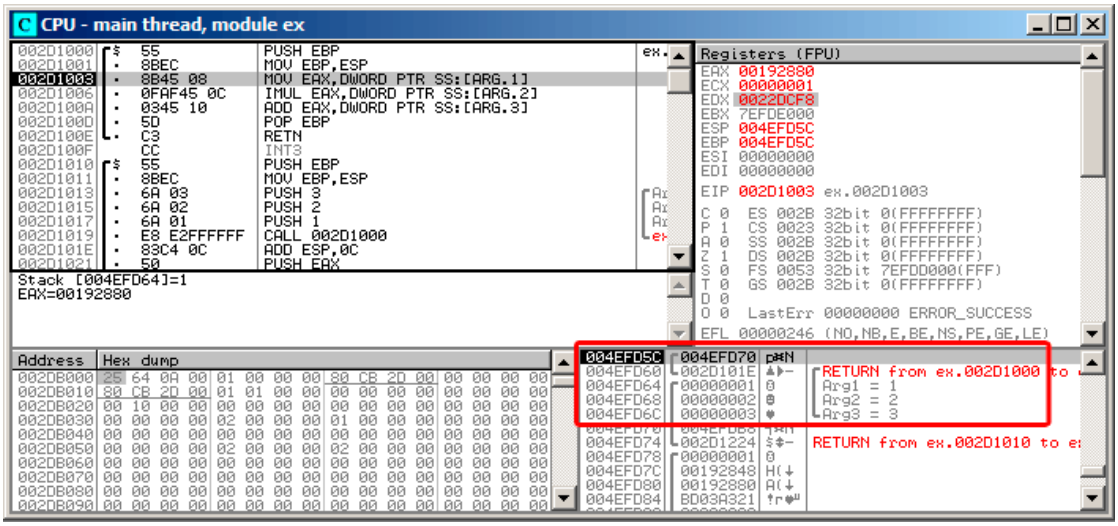


Figura 1.23: OllyDbg: dentro la funzione f()

GCC

Compiliamo lo stesso esempio con GCC 4.4.1 ed osserviamo il risultato con IDA:

Listing 1.92: GCC 4.4.1

```
        public f
f       proc near

arg_0   = dword ptr  8
arg_4   = dword ptr  0Ch
arg_8   = dword ptr  10h

        push    ebp
        mov     ebp, esp
        mov     eax, [ebp+arg_0] ; 1° argomento
        imul    eax, [ebp+arg_4] ; 2° argomento
        add     eax, [ebp+arg_8] ; 3° argomento
        pop     ebp
        retn
f       endp

        public main
main    proc near

var_10  = dword ptr -10h
```

```
var_C   = dword ptr -0Ch
var_8   = dword ptr -8

        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFFF0h
        sub     esp, 10h
        mov     [esp+10h+var_8], 3 ; 3° argomento
        mov     [esp+10h+var_C], 2 ; 2° argomento
        mov     [esp+10h+var_10], 1 ; 1° argomento
        call    f
        mov     edx, offset aD  ; "%d\n"
        mov     [esp+10h+var_C], eax
        mov     [esp+10h+var_10], edx
        call    _printf
        mov     eax, 0
        leave
        retn
main    endp
```

Il risultato è pressocheè identico, a meno di piccole differenze già discusse in precedenza.

Lo stack pointer non viene ripristinato dopo le due chiamate a funzione(f and printf), poichè se ne occupa la penultima istruzione LEAVE (**??** on page ??) alla fine della funzione.

1.14.2 x64

La situazione è leggemente diversa in x86-64. Gli argomenti della funzione (i primi 4 o 6) sono passati tramite i registri. La funzione chiamata (chiamata) legge quindi i parametri dai registri anzichè dallo stack.

MSVC

Con ottimizzazione MSVC:

Listing 1.93: Con ottimizzazione MSVC 2012 x64

```
$SG2997 DB      '%d', 0aH, 00H

main    PROC
        sub     rsp, 40
        mov     edx, 2
        lea     r8d, QWORD PTR [rdx+1] ; R8D=3
        lea     ecx, QWORD PTR [rdx-1] ; ECX=1
        call    f
        lea     rcx, OFFSET FLAT:$SG2997 ; '%d'
        mov     edx, eax
        call    printf
        xor     eax, eax
        add     rsp, 40
        ret     0
```

```
main    ENDP

f       PROC
        ; ECX - 1° argomento
        ; EDX - 2° argomento
        ; R8D - 3° argomento
        imul    ecx, edx
        lea     eax, DWORD PTR [r8+rcx]
        ret     0
f       ENDP
```

Come possiamo vedere, la piccola funzione `f()` prende tutti i suoi argomenti dai registri.

L'istruzione LEA qui è usata per l'addizione. Apparentemente il compilatore l'ha ritenuta più veloce di `ADD`.

LEA è anche usata nella funzione `main()` per preparare il primo e il tezo argomento di `f()`. Il compilatore deve aver deciso che questo approccio è più veloce del modo tradizionale di caricare valori nei registri usando l'istruzione `MOV`.

Diamo un'occhiata all'output di MSVC senza ottimizzazioni:

Listing 1.94: MSVC 2012 x64

```
f               proc near

; shadow space:
arg_0           = dword ptr  8
arg_8           = dword ptr  10h
arg_10          = dword ptr  18h

                ; ECX - 1° argomento
                ; EDX - 2° argomento
                ; R8D - 3° argomento
                mov     [rsp+arg_10], r8d
                mov     [rsp+arg_8], edx
                mov     [rsp+arg_0], ecx
                mov     eax, [rsp+arg_0]
                imul    eax, [rsp+arg_8]
                add     eax, [rsp+arg_10]
                retn
f               endp

main            proc near
                sub     rsp, 28h
                mov     r8d, 3 ; 3° argomento
                mov     edx, 2 ; 2° argomento
                mov     ecx, 1 ; 1° argomento
                call    f
                mov     edx, eax
                lea     rcx, $SG2931    ; "%d\n"
                call    printf

                ; ritorna 0
```

```
                xor     eax, eax
                add     rsp, 28h
                retn
main            endp
```

L'output può lasciarci un po' perplessi in quanto tutti i 3 argomenti nei registri sono anche salvati nello stack per qualche motivo. Ciò è chiamato «shadow space» [83]: ogni Win64 potrebbe (ma non deve necessariamente farlo) salvare tutti i 4 valori dei registri in questo spazio. E questo avviene per due ragioni: 1) è eccessivo allocare un intero registro (o addirittura 4) per un argomento in input, pertanto sarà acceduto tramite lo stack. 2) il debugger sa sempre dove trovare gli argomenti della funzione ad un break [84].

Quindi, alcune funzioni piuttosto estese potrebbero salvare i loro argomenti nello «shadow space» nel caso in cui abbiano necessità di utilizzarli durante l'esecuzione della funzione. Altre funzioni più piccole (come la nostra) potrebbero non farlo.

Allocare spazio nello «shadow space» è responsabilità del chiamante (chiamante).

GCC

Con ottimizzazione GCC genera codice più o meno comprensibile:

Listing 1.95: Con ottimizzazione GCC 4.4.6 x64

```
f:
        ; EDI - 1° argomento
        ; ESI - 2° argomento
        ; EDX - 3° argomento
        imul    esi, edi
        lea     eax, [rdx+rsi]
        ret

main:
        sub     rsp, 8
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    f
        mov     edi, OFFSET FLAT:.LC0 ; "%d\n"
        mov     esi, eax
        xor     eax, eax  ; numero dei registri vettore passati
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

Senza ottimizzazione GCC:

Listing 1.96: GCC 4.4.6 x64

[83] MSDN
[84] MSDN

```
f:
        ; EDI - 1° argomento
        ; ESI - 2° argomento
        ; EDX - 3° argomento
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        mov     DWORD PTR [rbp-12], edx
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, DWORD PTR [rbp-8]
        add     eax, DWORD PTR [rbp-12]
        leave
        ret
main:
        push    rbp
        mov     rbp, rsp
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    f
        mov     edx, eax
        mov     eax, OFFSET FLAT:.LC0 ; "%d\n"
        mov     esi, edx
        mov     rdi, rax
        mov     eax, 0  ; numero dei registri vettore passati
        call    printf
        mov     eax, 0
        leave
        ret
```

In System V *NIX ([Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] [85]) non è richiesto lo «shadow space», ma la funizone chiamata (chiamata) potrebbe aver bisogno di salvare i suoi argomenti da qualche parte in caso di scarsità di registri a disposizione.

GCC: uint64_t al posto di int

Il nostro esempio utilizza *int* a 32-bit, motivo per cui viene usata la parte a 32-bit del registro (con prefisso E-).

Può essere leggermente modificato per utilizzare valori a 64-bit:

```
#include <stdio.h>
#include <stdint.h>

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
{
        return a*b+c;
```

[85] Italian text placeholder https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

```
};

int main()
{
        printf ("%lld\n", f(0x1122334455667788,
                            0x1111111122222222,
                            0x3333333344444444));
        return 0;
};
```

Listing 1.97: Con ottimizzazione GCC 4.4.6 x64

```
f       proc near
        imul    rsi, rdi
        lea     rax, [rdx+rsi]
        retn
f       endp

main    proc near
        sub     rsp, 8
        mov     rdx, 3333333344444444h ; 3° argomento
        mov     rsi, 1111111122222222h ; 2° argomento
        mov     rdi, 1122334455667788h ; 1° argomento
        call    f
        mov     edi, offset format ; "%lld\n"
        mov     rsi, rax
        xor     eax, eax ; numero dei registri vettore passati
        call    _printf
        xor     eax, eax
        add     rsp, 8
        retn
main    endp
```

Il codice è lo stesso, ma in questo caso vengono usati i registri *completi* (con prefisso R-).

1.14.3 ARM

Senza ottimizzazione Keil 6/2013 (Modalità ARM)

```
.text:000000A4 00 30 A0 E1      MOV     R3, R0
.text:000000A8 93 21 20 E0      MLA     R0, R3, R1, R2
.text:000000AC 1E FF 2F E1      BX      LR
...
.text:000000B0              main
.text:000000B0 10 40 2D E9      STMFD   SP!, {R4,LR}
.text:000000B4 03 20 A0 E3      MOV     R2, #3
.text:000000B8 02 10 A0 E3      MOV     R1, #2
.text:000000BC 01 00 A0 E3      MOV     R0, #1
.text:000000C0 F7 FF FF EB      BL      f
.text:000000C4 00 40 A0 E1      MOV     R4, R0
.text:000000C8 04 10 A0 E1      MOV     R1, R4
```

```
.text:000000CC 5A 0F 8F E2      ADR     R0, aD_0        ; "%d\n"
.text:000000D0 E3 18 00 EB      BL      __2printf
.text:000000D4 00 00 A0 E3      MOV     R0, #0
.text:000000D8 10 80 BD E8      LDMFD   SP!, {R4,PC}
```

La funzione `main()` chiama altre due funzioni, con tre valori passati alla prima — (`f()`).

Come detto in precedenza, in ARM i primi 4 valori sono solitamente passati nei primi 4 registri (R0-R3).

La funzione `f()`, come si può osservare, usa i primi 3 registri (R0-R2) come argomenti.

L'istruzione `MLA` (*Multiply Accumulate*) moltiplica i suoi primi due operandi (R3 e R1), aggiunge al prodotto il terzo operando (R2) e salva il risultato nel zeresimo registro (R0), attraverso il quale, da standard, le funzioni restituiscono i valori.

La moltiplicazione e addizione fatte in una volta sola (*Fused multiply–add*) è un'operazione molto utile. Non vi era alcune funzione analoga in x86 prima dell'avvento delle istruzioni FMA in SIMD. [86].

La prima istruzione `MOV R3, R0`, è apparentemente ridondante (al suo posto sarebbe potuta essere usata una singola istruzione `MLA`). Il compilatore, come previsto, non ha quindi ottimizzato il codice.

L'istruzione BX restituisce il controllo all'indirizzo memorizzato nel registro LR e, se necessario, effettua lo switch della modalità del processore da Thumb a ARM o viceversa. Ciò può essere necessario in quanto, come possiamo vedere ,la funzione `f()` non è al corrente di che tipo di codice potrebbe esere chiamato in seguito (ARM o Thumb). Dunque, se viene chiamata da codice Thumb BX non resituisce soltanto il controllo alla funzione chiamante ma cambia anche la modalità del processore a Thumb. Se la funzione viene chiamata da codice ARM, non effettua lo scambio di modalità [*ARM(R) Architecture Reference Manual, ARMv7-A and ARMv7-R edition*, (2012)A2.3.2].

Con ottimizzazione Keil 6/2013 (Modalità ARM)

```
.text:00000098             f
.text:00000098 91 20 20 E0        MLA     R0, R1, R0, R2
.text:0000009C 1E FF 2F E1        BX      LR
```

Ecco la funzione `f()` compilata dal compilatore Keil con ottimizzazione completa (`-O3`).

L'istruzione `MOV` è stata ottimizzata (o ridotta), ora `MLA` usa tutti i registri di input e mete il risultato in R0, esattamente da dove la funzione chiamante leggerà il risultato.

Con ottimizzazione Keil 6/2013 (Modalità Thumb)

[86] wikipedia

```
.text:0000005E 48 43          MULS    R0, R1
.text:00000060 80 18          ADDS    R0, R0, R2
.text:00000062 70 47          BX      LR
```

L'istruzione MLA non è disponibile in modalità Thumb, pertanto il compilatore genera il codice effettuando le due operazioni (moltiplicazione e addizione) separatamente.

Per prima cosa l'istruzione MULS moltiplica R0 per R1, mettendo il risultato in R0. Successivamente la seconda istruzione (ADDS) somma al risultato precendete R2, e mette il risultato nel registro R0.

ARM64

Con ottimizzazione GCC (Linaro) 4.9

Appare tutto molto semplice. MADD è semplicemente un'istruzione che fa una moltiplicazione/addizione combinata (simile alla MLA vista in precedenza). Tutti i 3 argomenti sono passati tramite le parti a 32-bit dei registri X-. Infatti gli argomenti sono tutti di tipo *int* a 32-bit. Il risultato è restituito in W0.

Listing 1.98: Con ottimizzazione GCC (Linaro) 4.9

```
f:
        madd    w0, w0, w1, w2
        ret

main:
; salva il FP e il LR nello stack frame:
        stp     x29, x30, [sp, -16]!
        mov     w2, 3
        mov     w1, 2
        add     x29, sp, 0
        mov     w0, 1
        bl      f
        mov     w1, w0
        adrp    x0, .LC7
        add     x0, x0, :lo12:.LC7
        bl      printf
; ritorna 0
        mov     w0, 0
; ripristina FP and LR
        ldp     x29, x30, [sp], 16
        ret

.LC7:
        .string "%d\n"
```

Estendiamo anche questo esempio usando il tipo uint64_t a 64-bit e vediamo che succede:

```
#include <stdio.h>
#include <stdint.h>

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
{
        return a*b+c;
};

int main()
{
        printf ("%lld\n", f(0x1122334455667788,
                            0x1111111122222222,
                            0x3333333344444444));
        return 0;
};
```

```
f:
        madd    x0, x0, x1, x2
        ret
main:
        mov     x1, 13396
        adrp    x0, .LC8
        stp     x29, x30, [sp, -16]!
        movk    x1, 0x27d0, lsl 16
        add     x0, x0, :lo12:.LC8
        movk    x1, 0x122, lsl 32
        add     x29, sp, 0
        movk    x1, 0x58be, lsl 48
        bl      printf
        mov     w0, 0
        ldp     x29, x30, [sp], 16
        ret

.LC8:
        .string "%lld\n"
```

La funzione `f()` è rimasta invariata, ma adesso i registri a 64-bit X- sono utilizzati nella loro interezza. I valori grandi a 64-bit sono caricati nei registri per parti, come descritto anche qui: **??** on page ??.

Senza ottimizzazione GCC (Linaro) 4.9

L'output del compilatore non ottimizzante è più ridondante:

```
f:
        sub     sp, sp, #16
        str     w0, [sp,12]
        str     w1, [sp,8]
        str     w2, [sp,4]
        ldr     w1, [sp,12]
        ldr     w0, [sp,8]
```

```
        mul     w1, w1, w0
        ldr     w0, [sp,4]
        add     w0, w1, w0
        add     sp, sp, 16
        ret
```

Il codice salva gli argomenti in input nello stack locale, nel caso in cui qualcuno (o qualcosa) in questa funzione abbia necessità di usare i registri `W0...W2` Questo previene l'eventualità che gli argomenti originali siano sovrascritti, nel caso in cui servano nuovamente nel corso della funzione.

Ciò è detto *Register Save Area.* [*Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*, (2013)][87]. ed è vagamente simile allo «Shadow Space»: 1.14.2 on page 133. La funzione chiamata non è comunque obbligata a salvarli.

Perchè GCC 4.9 ottimizzante ha eliminato questa porzione di codice che salva gli argomenti? Lo ha fatto perchè, a seguito di un' ulteriore ottimizzazione, ha concluso che gli argomenti di questa funzione non sono riutilizzati in futuro e che i registri `W0...W2` non saranno utilizzati.

Notiamo anche una coppia di istruzioni `MUL/ADD` al posto della singola `MADD`.

1.14.4 MIPS

Listing 1.99: Con ottimizzazione GCC 4.4.5

```
.text:00000000 f:
; $a0=a
; $a1=b
; $a2=c
.text:00000000        mult    $a1, $a0
.text:00000004        mflo    $v0
.text:00000008        jr      $ra
.text:0000000C        addu    $v0, $a2, $v0     ; branch delay slot
; il risultato è in $v0 al ritorno
.text:00000010 main:
.text:00000010
.text:00000010 var_10 = -0x10
.text:00000010 var_4  = -4
.text:00000010
.text:00000010        lui     $gp, (__gnu_local_gp >> 16)
.text:00000014        addiu   $sp, -0x20
.text:00000018        la      $gp, (__gnu_local_gp & 0xFFFF)
.text:0000001C        sw      $ra, 0x20+var_4($sp)
.text:00000020        sw      $gp, 0x20+var_10($sp)
; imposta c:
.text:00000024        li      $a2, 3
; imposta a:
.text:00000028        li      $a0, 1
.text:0000002C        jal     f
; imposta b:
```

[87] Italian text placeholderhttp://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf

```
.text:00000030        li      $a1, 2            ; branch delay slot
; ora il risultato è in $v0
.text:00000034        lw      $gp, 0x20+var_10($sp)
.text:00000038        lui     $a0, ($LC0 >> 16)
.text:0000003C        lw      $t9, (printf & 0xFFFF)($gp)
.text:00000040        la      $a0, ($LC0 & 0xFFFF)
.text:00000044        jalr    $t9
; prendi il risultato della funzione f() e passalo
; come secondo argomento alla printf():
.text:00000048        move    $a1, $v0          ; branch delay slot
.text:0000004C        lw      $ra, 0x20+var_4($sp)
.text:00000050        move    $v0, $zero
.text:00000054        jr      $ra
.text:00000058        addiu   $sp, 0x20 ; branch delay slot
```

I primi quattro argomenti della funzione sono passati in quattro registri con prefisso A-.

Ci sono due registri speciali in MIPS: HI e LO che durante l'esecuzione dell' istruzione `MULT`, vengono riempiti con il risultato su 64-bit della moltiplicazione.

Questi registri sono accessibili solamente usando le istruzioni `MFLO` e `MFHI`. In questo caso `MFLO` prende la parte bassa del risultato della moltiplicazione e la salva in $V0. Di conseguenza i 32 bit della parte alta del risultato della moltiplicazione sono scartati (il contenuto del registro HI non viene usato). Infatti: noi qui lavoriamo con tipi di dati *int* a 32 bit.

Infine, `ADDU` («Add Unsigned») sommano il valore del terzo argomento al risultato.

Ci sono due tipi di istruzione addizione in MIPS: `ADD` and `ADDU`. La differnza non è legata al segno, ma alle eccezioni. `ADD` può sollevare un' eccezione in caso di overflow, che di solito è utile[88] e supportato in Ada PL, per esempio. `ADDU` non solleva eccezioni in caso di overlflow.

Siccome C/C++ non lo supporta, nei nostri esempi vediamo `ADDU` anzichè `ADD`.

Il risultato a 32 bit viene lasciato in $V0.

C'è una nuova istruzione per noi nel `main()`: `JAL` («Jump and Link»).

La differenza tra `JAL` e `JALR` è che l'offset relativo viene codificato nella prima istruzione, mentre `JALR` salta all'indirizzo assoluto salvato in un registro («Jump and Link Register»).

Entrambe le funzioni `f()` e `main()` si trovano nello stesso file oggetto, quindi l'indirizzo relativo di `f()` è conosciuto e fissato.

[88] http://blog.regehr.org/archives/1154

1.15 Ulteriori considerazioni sulla restituzione dei risultati

In x86, il risultato dell'esecuzione dei una funzione è generalmente restituito [89] nel registro EAX. Se il tipo del risultato è un byte o un *char*, viene utilizzata la parte bassa del registro EAX (AL). Se una funzione restituisce un numero di tipo *float*, viene invece utilizzato il registro FPU ST(0). In ARM, il risultato è solitamente restituito nel registro R0.

1.15.1 Tentativo di utilizzare il risultato di una funzione che resituisce *void*

Che succederebbe se la funzione main dichiarasse il valore di ritorno di tipo *void* invece di *int*? Il cosiddetto startup-code chiama `main()` più o meno così:

```
push envp
push argv
push argc
call main
push eax
call exit
```

In altre parole:

```
exit(main(argc,argv,envp));
```

Se dichiariamo `main()` come *void*, non viene esplicitamente restituito nulla (usando lo statement *return*), e quindi qualche valore casuale, che si trova memorizzato nel registro EAX alla fine di `main()`, diventa argomento della funzione exit(). Molto probabilmente si tratterà di un valore casuale, residuo dell'esecuzione della nostra funzione, quindi l'exit code del programma è pseudo-casuale.

Illustriamo meglio questo fatto. La funzione `main()` ha ora un valore di ritorno di tipo *void*:

```
#include <stdio.h>

void main()
{
        printf ("Hello, world!\n");
};
```

Compiliamo il programma su Linux.

GCC 4.8.1 ha sostituito `printf()` con `puts()` (abbiamo già visto questo caso: 1.5.3 on page 28), e va del tutto bene, poichè `puts()` restituisce il numero di caratteri stampati proprio come `printf()`. Notiamo che EAX non viene azzerato prima della fine di `main()`.

[89] Vedi anche: MSDN: Return Values (C++): MSDN

Ciò implica che il valore di EAX alla fine di `main()` conterrà il valore lasciato lì da `puts()`.

Listing 1.100: GCC 4.8.1

```
.LC0:
        .string "Hello, world!"
main:
        push    ebp
        mov     ebp, esp
        and     esp, -16
        sub     esp, 16
        mov     DWORD PTR [esp], OFFSET FLAT:.LC0
        call    puts
        leave
        ret
```

Scriviamo uno script bash che mostra l'exit status:

Listing 1.101: tst.sh

```
#!/bin/sh
./hello_world
echo $?
```

Eseguiamolo:

```
$ tst.sh
Hello, world!
14
```

14 è il numero di caratteri stampati. Il numero dei caratteri stampati *scivola* da `printf()` attraverso EAX/RAX nell' «exit code».

Un altro esempio nel libro: **??** on page ??.

Comunque, quando decompiliamo C++ in Hex-Rays, spesso possiamo trovare una funzione che termina con il distruttore di qualche classe:

```
...

call    ??1CString@@QAE@XZ ; CString:: CString(void)
mov     ecx, [esp+30h+var_C]
pop     edi
pop     ebx
mov     large fs:0, ecx
add     esp, 28h
retn
```

Dallo standard C++, i distruttori non ritornano nulla, ma quando Hex-Rays non lo capisce e pensa che sia il distruttore che la funzione ritornino *int*, possiamo avere un output simile a questo:

```
...
```

```
        return CString::~CString(&Str);
}
```

1.15.2 Che succede se il risultato della funzione non viene usato?

`printf()` restituisce il numero di caratteri mandati in output con successo, ma il risultato di questa funzione è usato molto raramente.

E' possibile anche chiamare una funzione la cui essenza risiede nel restituire un valore e non usarlo del tutto:

```
int f()
{
    // salta i primi 3 valori casuali:
    rand();
    rand();
    rand();
    // e usa il 4°:
    return rand();
};
```

Il risultato della funzione rand() è lasciato in EAX in tutti e quattro i casi. Nei primi 3 però il valore in EAX non viene usato.

1.15.3 Restituire una struttura

Torniamo al fatto che il valore di ritorno è lasciato nel registro EAX. Questo è il motivo per cui i vecchi compilatori C non possono creare funzioni in grado di restituire qualcosa che non entri perfettamente in un registro (solitamente un *int*). Se lo si vuole fare, è necessario restituire l'informazione attraverso puntatori passati come argomenti alla funzione.

Quindi, generalmente, se una funzione deve restituire più valori, ne restituisce (realmente) soltanto uno, ed il resto—tramite puntatori.

Oggi è possibile restituire anche un'intera struttura, ma non è ancora una pratica molto diffusa. Se una funzione deve restituire una struttura grande, il chiamante (chiamante) deve allocarla e passare come primo argomento della funzione un puntatore alla struttura, il tutto in modo trasparente per il programmatore. E' pressochè la stessa cosa di passare un puntatore manualmente come primo argomento, ma il compilatore "nasconde" questo passaggio.

Un piccolo esempio:

```
struct s
{
    int a;
    int b;
    int c;
};
```

```
struct s get_some_values (int a)
{
    struct s rt;

    rt.a=a+1;
    rt.b=a+2;
    rt.c=a+3;

    return rt;
};
```

...otteniamo (MSVC 2010 /0x):

```
$T3853 = 8                      ; size = 4
_a$ = 12                        ; size = 4
?get_some_values@@YA?AUs@@H@Z PROC              ; get_some_values
    mov    ecx, DWORD PTR _a$[esp-4]
    mov    eax, DWORD PTR $T3853[esp-4]
    lea    edx, DWORD PTR [ecx+1]
    mov    DWORD PTR [eax], edx
    lea    edx, DWORD PTR [ecx+2]
    add    ecx, 3
    mov    DWORD PTR [eax+4], edx
    mov    DWORD PTR [eax+8], ecx
    ret    0
?get_some_values@@YA?AUs@@H@Z ENDP              ; get_some_values
```

Il nome della macro per il passaggio interno del puntatore alla struttura è in questo caso $T3853.

Questo stesso esempio può essere riscritto utilizzando l'estensione del linguaggio C99:

```
struct s
{
    int a;
    int b;
    int c;
};

struct s get_some_values (int a)
{
    return (struct s){.a=a+1, .b=a+2, .c=a+3};
};
```

Listing 1.102: GCC 4.8.1

```
_get_some_values proc near

ptr_to_struct   = dword ptr  4
a               = dword ptr  8

                mov     edx, [esp+a]
                mov     eax, [esp+ptr_to_struct]
```

```
                lea     ecx, [edx+1]
                mov     [eax], ecx
                lea     ecx, [edx+2]
                add     edx, 3
                mov     [eax+4], ecx
                mov     [eax+8], edx
                retn
_get_some_values endp
```

Come possiamo vedere, la funzione chiamata non fa altro che riempire i campi della struttura allocata dalla funzione chiamante, come se un puntatore alla struttura fosse stato passato. Pertanto non ci sono neanche impatti negativi sulla performance.

1.16 Puntatori

1.16.1 Ritornare valori

I puntatori sono spesso usati per restituire valori dalle funzioni (come nel caso di `scanf()` (1.12 on page 89)). Ad esempio, quando una funzione deve restituire due valori.

Esempio variabili globali

```
#include <stdio.h>

void f1 (int x, int y, int *sum, int *product)
{
        *sum=x+y;
        *product=x*y;
};

int sum, product;

void main()
{
        f1(123, 456, &sum, &product);
        printf ("sum=%d, product=%d\n", sum, product);
};
```

Viene compilato in:

Listing 1.103: Con ottimizzazione MSVC 2010 (/Ob0)

```
COMM    _product:DWORD
COMM    _sum:DWORD
$SG2803 DB      'sum=%d, product=%d', 0aH, 00H

_x$ = 8                 ; size = 4
_y$ = 12                ; size = 4
_sum$ = 16              ; size = 4
```

```
_product$ = 20          ; size = 4
_f1     PROC
        mov     ecx, DWORD PTR _y$[esp-4]
        mov     eax, DWORD PTR _x$[esp-4]
        lea     edx, DWORD PTR [eax+ecx]
        imul    eax, ecx
        mov     ecx, DWORD PTR _product$[esp-4]
        push    esi
        mov     esi, DWORD PTR _sum$[esp]
        mov     DWORD PTR [esi], edx
        mov     DWORD PTR [ecx], eax
        pop     esi
        ret     0
_f1     ENDP

_main   PROC
        push    OFFSET _product
        push    OFFSET _sum
        push    456     ; 000001c8H
        push    123     ; 0000007bH
        call    _f1
        mov     eax, DWORD PTR _product
        mov     ecx, DWORD PTR _sum
        push    eax
        push    ecx
        push    OFFSET $SG2803
        call    DWORD PTR __imp__printf
        add     esp, 28
        xor     eax, eax
        ret     0
_main   ENDP
```

Esaminiamolo con OllyDbg:

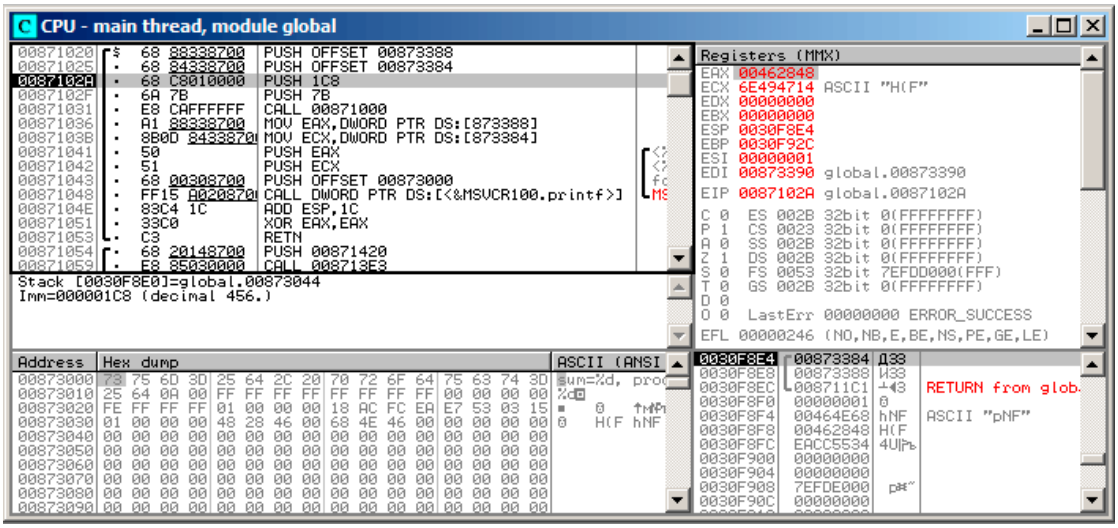


Figura 1.24: OllyDbg: gli indirizzi delle variabili globali sono passate a `f1()`

Prima di tutto, gli indirizzi delle variabili globali vengono passati a `f1()`. Possiamo cliccare su «Follow in dump» sull'elemento dello stack e vedere lo spazio nel data segment allocato per le due variabili.

Queste variabili sono azzerate, poichè i dati non inizializzati (dal segmento BSS) sono azzerati prima dell'inizio dell'esecuzione: [*ISO/IEC 9899:TC3 (C C99 standard)*, (2007)6.7.8p10].

Risiedono nel data segment, e possiamo verificarlo premendo Alt-M ed esaminando la mappa della memoria:

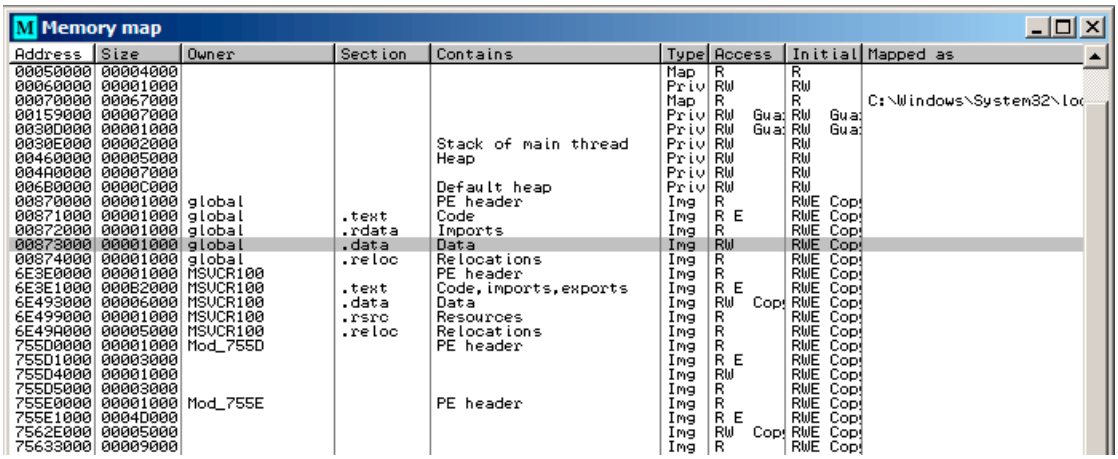


Figura 1.25: OllyDbg: memory map

Eseguiamo (F7) fino all'inizio di `f1()`:

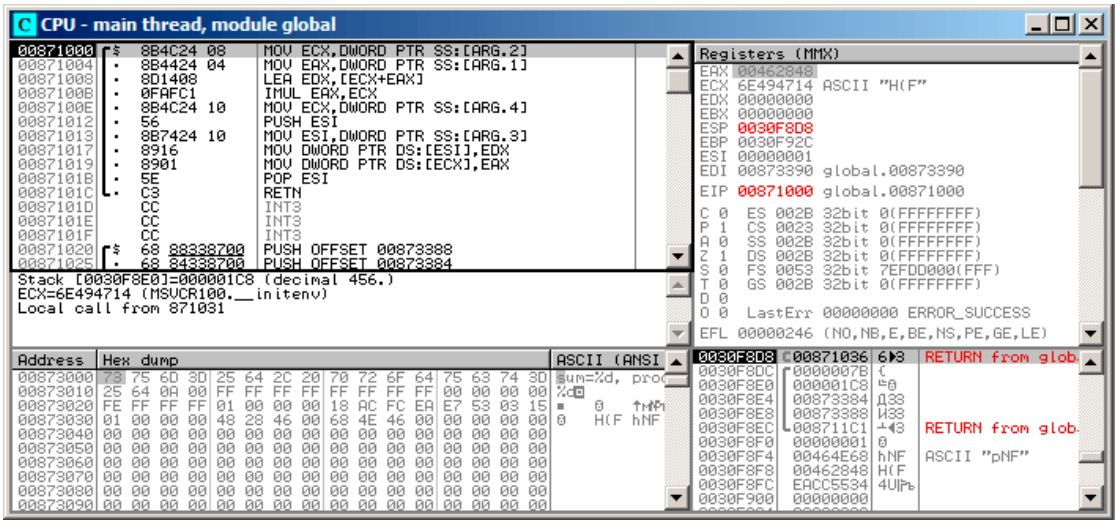


Figura 1.26: OllyDbg: `f1()` inizio

Nello stack sono visibili due avlori, 456 (0x1C8) e 123 (0x7B), oltre agli indirizzi delle due variabili globali.

Eseguiamo fino alla fine di `f1()`. Nella finestra in basso a sinistra vediamo come i risultati dei calcoli appaiono nelle viariabili globali:

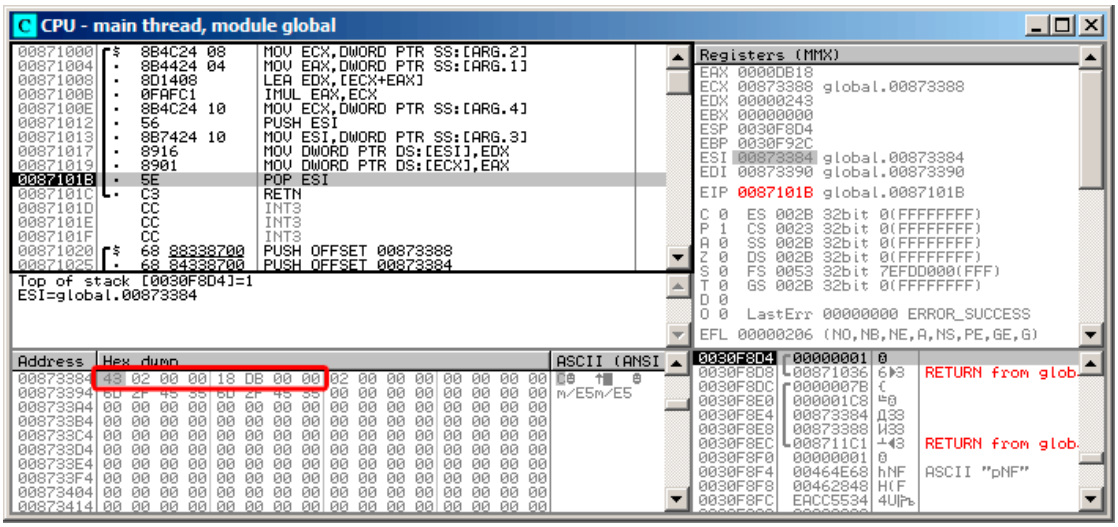


Figura 1.27: OllyDbg: esecuzione di `f1()` completata

Adesso i valori delle variabili globali sono caricati nei registri, pronti per essere passati a `printf()` (tramite lo stack):

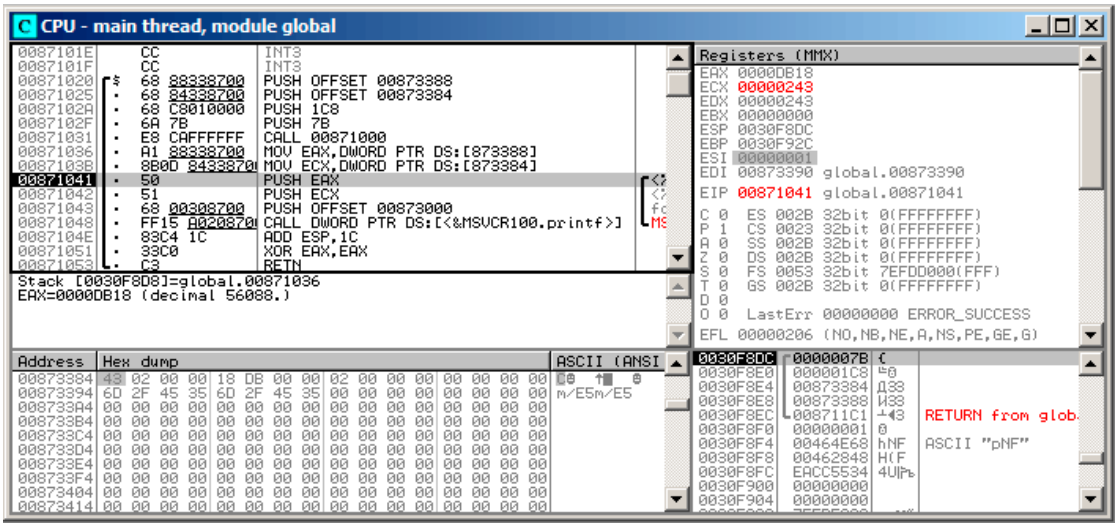


Figura 1.28: OllyDbg: gli indirizzi delle variabili globali sono passati alla `printf()`

Esempio variabili globali

Aggiustiamo leggermente l'esempio:

Listing 1.104: adesso le variabili `sum` e `product` sono locali

```
void main()
{
        int sum, product; // ora le veriabili sono locali nella funzione

        f1(123, 456, &sum, &product);
        printf ("sum=%d, product=%d\n", sum, product);
};
```

Il codice di `f1()` resterà invariato. Solo il `main()` cambierà in:

Listing 1.105: Con ottimizzazione MSVC 2010 (/Ob0)

```
_product$ = -8          ; size = 4
_sum$ = -4              ; size = 4
_main   PROC
; Line 10
        sub     esp, 8
; Line 13
        lea     eax, DWORD PTR _product$[esp+8]
        push    eax
        lea     ecx, DWORD PTR _sum$[esp+12]
        push    ecx
        push    456     ; 000001c8H
```

```
        push    123     ; 0000007bH
        call    _f1
; Line 14
        mov     edx, DWORD PTR _product$[esp+24]
        mov     eax, DWORD PTR _sum$[esp+24]
        push    edx
        push    eax
        push    OFFSET $SG2803
        call    DWORD PTR __imp__printf
; Line 15
        xor     eax, eax
        add     esp, 36
        ret     0
```

Esaminiamo nuovamente con OllyDbg. Gli indirizzi delle variabili locali nello stack sono 0x2EF854 e 0x2EF858. Vediamo come questi vengono messi nello stack:

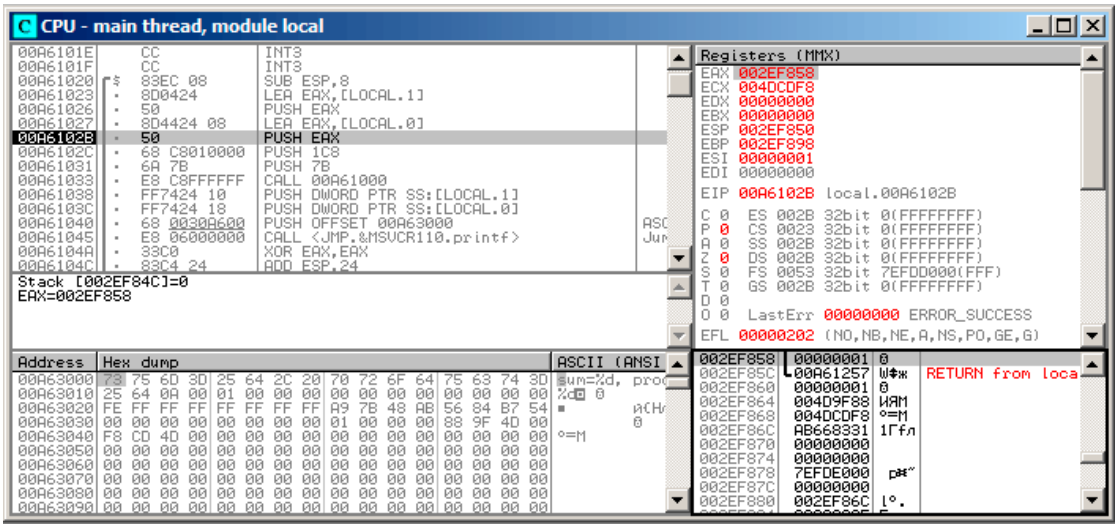


Figura 1.29: OllyDbg: gli indirizzi delle variabili locali sono inserite nello stack

Inizia `f1()`. A questo punto nello stack c'è solo spazzatura casuale a `0x2EF854` e `0x2EF858`:

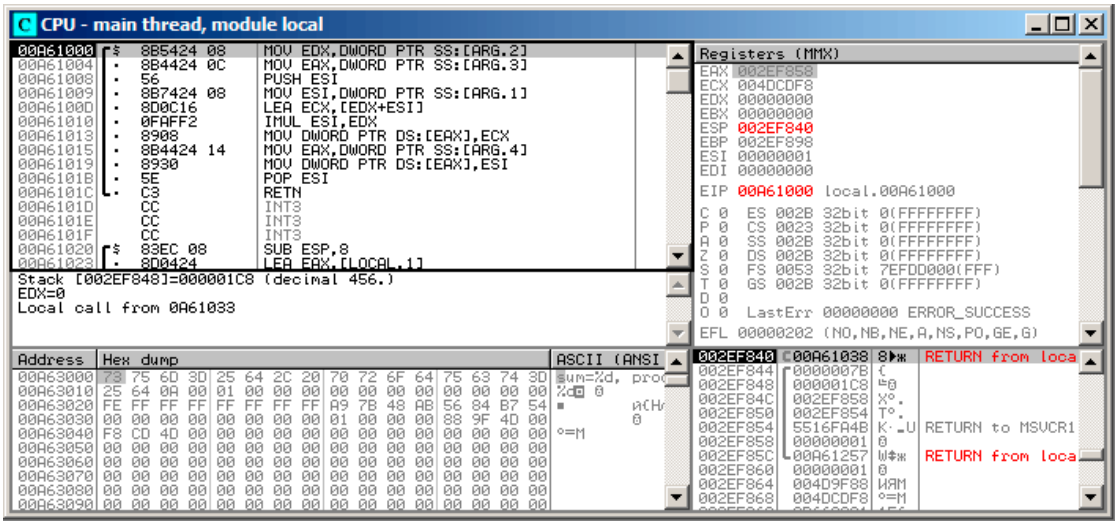


Figura 1.30: OllyDbg: inizia `f1()`

`f1()` finisce:

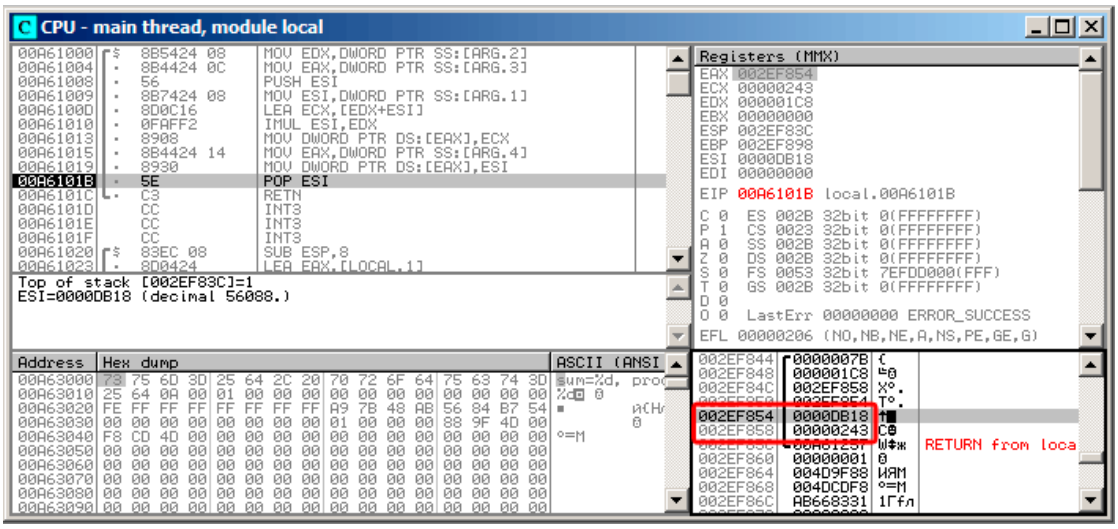


Figura 1.31: OllyDbg: esecuzione completata di `f1()`

Adesso agli indirizzi `0x2EF854` e `0x2EF858` vediamo `0xDB18` e `0x243`. Questi valori sono il risultato di `f1()`.

Conclusione

`f1()` può restituire puntatori ad un qualunque posto in memoria, a prescindere da dove si trovi. In definitiva è questa l'utilità dei puntatori.

A proposito, le *referenze* di C++ funzionano esattamente allo stesso modo. Maggiori dettagli qui: (**??** on page ??).

1.16.2 Valori di input in Swap

Questo è il codice:

```
#include <memory.h>
#include <stdio.h>

void swap_bytes (unsigned char* first, unsigned char* second)
{
        unsigned char tmp1;
        unsigned char tmp2;

        tmp1=*first;
        tmp2=*second;

        *first=tmp2;
        *second=tmp1;
```

```
};

int main()
{
        // copia la stringa nell' heap, così saremo in grado di modificarla
        char *s=strdup("string");

        // scambia il 2° e il 3° carattere
        swap_bytes (s+1, s+2);

        printf ("%s\n", s);
};
```

Come possiamo vedere, i byte sono caricati nelle parti basse a 8 bit di ECX e EBX usando MOVZX (quindi le parti alte di questi registri saranno pulite) e poi i byte saranno riscritti scambiati.

Listing 1.106: Optimizing GCC 5.4

```
swap_bytes:
        push    ebx
        mov     edx, DWORD PTR [esp+8]
        mov     eax, DWORD PTR [esp+12]
        movzx   ecx, BYTE PTR [edx]
        movzx   ebx, BYTE PTR [eax]
        mov     BYTE PTR [edx], bl
        mov     BYTE PTR [eax], cl
        pop     ebx
        ret
```

Gli indirizzi di entrambi i byte sono presi dagli argomenti e attraverso l' esecuzione della funzione sono allocati in EDX and EAX.

Abbiamo usato i puntatori: probabilmente, senza di essi non esiste metodo migliore per questo compito.

1.17 L'operatore GOTO

L'operatore GOTO è generalmente considerato un "anti-pattern", cfr. [Edgar Dijkstra, *Go To Statement Considered Harmful* (1968)[90]]. Ciononostante può essere usato ragionevolmente, [Donald E. Knuth, *Structured Programming with go to Statements* (1974)[91]] [92].

Ecco un esempio molto semplice:

```
#include <stdio.h>

int main()
{
```

[90] http://yurichev.com/mirrors/Dijkstra68.pdf
[91] http://yurichev.com/mirrors/KnuthStructuredProgrammingGoTo.pdf
[92] [Dennis Yurichev, *C/C++ programming language notes*] ha anche alcuni esempi

```
        printf ("begin\n");
        goto exit;
        printf ("skip me!\n");
exit:
        printf ("end\n");
};
```

...e quello che otteniamo con MSVC 2012:

Listing 1.107: MSVC 2012

```
$SG2934 DB      'begin', 0aH, 00H
$SG2936 DB      'skip me!', 0aH, 00H
$SG2937 DB      'end', 0aH, 00H

_main   PROC
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG2934 ; 'begin'
        call    _printf
        add     esp, 4
        jmp     SHORT $exit$3
        push    OFFSET $SG2936 ; 'skip me!'
        call    _printf
        add     esp, 4
$exit$3:
        push    OFFSET $SG2937 ; 'end'
        call    _printf
        add     esp, 4
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

Lo statement *goto* è stato semplicemente sostituito con un'istruzione JMP, che ha lo stesso effetto: un salto non condizionale ad un altro punto del codice. La seconda `printf()` può essere eseguita soltanto con l'intervento umano, utilizzando un debugger o patchando manualmente il codice.

Questo esempio può infatti essere utile come semplice esercizio di patching. Apriamo l'eseguibile con Hiew:

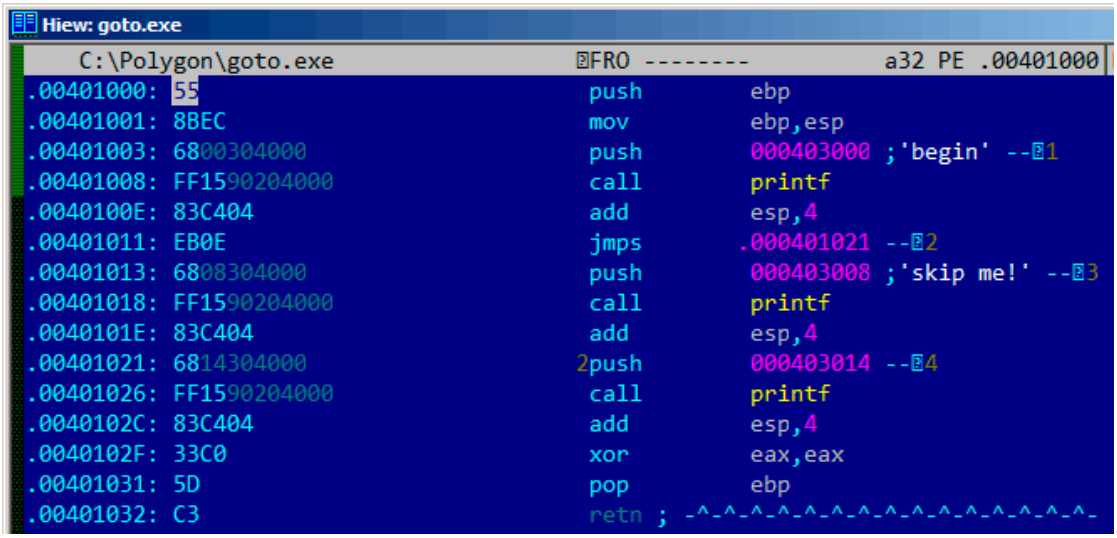


Figura 1.32: Hiew

Posizioniamo il cursore all'indirizzo di JMP (0x410), premiamo F3 (edit), premiamo due volte zero, così da modificare l'opcode in EB 00:

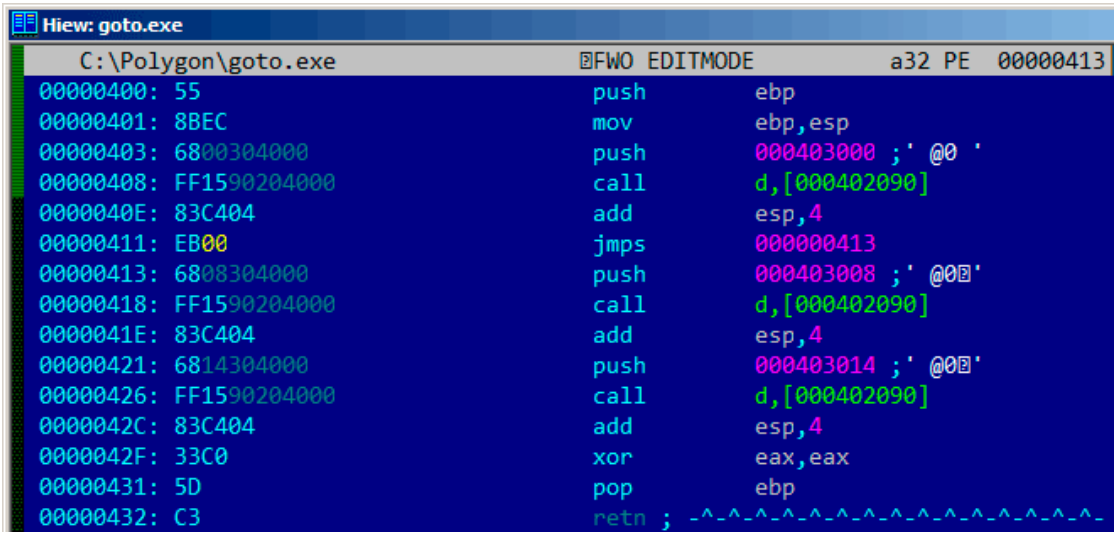


Figura 1.33: Hiew

Il secondo byte dell'opcode di JMP denota l'offset relativo per il salto, 0 significa il punto subito dopo l'istruzione corrente.

Adesso JMP non salterà la seconda chiamata a `printf()`.

Premiamo F9 (save) e usciamo da Hiew. Dopo aver eseguito il programma dovremmo vedere questo:

Listing 1.108: Output dell'eseguibile modificato

```
C:\...>goto.exe

begin
skip me!
end
```

Lo stesso risultato può essere ottenuto sostituendo l'istruzione JMP con 2 istruzioni NOP.

NOP ha opcode 0x90 ed è lunga 1 byte, quindi servono 2 istruzioni per rimpiazzare JMP (che è lunga 2 byte).

1.17.1 Dead code

In termini di compilatore, la seconda chiamata a `printf()` è anche detta «dead code» (codice morto). Sta a significare che quel codice non sarà mai eseguito. Se proviamo a compilare l'esempio con le ottimizzazioni, il compilatore rimuove completamente il «dead code», di cui non resta traccia:

Listing 1.109: Con ottimizzazione MSVC 2012

```
$SG2981 DB      'begin', 0aH, 00H
$SG2983 DB      'skip me!', 0aH, 00H
$SG2984 DB      'end', 0aH, 00H

_main   PROC
        push    OFFSET $SG2981 ; 'begin'
        call    _printf
        push    OFFSET $SG2984 ; 'end'
$exit$4:
        call    _printf
        add     esp, 8
        xor     eax, eax
        ret     0
_main   ENDP
```

Il compilatore si è però dimenticato di rimuovere la stringa «skip me!».

1.17.2 Esercizio

Provate ad ottenere lo stesso risultato utilizzato il vostro compilatore e debugger preferito.

1.18 Jump condizionali

1.18.1 Esempio semplice

```
#include <stdio.h>

void f_signed (int a, int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
};

void f_unsigned (unsigned int a, unsigned int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
};

int main()
```

```
{
    f_signed(1, 2);
    f_unsigned(1, 2);
    return 0;
};
```

x86

x86 + MSVC

La funzione `f_signed()` appare così:

Listing 1.110: Senza ottimizzazione MSVC 2010

```
_a$ = 8
_b$ = 12
_f_signed PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _a$[ebp]
    cmp    eax, DWORD PTR _b$[ebp]
    jle    SHORT $LN3@f_signed
    push   OFFSET $SG737        ; 'a>b'
    call   _printf
    add    esp, 4
$LN3@f_signed:
    mov    ecx, DWORD PTR _a$[ebp]
    cmp    ecx, DWORD PTR _b$[ebp]
    jne    SHORT $LN2@f_signed
    push   OFFSET $SG739        ; 'a==b'
    call   _printf
    add    esp, 4
$LN2@f_signed:
    mov    edx, DWORD PTR _a$[ebp]
    cmp    edx, DWORD PTR _b$[ebp]
    jge    SHORT $LN4@f_signed
    push   OFFSET $SG741        ; 'a<b'
    call   _printf
    add    esp, 4
$LN4@f_signed:
    pop    ebp
    ret    0
_f_signed ENDP
```

La prima istruzione, JLE, sta per *Jump if Less or Equal* (*salta se è minore o uguale*). In altre parole, se il secondo operando è maggiore o uguale al primo, il flusso di controllo sarà pasato all'indirizzo o alla label specificata nell'istruzione. Se questa condizione non è soddisfatta, poiché il secondo operando è più piccolo del primo, il flusso non viene alterato e la prima `printf()` sarà eseguita.

Il secondo controllo è JNE: *Jump if Not Equal*. Il flusso non cambia se i due operandi sono uguali.

Il terzo controllo è JGE: *Jump if Greater or Equal*—salta se il primo operando è maggiore del secondo, o se sono uguali. Quindi, se tutti i tre salti condizionali vengono innescati, nessuna delle chiamate a `printf()` sarà eseguita. Ciò è chiaramente impossibile, almeno senza un intervento speciale. Diamo ora un'occhiata alla funzione `f_unsigned()`. La funzione `f_unsigned()` è uguale a `f_signed()`, con l'eccezione che le istruzioni JBE e JAE sono utilizzate al posto di JLE e JGE:

Listing 1.111: GCC

```
_a$ = 8   ; size = 4
_b$ = 12  ; size = 4
_f_unsigned PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _a$[ebp]
    cmp    eax, DWORD PTR _b$[ebp]
    jbe    SHORT $LN3@f_unsigned
    push   OFFSET $SG2761    ; 'a>b'
    call   _printf
    add    esp, 4
$LN3@f_unsigned:
    mov    ecx, DWORD PTR _a$[ebp]
    cmp    ecx, DWORD PTR _b$[ebp]
    jne    SHORT $LN2@f_unsigned
    push   OFFSET $SG2763    ; 'a==b'
    call   _printf
    add    esp, 4
$LN2@f_unsigned:
    mov    edx, DWORD PTR _a$[ebp]
    cmp    edx, DWORD PTR _b$[ebp]
    jae    SHORT $LN4@f_unsigned
    push   OFFSET $SG2765    ; 'a<b'
    call   _printf
    add    esp, 4
$LN4@f_unsigned:
    pop    ebp
    ret    0
_f_unsigned ENDP
```

Come già detto, le istruzioni di salto (branch instructions) sono diverse: JBE—*Jump if Below or Equal* e JAE—*Jump if Above or Equal*. Queste istruzioni (JA/JAE/JB/JBE) differiscono da JG/JGE/JL/JLE in quanto operano con numeri senza segno (unsigned).

Questo è il motivo per cui se vediamo usare JG/JL al posto di JA/JB, o viceversa, possiamo essere quasi certi che le variabili sono rispettivamente di tipo signed o unsigned. Di seguito è riportata anche la funzione `main()`, dove non c'è niente di nuovo:

Listing 1.112: `main()`

```
_main   PROC
        push    ebp
        mov     ebp, esp
        push    2
```

```
        push    1
        call    _f_signed
        add     esp, 8
        push    2
        push    1
        call    _f_unsigned
        add     esp, 8
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

Possiamo vedere come vengono settati i flag facendo girare l'esempio in OllyDbg. Iniziamo con `f_unsigned()`, che funziona con numeri di tipo unsigned.

L'istruzione CMP è eseguita tre volte e con gli stessi argomenti, pertanto i flag saranno ogni volta gli stessi.

Risultato del primo confronto:

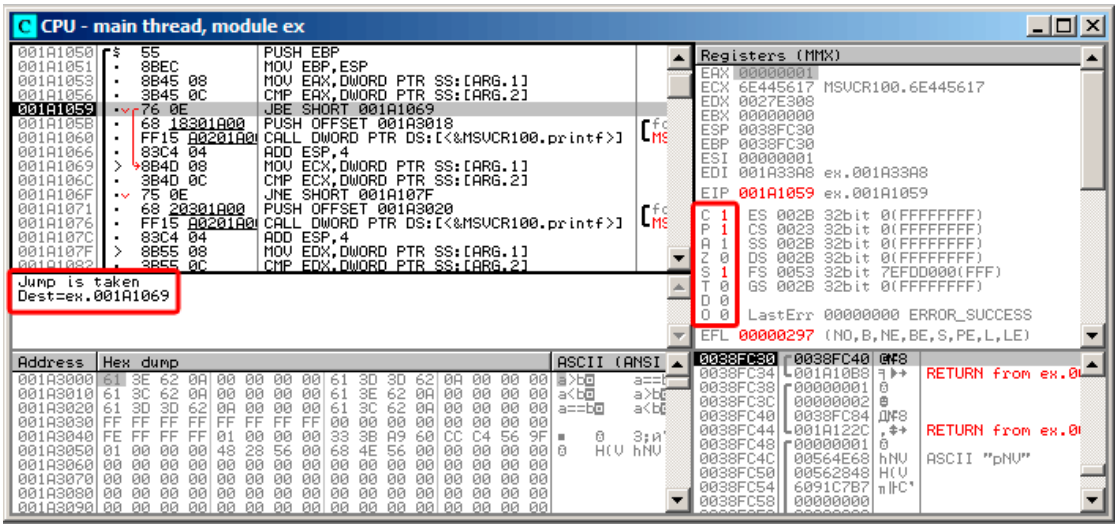


Figura 1.34: OllyDbg: `f_unsigned()`: primo salto condizionale

I flag sono: C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0. In OllyDbg sono riportati per brevità con la sola iniziale.

OllyDbg suggerisce che il jump (JBE) sarà innescato. Infatti, consultando i manuali Intel (8.1.4 on page 308), vediamo che JBE è innescato se CF=1 o ZF=1. La condizione è vera, e quindi il salto viene effettuato.

Jump condizionale successivo:

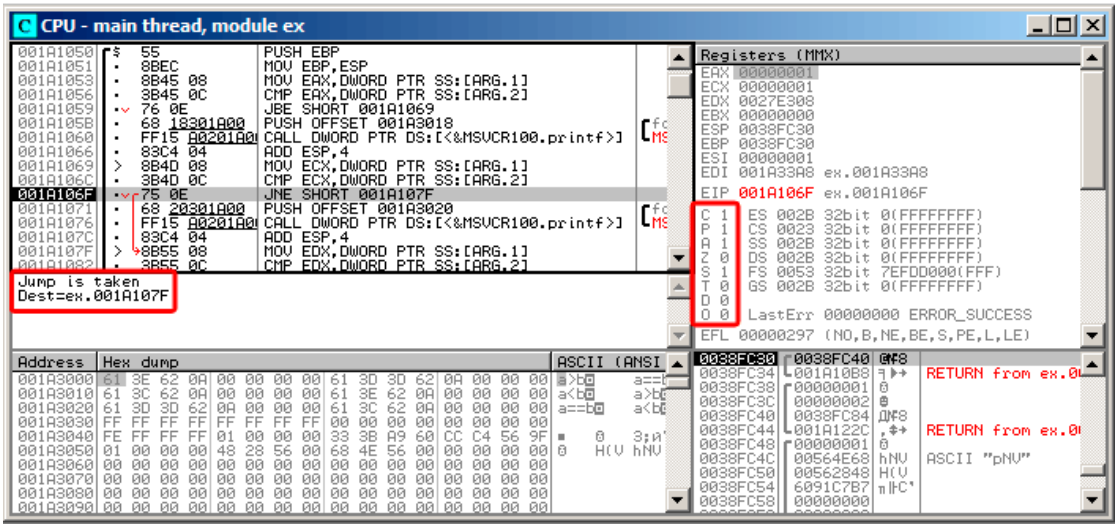


Figura 1.35: OllyDbg: `f_unsigned()`: secondo salto condizionale

OllyDbg dice che il JNZ verrà seguito. Infatti, JNZ è innescato se ZF=0 (zero flag).

Il terzo salto condizionale, JNB:

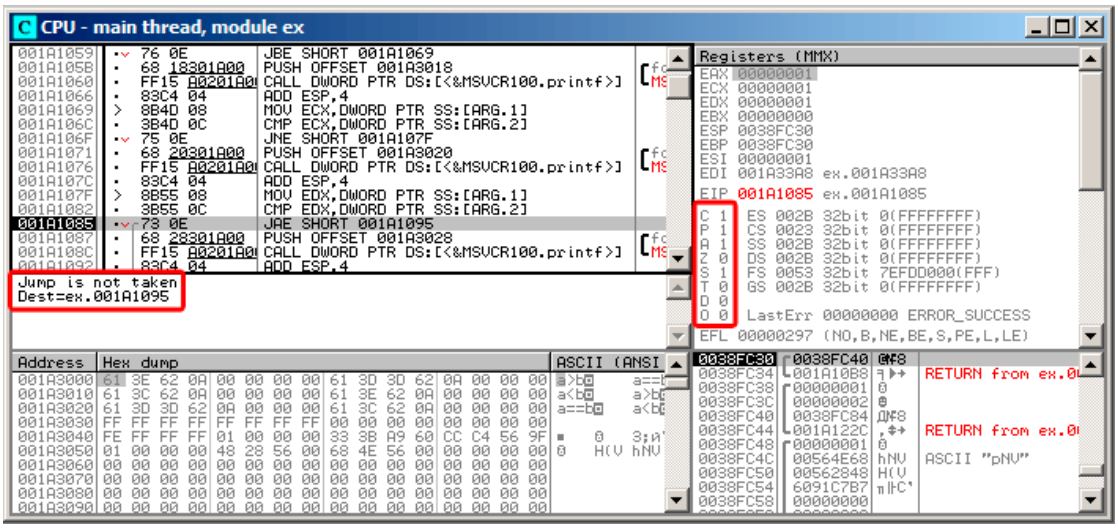


Figura 1.36: OllyDbg: `f_unsigned()`: terzo salto condizionale

Nei manuali Intel (8.1.4 on page 308) vediamo che JNB è innescato se CF=0 (carry flag). Nel nostro caso questa condizione non è vera, e quindi la terza `printf()` sarà eseguita.

Rivediamo ora la funzione `f_signed()`, che opera con valori signed, in OllyDbg. I flag sono settati allo stesso modo: C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0. Il primo salto condizionale JLE sarà eseguito:

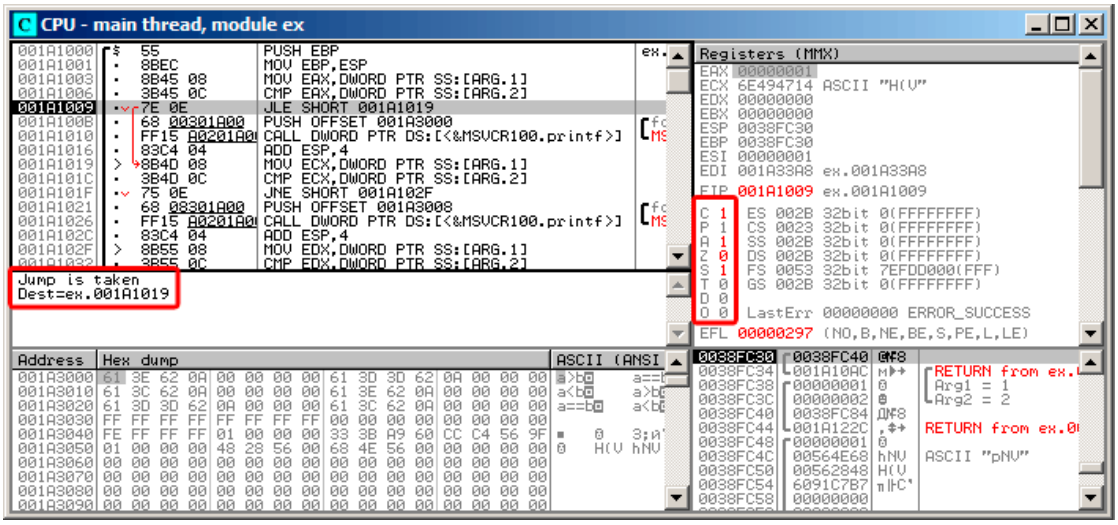


Figura 1.37: OllyDbg: `f_signed()`: primo salto condizionale

Nei manuali Intel (8.1.4 on page 308) vediamo che questa istruzione viene innescata se ZF=1 o SF≠OF. SF≠OF nel nostro caso, quindi il salto viene effettuato.

Il secondo salto condizionale JNZ viene innescato, se ZF=0 (zero flag):

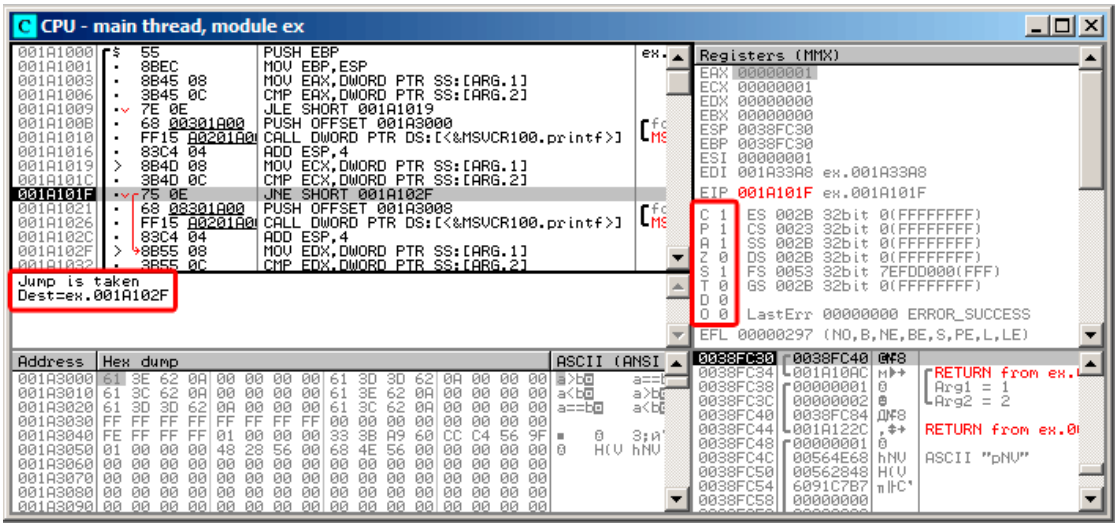


Figura 1.38: OllyDbg: `f_signed()`: secondo salto condizionale

Il terzo jump JGE non sarà innescato in quanto lo sarebbe solo se SF=OF, condizione non vera nel nostro caso:

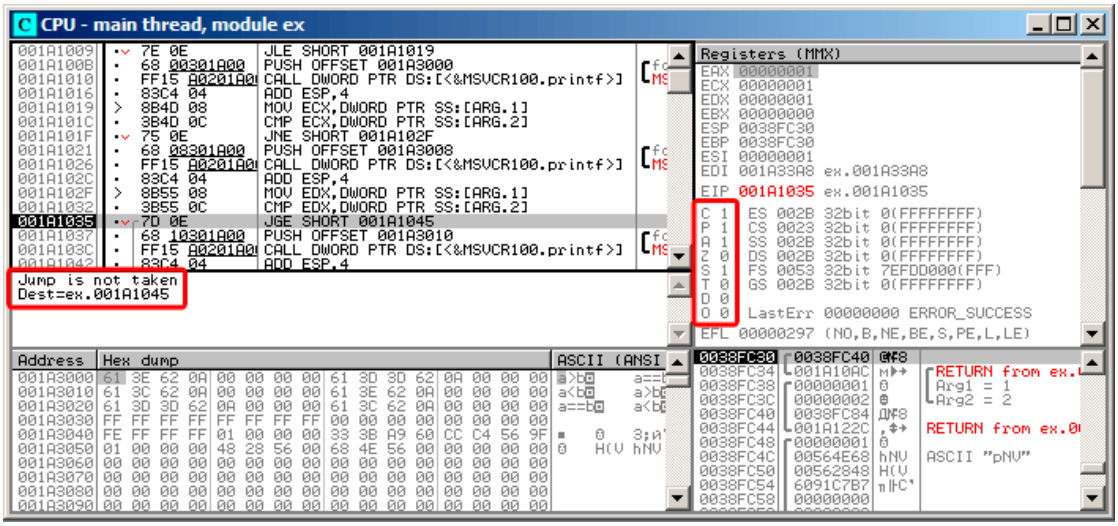


Figura 1.39: OllyDbg: `f_signed()`: terzo salto condizionale

Possiamo provare ad applicare una patch all' eseguibile in maniera tale che la funzione `f_unsigned()` stampi sempre «a==b», a prescindere dai valori in input.

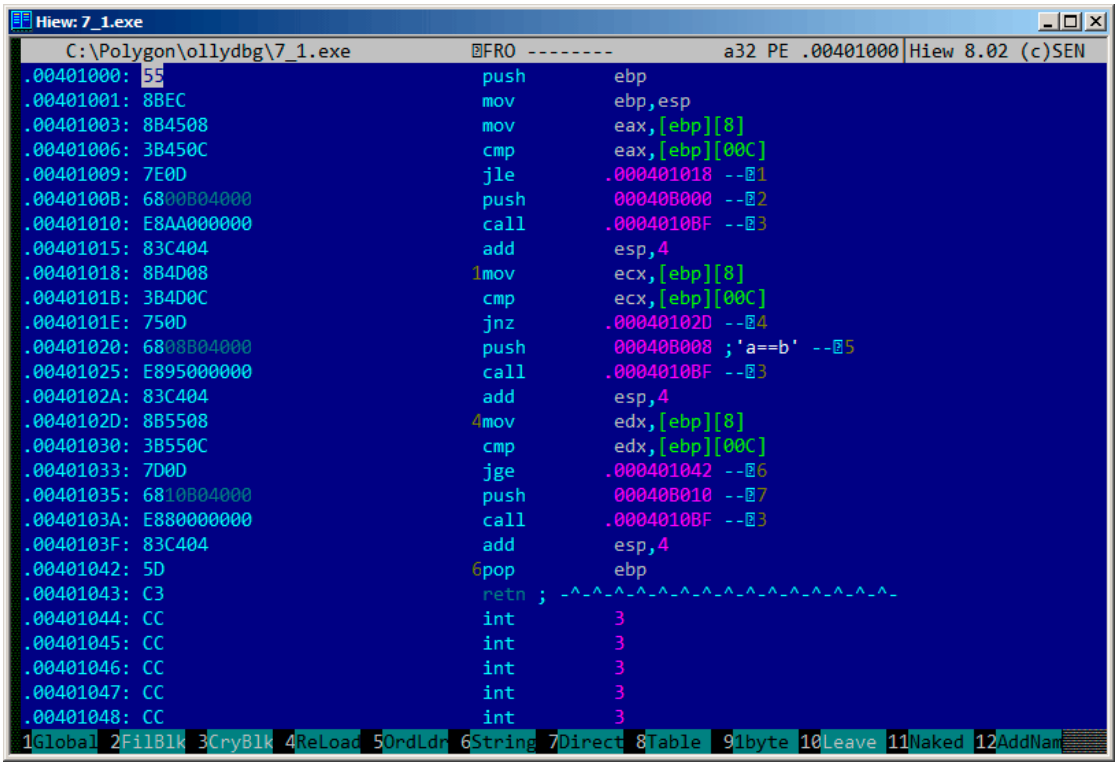


Figura 1.40: Hiew: funzione `f_unsigned()`

Essenzialmente, per ottenere il risultato desiderato, dobbiamo:

- forzare il primo jump in modo che sia sempre seguito;
- forzare il secondo jump a non essere mai seguito;
- forzare il terzo jump ad essere sempre seguito.

Possiamo così dirigere il flusso di esecuzione in modo tale da farlo sempre passare attraverso la seconda `printf()`, dando in output «a==b».

Devono essere corrette (patchate) tre istruzioni (o byte):

- Il primo jump diventa `JMP`, ma il offset di salto resta invariato.
- Il secondo jump potrebbe essere innescato in alcune occasioni, ma in ogni caso salterebbe alla prossima istruzione, poiché settiamo il offset di salto a 0.

In queste istruzioni il offset di salto viene sommato all'indirizzo della prossima istruzione. Quindi se l'offset è 0, il jump trasferirà il controllo all'istruzione successiva,

- Possiamo sostituire il terzo jump con `JMP` allo stesso modo del primo, in modo che sia sempre innescato.

Ecco il codice modificato:

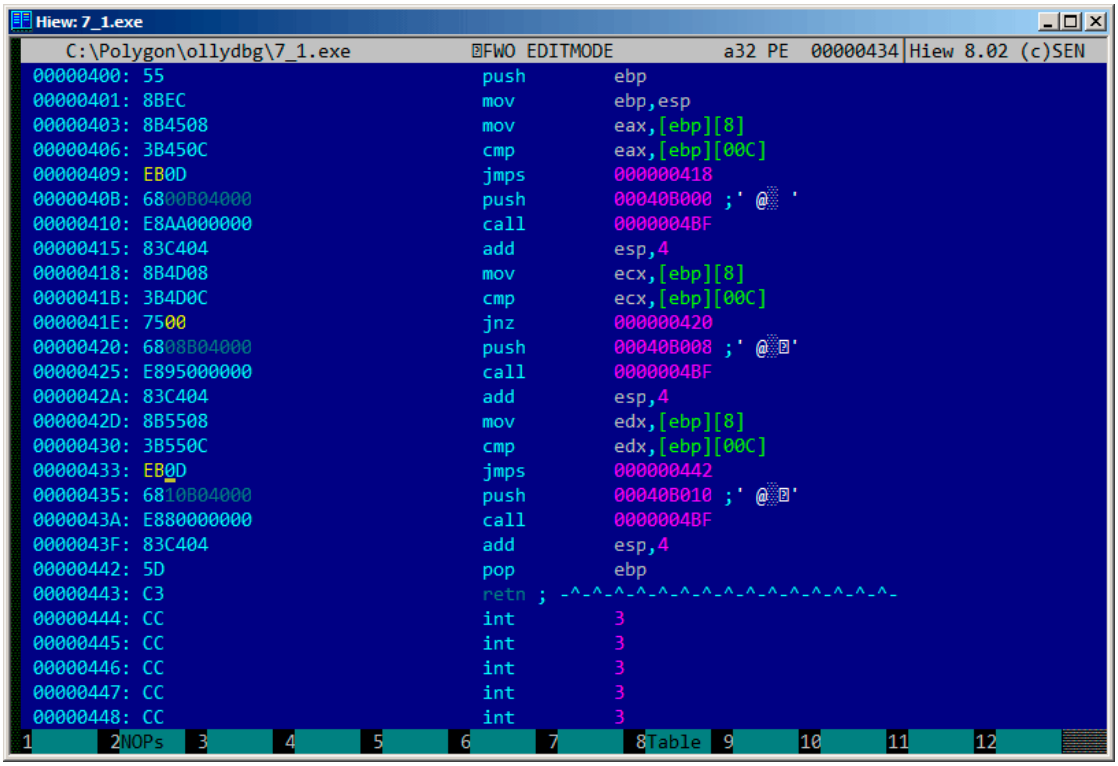


Figura 1.41: Hiew: funzione `f_unsigned()` modificata

Se ci dimentichiamo di cambiare uno di questi jump, potrebbero essere eseguite diverse chiamate a `printf()`, ma noi vogliamo eseguirne solo una.

Senza ottimizzazione GCC

Senza ottimizzazione GCC 4.4.1 produce pressoché lo stesso codice, ma usa `puts()` (1.5.3 on page 28) invece di `printf()`.

Con ottimizzazione GCC

Un lettore attento potrebbe domandare: perchè eseguire `CMP` più volte se i flag hanno gli stessi valori dopo ogni esecuzione?

Forse MSVC con ottimizzazioni non è in grado di applicare questa ottimizzazione, al contrario di GCC 4.8.1:

Listing 1.113: GCC 4.8.1 f_signed()

```
f_signed:
        mov     eax, DWORD PTR [esp+8]
        cmp     DWORD PTR [esp+4], eax
        jg      .L6
        je      .L7
        jge     .L1
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC2 ; "a<b"
        jmp     puts
.L6:
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC0 ; "a>b"
        jmp     puts
.L1:
        rep ret
.L7:
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC1 ; "a==b"
        jmp     puts
```

Notiamo anche l'uso di `JMP puts` al posto di `CALL puts / RETN`. Questo trucco sarà spiegato più avanti: 1.21.1 on page 201.

Questo tipo di codice x86 è piuttosto raro. MSVC 2012 apparentemente non è in grado di generarne di simile. Dall'altro lato, i programmatori assembly sanno perfettamente che le istruzioni `Jcc` possono essere disposte in fila.

Se vedete codice con una disposizione simile, è molto probabile che sia stato scritto a mano.

La funzione `f_unsigned()` non è esteticamente corta allo stesso modo:

Listing 1.114: GCC 4.8.1 f_unsigned()

```
f_unsigned:
        push    esi
        push    ebx
        sub     esp, 20
        mov     esi, DWORD PTR [esp+32]
        mov     ebx, DWORD PTR [esp+36]
        cmp     esi, ebx
        ja      .L13
        cmp     esi, ebx  ; questa istruzione può essere rimossa
        je      .L14
.L10:
        jb      .L15
        add     esp, 20
        pop     ebx
        pop     esi
        ret
.L15:
        mov     DWORD PTR [esp+32], OFFSET FLAT:.LC2 ; "a<b"
        add     esp, 20
        pop     ebx
        pop     esi
        jmp     puts
.L13:
        mov     DWORD PTR [esp], OFFSET FLAT:.LC0 ; "a>b"
```

```
        call    puts
        cmp     esi, ebx
        jne     .L10
.L14:
        mov     DWORD PTR [esp+32], OFFSET FLAT:.LC1 ; "a==b"
        add     esp, 20
        pop     ebx
        pop     esi
        jmp     puts
```

Ciò nonostante, ci sono due istruzioni CMP invece di tre. Gli algoritmi di ottimizzazione di GCC 4.8.1 probabilmente non sono ancora perfetti.

ARM

32-bit ARM

Con ottimizzazione Keil 6/2013 (Modalità ARM)

Listing 1.115: Con ottimizzazione Keil 6/2013 (Modalità ARM)

```
.text:000000B8                         EXPORT f_signed
.text:000000B8             f_signed                ; CODE XREF: main+C
.text:000000B8 70 40 2D E9             STMFD   SP!, {R4-R6,LR}
.text:000000BC 01 40 A0 E1             MOV     R4, R1
.text:000000C0 04 00 50 E1             CMP     R0, R4
.text:000000C4 00 50 A0 E1             MOV     R5, R0
.text:000000C8 1A 0E 8F C2             ADRGT   R0, aAB         ; "a>b\n"
.text:000000CC A1 18 00 CB             BLGT    __2printf
.text:000000D0 04 00 55 E1             CMP     R5, R4
.text:000000D4 67 0F 8F 02             ADREQ   R0, aAB_0       ; "a==b\n"
.text:000000D8 9E 18 00 0B             BLEQ    __2printf
.text:000000DC 04 00 55 E1             CMP     R5, R4
.text:000000E0 70 80 BD A8             LDMGEFD SP!, {R4-R6,PC}
.text:000000E4 70 40 BD E8             LDMFD   SP!, {R4-R6,LR}
.text:000000E8 19 0E 8F E2             ADR     R0, aAB_1       ; "a<b\n"
.text:000000EC 99 18 00 EA             B       __2printf
.text:000000EC             ; End of function f_signed
```

In modalità ARM molte istruzioni possono essere eseguite solo quando specifici flag sono settati. Es. sono spesso usate quando si confrontano numeri.

Ad esempio, l'istruzione ADD è infatti chiamata internamente ADDAL, il suffisso AL sta per *Always*, ad indicare che viene eseguita sempre. I predicati sono codificati nei 4 bit alti dell'istruzione ARM a 32-bit (*condition field*). L'istruzione B per effettuare un salto non condizionale è in realtà condizionale ed è codificata proprio come ogni altro jump condizionale, ma ha AL (*execute ALways*) nel *condition field*, e ciò implica che venga sempre eseguito, ignorando i flag.

L'istruzione ADRGT funziona come ADR, ma viene eseguita soltanto nel caso in cui la precedente istruzione CMP trovi uno dei due numeri a confronto più grande dell'altro, (*Greater Than*).

La successiva istruzione BLGT si comporta esattamente come BL ed il salto viene innescato solo se il risultato del confronto è (*Greater Than*). ADRGT scrive un putatore alla stringa a>b\n nel registro R0 e BLGT chiama `printf()`. Le istruzioni aventi il suffisso -GT in questo caso sono quindi eseguite solo se il valore in R0 (ovvero a) è maggiore del valore in R4 (ovvero b).

Andando avanti vediamo le istruzioni ADREQ e BLEQ. Si comporano come ADR e BL, ma vengono eseguite solo se gli operandi erano uguali al momento dell'ultimo confronto. Un altra CMP si trova subito prima di loro (poiché l'esecuzione di `printf()` potrebbe aver alterato i flag).

Ancora più avanti vediamo LDMGEFD, questa istruzione funziona come LDMFD[93], ma viene eseguita solo quando uno dei valori e maggiore di o uguale all'altro (*Greater or Equal*). L'istruzione `LDMGEFD SP!, {R4-R6,PC}` si comporta come un epilogo di funzione, ma viene eseguita solo se $a >= b$, e solo in tal caso avrà termine l'esecuzione della funzione.

Nel caso in cui questa condizione non venga soddisfatta, ovvero se $a < b$, il flusso continuerà alla successiva istruzione «`LDMFD SP!, {R4-R6,LR}`» , un altro epilogo di funzione. Questa istruzione non ripristina soltanto lo stato dei registri R4-R6 , ma anche LR invece di **PC!**, non ritornando così dalla funzione. Le due ultime istruzioni chiamano `printf()` con la stringa «a<b\n» come unico argomento. Abbiamo già visto un salto diretto non condizionale alla funzione `printf()` senza altro codice di uscita/ritorno dalla funzione nella sezione «printf() con più argomenti> > (1.11.2 on page 73).

`f_unsigned` è simile, e vengono utilizzate le funzioni ADRHI, BLHI, e LDMCSFD. Questi predicati (*HI = Unsigned higher, CS = Carry Set (maggiore di o uguale a)*) sono analoghi a quelli visti in precedenza, e operano su valori di tipo unsigned.

Nella funzione `main()` non c'è nulla di nuovo:

Listing 1.116: `main()`

```
.text:00000128                 EXPORT main
.text:00000128             main
.text:00000128 10 40 2D E9      STMFD   SP!, {R4,LR}
.text:0000012C 02 10 A0 E3      MOV     R1, #2
.text:00000130 01 00 A0 E3      MOV     R0, #1
.text:00000134 DF FF FF EB      BL      f_signed
.text:00000138 02 10 A0 E3      MOV     R1, #2
.text:0000013C 01 00 A0 E3      MOV     R0, #1
.text:00000140 EA FF FF EB      BL      f_unsigned
.text:00000144 00 00 A0 E3      MOV     R0, #0
.text:00000148 10 80 BD E8      LDMFD   SP!, {R4,PC}
.text:00000148             ; End of function main
```

In questo modo ci si può sbarazzare dei salti condizionali in modalità ARM. Perchè è bene? Leggi qui: 2.3.1 on page 284.

[93] LDMFD

Non esiste una funzionalità simile in x86, eccetto per l'istruzione CMOVcc , che è uguale a MOV ma viene eseguita solo se specifici flag sono settati, solitamente da CMP.

Con ottimizzazione Keil 6/2013 (Modalità Thumb)

Listing 1.117: Con ottimizzazione Keil 6/2013 (Modalità Thumb)

```
.text:00000072             f_signed ; CODE XREF: main+6
.text:00000072 70 B5         PUSH    {R4-R6,LR}
.text:00000074 0C 00         MOVS    R4, R1
.text:00000076 05 00         MOVS    R5, R0
.text:00000078 A0 42         CMP     R0, R4
.text:0000007A 02 DD         BLE     loc_82
.text:0000007C A4 A0         ADR     R0, aAB         ; "a>b\n"
.text:0000007E 06 F0 B7 F8   BL      __2printf
.text:00000082
.text:00000082             loc_82 ; CODE XREF: f_signed+8
.text:00000082 A5 42         CMP     R5, R4
.text:00000084 02 D1         BNE     loc_8C
.text:00000086 A4 A0         ADR     R0, aAB_0       ; "a==b\n"
.text:00000088 06 F0 B2 F8   BL      __2printf
.text:0000008C
.text:0000008C             loc_8C ; CODE XREF: f_signed+12
.text:0000008C A5 42         CMP     R5, R4
.text:0000008E 02 DA         BGE     locret_96
.text:00000090 A3 A0         ADR     R0, aAB_1       ; "a<b\n"
.text:00000092 06 F0 AD F8   BL      __2printf
.text:00000096
.text:00000096             locret_96 ; CODE XREF: f_signed+1C
.text:00000096 70 BD         POP     {R4-R6,PC}
.text:00000096             ; End of function f_signed
```

Solo le istruzioni B in modalità Thumb possono essere supplementate da *condition codes*, pertanto il codice Thumb ha un aspetto più ordinario.

BLE è un normale jump condizionale *Less than or Equal*, BNE—*Not Equal*, BGE—*Greater than or Equal*.

f_unsigned è simile, con la differenza che vengono usate altre istruzioni per operare con valori di tipo unsigned: BLS (*Unsigned lower or same*) e BCS (*Carry Set (Greater than or equal)*).

ARM64: Con ottimizzazione GCC (Linaro) 4.9

Listing 1.118: f_signed()

```
f_signed:
; W0=a, W1=b
        cmp     w0, w1
        bgt     .L19    ; Branch if Greater Than (a>b)
```

```
        beq     .L20    ; Branch if Equal (a==b)
        bge     .L15    ; Branch if Greater than or Equal (a>=b) (condizione
    impossibile, a questo punto)
        ; a<b
        adrp    x0, .LC11       ; "a<b"
        add     x0, x0, :lo12:.LC11
        b       puts
.L19:
        adrp    x0, .LC9        ; "a>b"
        add     x0, x0, :lo12:.LC9
        b       puts
.L15:   ; impossibile arrivare qui
        ret
.L20:
        adrp    x0, .LC10       ; "a==b"
        add     x0, x0, :lo12:.LC10
        b       puts
```

Listing 1.119: f_unsigned()

```
f_unsigned:
        stp     x29, x30, [sp, -48]!
; W0=a, W1=b
        cmp     w0, w1
        add     x29, sp, 0
        str     x19, [sp,16]
        mov     w19, w0
        bhi     .L25    ; Branch if HIgher (a>b)
        cmp     w19, w1
        beq     .L26    ; Branch if Equal (a==b)
.L23:
        bcc     .L27    ; Branch if Carry Clear (if less than) (a<b)
; epilogo della funzione, impossibile arrivare qui
        ldr     x19, [sp,16]
        ldp     x29, x30, [sp], 48
        ret
.L27:
        ldr     x19, [sp,16]
        adrp    x0, .LC11       ; "a<b"
        ldp     x29, x30, [sp], 48
        add     x0, x0, :lo12:.LC11
        b       puts
.L25:
        adrp    x0, .LC9        ; "a>b"
        str     x1, [x29,40]
        add     x0, x0, :lo12:.LC9
        bl      puts
        ldr     x1, [x29,40]
        cmp     w19, w1
        bne     .L23    ; Branch if Not Equal
.L26:
        ldr     x19, [sp,16]
        adrp    x0, .LC10       ; "a==b"
        ldp     x29, x30, [sp], 48
```

```
        add     x0, x0, :lo12:.LC10
        b       puts
```

I commenti nel codice sono stati inseriti dall'autore di questo libro. E' impressionante notare come il compilatore non si sia reso conto che alcuni condizioni sono del tutto impossibili, e per questo motivo si trovano delle parti con codice "morto" (dead code), che non può mai essere eseguito.

Esercizio

Prova ad ottimizzare manualmente queste funzioni per ottenere una versione più compatta, rimuovendo istruzioni ridondanti e senza aggiungerne di nuove.

MIPS

Una caratteristica distintiva di MIPS è l'assenza dei flag. Apparentemente è una scelta fatta per semplificare l'analisi della dipendenza dai dati.

Esistono istruzioni simili a SETcc in x86: SLT («Set on Less Than»: versione signed) e SLTU (versione unsigned). Queste istruzioni impostano il valore del registro di destinazione a 1 se la condizione è vera, a 0 se è falsa.

Il registro di destinazione viene quindi controllato usando BEQ («Branch on Equal») oppure BNE («Branch on Not Equal») ed in base al caso si può verificare un salto. Questa coppia di istruzioni è usata in MIPS per eseguire confronti e conseguenti branch. Iniziamo con la versione signed della nostra funzione:

Listing 1.120: Senza ottimizzazione GCC 4.4.5 (IDA)

```
.text:00000000 f_signed:  # CODE XREF: main+18
.text:00000000
.text:00000000 var_10  = -0x10
.text:00000000 var_8   = -8
.text:00000000 var_4   = -4
.text:00000000 arg_0   =  0
.text:00000000 arg_4   =  4
.text:00000000
.text:00000000         addiu   $sp, -0x20
.text:00000004         sw      $ra, 0x20+var_4($sp)
.text:00000008         sw      $fp, 0x20+var_8($sp)
.text:0000000C         move    $fp, $sp
.text:00000010         la      $gp, __gnu_local_gp
.text:00000018         sw      $gp, 0x20+var_10($sp)
; memorizza i valori di input nello stack locale:
.text:0000001C         sw      $a0, 0x20+arg_0($fp)
.text:00000020         sw      $a1, 0x20+arg_4($fp)
; reload them.
.text:00000024         lw      $v1, 0x20+arg_0($fp)
.text:00000028         lw      $v0, 0x20+arg_4($fp)
; $v0=b
; $v1=a
.text:0000002C         or      $at, $zero ; NOP
```

```
; questa è una pseudoistruzione. In realtà sarebbe "slt $v0,$v0,$v1".
; Quindi $v0 sarà settato a 1 se $v0<$v1 (b<a), altrimenti a 0:
.text:00000030         slt     $v0, $v1
; salta a loc_5c, se la condizione non è vera.
; pseudoistruzione, sarebbe "beq $v0,$zero,loc_5c":
.text:00000034         beqz    $v0, loc_5C
; stampa "a>b" ed esce
.text:00000038         or      $at, $zero ; branch delay slot, NOP
.text:0000003C         lui     $v0, (unk_230 >> 16) # "a>b"
.text:00000040         addiu   $a0, $v0, (unk_230 & 0xFFFF) # "a>b"
.text:00000044         lw      $v0, (puts & 0xFFFF)($gp)
.text:00000048         or      $at, $zero ; NOP
.text:0000004C         move    $t9, $v0
.text:00000050         jalr    $t9
.text:00000054         or      $at, $zero ; branch delay slot, NOP
.text:00000058         lw      $gp, 0x20+var_10($fp)
.text:0000005C
.text:0000005C loc_5C:                          # CODE XREF: f_signed+34
.text:0000005C         lw      $v1, 0x20+arg_0($fp)
.text:00000060         lw      $v0, 0x20+arg_4($fp)
.text:00000064         or      $at, $zero ; NOP
; controlla se a==b, salta a loc_90 se la condizione non è vera:
.text:00000068         bne     $v1, $v0, loc_90
.text:0000006C         or      $at, $zero ; branch delay slot, NOP
; se la condizione è vera, stampa "a==b" ed esce:
.text:00000070         lui     $v0, (aAB >> 16)  # "a==b"
.text:00000074         addiu   $a0, $v0, (aAB & 0xFFFF)  # "a==b"
.text:00000078         lw      $v0, (puts & 0xFFFF)($gp)
.text:0000007C         or      $at, $zero ; NOP
.text:00000080         move    $t9, $v0
.text:00000084         jalr    $t9
.text:00000088         or      $at, $zero ; branch delay slot, NOP
.text:0000008C         lw      $gp, 0x20+var_10($fp)
.text:00000090
.text:00000090 loc_90:                          # CODE XREF: f_signed+68
.text:00000090         lw      $v1, 0x20+arg_0($fp)
.text:00000094         lw      $v0, 0x20+arg_4($fp)
.text:00000098         or      $at, $zero ; NOP
; controlla se $v1<$v0 (a<b), imposta $v0 a 1 se la condizione è vera:
.text:0000009C         slt     $v0, $v1, $v0
; se la condizione non è vera (es., $v0==0), salta a loc_c8:
.text:000000A0         beqz    $v0, loc_C8
.text:000000A4         or      $at, $zero ; branch delay slot, NOP
; condizione vera, stampa "a<b" ed esce:
.text:000000A8         lui     $v0, (aAB_0 >> 16)  # "a<b"
.text:000000AC         addiu   $a0, $v0, (aAB_0 & 0xFFFF)  # "a<b"
.text:000000B0         lw      $v0, (puts & 0xFFFF)($gp)
.text:000000B4         or      $at, $zero ; NOP
.text:000000B8         move    $t9, $v0
.text:000000BC         jalr    $t9
.text:000000C0         or      $at, $zero ; branch delay slot, NOP
.text:000000C4         lw      $gp, 0x20+var_10($fp)
.text:000000C8
```

```
; tutte le 3 condizioni sono false, esce dalla funzione:
.text:000000C8 loc_C8:                                 # CODE XREF:
    f_signed+A0
.text:000000C8         move    $sp, $fp
.text:000000CC         lw      $ra, 0x20+var_4($sp)
.text:000000D0         lw      $fp, 0x20+var_8($sp)
.text:000000D4         addiu   $sp, 0x20
.text:000000D8         jr      $ra
.text:000000DC         or      $at, $zero ; branch delay slot, NOP
.text:000000DC  # End of function f_signed
```

SLT REG0, REG0, REG1 è stata ridotta da Ida nella sua forma breve: SLT REG0, REG1.

Notiamo anche la pseudo istruzione BEQZ («Branch if Equal to Zero»), che è in realtà BEQ REG, $ZERO, LABEL.

La versione unsigned è uguale, SLTU (versione unsigned, da cui la «U» nel nome) è usata al posto di SLT:

Listing 1.121: Senza ottimizzazione GCC 4.4.5 (IDA)

```
.text:000000E0 f_unsigned:  # CODE XREF: main+28
.text:000000E0
.text:000000E0 var_10  = -0x10
.text:000000E0 var_8   = -8
.text:000000E0 var_4   = -4
.text:000000E0 arg_0   =  0
.text:000000E0 arg_4   =  4
.text:000000E0
.text:000000E0         addiu   $sp, -0x20
.text:000000E4         sw      $ra, 0x20+var_4($sp)
.text:000000E8         sw      $fp, 0x20+var_8($sp)
.text:000000EC         move    $fp, $sp
.text:000000F0         la      $gp, __gnu_local_gp
.text:000000F8         sw      $gp, 0x20+var_10($sp)
.text:000000FC         sw      $a0, 0x20+arg_0($fp)
.text:00000100         sw      $a1, 0x20+arg_4($fp)
.text:00000104         lw      $v1, 0x20+arg_0($fp)
.text:00000108         lw      $v0, 0x20+arg_4($fp)
.text:0000010C         or      $at, $zero
.text:00000110         sltu    $v0, $v1
.text:00000114         beqz    $v0, loc_13C
.text:00000118         or      $at, $zero
.text:0000011C         lui     $v0, (unk_230 >> 16)
.text:00000120         addiu   $a0, $v0, (unk_230 & 0xFFFF)
.text:00000124         lw      $v0, (puts & 0xFFFF)($gp)
.text:00000128         or      $at, $zero
.text:0000012C         move    $t9, $v0
.text:00000130         jalr    $t9
.text:00000134         or      $at, $zero
.text:00000138         lw      $gp, 0x20+var_10($fp)
.text:0000013C
.text:0000013C loc_13C:                                # CODE XREF: f_unsigned+34
.text:0000013C         lw      $v1, 0x20+arg_0($fp)
```

```
.text:00000140         lw      $v0, 0x20+arg_4($fp)
.text:00000144         or      $at, $zero
.text:00000148         bne     $v1, $v0, loc_170
.text:0000014C         or      $at, $zero
.text:00000150         lui     $v0, (aAB >> 16)  # "a==b"
.text:00000154         addiu   $a0, $v0, (aAB & 0xFFFF)  # "a==b"
.text:00000158         lw      $v0, (puts & 0xFFFF)($gp)
.text:0000015C         or      $at, $zero
.text:00000160         move    $t9, $v0
.text:00000164         jalr    $t9
.text:00000168         or      $at, $zero
.text:0000016C         lw      $gp, 0x20+var_10($fp)
.text:00000170
.text:00000170 loc_170:                          # CODE XREF: f_unsigned+68
.text:00000170         lw      $v1, 0x20+arg_0($fp)
.text:00000174         lw      $v0, 0x20+arg_4($fp)
.text:00000178         or      $at, $zero
.text:0000017C         sltu    $v0, $v1, $v0
.text:00000180         beqz    $v0, loc_1A8
.text:00000184         or      $at, $zero
.text:00000188         lui     $v0, (aAB_0 >> 16)  # "a<b"
.text:0000018C         addiu   $a0, $v0, (aAB_0 & 0xFFFF)  # "a<b"
.text:00000190         lw      $v0, (puts & 0xFFFF)($gp)
.text:00000194         or      $at, $zero
.text:00000198         move    $t9, $v0
.text:0000019C         jalr    $t9
.text:000001A0         or      $at, $zero
.text:000001A4         lw      $gp, 0x20+var_10($fp)
.text:000001A8
.text:000001A8 loc_1A8:                          # CODE XREF: f_unsigned+A0
.text:000001A8         move    $sp, $fp
.text:000001AC         lw      $ra, 0x20+var_4($sp)
.text:000001B0         lw      $fp, 0x20+var_8($sp)
.text:000001B4         addiu   $sp, 0x20
.text:000001B8         jr      $ra
.text:000001BC         or      $at, $zero
.text:000001BC  # End of function f_unsigned
```

1.18.2 Calcolo del valore assoluto

Una funzione semplice:

```
int my_abs (int i)
{
        if (i<0)
                return -i;
        else
                return i;
};
```

Con ottimizzazione MSVC

Il codice solitamente generato è questo:

Listing 1.122: Con ottimizzazione MSVC 2012 x64

```
i$ = 8
my_abs  PROC
; ECX = input
        test    ecx, ecx
; controllo del segno del valore in input
; salta l'istruzione NEG se il segno è positivo
        jns     SHORT $LN2@my_abs
; valore negato
        neg     ecx
$LN2@my_abs:
; prepara il risultato in EAX:
        mov     eax, ecx
        ret     0
my_abs  ENDP
```

GCC 4.9 fa più o meno lo stesso.

Con ottimizzazione Keil 6/2013: Modalità Thumb

Listing 1.123: Con ottimizzazione Keil 6/2013: Modalità Thumb

```
my_abs PROC
        CMP      r0,#0
; il valore in input è maggiore o uguale a zero?
; allora salta l'istruzione RSBS
        BGE      |L0.6|
; sottrai a 0 il valore di input:
        RSBS     r0,r0,#0
|L0.6|
        BX       lr
        ENDP
```

In ARM manca l'istruzione di negazione, quindi il compilatore Keil usa l'istruzione «Reverse Subtract», che semplicemente sottrae gli operandi in modo inverso.

Con ottimizzazione Keil 6/2013: Modalità ARM

In modalità ARM è possibile aggiungere codice condizionale ad alcune istruzioni, e questo è ciò che fa il compilatore keil:

Listing 1.124: Con ottimizzazione Keil 6/2013: Modalità ARM

```
my_abs PROC
        CMP      r0,#0
; esegui l'istruzione "Reverse Subtract" solo se
; il valore di input è minore di 0:
        RSBLT    r0,r0,#0
        BX       lr
```

```
        ENDP
```

Adesso non ci sono più jump condizionali, e ciò è bene: 2.3.1 on page 284.

Senza ottimizzazione GCC 4.9 (ARM64)

ARM64 ha un'istruzione NEG per la negazione:

Listing 1.125: Con ottimizzazione GCC 4.9 (ARM64)

```
my_abs:
        sub     sp, sp, #16
        str     w0, [sp,12]
        ldr     w0, [sp,12]
; confronta il valore di input con il contenuto del registro WZR
; (che ha sempre zero)
        cmp     w0, wzr
        bge     .L2
        ldr     w0, [sp,12]
        neg     w0, w0
        b       .L3
.L2:
        ldr     w0, [sp,12]
.L3:
        add     sp, sp, 16
        ret
```

MIPS

Listing 1.126: Con ottimizzazione GCC 4.4.5 (IDA)

```
my_abs:
; salta se $a0<0:
                bltz    $a0, locret_10
; ritorna il valore di input ($a0) in $v0:
                move    $v0, $a0
                jr      $ra
                or      $at, $zero ; branch delay slot, NOP
locret_10:
; nega il valore di input e salvalo in $v0:
                jr      $ra
; questa è una pseudoistruzione. Infatti, questa è "subu $v0,$zero,$a0"
    ($v0=0-$a0)
                negu    $v0, $a0
```

Qui vediamo una nuova istruzione: BLTZ («Branch if Less Than Zero»).

C'è anche la pseudoistruzione NEGU , che semplicemente fa la sottrazione da zero. Il suffisso «U» suffix in entrambe SUBU e NEGU implica che non verrà sollevata nessuna eccezione in caso di integer overflow.

Branchless version?

C'è anche una versione senza diramazioni (branchless) di questo codice. Ne riparleremo più avanti: **??** on page ??.

1.18.3 Operatore ternario

L'operatore ternario in C/C++ ha la seguente sintassi:

```
espressione ? espressione : espressione
```

Ecco un semplice esempio:

```
const char* f (int a)
{
        return a==10 ? "it is ten" : "it is not ten";
};
```

x86

I vecchi compilatori e quelli non ottimizzanti generano codice assembly come se fosse stata usata una coppia `if/else`:

Listing 1.127: Senza ottimizzazione MSVC 2008

```
$SG746  DB      'it is ten', 00H
$SG747  DB      'it is not ten', 00H

tv65 = -4 ; questa verrà utilizzata come variabile temporanea
_a$ = 8
_f      PROC
        push    ebp
        mov     ebp, esp
        push    ecx
; confronta il valore in input con 10
        cmp     DWORD PTR _a$[ebp], 10
; se non è uguale, salta a $LN3@f
        jne     SHORT $LN3@f
; salva il puntatore alla stringa nella variabile temporanea:
        mov     DWORD PTR tv65[ebp], OFFSET $SG746 ; 'it is ten'
; salta a exit
        jmp     SHORT $LN4@f
$LN3@f:
; salva il puntatore alla stringa nella variabile temporanea:
        mov     DWORD PTR tv65[ebp], OFFSET $SG747 ; 'it is not ten'
$LN4@f:
; questa è l' exit.
; copia il puntatore alla stringa dalla variabile temporanea in EAX.
        mov     eax, DWORD PTR tv65[ebp]
        mov     esp, ebp
        pop     ebp
        ret     0
_f      ENDP
```

Listing 1.128: Con ottimizzazione MSVC 2008

```
$SG792  DB      'it is ten', 00H
$SG793  DB      'it is not ten', 00H

_a$ = 8 ; dimensione = 4
_f      PROC
; confronta il valore input con 10
        cmp     DWORD PTR _a$[esp-4], 10
        mov     eax, OFFSET $SG792 ; 'it is ten'
; se è uguale, salta a $LN4@f
        je      SHORT $LN4@f
        mov     eax, OFFSET $SG793 ; 'it is not ten'
$LN4@f:
        ret     0
_f      ENDP
```

I nuovi compilatori sono più concisi:

Listing 1.129: Con ottimizzazione MSVC 2012 x64

```
$SG1355 DB      'it is ten', 00H
$SG1356 DB      'it is not ten', 00H

a$ = 8
f       PROC
; carica i puntatore ad entrambe le stringhe
        lea     rdx, OFFSET FLAT:$SG1355 ; 'it is ten'
        lea     rax, OFFSET FLAT:$SG1356 ; 'it is not ten'
; confronta il valore di input con 10
        cmp     ecx, 10
; se è uguale, copia il valore da RDX ("it is ten")
; altrimenti, non fare niente. Il puntatore alla strunga
; "it is not ten" è già in RAX.
        cmove   rax, rdx
        ret     0
f       ENDP
```

Con ottimizzazione GCC 4.8 per x86 usa anche l'istruzione `CMOVcc`, mentre la versione non-optimizing usa jump condizionali.

ARM

Anche l'ottimizzazione Keil per ARM mode usa le istruzioni condizionali `ADRcc`:

Listing 1.130: Con ottimizzazione Keil 6/2013 (Modalità ARM)

```
f PROC
; confronta il valore in input con 10
        CMP      r0,#0xa
; se il risultato del confronto è uguale, copia il puntatore alla stringa "it
    is ten" in R0
        ADREQ    r0,|L0.16| ; "it is ten"
; se il risultato del confronto è diverso, copia il puntatore alla stringa
; "it is not ten" in R0
```

```
        ADRNE    r0,|L0.28| ; "it is not ten"
        BX       lr
        ENDP

|L0.16|
        DCB      "it is ten",0
|L0.28|
        DCB      "it is not ten",0
```

Senza alcun intervento manuale, le due istruzioni ADREQ e ADRNE non possono essere eseguite nella stessa istanza.

Con ottimizzazione Keil per Thumb mode deve usare i jump condizionali, in quanto non esistono istruzioni di caricamento che supportano i flag condizionali:

Listing 1.131: Con ottimizzazione Keil 6/2013 (Modalità Thumb)

```
f PROC
; confronta il valore in input con 10
        CMP      r0,#0xa
; Se è uguale, salta a |L0.8|
        BEQ      |L0.8|
        ADR      r0,|L0.12| ; "it is not ten"
        BX       lr
|L0.8|
        ADR      r0,|L0.28| ; "it is ten"
        BX       lr
        ENDP

|L0.12|
        DCB      "it is not ten",0
|L0.28|
        DCB      "it is ten",0
```

ARM64

L' Con ottimizzazione GCC (Linaro) 4.9 per ARM64 usa anch'esso i jump condizionali:

Listing 1.132: Con ottimizzazione GCC (Linaro) 4.9

```
f:
        cmp     x0, 10
        beq     .L3             ; salta se è uguale
        adrp    x0, .LC1        ; "it is ten"
        add     x0, x0, :lo12:.LC1
        ret
.L3:
        adrp    x0, .LC0        ; "it is not ten"
        add     x0, x0, :lo12:.LC0
        ret
.LC0:
        .string "it is ten"
.LC1:
        .string "it is not ten"
```

Ciò avviene perchè ARM64 non ha una semplice istruzione di caricamento con flag condizionali, come ADRcc in ARM mode a 32-bit o CMOVcc in x86.

Tuttavia ha l'istruzione «Conditional SELect» (CSEL)[*ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, (2013)p390, C5.5], ma GCC 4.9 non sembra essere abbastanza intelligente da usarla in un simile pezzo di codice.

MIPS

Sfortunatamente, anche GCC 4.4.5 per MIPS non è molto intelligente:

Listing 1.133: Con ottimizzazione GCC 4.4.5 (risultato dell'assembly)

```
$LC0:
        .ascii  "it is not ten\000"
$LC1:
        .ascii  "it is ten\000"
f:
        li      $2,10                   # 0xa
; confronta $a0 e 10, se sono uguali salta:
        beq     $4,$2,$L2
        nop ; branch delay slot

; metti l'indirizzo della stringa "it is not ten" in $v0 e ritorna:
        lui     $2,%hi($LC0)
        j       $31
        addiu   $2,$2,%lo($LC0)

$L2:
; metti l'indirizzo della stringa "it is ten" in $v0 e return:
        lui     $2,%hi($LC1)
        j       $31
        addiu   $2,$2,%lo($LC1)
```

Riscriviamolo come un `if/else`

```
const char* f (int a)
{
        if (a==10)
                return "it is ten";
        else
                return "it is not ten";
};
```

E' interessante notare che GCC 4.8 ottimizzante per x86 è stato in grado di usare CMOVcc in questo caso:

Listing 1.134: Con ottimizzazione GCC 4.8

```
.LC0:
        .string "it is ten"
.LC1:
        .string "it is not ten"
```

```
f:
.LFB0:
; confronta il valode di input con 10
        cmp     DWORD PTR [esp+4], 10
        mov     edx, OFFSET FLAT:.LC1 ; "it is not ten"
        mov     eax, OFFSET FLAT:.LC0 ; "it is ten"
; se il rislutato del confronto non è uguale, copia il valore di EDX in EAX
; altrimenti, non fare nulla
        cmovne  eax, edx
        ret
```

Keil ottimizzante in ARM mode genera codice identico a listato.1.130.

Ma MSVC 2012 ottimizzante non è (ancora) così in gamba.

Conclusione

Perchè i compilatori ottimizzanti cercano di sbarazzarsi dei jump condizionali? Leggi qui: 2.3.1 on page 284.

1.18.4 Ottenere i valori massimo e minimo

32-bit

```
int my_max(int a, int b)
{
        if (a>b)
                return a;
        else
                return b;
};

int my_min(int a, int b)
{
        if (a<b)
                return a;
        else
                return b;
};
```

Listing 1.135: Senza ottimizzazione MSVC 2013

```
_a$ = 8
_b$ = 12
_my_min PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
; confronta A e B:
        cmp     eax, DWORD PTR _b$[ebp]
; se A è maggiore o uguale a B, salta:
        jge     SHORT $LN2@my_min
```

```
; altrimenti ricarica A in EAX EAX e salta a exit
        mov     eax, DWORD PTR _a$[ebp]
        jmp     SHORT $LN3@my_min
        jmp     SHORT $LN3@my_min ; questo JMP è rindondante
$LN2@my_min:
; ritorna B
        mov     eax, DWORD PTR _b$[ebp]
$LN3@my_min:
        pop     ebp
        ret     0
_my_min ENDP

_a$ = 8
_b$ = 12
_my_max PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
; confronta A e B:
        cmp     eax, DWORD PTR _b$[ebp]
; se A è minore o uguale a B, salta:
        jle     SHORT $LN2@my_max
; altrimenti ricarica A in EAX e salta a exit
        mov     eax, DWORD PTR _a$[ebp]
        jmp     SHORT $LN3@my_max
        jmp     SHORT $LN3@my_max ; questo JMP è rindondante
$LN2@my_max:
; ritorna B
        mov     eax, DWORD PTR _b$[ebp]
$LN3@my_max:
        pop     ebp
        ret     0
_my_max ENDP
```

Queste due funzioni differiscono solo per l'istruzione di salto condizionale: JGE («Jump if Greater or Equal») è usata nella prima e JLE («Jump if Less or Equal») nella seconda.

In ciascuna funzione c'è un'istruzione JMP non necessaria, che MSVC ha probabilmente lasciato per sbaglio.

Branchless

ARM in modalità Thumb ci ricorda molto codice x86:

Listing 1.136: Con ottimizzazione Keil 6/2013 (Modalità Thumb)

```
my_max PROC
; R0=A
; R1=B
; confronta A e B:
        CMP      r0,r1
; se A è maggiore di B salta:
```

```
        BGT      |L0.6|
; altrimenti (A<=B) ritorna R1 (B):
        MOVS     r0,r1
|L0.6|
; ritorna
        BX       lr
        ENDP

my_min PROC
; R0=A
; R1=B
; confronta A e B:
        CMP      r0,r1
; se A è minore di B, salta:
        BLT      |L0.14|
; altrimenti (A>=B) ritorna R1 (B):
        MOVS     r0,r1
|L0.14|
; ritorna
        BX       lr
        ENDP
```

Le funzioni differiscono per le istruzioni di branching: BGT e BLT. Essendo possibile usare suffissi condizionali in modalità ARM, il codice è più conciso.

MOVcc viene eseguita se la condizione è soddisfatta:

Listing 1.137: Con ottimizzazione Keil 6/2013 (Modalità ARM)

```
my_max PROC
; R0=A
; R1=B
; confronta A e B:
        CMP      r0,r1
; restituisci B al posto di A inserendo B in R0
; questa istruzione si attiva solo se A<=B (quindi, LE - Less or Equal)
; se l' istruzione non si attiva (cioè se A>B),
; A è ancora nel registro R0
        MOVLE    r0,r1
        BX       lr
        ENDP

my_min PROC
; R0=A
; R1=B
; confronta A e B:
        CMP      r0,r1
; ritorna B al posto di A inserendo B in R0
; questa istruzione si attiva solo se A>=B (quindi, GE - Greater or Equal)
; se l'istruzione non si attiva (cioè A<B),
; il valore A è ancora nel registro R0
        MOVGE    r0,r1
        BX       lr
        ENDP
```

Con ottimizzazione, GCC 4.8.1 e MSVC 2013 possono usare l'istruzione `CMOVcc`, che è analoga a `MOVcc` in ARM:

Listing 1.138: Con ottimizzazione MSVC 2013

```
my_max:
        mov     edx, DWORD PTR [esp+4]
        mov     eax, DWORD PTR [esp+8]
; EDX=A
; EAX=B
; confronta A e B:
        cmp     edx, eax
; se A>=B, carica il valore di A in EAX
; altrimenti questa istruzione è inutile (se A<B)
        cmovge  eax, edx
        ret

my_min:
        mov     edx, DWORD PTR [esp+4]
        mov     eax, DWORD PTR [esp+8]
; EDX=A
; EAX=B
; confronta A e B:
        cmp     edx, eax
; se A<=B, carica il valore di A in EAX
; altrimenti questa istruzione è inutile (se A>B)
        cmovle  eax, edx
        ret
```

64-bit

```
#include <stdint.h>

int64_t my_max(int64_t a, int64_t b)
{
        if (a>b)
                return a;
        else
                return b;
};

int64_t my_min(int64_t a, int64_t b)
{
        if (a<b)
                return a;
        else
                return b;
};
```

C'è un po' di spostamento di valori non necessario, ma il codice è comprensibile:

Listing 1.139: Senza ottimizzazione GCC 4.9.1 ARM64

```
my_max:
        sub     sp, sp, #16
        str     x0, [sp,8]
        str     x1, [sp]
        ldr     x1, [sp,8]
        ldr     x0, [sp]
        cmp     x1, x0
        ble     .L2
        ldr     x0, [sp,8]
        b       .L3
.L2:
        ldr     x0, [sp]
.L3:
        add     sp, sp, 16
        ret

my_min:
        sub     sp, sp, #16
        str     x0, [sp,8]
        str     x1, [sp]
        ldr     x1, [sp,8]
        ldr     x0, [sp]
        cmp     x1, x0
        bge     .L5
        ldr     x0, [sp,8]
        b       .L6
.L5:
        ldr     x0, [sp]
.L6:
        add     sp, sp, 16
        ret
```

Branchless

Non è necessario caricare gli argomenti della funzione dallo stack poiché sono già nei registri:

Listing 1.140: Con ottimizzazione GCC 4.9.1 x64

```
my_max:
; RDI=A
; RSI=B
; confronta A e B:
        cmp     rdi, rsi
; carica B in RAX per ritornarlo:
        mov     rax, rsi
; se A>=B, carica A (RDI) in RAX per ritornarlo.
; altrimenti questa istruzione è inutile (se A<B)
        cmovge  rax, rdi
        ret

my_min:
```

```
; RDI=A
; RSI=B
; confronta A e B:
        cmp     rdi, rsi
; carica B in RAX per ritornarlo:
        mov     rax, rsi
; se A<=B, carica A (RDI) in RAX per ritornarlon.
; altrimenti questa istruzione è inutile (se A>B)
        cmovle  rax, rdi
        ret
```

MSVC 2013 fa pressoché lo stesso.

ARM64 ha l'istruzione CSEL, che funziona esattamente come MOVcc in ARM o CMOVcc in x86, cambia soltanto il nome: «Conditional SELect».

Listing 1.141: Con ottimizzazione GCC 4.9.1 ARM64

```
my_max:
; X0=A
; X1=B
; confronta A e B:
        cmp     x0, x1
; seleziona X0 (A) a X0 se X0>=X1 o A>=B (Greater or Equal)
; seleziona X1 (B) a X0 se A<B
        csel    x0, x0, x1, ge
        ret

my_min:
; X0=A
; X1=B
; confronta A e B:
        cmp     x0, x1
; seleziona X0 (A) a X0 se X0<=X1 o A<=B (Less or Equal)
; seleziona X1 (B) a X0 se A>B
        csel    x0, x0, x1, le
        ret
```

MIPS

Sfortunatamente GCC 4.4.5 per MIPS non è altrettanto bravo:

Listing 1.142: Con ottimizzazione GCC 4.4.5 (IDA)

```
my_max:
; se $a1<$a0 imposta $v1 a 1 $a1<$a0, altrimenti pulisci (se $a1>$a0):
                slt     $v1, $a1, $a0
; se $v1 è 0 (o $a1>$a0), salta:
                beqz    $v1, locret_10
; this is branch delay slot
; carica $a1 in $v0 nel caso il salto sia attivato:
                move    $v0, $a1
; se il salto non è attivato, carica $a0 in $v0:
                move    $v0, $a0
```

```
locret_10:
                jr      $ra
                or      $at, $zero ; branch delay slot, NOP

; la funzione min() è uguale, ma gli operandi in input nell'
; istruzione SLT sono scambiati:
my_min:
                slt     $v1, $a0, $a1
                beqz    $v1, locret_28
                move    $v0, $a1
                move    $v0, $a0

locret_28:
                jr      $ra
                or      $at, $zero ; branch delay slot, NOP
```

Non ci dimentichiamo dei *branch delay slot*: la prima MOVE è eseguita *prima* di BEQZ, la seconda MOVE viene eseguita solo se il branch non è stato seguito.

1.18.5 Conclusione

x86

La forma grezza di un jump condizionale è la seguente:

Listing 1.143: x86

```
CMP register, register/value
Jcc true ; cc=condition code
false:
;... codice da eseguire se il risultato del confronto è false ...
JMP exit
true:
;... codice da eseguire se il risultato del confronto è true ...
exit:
```

ARM

Listing 1.144: ARM

```
CMP register, register/value
Bcc true ; cc=condition code
false:
;... codice da eseguire se il risultato del confronto è false ...
JMP exit
true:
;... codice da eseguire se il risultato del confronto è true ...
exit:
```

MIPS

Listing 1.145: Check for zero

```
BEQZ REG, label
...
```

Listing 1.146: Check for less than zero (using pseudoinstruction)

```
BLTZ REG, label
...
```

Listing 1.147: Check for equal values

```
BEQ REG1, REG2, label
...
```

Listing 1.148: Check for non-equal values

```
BNE REG1, REG2, label
...
```

Listing 1.149: Check for less than (signed)

```
SLT REG1, REG2, REG3
BEQ REG1, label
...
```

Listing 1.150: Check for less than (unsigned)

```
SLTU REG1, REG2, REG3
BEQ REG1, label
...
```

Branchless

Se il corpo di uno statement condizionale è molto piccolo, può essere utilizzata l'istruzione "move" condizionale: `MOVcc` in ARM (in ARM mode), `CSEL` in ARM64, `CMOVcc` in x86.

ARM

In ARM è possibile usare suffissi condizionali per alcune istruzioni:

Listing 1.151: ARM (Modalità ARM)

```
CMP register, register/value
instr1_cc ; istruzione che sarà eseguita se il condition code è true
instr2_cc ; altra istruzione che sarà eseguita se il condition code è true
;... etc...
```

Ovviamente non c'è limite al numero di istruzioni con il suffisso condizionale, a patto che le flag CPU non siano modificate da nessuna istruzione.

La modalità Thumb ha l'istruzione IT, che permette di aggiungere suffissi condizionali alle prossime quattro istruzioni. Maggiori informazioni qui: **??** on page ??.

Listing 1.152: ARM (Modalità Thumb)

```
CMP register, register/value
ITEEE EQ ; imposta questi suffissi: if-then-else-else-else
instr1   ; istruzione da eseguire se la condizione è true
instr2   ; istruzione da eseguire se la condizione è false
instr3   ; istruzione da eseguire se la condizione è false
instr4   ; istruzione da eseguire se la condizione è false
```

1.18.6 Esercizio

(ARM64) Prova a riscrivere il codice in listato.1.132 rimuovendo tutti i jump condizionali e usando al loro posto l'istruzione CSEL instruction.

1.19 Software cracking

La maggior parte dei software può essere craccata in questo modo — cercando la reale posizione dove la protezione viene controllata, un dongle (**??** on page ??), license key, serial number, etc.

Solitamente è del tipo:

```
...
call check_protection
jz   all_OK
call message_box_protection_missing
call exit
all_OK:
; procedi
...
```

Quindi se vedete una patch (o "crack"), che cracca un software, e questa patch rimpiazza i byte 0x74/0x75 (JZ/JNZ) con 0xEB (JMP), è tutto qui.

Il processo di cracking del software si riduce alla ricerca di quel JMP.

Ci sono anche casi, dove un software controlla la protezione ogni tanto, questo può essere un dongle, o un server di licenza può richiederlo attraverso internet. In questo caso bisogna cercare la funzione che controlla la protezione. Quindi per applicare una patch, inserire `xor eax, eax / retn`, o `mov eax, 1 / retn`.

E' importante capire che dopo aver applicato una patch all'inizio di una funzione, di solito, un garbage esegue queste due istruzioni. Il garbage è composto da una parte di un' istruzione e diverse altre succesive.

Questo è un caso reale. L'inizio della funzione che vogliamo *rimpiazzare* con `return 1;`

Listing 1.153: Prima

```
8BFF                    mov         edi,edi
55                      push        ebp
8BEC                    mov         ebp,esp
81EC68080000            sub         esp,000000868
A110C00001              mov         eax,[00100C010]
33C5                    xor         eax,ebp
8945FC                  mov         [ebp][-4],eax
53                      push        ebx
8B5D08                  mov         ebx,[ebp][8]
...
```

Listing 1.154: Dopo

```
B801000000              mov         eax,1
C3                      retn
EC                      in          al,dx
68080000A1              push        0A1000008
10C0                    adc         al,al
0001                    add         [ecx],al
33C5                    xor         eax,ebp
8945FC                  mov         [ebp][-4],eax
53                      push        ebx
8B5D08                  mov         ebx,[ebp][8]
...
```

Diverse istruzioni sbagliate appaiono — `IN`, `PUSH`, `ADC`, `ADD`, dopo che, il disassemblatore Hiew (che ho appena usato) ha sincronizzato e continuato a disassemblare tutto ilresto.

Ciò non è importante — tutte queste istruzioni successivve a `RETN` non saranno mai eseguited, a meno che da qualche parte avvenga un salto diretto, e questo non dovrebbe essere possibile in generale.

Inoltre, potrebbe essere presente una variabile booleana globale, con una flag, se il software è registrato oppure no.

```
init_etc proc
...
call check_protection_or_license_file
mov  is_demo, eax
...
retn
init_etc endp

...

save_file proc
...
```

```
mov  eax, is_demo
cmp  eax, 1
jz   all_OK1

call message_box_it_is_a_demo_no_saving_allowed
retn

:all_OK1
; continua a salvare il file

...

save_proc endp

somewhere_else proc

mov  eax, is_demo
cmp  eax, 1
jz   all_OK

; controlla se siamo in esecuzione da 15 minuti
; esci se è così
; o mostra sullo schermo qualcosa di fastidioso

:all_OK2
; continua

somewhere_else endp
```

All'inizio di una funzione `check_protection_or_license_file()` si potrebbe applicare una patch, cosicchè ritorni sempre 1 o se conviene, per diverse ragioni, applicare una patch a tutte le funzioni JZ/JNZ.

Altro sulle patch: **??**.

1.20 Impossible shutdown practical joke (Windows 7)

Non ricordo quasi più come ho trovato la funzione `ExitWindowsEx()` nel file user32.dll di Windows (era la fine degli anni '90). Probabilmente, notai solo in suo nome autoesplicativo. E poi provai a *bloccarla* applicando una patch al suo inizio con il byte 0xC3 (RETN).

Il resultato fu divertente: Windows 98 non poteva più essere spento. Dovetti premere il tasto reset.

Recentemente ho provato a replicare su Windows 7, il quale è stato creato quasi 10 anni dopo e si basa su una base Windows NT completamente diversa. Ancora la funzione `ExitWindowsEx()` è presente nel file user32.dll soddisfa lo stesso compito.

Innanzitutto, ho spento la *Windows File Protection* aggiungendola al registro (Windows would silently restore modified system files otherwise):

```
Windows Registry Editor Version 5.00

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon]
"SFCDisable"=dword:ffffff9d
```

Poi ho rinominato `c:\windows\system32\user32.dll` in `user32.dll.bak`. Ho trovato `ExitWindowsEx()` usando Hiew (IDA può essere altrettanto utile) e ho inserito il byte 0xC3. Ho riavviato Windows 7 e ora non può più essere spento. I pulsanti "Restart" e "Logoff" non funzionano più.

Non so se è ancora divertente, ma alla fine degli anni '90, un mio amico copiò il file user32.dll con la patch su un floppy disk e lo mise in tutti i computer (al quale poteva accedere, che avevano Windows 98 (quasi tutti)) nella sua università. Nessun Windows poteva essere più spento e il suo insegnante di informatica si spaventò a morte. (Nel caso stesse leggendo, speriamo ci possa perdonare.)

Se volete farlo, eseguite un backup completo. L' idea migliore sarebbe quella di eseguire Windows in una macchina virtuale.

1.21 switch()/case/default

1.21.1 Pochi casi

```
#include <stdio.h>

void f (int a)
{
    switch (a)
    {
    case 0: printf ("zero\n"); break;
    case 1: printf ("one\n"); break;
    case 2: printf ("two\n"); break;
    default: printf ("something unknown\n"); break;
    };
};

int main()
{
    f (2); // test
};
```

x86

Senza ottimizzazione MSVC

Risultato (MSVC 2010):

Listing 1.155: MSVC 2010

```
tv64 = -4 ; size = 4
_a$ = 8   ; size = 4
_f    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    eax, DWORD PTR _a$[ebp]
    mov    DWORD PTR tv64[ebp], eax
    cmp    DWORD PTR tv64[ebp], 0
    je     SHORT $LN4@f
    cmp    DWORD PTR tv64[ebp], 1
    je     SHORT $LN3@f
    cmp    DWORD PTR tv64[ebp], 2
    je     SHORT $LN2@f
    jmp    SHORT $LN1@f
$LN4@f:
    push   OFFSET $SG739 ; 'zero', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN7@f
$LN3@f:
    push   OFFSET $SG741 ; 'one', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN7@f
$LN2@f:
    push   OFFSET $SG743 ; 'two', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN7@f
$LN1@f:
    push   OFFSET $SG745 ; 'something unknown', 0aH, 00H
    call   _printf
    add    esp, 4
$LN7@f:
    mov    esp, ebp
    pop    ebp
    ret    0
_f    ENDP
```

La nostra funzione con pochi casi nello switch() è praticamente analogo a questa costruzione:

```
void f (int a)
{
    if (a==0)
        printf ("zero\n");
    else if (a==1)
        printf ("one\n");
    else if (a==2)
        printf ("two\n");
    else
        printf ("something unknown\n");
};
```

Nel caso di switch() con un piccolo numero di casi, è impossibile essere sicuri che nel sorgente originale ci fosse veramente la funzione switch() o soltanto una serie di costrutti if().

Ciò vuol dire che switch() si comporta come ”zucchero sintattico”, equivalente ad un alto numero di if() annidati.

Dal nostro punto di vista non c’è niente di particolarmente nuovo nel codice generato, con l’eccezione dello spostamento (da parte del compilatore) della variabile in input a in una variabile locale temporanea `tv64` [94].

Se compiliamo questo codice con 4.4.1 otteniamo pressoché lo stesso risultato, anche con il maggior livello di ottimizzazione (`-03` option).

Con ottimizzazione MSVC

Ora attiviamo l’ottimizzaione in MSVC (`/0x`): `cl 1.c /Fa1.asm /0x`

Listing 1.156: MSVC

```
_a$ = 8 ; size = 4
_f    PROC
    mov    eax, DWORD PTR _a$[esp-4]
    sub    eax, 0
    je     SHORT $LN4@f
    sub    eax, 1
    je     SHORT $LN3@f
    sub    eax, 1
    je     SHORT $LN2@f
    mov    DWORD PTR _a$[esp-4], OFFSET $SG791 ; 'something unknown', 0aH,
    00H
    jmp    _printf
$LN2@f:
    mov    DWORD PTR _a$[esp-4], OFFSET $SG789 ; 'two', 0aH, 00H
    jmp    _printf
$LN3@f:
    mov    DWORD PTR _a$[esp-4], OFFSET $SG787 ; 'one', 0aH, 00H
    jmp    _printf
$LN4@f:
    mov    DWORD PTR _a$[esp-4], OFFSET $SG785 ; 'zero', 0aH, 00H
    jmp    _printf
_f    ENDP
```

Qui possiamo vedere un po’ di trucchetti.

Primo: il valore di a è messo in EAX, e gli viene sottratto 0. Sembra assurdo, ma è fatto per controllare se il valore in EAX è 0. Se si, il flag ZF viene settato (i.e. sottrarre 0 a 0 da 0) e il primo jump condizionale JE (*Jump if Equal* o il sinonimo JZ —*Jump if Zero*) è innescato e il controllo del flusso passato alla label `$LN4@f`, dove il messaggio

[94] Le variabili locali nello stack hanno il prefisso `tv`— MSVC nomina così le varibili locali per i suoi scopi

`'zero'` viene stampato. Se il primo jump non viene innescato, 1 viene sottratto dal valore in input e se ad un certo punto il risultato è 0, il jump corrispondente viene innescato.

Se nessun jump viene innescato, il controllo del flusso passa a `printf()` con l'argomento
`'something unknown'`.

Secondo: notiamo qualcosa di inusuale per noi: un puntatore a stringa viene messo nella variabile a, e successivamente viene chiamata `printf()` non tramite `CALL`, ma via `JMP`. C'è una spiegazione semplice dietro ciò: il chiamante mette un valore sullo stack e chiama la nostra funzione tramite `CALL`. La stessa `CALL` fa il push del return address (RA) sullo stack e fa un salto non condizionale all'indirizzo della nostra funzione. La nostra funzione, in qualunque punto della sua esecuzione (poiché non contiene istruzioni che spostano lo stack pointer) ha il seguente layout dello stack:

- ESP—punta a RA
- ESP+4—punta alla variabile a

Dall'altro lato, quando dobbiamo chiamare `printf()` qui abbiamo bisogno esattamente dello stesso layout dello stack, eccetto per il primo argomento di `printf()`, che deve puntare alla stringa. E questo è ciò che fa il codice.

Sostituisce il primo argomento della funzione con l'indirizzo della stringa, e salta a `printf()`, come se non avessimo chiamato la nostra funzione `f()`, ma direttamente `printf()`. `printf()` stampa una stringa su stdout e successivamente esegue l'istruzione RET, che fa il POP del RA dallo stack, e il controllo del flusso viene restituito non a `f()` ma al chiamante di `f()`, bypassando la fine della funzione `f()`.

Tutto ciò è possibile perchè `printf()` è chiamata proprio alla fine della funzione `f()` in tutti i casi. In qualche modo è simile alla funzione `longjmp()`[95] function. E, ovviamente, tutto ciò viene fatto a favore della velocità di esecuzione.

Un simile caso con il compilatore ARM è descritto nella sezione «printf() con più argomenti», qui (1.11.2 on page 73).

[95] wikipedia

OllyDbg

Visto che questo esempio è ingannevole, seguiamolo da OllyDbg.

OllyDbg può rilevare tali costrutti switch () e può aggiungere alcuni commenti utili. EAX vale 2 all' inizio, che è il valore di input della funzione:

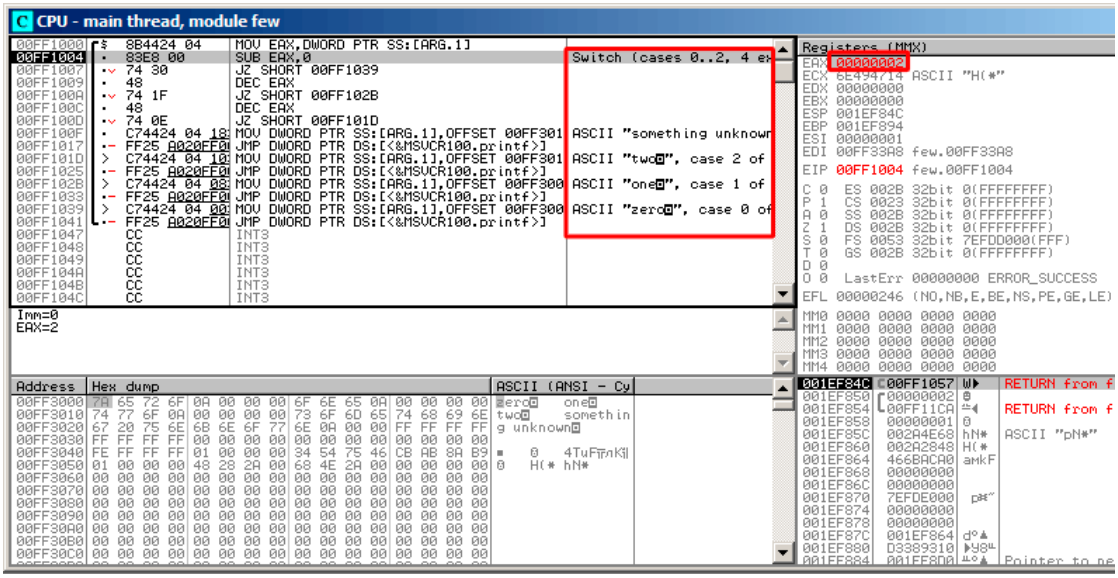


Figura 1.42: OllyDbg: EAX ora contiente il primo (e solo) argomento della funzione

0 è sottrato da 2 in EAX. Ovviamente, EAX contiene ancora 2. Ma lo ZF flag è ora 0, ad indicare che il risultato è diverso da zero:

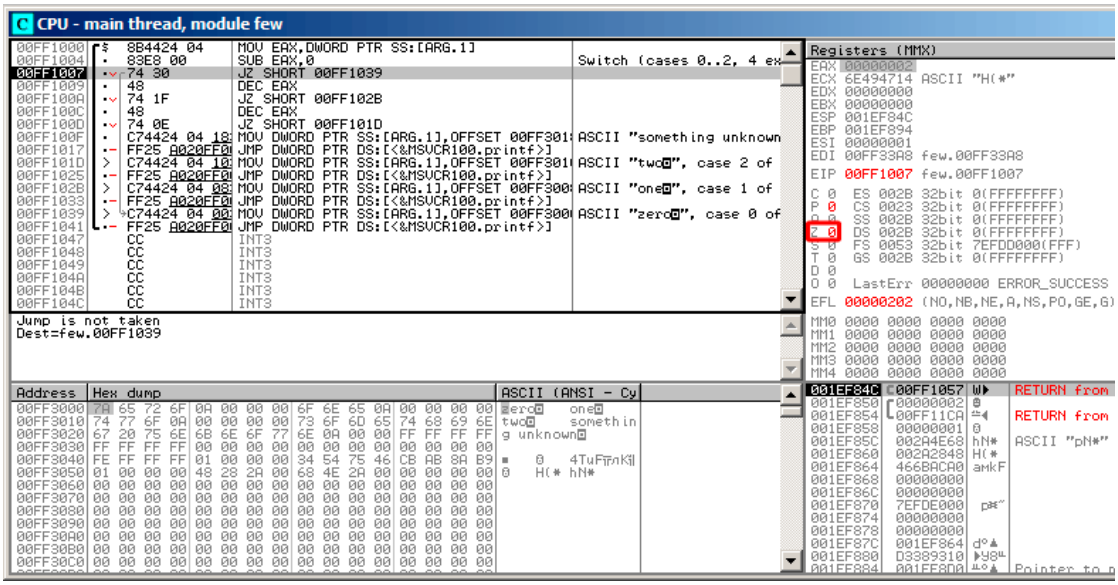


Figura 1.43: OllyDbg: SUB eseguito

E’ stato eseguito DEC e ora EAX contiene 1. Ma 1 è diverso da zero, quindi lo ZF flag è ancora 0:

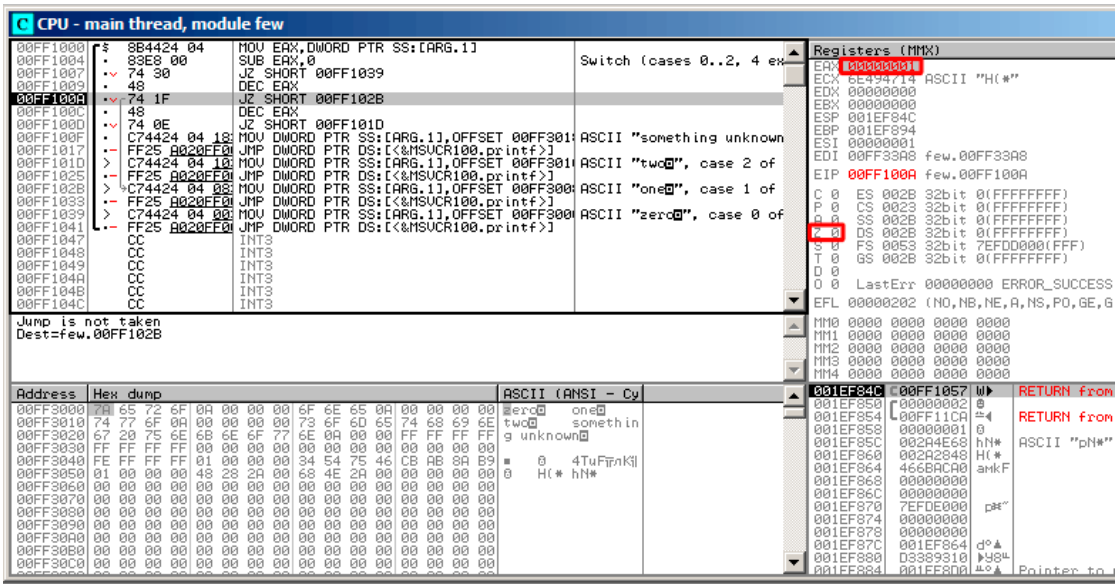


Figura 1.44: OllyDbg: primo DEC eseguito

Il seguente DEC è stato eseguito. EAX è finalmente 0 e lo ZF flag viene impostato, perchè il risultato è zero:

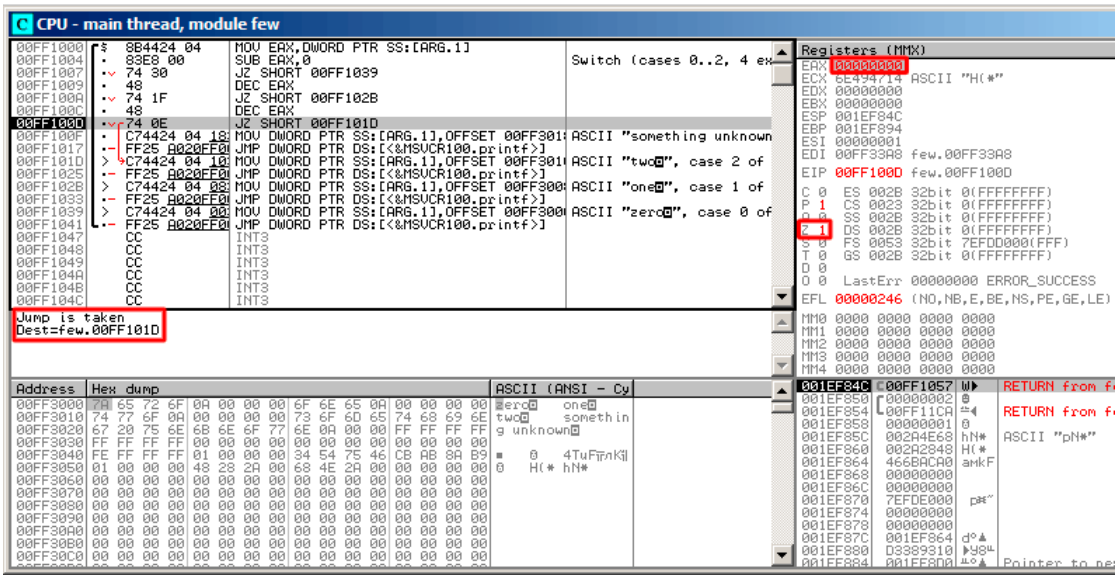


Figura 1.45: OllyDbg: secondo DEC eseguito

OllyDbg mostra che questo salto ora verrà eseguito.

Ora, il puntatore alla stringa «two», verrà scritto nello stack:

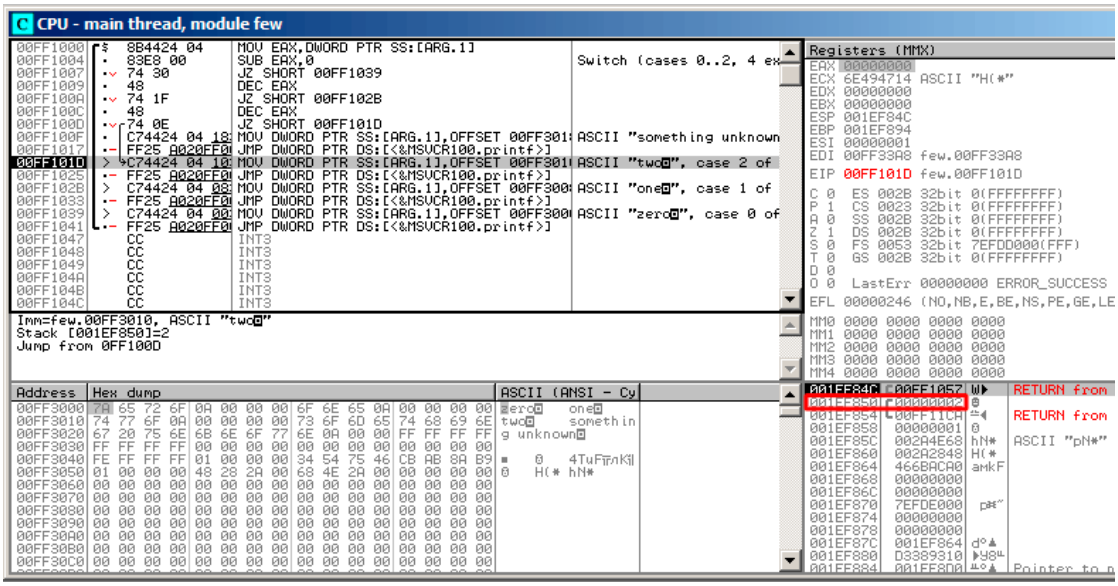


Figura 1.46: OllyDbg: il puntatore alla stringa verrà scritto nel posto del primo argomento

Nota: l'argomento corrente della funzione è 2 e 2 ora si trova nello stack all'indirizzo `0x001EF850`.

MOV scrive scrive il puntatore alla stringa all' indirizzo `0x001EF850` (notare la finestra dello stack). Dopodichè, avviene il salto. Questa è la prima istruzione della funzione `printf()` in MSVCR100.DLL (Questo esempio è stato compilato con lo switch /MD):

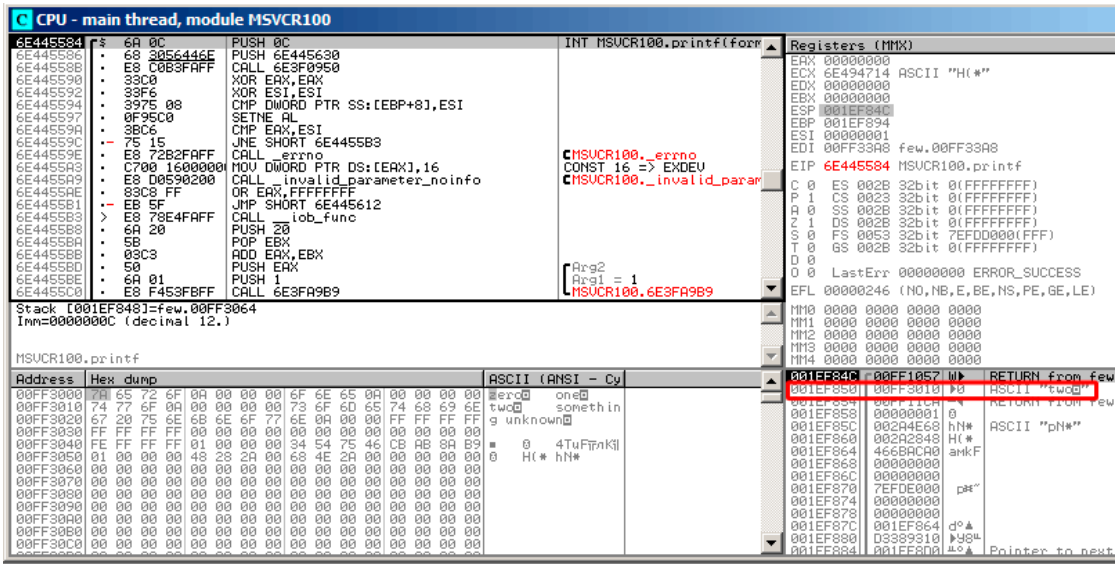


Figura 1.47: OllyDbg: prima istruzione della `printf()` in MSVCR100.DLL

Ora la `printf()` tratta la stringa a `0x00FF3010` come unico argomento e stampa la strunga.

Questa è l'ultima istruzione della `printf()`:

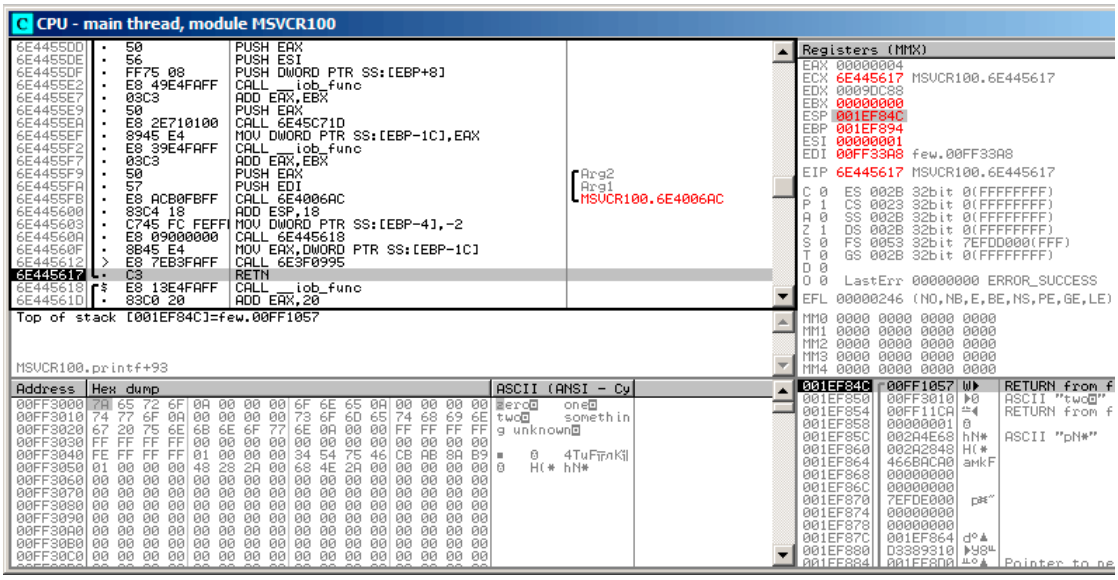


Figura 1.48: OllyDbg: ultima istruzione della `printf()` in MSVCR100.DLL

La stringa «two» è appena stata stampata nella finestra della console.

Ora premiamo F7 o F8 (step over) e ritorniamo...non nella `f()`, ma piuttosto nel `main()`:

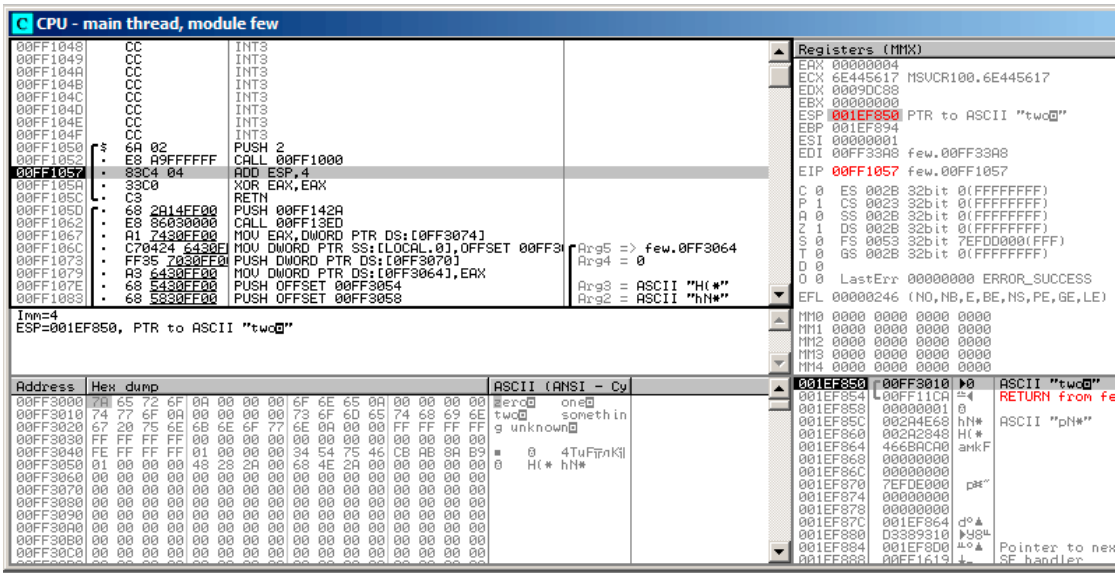


Figura 1.49: OllyDbg: ritorno al `main()`

Si, il salto è stato diretto, dalla `printf()` al `main()`. Perchè RA nello stack, non punta da qualche parte dentro `f()`, ma piuttosto nel `main()`. E `CALL 0x00FF1000` è stata l'effettiva istruzione che ha chiamato `f()`.

ARM: Con ottimizzazione Keil 6/2013 (Modalità ARM)

```
.text:0000014C                f1:
.text:0000014C 00 00 50 E3    CMP     R0, #0
.text:00000150 13 0E 8F 02    ADREQ   R0, aZero ; "zero\n"
.text:00000154 05 00 00 0A    BEQ     loc_170
.text:00000158 01 00 50 E3    CMP     R0, #1
.text:0000015C 4B 0F 8F 02    ADREQ   R0, aOne  ; "one\n"
.text:00000160 02 00 00 0A    BEQ     loc_170
.text:00000164 02 00 50 E3    CMP     R0, #2
.text:00000168 4A 0F 8F 12    ADRNE   R0, aSomethingUnkno ; "something
    unknown\n"
.text:0000016C 4E 0F 8F 02    ADREQ   R0, aTwo  ; "two\n"
.text:00000170
.text:00000170                loc_170: ; CODE XREF: f1+8
.text:00000170                         ; f1+14
.text:00000170 78 18 00 EA    B        __2printf
```

Nuovamente, analizzando questo codice, non possiamo dire con certezza se originariamente nel sorgente ci fosse uno vero e proprio switch o una serie di istruzioni if().

Ad ogni modo vediamo istruzioni condizionali (dette anche *predicated instructions*) come ADREQ (*Equal*) che viene eseguita solo nel caso $R0 = 0$, e carica l'indirizzo della stringa *«zero\n»* into R0. La successiva istruzione BEQ redirige il controllo del flusso a `loc_170`, se $R0 = 0$.

Un lettore attento potrebbe chiedersi se BEQ sarà attivata correttamente, dal momento che ADREQ ha riempito prima il registro R0 con un altro valore.

Si, sarà eseguita correttamente perchè BEQ controlla i flag settati dall'istruzione CMP, e ADREQ non modifica alcun flag.

Il resto delle istruzioni ci sono già familiari. C'è solo una chiamata a `printf()`, alla fine, ed abbiamo già esaminato questo trucco qui (1.11.2 on page 73). A conti fatti, ci sono tre percorsi che portano alla `printf()`.

L'ultima istruzione, `CMP R0, #2`, è necessria per controllare se $a = 2$.

Se la condizione non è vera, allora ADRNE carica un puntatore alla stringa *«something unknown \n»* nel registro R0, poiché la variabile a è stata già confrontata 0 e 1 e siamo certi, a questo punto, che non sia uguale a tali valori. E se $R0 = 2$, il puntatore alla stringa *«two\n»* sarà caricato in R0 da ADREQ.

ARM: Con ottimizzazione Keil 6/2013 (Modalità Thumb)

```
.text:000000D4                f1:
.text:000000D4 10 B5          PUSH    {R4,LR}
.text:000000D6 00 28          CMP     R0, #0
.text:000000D8 05 D0          BEQ     zero_case
.text:000000DA 01 28          CMP     R0, #1
.text:000000DC 05 D0          BEQ     one_case
.text:000000DE 02 28          CMP     R0, #2
.text:000000E0 05 D0          BEQ     two_case
.text:000000E2 91 A0          ADR     R0, aSomethingUnkno ; "something
    unknown\n"
.text:000000E4 04 E0          B       default_case

.text:000000E6                zero_case: ; CODE XREF: f1+4
.text:000000E6 95 A0          ADR     R0, aZero ; "zero\n"
.text:000000E8 02 E0          B       default_case

.text:000000EA                one_case: ; CODE XREF: f1+8
.text:000000EA 96 A0          ADR     R0, aOne ; "one\n"
.text:000000EC 00 E0          B       default_case

.text:000000EE                two_case: ; CODE XREF: f1+C
.text:000000EE 97 A0          ADR     R0, aTwo ; "two\n"
.text:000000F0                default_case ; CODE XREF: f1+10
.text:000000F0                                     ; f1+14
.text:000000F0 06 F0 7E F8    BL      __2printf
.text:000000F4 10 BD          POP     {R4,PC}
```

Come già detto in precedenza, non è possibile aggiungere predicati condizionali alla maggior parte di istruzioni in modalità Thumb, pertanto il codice Thumb qui mostrato è piuttosto simile a quello x86 CISC-style facilmente comprensibile.

ARM64: Senza ottimizzazione GCC (Linaro) 4.9

```
.LC12:
        .string "zero"
.LC13:
        .string "one"
.LC14:
        .string "two"
.LC15:
        .string "something unknown"
f12:
        stp     x29, x30, [sp, -32]!
        add     x29, sp, 0
        str     w0, [x29,28]
        ldr     w0, [x29,28]
        cmp     w0, 1
        beq     .L34
        cmp     w0, 2
        beq     .L35
        cmp     w0, wzr
        bne     .L38            ; salta alla sezione di default
        adrp    x0, .LC12       ; "zero"
        add     x0, x0, :lo12:.LC12
        bl      puts
        b       .L32
.L34:
        adrp    x0, .LC13       ; "one"
        add     x0, x0, :lo12:.LC13
        bl      puts
        b       .L32
.L35:
        adrp    x0, .LC14       ; "two"
        add     x0, x0, :lo12:.LC14
        bl      puts
        b       .L32
.L38:
        adrp    x0, .LC15       ; "something unknown"
        add     x0, x0, :lo12:.LC15
        bl      puts
        nop
.L32:
        ldp     x29, x30, [sp], 32
        ret
```

Il tipo di valore in input è *int*, perciò per memorizzarlo viene usato il registro `W0` anziché l'intero registro `X0`.

I puntatori alle stringhe sono passati a `puts()` tramite una coppia di istruzioni `ADRP`/`ADD` secondo quanto già dimostrato nell'esempio «Hello, world!»: 1.5.3 on page 32.

ARM64: Con ottimizzazione GCC (Linaro) 4.9

```
f12:
```

```
        cmp     w0, 1
        beq     .L31
        cmp     w0, 2
        beq     .L32
        cbz     w0, .L35
; caso di default
        adrp    x0, .LC15       ; "something unknown"
        add     x0, x0, :lo12:.LC15
        b       puts
.L35:
        adrp    x0, .LC12       ; "zero"
        add     x0, x0, :lo12:.LC12
        b       puts
.L32:
        adrp    x0, .LC14       ; "two"
        add     x0, x0, :lo12:.LC14
        b       puts
.L31:
        adrp    x0, .LC13       ; "one"
        add     x0, x0, :lo12:.LC13
        b       puts
```

Codice maggiormente ottimizzato. L'istruzione CBZ (*Compare and Branch on Zero*) salta se W0 è zero. C'è anche un salto diretto a puts() invece di una chiamata, come spiegato in precedenza: 1.21.1 on page 201.

MIPS

Listing 1.157: Con ottimizzazione GCC 4.4.5 (IDA)

```
f:
                lui     $gp, (__gnu_local_gp >> 16)
; vale 1?
                li      $v0, 1
                beq     $a0, $v0, loc_60
                la      $gp, (__gnu_local_gp & 0xFFFF) ; branch delay slot
; vale 2?
                li      $v0, 2
                beq     $a0, $v0, loc_4C
                or      $at, $zero ; branch delay slot, NOP
; se è diverso da 0, salta:
                bnez    $a0, loc_38
                or      $at, $zero ; branch delay slot, NOP
; caso zero:
                lui     $a0, ($LC0 >> 16)  # "zero"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; load delay slot, NOP
                jr      $t9 ; branch delay slot, NOP
                la      $a0, ($LC0 & 0xFFFF)  # "zero" ; branch delay slot

loc_38:                                 # CODE XREF: f+1C
                lui     $a0, ($LC3 >> 16)  # "something unknown"
                lw      $t9, (puts & 0xFFFF)($gp)
```

```
                or      $at, $zero ; load delay slot, NOP
                jr      $t9
                la      $a0, ($LC3 & 0xFFFF)  # "something unknown" ; branch
    delay slot

loc_4C:                                 # CODE XREF: f+14
                lui     $a0, ($LC2 >> 16)  # "two"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; load delay slot, NOP
                jr      $t9
                la      $a0, ($LC2 & 0xFFFF)  # "two" ; branch delay slot

loc_60:                                 # CODE XREF: f+8
                lui     $a0, ($LC1 >> 16)  # "one"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; load delay slot, NOP
                jr      $t9
                la      $a0, ($LC1 & 0xFFFF)  # "one" ; branch delay slot
```

La funzione finisce sempre col chiamare `puts()`, e quindi qui vediamo un salto a `puts()` (JR: «Jump Register») invece di «jump and link». Abbiamo già discusso questo argomento qui: 1.21.1 on page 201.

Vediamo anche spesso delle istruzioni `NOP` dopo le istruzioni `LW`. Si tratta di «load delay slot»: un altro tipo di *delay slot* in MIPS.

Un'istruzione immediatamente successiva a `LW` potrebbe essere eseguita mentre `LW` carica il valore dalla memoria. Questa istruzione, comunque, non deve usare il risultato di `LW`.

Le moderne CPU MIPS hanno una funzionalità che consente di attendere, nel caso in cui l'istruzione successiva usi il risultato di `LW`, peranto questo tipo codice è piuttosto antiquato e può essere ignorato. GCC continua ad aggiungere i NOP a favore delle cpu MIPS più vecchie.

Conclusione

Uno *switch()* con un piccolo numero di casi è indistinguibile da un costrutto *if/else* , per esempio: listato.1.21.1.

1.21.2 Molti casi

Se uno statement `switch()` contiene molti casi, per il compilatore non è molto conveniente emettere codice troppo lungo con un sacco di istruzioni JE/JNE.

```
#include <stdio.h>

void f (int a)
{
    switch (a)
    {
    case 0: printf ("zero\n"); break;
    case 1: printf ("one\n"); break;
```

```
    case 2: printf ("two\n"); break;
    case 3: printf ("three\n"); break;
    case 4: printf ("four\n"); break;
    default: printf ("something unknown\n"); break;
    };
};

int main()
{
    f (2); // test
};
```

x86

Senza ottimizzazione MSVC

Con MSVC 2010 otteniamo:

Listing 1.158: MSVC 2010

```
tv64 = -4  ; dimensione = 4
_a$ = 8    ; dimensione = 4
_f    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    eax, DWORD PTR _a$[ebp]
    mov    DWORD PTR tv64[ebp], eax
    cmp    DWORD PTR tv64[ebp], 4
    ja     SHORT $LN1@f
    mov    ecx, DWORD PTR tv64[ebp]
    jmp    DWORD PTR $LN11@f[ecx*4]
$LN6@f:
    push   OFFSET $SG739 ; 'zero', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN5@f:
    push   OFFSET $SG741 ; 'one', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN4@f:
    push   OFFSET $SG743 ; 'two', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN3@f:
    push   OFFSET $SG745 ; 'three', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
```

```
$LN2@f:
    push   OFFSET $SG747 ; 'four', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN1@f:
    push   OFFSET $SG749 ; 'something unknown', 0aH, 00H
    call   _printf
    add    esp, 4
$LN9@f:
    mov    esp, ebp
    pop    ebp
    ret    0
    npad    2 ; allinea la prossima label
$LN11@f:
    DD    $LN6@f ; 0
    DD    $LN5@f ; 1
    DD    $LN4@f ; 2
    DD    $LN3@f ; 3
    DD    $LN2@f ; 4
_f    ENDP
```

Vediamo una serie di chiamate a `printf()` con vari argomenti. Hanno tutte non solo indirizzi nella memoria del processo, ma anche etichette simboliche assegnate dal compilatore. Queste label sono anche menzionate nalle tabella interna `$LN11@f`.

All'inizio della funzione, se a è maggiore di 4, il controllo del flusso è passato alla label `$LN1@f`, dove viene chiamata `printf()` con argomento `'something unknown'`.

Se invece il valore di a è minore o uguale a 4, viene moltiplicato per 4 e sommato all'indirizzo della tabella `$LN11@f`. In questo modo vengono costruiti gli indirizzi della tabella, facendo puntare esattamente all'elemento giusto per ogni caso,

Poniamo ad esempio che a sia uguale a 2. $2 * 4 = 8$ (tutti gli elementi della tabella sono indirizzi in un processo a 32-bit, perciò tutti gli elementi sono larghi 4 byte). L'indirizzo della tabella `$LN11@f + 8` corrisponde all'elemento della tabella in cui è memorizzata la label `$LN4@f`. L'istruzione `JMP` recupera quindi l'indirizzo di `$LN4@f` dalla tabella e salta.

Questa tabella è talvolta detta *jumptable* o *branch table*[96].

Successivamente la corrispondente `printf()` viene chiamata con argomento `'two'`. Letteralmente, l'istruzione `jmp DWORD PTR $LN11@f[ecx*4]` corrisponde a *salta alla DWORD che è memorizzata all'indirizzo* `$LN11@f + ecx * 4`.

npad (.1.2 on page 314) è una macro del linguaggio assembly che allinea la prossima label in modo tale che sia memorizzata ad un indirizzo allineato a 4 byte (or a 16 byte).

Ciò è molto utile in termini di performance poiché così il processore è in grado di recuperare valori a 32-bit dalla memoria attraverso il memory bus, la cache, etc., in maniera più efficiente se è allineata.

[96] L'intero metodo una volta era noto come *computed GOTO* nelle prime versioni di Fortran: wikipedia. Non è molto rilevante oggigiorno, ma che termine!

OllyDbg

Esaminiamo questo esempio con OllyDbg. Il valore di input della funzione (2) viene caricato EAX:

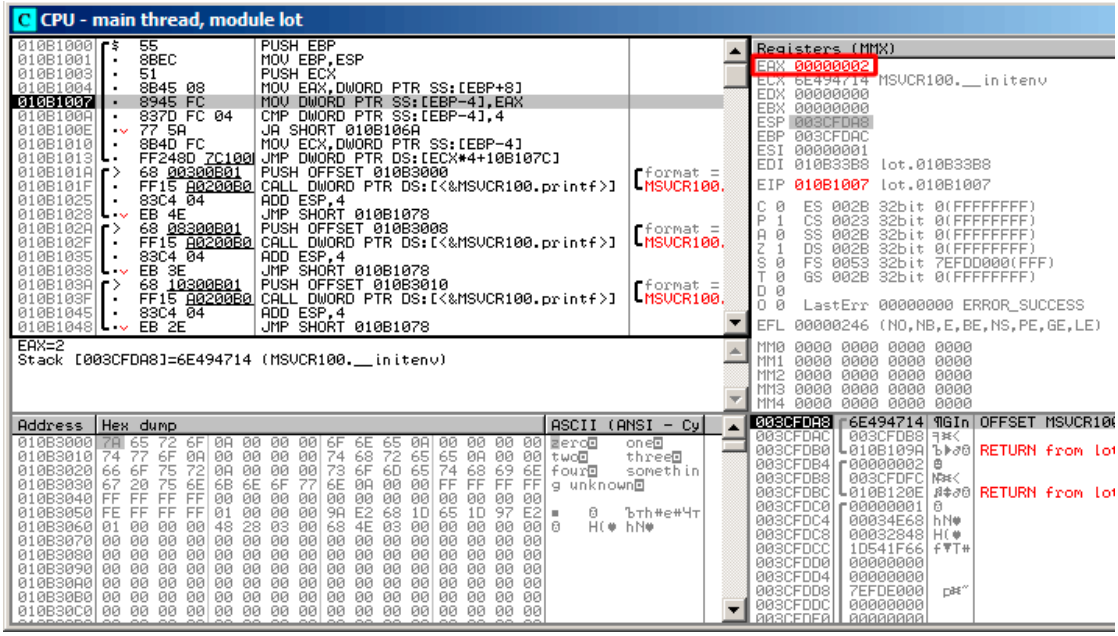


Figura 1.50: OllyDbg: il valore di input è caricato in EAX

Il valore viene controllato, è maggiore di 4? Se no, il «default» jump non viene innescato:

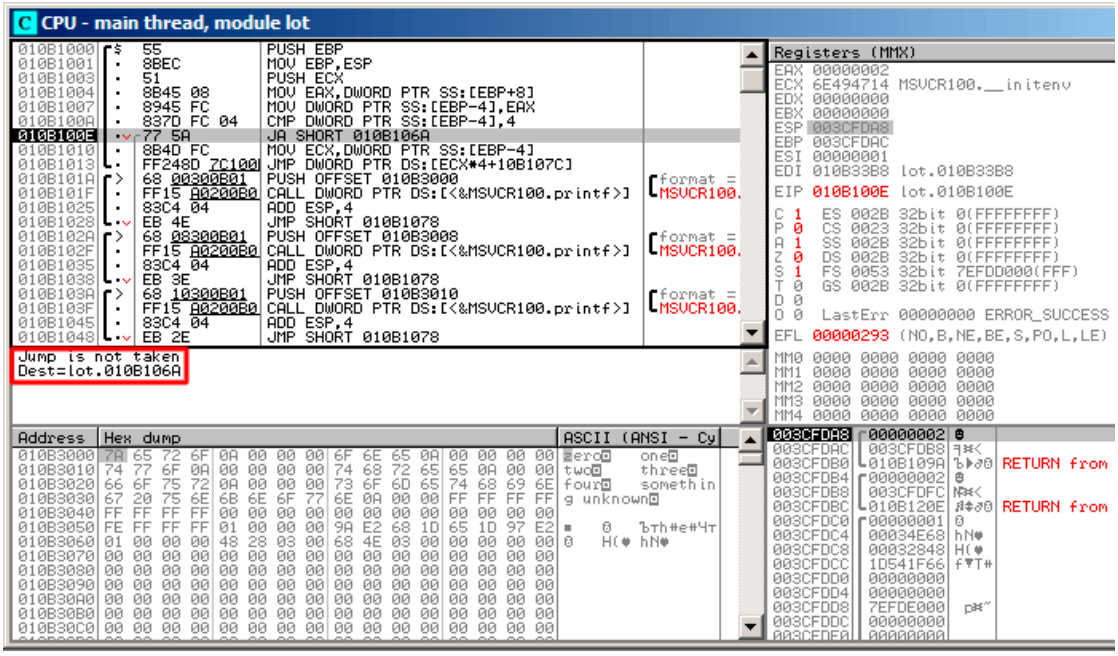


Figura 1.51: OllyDbg: 2 non è maggiore di 4: il salto non viene fatto

Qui vediamo una jumptable:

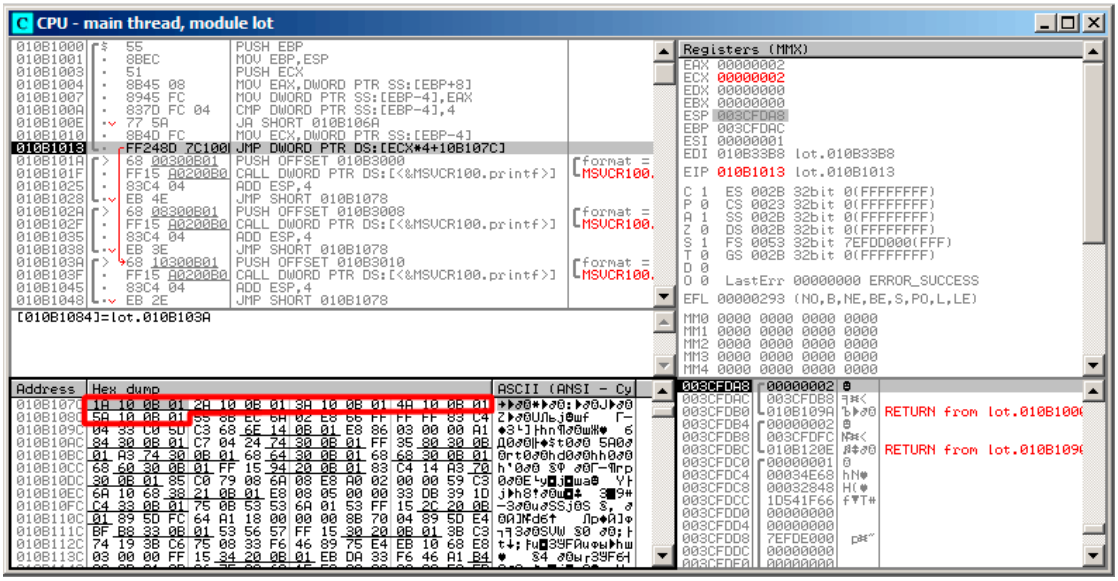


Figura 1.52: OllyDbg: calcolo dell'indirizzo di destinazione mediante jumptable

Qui abbiamo cliccato «Follow in Dump» → «Address constant», così da vedere la *jumptable* nella finestra dati. Sono 5 valori a 32-bit [97]. ECX adesso è 2, quindi il terzo elemento (avente indice 2[98]) della tabella. E' anche possibile cliccare su «Follow in Dump» → «Memory address» e OllyDbg mostrerà l'elemento a cui punta l'istruzione JMP. In questo caso è 0x010B103A.

[97] Sono sottolineati da OllyDbg poiché sono anche FIXUPs: **??** on page ??, torneremo su questo argomento più avanti

[98] Per l'indicizzazione, vedi anche: **??** on page ??

Dopo il salto ci troviamo a 0x010B103A: il codice che stampa «two» sarà ora eseguito:

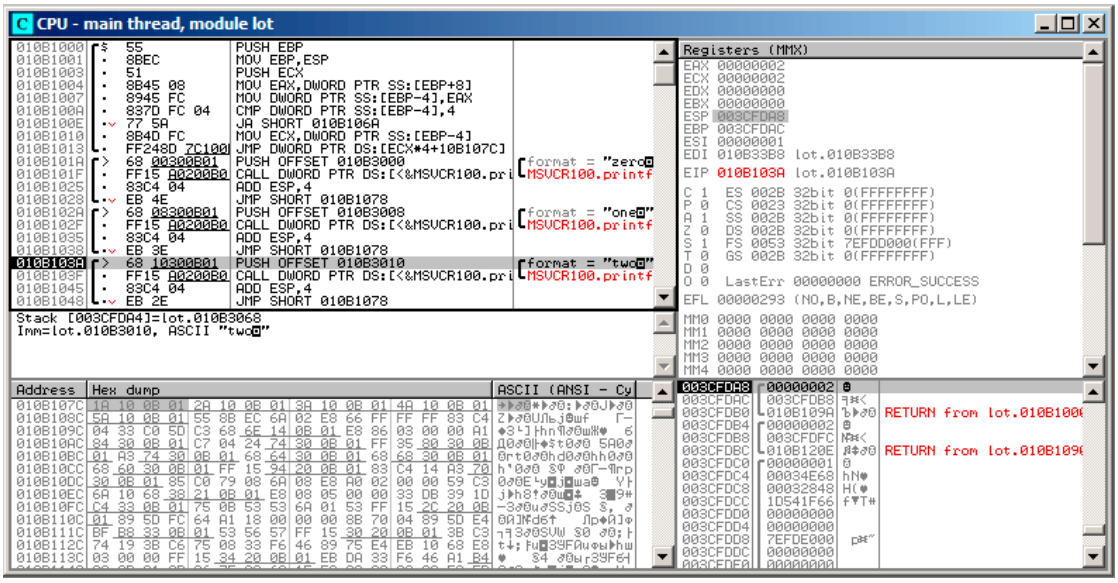


Figura 1.53: OllyDbg: ora ci troviamo alla label *case:*

Senza ottimizzazione GCC

Vediamo il codice generato da GCC 4.4.1:

Listing 1.159: GCC 4.4.1

```
        public f
f       proc near ; CODE XREF: main+10

var_18 = dword ptr -18h
arg_0  = dword ptr  8

        push    ebp
        mov     ebp, esp
        sub     esp, 18h
        cmp     [ebp+arg_0], 4
        ja      short loc_8048444
        mov     eax, [ebp+arg_0]
        shl     eax, 2
        mov     eax, ds:off_804855C[eax]
        jmp     eax

loc_80483FE: ; DATA XREF: .rodata:off_804855C
        mov     [esp+18h+var_18], offset aZero ; "zero"
        call    _puts
        jmp     short locret_8048450
```

```
loc_804840C: ; DATA XREF: .rodata:08048560
        mov     [esp+18h+var_18], offset aOne ; "one"
        call    _puts
        jmp     short locret_8048450

loc_804841A: ; DATA XREF: .rodata:08048564
        mov     [esp+18h+var_18], offset aTwo ; "two"
        call    _puts
        jmp     short locret_8048450

loc_8048428: ; DATA XREF: .rodata:08048568
        mov     [esp+18h+var_18], offset aThree ; "three"
        call    _puts
        jmp     short locret_8048450

loc_8048436: ; DATA XREF: .rodata:0804856C
        mov     [esp+18h+var_18], offset aFour ; "four"
        call    _puts
        jmp     short locret_8048450

loc_8048444: ; CODE XREF: f+A
        mov     [esp+18h+var_18], offset aSomethingUnkno ; "something unknown"
        call    _puts

locret_8048450: ; CODE XREF: f+26
                ; f+34...
        leave
        retn
f       endp

off_804855C dd offset loc_80483FE   ; DATA XREF: f+12
            dd offset loc_804840C
            dd offset loc_804841A
            dd offset loc_8048428
            dd offset loc_8048436
```

E' pressoché identico, con una leggera variazione: l'argomento `arg_0` è moltiplicato per 4 effettuando uno shift a sinistra di 2 bit (quasi identico alla moltiplicazione per 4) (1.24.2 on page 278). Successivamente l'indirizzo della label è preso dall'array `off_804855C`, memorizzato in EAX, e infine `JMP EAX` effettua il salto.

ARM: Con ottimizzazione Keil 6/2013 (Modalità ARM)

Listing 1.160: Con ottimizzazione Keil 6/2013 (Modalità ARM)

```
00000174             f2
00000174 05 00 50 E3     CMP     R0, #5           ; switch 5 cases
00000178 00 F1 8F 30     ADDCC   PC, PC, R0,LSL#2 ; switch jump
0000017C 0E 00 00 EA     B       default_case     ; jumptable 00000178
    default case

00000180
00000180             loc_180 ; CODE XREF: f2+4
```

```
00000180 03 00 00 EA     B       zero_case       ; jumptable 00000178 case 0

00000184
00000184             loc_184 ; CODE XREF: f2+4
00000184 04 00 00 EA     B       one_case        ; jumptable 00000178 case 1

00000188
00000188             loc_188 ; CODE XREF: f2+4
00000188 05 00 00 EA     B       two_case        ; jumptable 00000178 case 2

0000018C
0000018C             loc_18C ; CODE XREF: f2+4
0000018C 06 00 00 EA     B       three_case      ; jumptable 00000178 case 3

00000190
00000190             loc_190 ; CODE XREF: f2+4
00000190 07 00 00 EA     B       four_case       ; jumptable 00000178 case 4

00000194
00000194             zero_case ; CODE XREF: f2+4
00000194                       ; f2:loc_180
00000194 EC 00 8F E2     ADR     R0, aZero       ; jumptable 00000178 case 0
00000198 06 00 00 EA     B       loc_1B8

0000019C
0000019C             one_case ; CODE XREF: f2+4
0000019C                      ; f2:loc_184
0000019C EC 00 8F E2     ADR     R0, aOne        ; jumptable 00000178 case 1
000001A0 04 00 00 EA     B       loc_1B8

000001A4
000001A4             two_case ; CODE XREF: f2+4
000001A4                      ; f2:loc_188
000001A4 01 0C 8F E2     ADR     R0, aTwo        ; jumptable 00000178 case 2
000001A8 02 00 00 EA     B       loc_1B8

000001AC
000001AC             three_case ; CODE XREF: f2+4
000001AC                        ; f2:loc_18C
000001AC 01 0C 8F E2     ADR     R0, aThree      ; jumptable 00000178 case 3
000001B0 00 00 00 EA     B       loc_1B8

000001B4
000001B4             four_case ; CODE XREF: f2+4
000001B4                       ; f2:loc_190
000001B4 01 0C 8F E2     ADR     R0, aFour       ; jumptable 00000178 case 4
000001B8
000001B8             loc_1B8   ; CODE XREF: f2+24
000001B8                       ; f2+2C
000001B8 66 18 00 EA     B       __2printf

000001BC
000001BC             default_case ; CODE XREF: f2+4
```

```
000001BC                                ; f2+8
000001BC D4 00 8F E2     ADR     R0, aSomethingUnkno ; jumptable 00000178
    default case
000001C0 FC FF FF EA     B       loc_1B8
```

Il codice fa uso della modalità ARM in cui tutte le istruzioni hanno dimensione fissa di 4 byte. Ricordiamoci che il massimo valore previsto per a è 4 e ogni valore maggiore causerà la stampa della stringa *«something unknown\n»*.

La prima istruzione `CMP R0, #5` confronta il valore di a con 5.

[99] La successiva `ADDCC PC, PC, R0,LSL#2` viene eseguita solo se $R0 < 5$ (*CC=Carry clear / Less than*). Di conseguenza, se `ADDCC` non viene innescata (è il caso $R0 \geq 5$), si verificherà un jump a *default_case*.

Se invece $R0 < 5$ e `ADDCC` viene innescata, succede quanto segue:

Il valore in `R0` viene moltiplicato per 4. Infatti `LSL#2` nel suffisso dell'istruzione sta per «shift left by 2 bits» (shift a sinistra di 2 bit). Come vedremo più avanti (1.24.2 on page 278) nella sezione «Italian text placeholder», uno shift a sinistra di 2 bit equivale a moltiplicare per 4.

In seguito viene aggiunto $R0 * 4$ all'attuale valore in **PC!**, saltando quindi ad una delle istruzioni `B` (*Branch*) poste sotto.

Al momento dell'esecuzione di `ADDCC`, il valore in **PC!** si trova 8 byte più avanti (`0x180`) rispetto all'indirizzo a cui si trova l'istruzione `ADDCC` (`0x178`), o, in altre parole, 2 istruzioni più avanti.

La pipeline nei processori ARM funziona così: nel momento in cui `ADDCC` viene eseguita, il processore sta iniziando a processare l'istruzione successiva, e questo è il motivo per cui **PC!** punta a quella. Bisogna ricordarsi di ciò e tenerne conto.

Se $a = 0$, viene aggiunta al valore in **PC!**, e l'attuale valore del **PC!** sarà scritto in **PC!** (che è 8 byte più avanti) e si verificherà un salto alla label *loc_180*, che si trova 8 byte più avanti rispetto al punto in cui si trova l'istruzione `ADDCC`.

Se $a = 1$, allora $PC + 8 + a * 4 = PC + 8 + 1 * 4 = PC + 12 = 0x184$ sarà scritto in **PC!**, ovvero l'indirizzo della label *loc_184*.

Ogni volta che si aggiunge 1 ad a, il risultante **PC!** è incrementato di 4.

4 è la lunghezza delle istruzioni in modalità ARM, comprese le istruzioni `B` di cui ne abbiamo 5.

Ognuna di queste istruzioni `B` passa il controllo più avanti, a quello che era previsto nello *switch()*. Lì avviene il caricamento del puntatore alla stringa corrispondente al caso, etc.

ARM: Con ottimizzazione Keil 6/2013 (Modalità Thumb)

Listing 1.161: Con ottimizzazione Keil 6/2013 (Modalità Thumb)

```
000000F6                        EXPORT f2
```

[99] ADD—addition

```
000000F6                    f2
000000F6 10 B5                  PUSH    {R4,LR}
000000F8 03 00                  MOVS    R3, R0
000000FA 06 F0 69 F8            BL      __ARM_common_switch8_thumb ; switch 6
    cases

000000FE 05                     DCB 5
000000FF 04 06 08 0A 0C 10      DCB 4, 6, 8, 0xA, 0xC, 0x10 ; jump table for
    switch statement
00000105 00                     ALIGN 2
00000106
00000106                    zero_case ; CODE XREF: f2+4
00000106 8D A0                  ADR     R0, aZero ; jumptable 000000FA case 0
00000108 06 E0                  B       loc_118

0000010A
0000010A                    one_case ; CODE XREF: f2+4
0000010A 8E A0                  ADR     R0, aOne ; jumptable 000000FA case 1
0000010C 04 E0                  B       loc_118

0000010E
0000010E                    two_case ; CODE XREF: f2+4
0000010E 8F A0                  ADR     R0, aTwo ; jumptable 000000FA case 2
00000110 02 E0                  B       loc_118

00000112
00000112                    three_case ; CODE XREF: f2+4
00000112 90 A0                  ADR     R0, aThree ; jumptable 000000FA case 3
00000114 00 E0                  B       loc_118

00000116
00000116                    four_case ; CODE XREF: f2+4
00000116 91 A0                  ADR     R0, aFour ; jumptable 000000FA case 4
00000118
00000118                    loc_118 ; CODE XREF: f2+12
00000118                            ; f2+16
00000118 06 F0 6A F8            BL      __2printf
0000011C 10 BD                  POP     {R4,PC}

0000011E
0000011E                    default_case ; CODE XREF: f2+4
0000011E 82 A0                  ADR     R0, aSomethingUnkno ; jumptable
    000000FA default case
00000120 FA E7                  B       loc_118


000061D0                        EXPORT __ARM_common_switch8_thumb
000061D0                    __ARM_common_switch8_thumb ; CODE XREF:
    example6_f2+4
000061D0 78 47                  BX      PC

000061D2 00 00                  ALIGN 4
000061D2                    ; End of function __ARM_common_switch8_thumb
000061D2
```

```
000061D4                    __32__ARM_common_switch8_thumb ; CODE XREF:
    __ARM_common_switch8_thumb
000061D4 01 C0 5E E5           LDRB    R12, [LR,#-1]
000061D8 0C 00 53 E1           CMP     R3, R12
000061DC 0C 30 DE 27           LDRCSB  R3, [LR,R12]
000061E0 03 30 DE 37           LDRCCB  R3, [LR,R3]
000061E4 83 C0 8E E0           ADD     R12, LR, R3,LSL#1
000061E8 1C FF 2F E1           BX      R12
000061E8                    ; End of function __32__ARM_common_switch8_thumb
```

Non possiamo essere certi che tutte le istruzioni in modalità Thumb e Thumb-2 siano della stessa lunghezza. Si può dire che in queste modalità le istruzioni hanno lunghezza variabile, proprio come in x86.

E' stata aggiunta una speciale tabella che contiene informazioni su quanti casi sono previsti (escluso il default-case), ed un offset per ciascuno di essi, con una label a cui deve essere passato il controllo per il caso corrispondente.

E' anche presente una funzione speciale per gestire la tabella e passare il controllo, chiamata *__ARM_common_switch8_thumb*. Inizia con l'istruzione BX PC, la cui funzione è quella di mettere il processore in ARM-mode. Subito dopo c'è la funzione per il processamento della tabella.

E' troppo avanzata per essere analizzata adesso, e per il momento la saltiamo.

E' interessante notare che la funzione usa il registro LR come puntatore alla tabella.

Infatti, dopo la chiamata a questa funzione, LR contiene l'indirizzo subito dopo l'istruzione
BL `__ARM_common_switch8_thumb`, dove inizia appunto la tabella.

Vale anche la pena notare che il codice è generato come una funzione separata, così che possa essere riutilizzata, e il compilatore debba generare lo stesso codice per ogni istruzione switch().

IDA ha correttamente capito che si tratta di una funzione di servizio e di una tabella, ed ha aggiunto i commenti alle label come
`jumptable 000000FA case 0`.

MIPS

Listing 1.162: Con ottimizzazione GCC 4.4.5 (IDA)

```
f:
                lui     $gp, (__gnu_local_gp >> 16)
; se il valore in input è minore di 5, salta a loc_24:
                sltiu   $v0, $a0, 5
                bnez    $v0, loc_24
                la      $gp, (__gnu_local_gp & 0xFFFF) ; branch delay slot
; valore in input è maggiore o uguale a 5.
; stampa "something unknown" e termina:
                lui     $a0, ($LC5 >> 16)  # "something unknown"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
```

```
                jr      $t9
                la      $a0, ($LC5 & 0xFFFF)  # "something unknown" ; branch
    delay slot

loc_24:                                 # CODE XREF: f+8
; carica l'indirizzo della jumptable
; LA è pseudoistruzione, infatti ci sono LUI e ADDIU qui:
                la      $v0, off_120
; moltiplica il valore di input per 4:
                sll     $a0, 2
; somma il valore moltiplicato e l' indirizzo della jumptable:
                addu    $a0, $v0, $a0
; carica un elemento dalla jumptable:
                lw      $v0, 0($a0)
                or      $at, $zero ; NOP
; salta all' indirizzo che c'è nella jumptable:
                jr      $v0
                or      $at, $zero ; branch delay slot, NOP

sub_44:                                 # DATA XREF: .rodata:0000012C
; stampa "three" e termina
                lui     $a0, ($LC3 >> 16)  # "three"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC3 & 0xFFFF)  # "three" ; branch delay slot

sub_58:                                 # DATA XREF: .rodata:00000130
; stampa "four" e termina
                lui     $a0, ($LC4 >> 16)  # "four"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC4 & 0xFFFF)  # "four" ; branch delay slot

sub_6C:                                 # DATA XREF: .rodata:off_120
; stampa "zero" e termina
                lui     $a0, ($LC0 >> 16)  # "zero"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC0 & 0xFFFF)  # "zero" ; branch delay slot

sub_80:                                 # DATA XREF: .rodata:00000124
; stampa "one" e termina
                lui     $a0, ($LC1 >> 16)  # "one"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC1 & 0xFFFF)  # "one" ; branch delay slot

sub_94:                                 # DATA XREF: .rodata:00000128
; stampa "two" e termina
                lui     $a0, ($LC2 >> 16)  # "two"
```

```
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC2 & 0xFFFF)  # "two" ; branch delay slot

; può essere inserita nella sezione .rodata:
off_120:        .word sub_6C
                .word sub_80
                .word sub_94
                .word sub_44
                .word sub_58
```

La nuova istruzione che incontriamo è SLTIU («Set on Less Than Immediate Unsigned»).

E' uguale a SLTU («Set on Less Than Unsigned»), e la «I» sta per «immediate», e prevede cioè che un valore sia specificato nell'istruzione stessa.

BNEZ is «Branch if Not Equal to Zero».

Il codice è molto simile a quello di altre ISA. SLL («Shift Word Left Logical») moltiplica per 4.

MIPS è una CPU a 32-bit, e tutti gli indirizzi contenuti nella *jumptable* sono quindi a 32-bit.

Conclusione

Stuttura approssimativa di *switch()*:

Listing 1.163: x86

```
MOV REG, input
CMP REG, 4 ; numero massimo di casi
JA default
SHL REG, 2 ; trova l'elemento nella tabella. Shift di 3 bit in x64.
MOV REG, jump_table[REG]
JMP REG

case1:
    ; gestione del caso
    JMP exit
case2:
    ; gestione del caso
    JMP exit
case3:
    ; gestione del caso
    JMP exit
case4:
    ; gestione del caso
    JMP exit
case5:
    ; gestione del caso
    JMP exit
```

```
default:

    ...

exit:

    ....

jump_table dd case1
           dd case2
           dd case3
           dd case4
           dd case5
```

Il salto agli indirizzi nella tabella di jump può anche essere implementato usando questa istruzione:
`JMP jump_table[REG*4]`. oppure `JMP jump_table[REG*8]` in x64.

Una *jumptable* è semplicemente un array di puntatori, come quello descritto più avanti: **??** on page ??.

1.21.3 Ancora più istruzioni *case* in un unico blocco

Ecco un costrutto molto diffuso: molti casi in un singolo blocco:

```
#include <stdio.h>

void f(int a)
{
        switch (a)
        {
        case 1:
        case 2:
        case 7:
        case 10:
                printf ("1, 2, 7, 10\n");
                break;
        case 3:
        case 4:
        case 5:
        case 6:
                printf ("3, 4, 5\n");
                break;
        case 8:
        case 9:
        case 20:
        case 21:
                printf ("8, 9, 21\n");
                break;
        case 22:
                printf ("22\n");
                break;
```

```
        default:
                printf ("default\n");
                break;
        };
};

int main()
{
        f(4);
};
```

Generare un blocco per ciascun caso possibile risulta poco efficiente, perciò solitamente quello che viene fatto è generare ogni blocco più una sorta di smistatore (dispatcher).

MSVC

Listing 1.164: Con ottimizzazione MSVC 2010

```
1  $SG2798 DB      '1, 2, 7, 10', 0aH, 00H
2  $SG2800 DB      '3, 4, 5', 0aH, 00H
3  $SG2802 DB      '8, 9, 21', 0aH, 00H
4  $SG2804 DB      '22', 0aH, 00H
5  $SG2806 DB      'default', 0aH, 00H
6
7  _a$ = 8
8  _f      PROC
9          mov     eax, DWORD PTR _a$[esp-4]
10         dec     eax
11         cmp     eax, 21
12         ja      SHORT $LN1@f
13         movzx   eax, BYTE PTR $LN10@f[eax]
14         jmp     DWORD PTR $LN11@f[eax*4]
15 $LN5@f:
16         mov     DWORD PTR _a$[esp-4], OFFSET $SG2798 ; '1, 2, 7, 10'
17         jmp     DWORD PTR __imp__printf
18 $LN4@f:
19         mov     DWORD PTR _a$[esp-4], OFFSET $SG2800 ; '3, 4, 5'
20         jmp     DWORD PTR __imp__printf
21 $LN3@f:
22         mov     DWORD PTR _a$[esp-4], OFFSET $SG2802 ; '8, 9, 21'
23         jmp     DWORD PTR __imp__printf
24 $LN2@f:
25         mov     DWORD PTR _a$[esp-4], OFFSET $SG2804 ; '22'
26         jmp     DWORD PTR __imp__printf
27 $LN1@f:
28         mov     DWORD PTR _a$[esp-4], OFFSET $SG2806 ; 'default'
29         jmp     DWORD PTR __imp__printf
30         npad    2 ; allinea la tabella $LN11@f ad un margine di 16 byte
31 $LN11@f:
32         DD      $LN5@f ; stampa '1, 2, 7, 10'
33         DD      $LN4@f ; stampa '3, 4, 5'
34         DD      $LN3@f ; stampa '8, 9, 21'
```

```
35          DD      $LN2@f ; stampa '22'
36          DD      $LN1@f ; stampa 'default'
37  $LN10@f:
38          DB      0 ; a=1
39          DB      0 ; a=2
40          DB      1 ; a=3
41          DB      1 ; a=4
42          DB      1 ; a=5
43          DB      1 ; a=6
44          DB      0 ; a=7
45          DB      2 ; a=8
46          DB      2 ; a=9
47          DB      0 ; a=10
48          DB      4 ; a=11
49          DB      4 ; a=12
50          DB      4 ; a=13
51          DB      4 ; a=14
52          DB      4 ; a=15
53          DB      4 ; a=16
54          DB      4 ; a=17
55          DB      4 ; a=18
56          DB      4 ; a=19
57          DB      2 ; a=20
58          DB      2 ; a=21
59          DB      3 ; a=22
60  _f      ENDP
```

Notiamo due tabelle: la prima, (`$LN10@f`), è una tabella di indici. La seconda, (`$LN11@f`), è un array di puntatori ai blocchi.

Per cominciare, il valore di input è usato come indice nella tabella di indici (riga 13).

Ecco una piccola legenda di valori in questa tabella: 0 è il blocco relativo al primo *case* (per i valori 1, 2, 7, 10), 1 al secondo (per i valori 3, 4, 5), 2 al terzo (per i valori 8, 9, 21), 3 al quarto (per i valori 22), 4 è relativo al blocco default.

Da qui ottieniamo un indice per la seconda tabella di puntatori al codice, e vi saltiamo (riga 14).

Vale anche la pena notare che non c'è alcun caso per il valore 0 in input.

Per questo motivo vediamo l'istruzione DEC a riga 10, e la tabella inizia da $a = 1$, non c'è bisogno di allocare un elemento nella tabella per $a = 0$.

Questo è un pattern molto diffuso.

Perchè è economico? Perchè non è possibile avere lo stesso risultato con le tecniche viste prima (1.21.2 on page 220), solo con una tabella di puntatori ai blocchi? La risposta sta nel fatto che gli elementi nella tabella di indici sono di 8-bit, dunque è tutto più compatto.

GCC

GCC GCC utilizza la tecnica già vista (1.21.2 on page 220), usando solo una tabella di puntatori.

ARM64: Con ottimizzazione GCC 4.9.1

Non c'è codice da eseguire se il valore in input è 0, perciò GCC prova a rendere la jump table più compatta iniziando da 1 come valore di input.

GCC 4.9.1 per ARM64 usa un trucco ancora migliore. Riesce a codificare tutti gli offset con byte (8-bit).

Ricordiamoci che tutte le istruzioni ARM64 sono lunghe 4 byte.

GCC sfrutta il fatto che tutti gli offset nel nostro piccolo esempio si trovano molto vicini tra di loro. In questo modo la jump table consiste di singoli byte.

Listing 1.165: Con ottimizzazione GCC 4.9.1 ARM64

```
f14:
; valore di input in W0
        sub     w0, w0, #1
        cmp     w0, 21
; salta se è minore o uguale (unsigned):
        bls     .L9
.L2:
; stampa "default":
        adrp    x0, .LC4
        add     x0, x0, :lo12:.LC4
        b       puts
.L9:
; carica l' indirizzo della jumptable in X1:
        adrp    x1, .L4
        add     x1, x1, :lo12:.L4
; W0=input_value-1
; carica un byte dalla tabella:
        ldrb    w0, [x1,w0,uxtw]
; carica l'indirizzo della label Lrtx:
        adr     x1, .Lrtx4
; moltiplica l' elemento della tabella per 4(con uno shift di 2 bit a
    sinistra) e aggiungi (o sottrai) all indirizzo di Lrtx:
        add     x0, x1, w0, sxtb #2
; salta all' indirizzo calcolato:
        br      x0
; questa label sta puntando nel segmento di codice (text):
.Lrtx4:
        .section        .rodata
; tutto dopo lo statement ".section" è allocato nel segmento read-only data
    (rodata):
.L4:
        .byte   (.L3 - .Lrtx4) / 4     ; caso 1
        .byte   (.L3 - .Lrtx4) / 4     ; caso 2
        .byte   (.L5 - .Lrtx4) / 4     ; caso 3
        .byte   (.L5 - .Lrtx4) / 4     ; caso 4
```

```
        .byte   (.L5 - .Lrtx4) / 4      ; caso 5
        .byte   (.L5 - .Lrtx4) / 4      ; caso 6
        .byte   (.L3 - .Lrtx4) / 4      ; caso 7
        .byte   (.L6 - .Lrtx4) / 4      ; caso 8
        .byte   (.L6 - .Lrtx4) / 4      ; caso 9
        .byte   (.L3 - .Lrtx4) / 4      ; caso 10
        .byte   (.L2 - .Lrtx4) / 4      ; caso 11
        .byte   (.L2 - .Lrtx4) / 4      ; caso 12
        .byte   (.L2 - .Lrtx4) / 4      ; caso 13
        .byte   (.L2 - .Lrtx4) / 4      ; caso 14
        .byte   (.L2 - .Lrtx4) / 4      ; caso 15
        .byte   (.L2 - .Lrtx4) / 4      ; caso 16
        .byte   (.L2 - .Lrtx4) / 4      ; caso 17
        .byte   (.L2 - .Lrtx4) / 4      ; caso 18
        .byte   (.L2 - .Lrtx4) / 4      ; caso 19
        .byte   (.L6 - .Lrtx4) / 4      ; caso 20
        .byte   (.L6 - .Lrtx4) / 4      ; caso 21
        .byte   (.L7 - .Lrtx4) / 4      ; caso 22
        .text
; tutto dopo lo statement ".text" è allocato nel segmento di codice (text):
.L7:
; stampa "22"
        adrp    x0, .LC3
        add     x0, x0, :lo12:.LC3
        b       puts
.L6:
; stampa "8, 9, 21"
        adrp    x0, .LC2
        add     x0, x0, :lo12:.LC2
        b       puts
.L5:
; stampa "3, 4, 5"
        adrp    x0, .LC1
        add     x0, x0, :lo12:.LC1
        b       puts
.L3:
; stampa "1, 2, 7, 10"
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
        b       puts
.LC0:
        .string "1, 2, 7, 10"
.LC1:
        .string "3, 4, 5"
.LC2:
        .string "8, 9, 21"
.LC3:
        .string "22"
.LC4:
        .string "default"
```

Compiliamo questo esempio in un file oggetto e apriamolo con IDA. Questa è la jump table:

Listing 1.166: jumptable in IDA

```
.rodata:0000000000000064                AREA .rodata, DATA, READONLY
.rodata:0000000000000064                ; ORG 0x64
.rodata:0000000000000064 $d             DCB    9    ; case 1
.rodata:0000000000000065                DCB    9    ; case 2
.rodata:0000000000000066                DCB    6    ; case 3
.rodata:0000000000000067                DCB    6    ; case 4
.rodata:0000000000000068                DCB    6    ; case 5
.rodata:0000000000000069                DCB    6    ; case 6
.rodata:000000000000006A                DCB    9    ; case 7
.rodata:000000000000006B                DCB    3    ; case 8
.rodata:000000000000006C                DCB    3    ; case 9
.rodata:000000000000006D                DCB    9    ; case 10
.rodata:000000000000006E                DCB 0xF7    ; case 11
.rodata:000000000000006F                DCB 0xF7    ; case 12
.rodata:0000000000000070                DCB 0xF7    ; case 13
.rodata:0000000000000071                DCB 0xF7    ; case 14
.rodata:0000000000000072                DCB 0xF7    ; case 15
.rodata:0000000000000073                DCB 0xF7    ; case 16
.rodata:0000000000000074                DCB 0xF7    ; case 17
.rodata:0000000000000075                DCB 0xF7    ; case 18
.rodata:0000000000000076                DCB 0xF7    ; case 19
.rodata:0000000000000077                DCB    3    ; case 20
.rodata:0000000000000078                DCB    3    ; case 21
.rodata:0000000000000079                DCB    0    ; case 22
.rodata:000000000000007B ; .rodata ends
```

Riassumendo, nel caso 1, 9 viene moltiplicato per 4 e aggiunto all'indirizzo della label `Lrtx4`. Nel caso 22, 0 viene moltiplicato per 4, risultando 0.

Subito dopo la label `Lrtx4` si trova label L7, dove c'è il codice che stampa «22».

Non c'è alcuna jump table nel code segnment, è allocata in una sezione .rodata separata (non vi è alcun motivo particolare per cui sarebbe necessario metterla nella sezione del codice).

Ci sono anche byte negativi (0xF7), usati per saltare indietro al codice che stampa la stringa «default» (a `.L2`).

1.21.4 Fall-through

Un altro uso diffuso dell'operatore `switch()` è il cosiddetto «fallthrough». Ecco un semplice esempio [100]:

```
1 bool is_whitespace(char c) {
2         switch (c) {
3                 case ' ': // fallthrough
4                 case '\t': // fallthrough
5                 case '\r': // fallthrough
6                 case '\n':
```

[100] Preso da https://github.com/azonalon/prgraas/blob/master/prog1lib/lecture_examples/is_whitespace.c

```
7                       return true;
8               default: // not whitespace
9                       return false;
10      }
11  }
```

Uno leggermente più difficile, dal kernel di Linux [101]:

```
1   char nco1, nco2;
2
3   void f(int if_freq_khz)
4   {
5
6           switch (if_freq_khz) {
7                   default:
8                           printf("IF=%d KHz is not supportted, 3250 assumed\n↙
        ↳ ", if_freq_khz);
9                           /* fallthrough */
10                  case 3250: /* 3.25Mhz */
11                          nco1 = 0x34;
12                          nco2 = 0x00;
13                          break;
14                  case 3500: /* 3.50Mhz */
15                          nco1 = 0x38;
16                          nco2 = 0x00;
17                          break;
18                  case 4000: /* 4.00Mhz */
19                          nco1 = 0x40;
20                          nco2 = 0x00;
21                          break;
22                  case 5000: /* 5.00Mhz */
23                          nco1 = 0x50;
24                          nco2 = 0x00;
25                          break;
26                  case 5380: /* 5.38Mhz */
27                          nco1 = 0x56;
28                          nco2 = 0x14;
29                          break;
30          }
31  };
```

Listing 1.167: Optimizing GCC 5.4.0 x86

```
1   .LC0:
2           .string "IF=%d KHz is not supportted, 3250 assumed\n"
3   f:
4           sub     esp, 12
5           mov     eax, DWORD PTR [esp+16]
6           cmp     eax, 4000
7           je      .L3
8           jg      .L4
```

[101] Preso da https://github.com/torvalds/linux/blob/master/drivers/media/dvb-frontends/lgdt3306a.c

```
 9         cmp     eax, 3250
10         je      .L5
11         cmp     eax, 3500
12         jne     .L2
13         mov     BYTE PTR nco1, 56
14         mov     BYTE PTR nco2, 0
15         add     esp, 12
16         ret
17 .L4:
18         cmp     eax, 5000
19         je      .L7
20         cmp     eax, 5380
21         jne     .L2
22         mov     BYTE PTR nco1, 86
23         mov     BYTE PTR nco2, 20
24         add     esp, 12
25         ret
26 .L2:
27         sub     esp, 8
28         push    eax
29         push    OFFSET FLAT:.LC0
30         call    printf
31         add     esp, 16
32 .L5:
33         mov     BYTE PTR nco1, 52
34         mov     BYTE PTR nco2, 0
35         add     esp, 12
36         ret
37 .L3:
38         mov     BYTE PTR nco1, 64
39         mov     BYTE PTR nco2, 0
40         add     esp, 12
41         ret
42 .L7:
43         mov     BYTE PTR nco1, 80
44         mov     BYTE PTR nco2, 0
45         add     esp, 12
46         ret
```

Possiamo arrivare alla label `.L5` se all'input della funzione viene dato il valore 3250. Ma si può anche giungere allo stesso punto da un altro percorso: notiamo che non ci sono jump tra la chiamata a `printf()` e la label `.L5`.

Questo spiega facilmente perchè i costrutti con *switch()* sono spesso fonte di bug: è sufficiente dimenticare un *break* per trasformare il costrutto *switch()* in un *fallthrough* , in cui vengono eseguiti più blocchi invece di uno solo.

1.21.5 Esercizi

Esercizio#1

E' possibile riscrivere l'esempio C da 1.21.2 on page 214 in modo tale che il compilatore riesca a produrre codice ancora più breve e che funzioni allo stesso modo.

Prova a farlo.

1.22 Cicli

1.22.1 Semplice esempio

x86

Nell' instruction set x86, c'è una speciale istruzione di L00P per controllare il valore nel registro ECX e se non è 0, decrementa ECX e passa il controllo del flusso alla label nell' operando di L00P. Probabilmente questa istruzione non è molto conveniente, e non ci sono moderni compilatori che la inseriscono automaticamente. Di conseguenza, se la vedete da qualche parte, probabilmente quella parte di codice assembly è stata scritta a mano.

In C/C++ i cicli sono solitamente costruiti usando le istruzioni `for()`, `while()` o `do/while()`.

Iniziamo con `for()`.

Questa istruzione definisce l'inizializzazione del ciclo (imposta un contatore di cicli ad un valore iniziale), la condizione di ciclo (il contatore è maggiore di un valore limite?), cosa viene eseguito ad ogni iterazione (incrementa/decrementa il contatore) e ovviamente il corpo del ciclo.

```
for (inizializzazione; condizione; ad ogni iterazione)
{
    corpo_ciclo;
}
```

Anche il codice generato è composto da quattro parti.

Iniziamo con un semplice esempio:

```
#include <stdio.h>

void printing_function(int i)
{
        printf ("f(%d)\n", i);
};

int main()
{
        int i;

        for (i=2; i<10; i++)
                printing_function(i);

        return 0;
};
```

Il risultato (MSVC 2010):

Listing 1.168: MSVC 2010

```
_i$ = -4
_main    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    DWORD PTR _i$[ebp], 2   ; inizializzazione ciclo
    jmp    SHORT $LN3@main
$LN2@main:
    mov    eax, DWORD PTR _i$[ebp] ; qui c'è ciò che facciamo dopo ogni
    iterazione:
    add    eax, 1                  ; aggiungi 1 al valore (i)
    mov    DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp    DWORD PTR _i$[ebp], 10  ; questa condizione è controllata prima di
    ogni iterazione
    jge    SHORT $LN1@main         ; se (i) è maggiore o uguale a 10, il
    ciclo termina
    mov    ecx, DWORD PTR _i$[ebp] ; corpo del ciclo: call
    printing_function(i)
    push   ecx
    call   _printing_function
    add    esp, 4
    jmp    SHORT $LN2@main         ; salta all' inizio del ciclo
$LN1@main:                         ; fine del ciclo
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
```

Come possiamo vedere, nulla di speciale.

GCC 4.4.1 emette quasi lo stesso codice, con una sottile differenza:

Listing 1.169: GCC 4.4.1

```
main            proc near

var_20          = dword ptr -20h
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 20h
                mov     [esp+20h+var_4], 2  ; inizializzazione (i)
                jmp     short loc_8048476

loc_8048465:
                mov     eax, [esp+20h+var_4]
                mov     [esp+20h+var_20], eax
                call    printing_function
                add     [esp+20h+var_4], 1  ; incrementa (i)
```

```
loc_8048476:
                cmp     [esp+20h+var_4], 9
                jle     short loc_8048465   ; se i<=9, continua il ciclo
                mov     eax, 0
                leave
                retn
main            endp
```

Ora vediamo cosa ottieniamo con l'ottimizzazione impostata su (/0x):

Listing 1.170: Con ottimizzazione MSVC

```
_main    PROC
    push   esi
    mov    esi, 2
$LL3@main:
    push   esi
    call   _printing_function
    inc    esi
    add    esp, 4
    cmp    esi, 10     ; 0000000aH
    jl     SHORT $LL3@main
    xor    eax, eax
    pop    esi
    ret    0
_main    ENDP
```

Quello che abbiamo è che lo spazio per la veriabile i non è più allocato nello stack, ma viene utilizzato un registro, ESI. Questo è possibile nelle piccole funzioni dove non ci somo molte variabili locali.

Una cosa importante è che la funzione f() non deve cambiare il valore in ESI. Il nostro compilatore c'è lo assicura. E se il compilatore decide di usare il registro ESI anche nella funzione f(), il suo valore viene salvato durante il prologo della funzione e ripristinato durante l'epilogo della funzione, similmente al nostro esempio: notare PUSH ESI/POP ESI all'inizio e fine della funzione.

Proviamo GCC 4.4.1 con la massima ottimizzazione impostata (opzione -03):

Listing 1.171: Con ottimizzazione GCC 4.4.1

```
main            proc near

var_10          = dword ptr -10h

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     [esp+10h+var_10], 2
                call    printing_function
                mov     [esp+10h+var_10], 3
                call    printing_function
                mov     [esp+10h+var_10], 4
```

```
                call    printing_function
                mov     [esp+10h+var_10], 5
                call    printing_function
                mov     [esp+10h+var_10], 6
                call    printing_function
                mov     [esp+10h+var_10], 7
                call    printing_function
                mov     [esp+10h+var_10], 8
                call    printing_function
                mov     [esp+10h+var_10], 9
                call    printing_function
                xor     eax, eax
                leave
                retn
main            endp
```

Huh, GCC ha appena "srotolato" il nostro ciclo.

Srotolamente del ciclo è vantaggioso nel caso in cui non ci siano molte iterazioni, perchè possiamo ridurre il tempo di esecuzione rimuovendo tutte le istruzioni di supporto ai cicli. Dall' altro lato, il codice risultante è ovviamente maggiore.

Srotolare grossi cicli non è raccomandato al giorno d'oggi, perchè grosse funzioni possono richiedere un ingombro della cache maggiore[102].

OK, aumentiamo a 100 il massimo valore della variabile i e proviamo nuovamente. GCC fa:

Listing 1.172: GCC

```
                public main
main            proc near

var_20          = dword ptr -20h

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                push    ebx
                mov     ebx, 2    ; i=2
                sub     esp, 1Ch

; allinea la label loc_80484D0 (inizio del corpo del ciclo) con un bordo di
    16-byte:
                nop

loc_80484D0:
; passa (i) come primo argomento a printing_function():
                mov     [esp+20h+var_20], ebx
                add     ebx, 1    ; i++
                call    printing_function
```

[102] Un ottimo articolo a riguardo: [Ulrich Drepper, *What Every Programmer Should Know About Memory*, (2007)][103]. Qui ci sono altre raccomandazioni da Intel riguardo lo srotolamento dei cicli: [*Intel® 64 and IA-32 Architectures Optimization Reference Manual*, (2014)3.4.1.7].

```
                cmp     ebx, 64h  ; i==100?
                jnz     short loc_80484D0 ; altrimenti, continua
                add     esp, 1Ch
                xor     eax, eax  ; ritorna 0
                pop     ebx
                mov     esp, ebp
                pop     ebp
                retn
main            endp
```

E' abbastanza simile a quello che produce MSVC 2010 con ottimizzazione (/0x), con l'eccezione che il registro EBX è allocato per la variabile i.

GCC è sicuro che questo registro non verrà modificato nella funzione `f()` e nel caso, verrà salvato durante il prologo della funzione e verrà ripristinato durante l'epilogo, proprio come in questo caso nella funzione `main()`.

x86: OllyDbg

Compiliamo il nostro esempio con MSVC 2010 con le opzioni `/Ox` e `/Ob0`, carichiamolo poi in OllyDbg.

Sembrerebbe che OllyDbg sia in grado di rilevare dei semplici cicli e ce li mostra tra parentesi quadre, per convenienza:

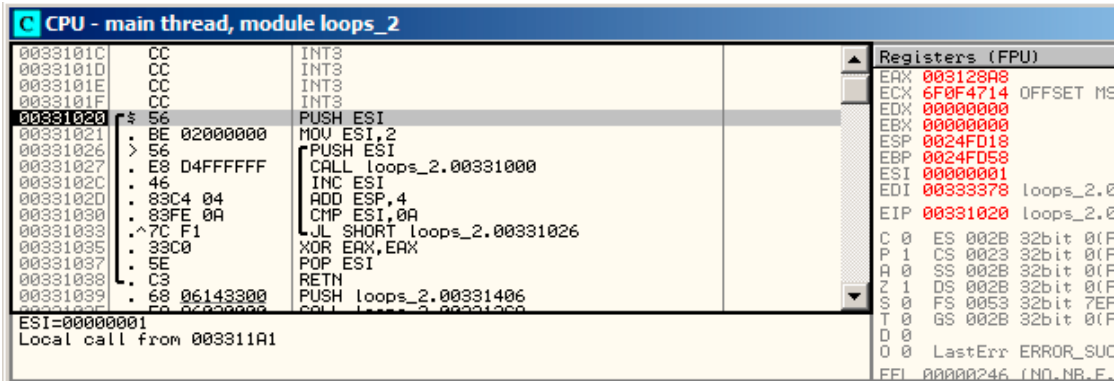


Figura 1.54: OllyDbg: inizio `main()`

Tracciando (F8 — step over) vediamo ESI incrementare. Qui, per esempio, $ESI = i = 6$:

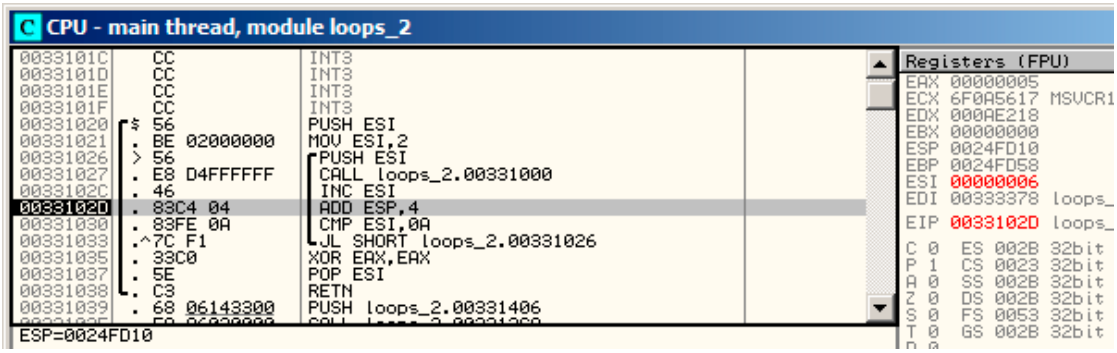


Figura 1.55: OllyDbg: corpo del ciclo appena eseguito con $i = 6$

9 è l'ultimo valore del ciclo. Motivo per il quale, JL non si attiva dopo incrementa e la funzione concluderà:

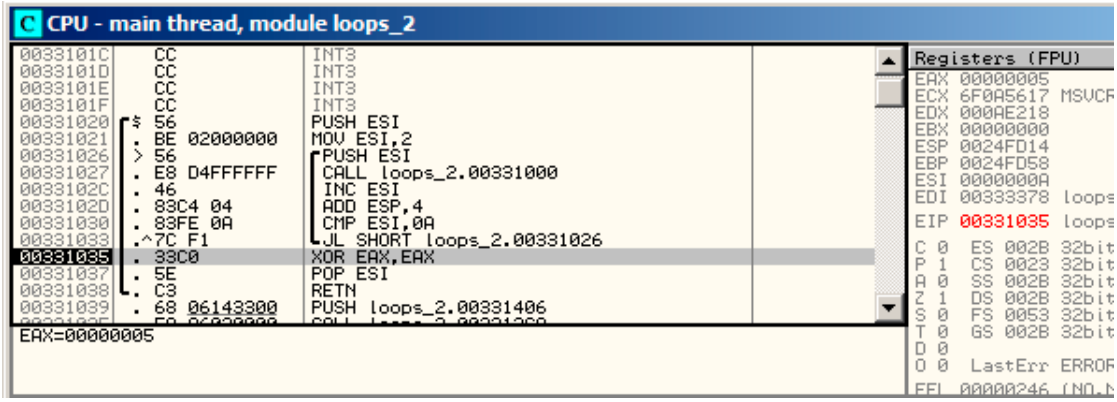


Figura 1.56: OllyDbg: $ESI = 10$, fine ciclo

x86: tracer

Come possimo vedere, non è molto comodo tracciare manualmente nel debugger. Questa è la ragione per cui proveremo ad usare Italian text placeholder.

Apriamo l'esempio compilato in IDA, cerchiamo l'indirizzo dell' istruzione `PUSH ESI` (che passa l'unico argomento a `f()`), che è `0x401026` in questo caso ed eseguiamo il Italian text placeholder:

```
tracer.exe -l:loops_2.exe bpx=loops_2.exe!0x00401026
```

BPX imposta solamente un breakpoint all' indirizzo e Italian text placeholder stamperà poi lo stato dei registri.

Questo è ciò che vediamo in `tracer.log`:

```
PID=12884|New process loops_2.exe
(0) loops_2.exe!0x401026
EAX=0x00a328c8 EBX=0x00000000 ECX=0x6f0f4714 EDX=0x00000000
ESI=0x00000002 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=PF ZF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000003 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000004 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000005 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
```

```
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000006 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000007 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000008 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000009 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
PID=12884|Process loops_2.exe exited. ExitCode=0 (0x0)
```

Vediamo il valore del registro ESI, cambiare da 2 a 9.

Oltre a ciò, il Italian text placeholder può collezionare i valori del registro a tutti gli indirizzi all' interno della funzione. Questo si chiama *trace*. Ogni istruzione viene tracciata, tutti i valori interessanti del registro vengono collezionati.

Dopodichè, viene generato un IDA.idc-script, che aggiunge i commenti. Quindi, abbiamo appreso in IDA che l' indirizzo della funzione `main()` è `0x00401020`, quindi eseguiamo:

```
tracer.exe -l:loops_2.exe bpf=loops_2.exe!0x00401020,trace:cc
```

BPF sta per "imposta breakpoint alla funzione".

Come risultato, otteniamo gli script `loops_2.exe.idc` e `loops_2.exe_clear.idc`.

Carichiamo `loops_2.exe.idc` in IDA e osserviamo:

```
.text:00401020
.text:00401020 ; =============== S U B R O U T I N E =======================================
.text:00401020
.text:00401020
.text:00401020 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401020 _main           proc near               ; CODE XREF: ___tmainCRTStartup+11D↓p
.text:00401020
.text:00401020 argc            = dword ptr  4
.text:00401020 argv            = dword ptr  8
.text:00401020 envp            = dword ptr  0Ch
.text:00401020
.text:00401020                 push    esi             ; ESI=1
.text:00401021                 mov     esi, 2
.text:00401026
.text:00401026 loc_401026:                             ; CODE XREF: _main+13↓j
.text:00401026                 push    esi             ; ESI=2..9
.text:00401027                 call    sub_401000      ; tracing nested maximum level (1) reached,
.text:0040102C                 inc     esi             ; ESI=2..9
.text:0040102D                 add     esp, 4          ; ESP=0x38fcbc
.text:00401030                 cmp     esi, 0Ah        ; ESI=3..0xa
.text:00401033                 jl      short loc_401026 ; SF=false,true OF=false
.text:00401035                 xor     eax, eax
.text:00401037                 pop     esi
.text:00401038                 retn                    ; EAX=0
.text:00401038 _main           endp
```

Figura 1.57: IDA con .idc-script caricato

Notiamo che ESI può assumere valori da 2 a 9 all' inizio del corpo del ciclo, ma da 3 a 0xA (10) dopo l'incremento. Notiamo inoltre che il `main()` termina con 0 in EAX.

Italian text placeholder genera inoltre `loops_2.exe.txt`, che contiene informazioni riguardo a quante volte ogni istruzione è stata eseguita e i valori del registro:

Listing 1.173: loops_2.exe.txt

```
0x401020 (.text+0x20), e=       1 [PUSH ESI] ESI=1
0x401021 (.text+0x21), e=       1 [MOV ESI, 2]
0x401026 (.text+0x26), e=       8 [PUSH ESI] ESI=2..9
0x401027 (.text+0x27), e=       8 [CALL 8D1000h] tracing nested maximum ↙
    ↳ level (1) reached, skipping this CALL 8D1000h=0x8d1000
0x40102c (.text+0x2c), e=       8 [INC ESI] ESI=2..9
0x40102d (.text+0x2d), e=       8 [ADD ESP, 4] ESP=0x38fcbc
0x401030 (.text+0x30), e=       8 [CMP ESI, 0Ah] ESI=3..0xa
0x401033 (.text+0x33), e=       8 [JL 8D1026h] SF=false,true OF=false
0x401035 (.text+0x35), e=       1 [XOR EAX, EAX]
0x401037 (.text+0x37), e=       1 [POP ESI]
0x401038 (.text+0x38), e=       1 [RETN] EAX=0
```

Possiamo usare grep qui.

ARM

Senza ottimizzazione Keil 6/2013 (Modalità ARM)

```
main
        STMFD   SP!, {R4,LR}
        MOV     R4, #2
        B       loc_368
loc_35C  ; CODE XREF: main+1C
        MOV     R0, R4
        BL      printing_function
        ADD     R4, R4, #1

loc_368  ; CODE XREF: main+8
        CMP     R4, #0xA
        BLT     loc_35C
        MOV     R0, #0
        LDMFD   SP!, {R4,PC}
```

Il contatore di iterazioni i viene salvato nel registro R4. L' istruzione `MOV R4, #2` inizializza solamente i. Le istruzioni `MOV R0, R4` e `BL printing_function` compongono il corpo del ciclo, la prima istruzione prepara l'argomento per la funzione `f()` e la seconda la chiama. L'istruzione `ADD R4, R4, #1` aggiunge solamente 1 alla variabie i ad ogni iterazione. `CMP R4, #0xA` compara i con `0xA` (10). L' istruzione successiva `BLT` (*Branch Less Than*) salta se i è minore di 10. Altrimenti, 0 deve essere scritto in `R0` (poiché la nostra funzione restituisce 0) e termina l'esecuzione della funzione.

Con ottimizzazione Keil 6/2013 (Modalità Thumb)

```
_main
                PUSH    {R4,LR}
                MOVS    R4, #2

loc_132                                 ; CODE XREF: _main+E
                MOVS    R0, R4
                BL      printing_function
                ADDS    R4, R4, #1
                CMP     R4, #0xA
                BLT     loc_132
                MOVS    R0, #0
                POP     {R4,PC}
```

Sostanzialmente è uguale.

Con ottimizzazione Xcode 4.6.3 (LLVM) (Modalità Thumb-2)

```
_main
       PUSH            {R4,R7,LR}
       MOVW            R4, #0x1124 ; "%d\n"
       MOVS            R1, #2
       MOVT.W          R4, #0
       ADD             R7, SP, #4
       ADD             R4, PC
```

```
        MOV             R0, R4
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #3
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #4
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #5
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #6
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #7
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #8
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #9
        BLX             _printf
        MOVS            R0, #0
        POP             {R4,R7,PC}
```

Infatti, questo era nella mia funzione `f()`:

```
void printing_function(int i)
{
    printf ("%d\n", i);
};
```

Quindi, LLVM non ha solamente *srotolato* il ciclo, ma ha anche ”inserito tra le linee” la mia semplice funzione `f()`, inserendo il suo codice 8 volte anzichè chiamarla.

Ciò è possibile nel caso in cui la funzione sia molto semplice (come la mia) e non venga chiamata molto spesso (come qui).

ARM64: Con ottimizzazione GCC 4.9.1

Listing 1.174: Con ottimizzazione GCC 4.9.1

```
printing_function:
; prepara il secondo argomento di printf():
        mov     w1, w0
; carica l' indirizzo della stringa "f(%d)\n"
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
; salta solo qui invece di saltare con link e ritorno:
        b       printf
main:
```

```
; salva FP e LR nello stack locale:
        stp     x29, x30, [sp, -32]!
; prepara lo stack frame:
        add     x29, sp, 0
; salva il contenuto del registro X19 nello stack locale:
        str     x19, [sp,16]
; useremo il registro W19 come contatore.
; imposta 2 come valore iniziale:
        mov     w19, 2
.L3:
; prepara il primo argomento di printing_function():
        mov     w0, w19
; incrementa il registro contatore.
        add     w19, w19, 1
; qui W0 ha ancora il valore del contatore prima dell' incremento.
        bl      printing_function
; è finita?
        cmp     w19, 10
; no, salta all' inizio del corpo del ciclo:
        bne     .L3
; ritorna 0
        mov     w0, 0
; ripristina il contenuto del registro X19:
        ldr     x19, [sp,16]
; ripristina i valori di FP e LR:
        ldp     x29, x30, [sp], 32
        ret
.LC0:
        .string "f(%d)\n"
```

ARM64: Senza ottimizzazione GCC 4.9.1

Listing 1.175: Senza ottimizzazione GCC 4.9.1 -fno-inline

```
.LC0:
        .string "f(%d)\n"
printing_function:
; salva FP e LR nello stack locale:
        stp     x29, x30, [sp, -32]!
; prepara lo stack frame:
        add     x29, sp, 0
; salva il contenuto del registro W0:
        str     w0, [x29,28]
; carica l'indirizzo della stringa "f(%d)\n"
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
; ricarica il valore di input dallo stack locale nel registro W1:
        ldr     w1, [x29,28]
; chiama printf()
        bl      printf
; ripristina i valori di FP e LR:
        ldp     x29, x30, [sp], 32
```

```
        ret
main:
; salva FP e LR nello stack locale:
        stp     x29, x30, [sp, -32]!
; prepara lo stack frame:
        add     x29, sp, 0
; inizializza il contatore
        mov     w0, 2
; salvalo nello spazio allocato per lui nello stack locale:
        str     w0, [x29,28]
; sorvola il corpo del ciclo e salta all' istruzione di controllo della
    condizione di ciclo:
        b       .L3
.L4:
; carica il valore del contatore in W0.
; sarà il primo argomento di printing_function():
        ldr     w0, [x29,28]
; chiama printing_function():
        bl      printing_function
; incrementa il valore del contatore:
        ldr     w0, [x29,28]
        add     w0, w0, 1
        str     w0, [x29,28]
.L3:
; controllo della condizione di ciclo.
; carica il valore del contatore:
        ldr     w0, [x29,28]
; vale 9?
        cmp     w0, 9
; minore o uguale? allora salta all' inizio del corpo del ciclo:
; altrimenti, non fare nulla.
        ble     .L4
; ritorna 0
        mov     w0, 0
; ripristina i valori di FP e LR:
        ldp     x29, x30, [sp], 32
        ret
```

MIPS

Listing 1.176: Senza ottimizzazione GCC 4.4.5 (IDA)

```
main:

; IDA non è al corrente dei nomi delle variabili nello stack locale
; Gli abbiamo dato i nomi manualmente:
i               = -0x10
saved_FP        = -8
saved_RA        = -4

; prologo funzione:
                addiu   $sp, -0x28
                sw      $ra, 0x28+saved_RA($sp)
```

```
                sw      $fp, 0x28+saved_FP($sp)
                move    $fp, $sp
; inizializza il contatore a 2 e salva questo valore nello stack locale
                li      $v0, 2
                sw      $v0, 0x28+i($fp)
; pseudoistruzione. "BEQ $ZERO, $ZERO, loc_9C" qui è:
                b       loc_9C
                or      $at, $zero ; branch delay slot, NOP

loc_80:                                 # CODE XREF: main+48
; carica il valore del contatore dallo stack locale e chiama
    printing_function():
                lw      $a0, 0x28+i($fp)
                jal     printing_function
                or      $at, $zero ; branch delay slot, NOP
; carica il contatore, incrementalo, salvalo nuovamente:
                lw      $v0, 0x28+i($fp)
                or      $at, $zero ; NOP
                addiu   $v0, 1
                sw      $v0, 0x28+i($fp)

loc_9C:                                 # CODE XREF: main+18
; controlla il contatore, vale 10?
                lw      $v0, 0x28+i($fp)
                or      $at, $zero ; NOP
                slti    $v0, 0xA
; se è minore di, salta a loc_80 (inizio del corpo del ciclo):
                bnez    $v0, loc_80
                or      $at, $zero ; branch delay slot, NOP
; terminando, ritorna 0:
                move    $v0, $zero
; epilogo funzione:
                move    $sp, $fp
                lw      $ra, 0x28+saved_RA($sp)
                lw      $fp, 0x28+saved_FP($sp)
                addiu   $sp, 0x28
                jr      $ra
                or      $at, $zero ; branch delay slot, NOP
```

L'istruzione per noi nuova è B. E' in reltà la pseudo istruzione(BEQ).

Un' ulteriore cosa

Nel codice generato possiamo notare che: dopo aver inizializzato i, il corpo del ciclo non viene eseguito, questo perchè viene prima controllata la condizione su i, solamente dopo il corpo del ciclo può essere eseguito. E questo è corretto.

Perchè, se la condizione del ciclo non è rispettata all' inizio, il corpo del ciclo non deve essere eseguito. Ciò è possibile nel caso seguente:

```
for (i=0; i<total_entries_to_process; i++)
    loop_body;
```

Se *total_entries_to_process* vale 0, il corpo del ciclo non deve proprio essere eseguito.

Questo è il motivo del controllo della condizione prima dell' esecuzione.

In ogni caso, un compilatore ottimizzato potrebbe scambiare il controllo della condizione e il corpo del ciclo, se è sicuro che la precedente situazione non sia possibile (come nel caso del nostro semplice esempio e usando compilatori come Keil, Xcode (LLVM), MSVC in modalità ottimizzato).

1.22.2 Routine di copia blocchi di memoria

Le routine di copia della memoria, nel mondo reale, possono copiare 4 o 8 byte ad ogni iterazione, usando SIMD[104], vettorizzazione, etc. Ma per maggiore semplicità, questo esempio è il più semplice possibile.

```
#include <stdio.h>

void my_memcpy (unsigned char* dst, unsigned char* src, size_t cnt)
{
        size_t i;
        for (i=0; i<cnt; i++)
                dst[i]=src[i];
};
```

Implementazione diretta

Listing 1.177: GCC 4.9 x64 ottimizzato per la dimensione (-Os)

```
my_memcpy:
; RDI = indirizzo destinazione
; RSI = indirizzo sorgente
; RDX = dimensione del blocco

; inizializza il contatore (i) a 0
        xor     eax, eax
.L2:
; sono stati copiati tutti i byte? allora esci:
        cmp     rax, rdx
        je      .L5
; carica il byte a RSI+i:
        mov     cl, BYTE PTR [rsi+rax]
; salva il byte a RDI+i:
        mov     BYTE PTR [rdi+rax], cl
        inc     rax ; i++
        jmp     .L2
.L5:
        ret
```

[104] Single Instruction, Multiple Data

Listing 1.178: GCC 4.9 ARM64 ottimizzato per la dimensione (-Os)

```
my_memcpy:
; X0 = indirizzo destinazione
; X1 = indirizzo sorgente
; X2 = dimensione del blocco

; inizializza il contaore (i) a 0
        mov     x3, 0
.L2:
; sono stati copiati tutti i byte? allora esci:
        cmp     x3, x2
        beq     .L5
; carica il byte a X1+i:
        ldrb    w4, [x1,x3]
; salva il byte a X0+i:
        strb    w4, [x0,x3]
        add     x3, x3, 1 ; i++
        b       .L2
.L5:
        ret
```

Listing 1.179: Con ottimizzazione Keil 6/2013 (Modalità Thumb)

```
my_memcpy PROC
; R0 = indirizzo destinazione
; R1 = indirizzo sorgente
; R2 = dimensione del blocco

        PUSH    {r4,lr}
; inizializza il contatore (i) a 0
        MOVS    r3,#0
; condizione controllata alla fine della funzione, quindi salta li:
        B       |L0.12|
|L0.6|
; carica il byte a R1+i:
        LDRB    r4,[r1,r3]
; salva il byte a R0+i:
        STRB    r4,[r0,r3]
; i++
        ADDS    r3,r3,#1
|L0.12|
; i<size?
        CMP     r3,r2
; se è così, salta all' inizio del ciclo:
        BCC     |L0.6|
        POP     {r4,pc}
        ENDP
```

ARM in modalità ARM

Keil in modalità ARM sfrutta appieno i suffissi condizionali:

Listing 1.180: Con ottimizzazione Keil 6/2013 (Modalità ARM)

```
my_memcpy PROC
; R0 = indirizzo destinazione
; R1 = indirizzo sorgente
; R2 = dimensione del blocco

; inizializza il contatore (i) a 0
        MOV      r3,#0
|L0.4|
; sono stati copiati tutti i byte?
        CMP      r3,r2
; il seguente blocco è eseguito solo se la condizione è minore di,
; i.e., se R2<R3 o i<size.
; carica il byte a R1+i:
        LDRBCC   r12,[r1,r3]
; salva il byte a R0+i:
        STRBCC   r12,[r0,r3]
; i++
        ADDCC    r3,r3,#1
; l' ultima istruzione del blocco condizionale.
; salta all' inizio del ciclo se i<size
; non fare niente altrimenti (i.e., if i>=size)
        BCC      |L0.4|
; ritorna
        BX       lr
        ENDP
```

Ecco perchè c'è solo un istruzione di diramazione anzichè 2.

MIPS

Listing 1.181: GCC 4.4.5 ottimizzato per la dimensione (-Os) (IDA)

```
my_memcpy:
; salta alla parte di controllo del ciclo:
                b       loc_14
; inizializza il contatore (i) a 0
; risiederà sempre in $v0:
                move    $v0, $zero ; branch delay slot

loc_8:                                  # CODE XREF: my_memcpy+1C
; carica il byte senza segno da $t0 in $v1:
                lbu     $v1, 0($t0)
; incrementa il contatore (i):
                addiu   $v0, 1
; salva il byte in $a3
                sb      $v1, 0($a3)

loc_14:                                 # CODE XREF: my_memcpy
; controlla se il contatore (i) in $v0 è ancora minore del 3° argomento della
    funzione ("cnt" in $a2):
                sltu    $v1, $v0, $a2
; forma l'indirizzo del byte nel blocco sorgente:
```

```
                addu    $t0, $a1, $v0
; $t0 = $a1+$v0 = src+i
; se il contatore è ancora minore di "cnt", salta al corpo del ciclo:
                bnez    $v1, loc_8
; forma l'indirizzo del byte nel blocco destinazione ($a3 = $a0+$v0 = dst+i):
                addu    $a3, $a0, $v0 ; branch delay slot
; termina se BNEZ non è stato attivato:
                jr      $ra
                or      $at, $zero ; branch delay slot, NOP
```

Qui abbiamo due nuove istruzioni: LBU («Load Byte Unsigned») e SB («Store Byte»).

Esattamente come in ARM, tutti i registri in MIPS sono larghi 32 bit, non ci sono parti larghe un byte come in x86.

Quindi quando maneggiamo dei singoli byte, dobbiamo allocare interi registri a 32 bit per loro.

LBU carica un byte e pulisce tutti gli altri bit («Unsigned»).

D' altro canto, l' istruzione LB («Load Byte») estende il segno del byte caricato a un valore su 32 bit.

SB scrive solamente in memoria un byte dagli ultimi 8 bit del registro.

Vettorizzazione

Con ottimizzazione GCC può fare molto di più con questo esempio: **??** on page ??.

1.22.3 Controllo condizione

E' importante tenere a mente che nel costrutto *for()*, la condizione non è controllata alla fine, ma all' inizio, prima che il corpo del ciclo venga eseguito. Ma spesso, è più conveniente per il compilatore controllarla alla fine, dopo il corpo. A volte, può essere aggiunto un controllo aggiuntivo all'inizio.

Per esempio:

```
#include <stdio.h>

void f(int start, int finish)
{
        for (; start<finish; start++)
                printf ("%d\n", start);
};
```

GCC 5.4.0 x64 ottimizzato:

```
f:
; check condition (1):
        cmp     edi, esi
        jge     .L9
        push    rbp
        push    rbx
```

```
        mov     ebp, esi
        mov     ebx, edi
        sub     rsp, 8
.L5:
        mov     edx, ebx
        xor     eax, eax
        mov     esi, OFFSET FLAT:.LC0 ; "%d\n"
        mov     edi, 1
        add     ebx, 1
        call    __printf_chk
; check condition (2):
        cmp     ebp, ebx
        jne     .L5
        add     rsp, 8
        pop     rbx
        pop     rbp
.L9:
        rep ret
```

Notiamo due controlli.

Hex-Rays (alla versione 2.2.0) lo decompila così:

```
void __cdecl f(unsigned int start, unsigned int finish)
{
  unsigned int v2; // ebx@2
  __int64 v3; // rdx@3

  if ( (signed int)start < (signed int)finish )
  {
    v2 = start;
    do
    {
      v3 = v2++;
      _printf_chk(1LL, "%d\n", v3);
    }
    while ( finish != v2 );
  }
}
```

In questo caso, *do/while()* può essere senza alcun dubbio rimpiazzato con *for()*, e il primo controllo può essere rimosso.

1.22.4 Conclusione

Scheletro grezzo del ciclo da 2 a 9 inclusi:

Listing 1.182: x86

```
    mov [counter], 2 ; inizializzazione
    jmp check
body:
    ; corpo del ciclo
```

```
    ; fai qualcosa qui
    ; utilizza la variabile contatore nello stack locale
    add [counter], 1 ; incrementa
check:
    cmp [counter], 9
    jle body
```

L' operazione di incremento può essere rappresentata con 3 istruzioni nel codice non ottimizzato:

Listing 1.183: x86

```
    MOV [counter], 2 ; inizializzazione
    JMP check
body:
    ; corpo del ciclo
    ; fai qualcosa qui
    ; utilizza la variabile contatore nello stack locale
    MOV REG, [counter] ; incrementa
    INC REG
    MOV [counter], REG
check:
    CMP [counter], 9
    JLE body
```

Se il corpo del ciclo è corto, un intero registro può essere dedicato alla variabile contatore:

Listing 1.184: x86

```
    MOV EBX, 2 ; inizializzazione
    JMP check
body:
    ; corpo del ciclo
    ; fai qualcosa qui
    ; usa il contatore in EBX, ma non modificarlo!
    INC EBX ; incrementa
check:
    CMP EBX, 9
    JLE body
```

Molte parti del ciclo possono essere generate in oridine diverso dal compilatore:

Listing 1.185: x86

```
    MOV [counter], 2 ; inizializzazione
    JMP label_check
label_increment:
    ADD [counter], 1 ; incrementa
label_check:
    CMP [counter], 10
    JGE exit
    ; corpo del ciclo
    ; fai qualcosa qui
```

```
    ; usa la variabile contatore nello stack locale
    JMP label_increment
exit:
```

Di solito la condizione è controllata *prima* del corpo del ciclo, ma il compilatore potrebbe riarrangiarlo in maniera che la condizione sia controllata *dopo* il corpo del ciclo.

Questo avviene quando il compilatore è sicuro che la condizione è sempre *vera* per la prima iterazione, di conseguenza il corpo del ciclo verrà eseguito almeno una volta:

Listing 1.186: x86

```
    MOV REG, 2 ; inizializzazione
body:
    ; corpo del ciclo
    ; fai qualcosa qui
    ; usa il contatore in REG, ma non modificarlo!
    INC REG ; incrementa
    CMP REG, 10
    JL body
```

Utilizzando l' istruzione di L00P. E' raro che il compilatore non lo utilizzi. Se ciò avviene, è segno che il codice è stato scritto a mano:

Listing 1.187: x86

```
    ; conta da 10 a 1
    MOV ECX, 10
body:
    ; corpo del ciclo
    ; dai qualcosa qui
    ; utilizza il contatore in ECX, ma non modificarlo!
    LOOP body
```

ARM.

Il registro R4 è dedicato alla variabile contatore in questo esempio:

Listing 1.188: ARM

```
    MOV R4, 2 ; inizializzazione
    B check
body:
    ; corpo del ciclo
    ; fai qualcosa qui
    ; utilizza il contatore in R4, ma non modificarlo!
    ADD R4,R4, #1 ; incrementa
check:
    CMP R4, #10
    BLT body
```

1.22.5 Esercizi

- http://challenges.re/54
- http://challenges.re/55
- http://challenges.re/56
- http://challenges.re/57

1.23 Maggiori informazioni sulle stringhe

1.23.1 strlen()

Parliamo ancora una volta dei cicli. Spesso, la funzione `strlen()` [105] è implementata utilizzando il costrutto `while()`. Ecco come è fatta nelle librerie standard di MSVC:

```
int my_strlen (const char * str)
{
        const char *eos = str;

        while( *eos++ ) ;

        return( eos - str - 1 );
}

int main()
{
        // test
        return my_strlen("hello!");
};
```

x86

Senza ottimizzazione MSVC

Compiliamolo:

```
_eos$ = -4                      ; dimensione = 4
_str$ = 8                       ; dimensione = 4
_strlen PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _str$[ebp]  ; prendi il puntatore alla stringa da
    "str"
    mov     DWORD PTR _eos$[ebp], eax  ; piazzalo nella variabile locale
    "eos"
$LN2@strlen_:
    mov     ecx, DWORD PTR _eos$[ebp]  ; ECX=eos
```

[105] conta i caratteri in una stringa, nel linguaggio C

```
    ; prendi un byte (8-bit) dall' indirizzo in ECX e piazzalo
    ; in EDX come valore a 32-bit con estensione del segno

    movsx   edx, BYTE PTR [ecx]
    mov     eax, DWORD PTR _eos$[ebp]  ; EAX=eos
    add     eax, 1                     ; incrementa EAX
    mov     DWORD PTR _eos$[ebp], eax  ; rimetti EAX in "eos"
    test    edx, edx                   ; EDX è zero?
    je      SHORT $LN1@strlen_         ; si, allora termina il ciclo
    jmp     SHORT $LN2@strlen_         ; continua il ciclo
$LN1@strlen_:

    ; qui calcoliamo la differena tra 2 puntatori

    mov    eax, DWORD PTR _eos$[ebp]
    sub    eax, DWORD PTR _str$[ebp]
    sub    eax, 1                     ; sottrai 1 e ritorna il risultato
    mov    esp, ebp
    pop    ebp
    ret    0
_strlen_ ENDP
```

Qui, abbiamo due nuove istruzioni: MOVSX e TEST.

La prima—MOVSX—prende un byte da un indirizzo in memoria e salva il valore in un registro a 32-bit. MOVSX sta per *MOV with Sign-Extend*. MOVSX imposta i restanti bit, dal 8 al 31, a 1 se il byte sorgente è *negativo* o a 0 se *positivo*.

Vediamo il perchè.

Di default, il tipo *char* è con segno in MSVC e GCC. Se abbiamo due valori, uno dei quali è *char* mentre l' altro è *int*, (anche *int* è con segno), se il primo valore contiene -2 (codificato come 0xFE) e copiamo il byte nel contenitore *int*, sarebbe 0x000000FE e ciò dal punto di vista di un *int* con segno è 254, non -2. Negli interi con segno, -2 è codificato come 0xFFFFFFFE. Quindi se vogliamo trasferire 0xFE da una variabile di tipo *char* a *int*, dobbiamo identficare il suo segno estenderlo. Questo è ciò che fa MOVSX.

E difficile dire se il compilatore necessiti di salvare una varibile *char* in EDX, potrebbe prendere solo la parte a 8-bit di un registro (per esempio DL). Apparentemente, il registro allocatore del compilatore funziona così.

Dopodichè vediamo TEST EDX, EDX. Maggiori dettagli riguardo all' istruzione TEST nella sezione dei campi di bit (**??** on page ??). In questo caso, questa istruzione controlla solamente se il valore in EDX è pari a 0.

Senza ottimizzazione GCC

Proviamo GCC 4.4.1:

```
                public strlen
strlen          proc near
```

```
eos             = dword ptr -4
arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 10h
                mov     eax, [ebp+arg_0]
                mov     [ebp+eos], eax

loc_80483F0:
                mov     eax, [ebp+eos]
                movzx   eax, byte ptr [eax]
                test    al, al
                setnz   al
                add     [ebp+eos], 1
                test    al, al
                jnz     short loc_80483F0
                mov     edx, [ebp+eos]
                mov     eax, [ebp+arg_0]
                mov     ecx, edx
                sub     ecx, eax
                mov     eax, ecx
                sub     eax, 1
                leave
                retn
strlen          endp
```

Il risultato è quasi lo stesso di MSVC, ma qui possiamo notare `MOVZX` al posto di `MOVSX`. `MOVZX` sta per *MOV with Zero-Extend*. Questa istruzione copia un valore a 8 o 16 bit in un registro a 32-bit e imposta i restanti bit a 0. Infatti, questa istruzione è conveniente solo perchè ci permette di rimpiazzare questa coppia di istruzioni: `xor eax, eax / mov al, [...]`.

D' altronde, è ovvio che il compilatore possa produrre questo codice: `mov al, byte ptr [eax] / test al, al`—è quasi lo stesso, tuttavia, i bit più alti del registro EAX conterranno rumore casuale. Ma supponiamo sia un ostacolo del compilatore—non potrebbe più produrre codice leggibile. Parlando francamente, il compilatore non è obbligato ad emettere codice del tutto comprensibile (agli umani).

La prossima nuova istruzione è `SETNZ`. In questo caso, se `AL` non contiene zero, `test al, al` imposta la flag `ZF` a 0, ma `SETNZ`, se `ZF==0` (*NZ* sta per *not zero*) imposta `AL` a 1. Parlando con un liguaggio naturale, *se AL non è zero, salta a loc_80483F0*. Il compilatore emette molto codice rindondante, ma non dimentichiamo che le ottimizzazioni sono spente.

Con ottimizzazione MSVC

Ora compiliamo il tutto in MSVC 2012, con le ottimizzazioni attivate (`/Ox`):

Listing 1.189: Con ottimizzazione MSVC 2012 /Ob0

```
_str$ = 8                       ; dimensione = 4
```

```
_strlen PROC
        mov     edx, DWORD PTR _str$[esp-4] ; EDX -> puntatore alla stringa
        mov     eax, edx                    ; spostalo in EAX
$LL2@strlen:
        mov     cl, BYTE PTR [eax]          ; CL = *EAX
        inc     eax                         ; EAX++
        test    cl, cl                      ; CL==0?
        jne     SHORT $LL2@strlen           ; no, continua il ciclo
        sub     eax, edx                    ; calcola la differenza tra
    puntatori
        dec     eax                         ; decrementa EAX
        ret     0
_strlen ENDP
```

Ora è tutto più semplice. Inutile dire che il compilatore può usare i registri con tale efficienza solo in piccole funzioni con poche variabili locali.

INC/DEC—sono le istruzioni incrementa/decrementa, in altre parole: aggiunge o sottrae 1 a/da una variable.

Con ottimizzazione MSVC + OllyDbg

Testiamo questo esempio (ottimizzato) in OllyDbg. Questa è la prima iterazione:

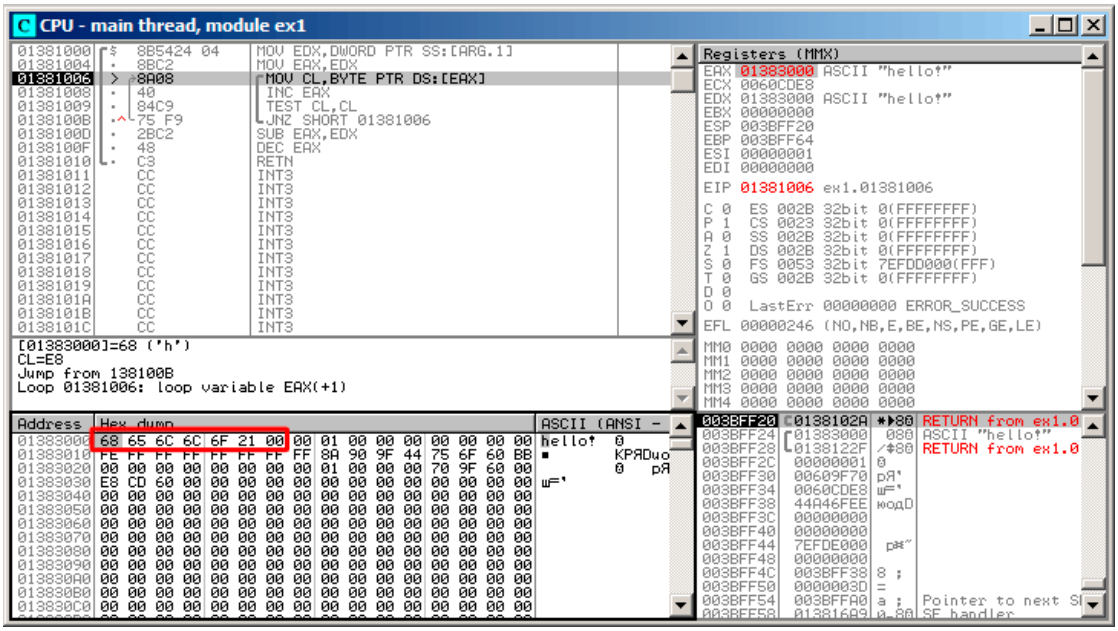


Figura 1.58: OllyDbg: inizio prima iterazione

Notiamo che OllyDbg ha trovato un ciclo e per convenienza, ha *avvolto* le sue istruzioni dentro le parentesi. Cliccando con il tasto destro su EAX e scegliendo «Follow in Dump», la finestra della memoria scorrerà fino al punto giusto. Possiamo vedere la stringa «hello!» in memoria. C'è almeno uno zero byte dopo la stringa e poi spazzatura casuale.

Se OllyDbg vede un registro contenente un indirizzo valido, che punta ad una stringa, lo mostra come stringa.

Premiamo F8 (step over) un paio di volte, per arrivare all' inizio del corpo del ciclo:

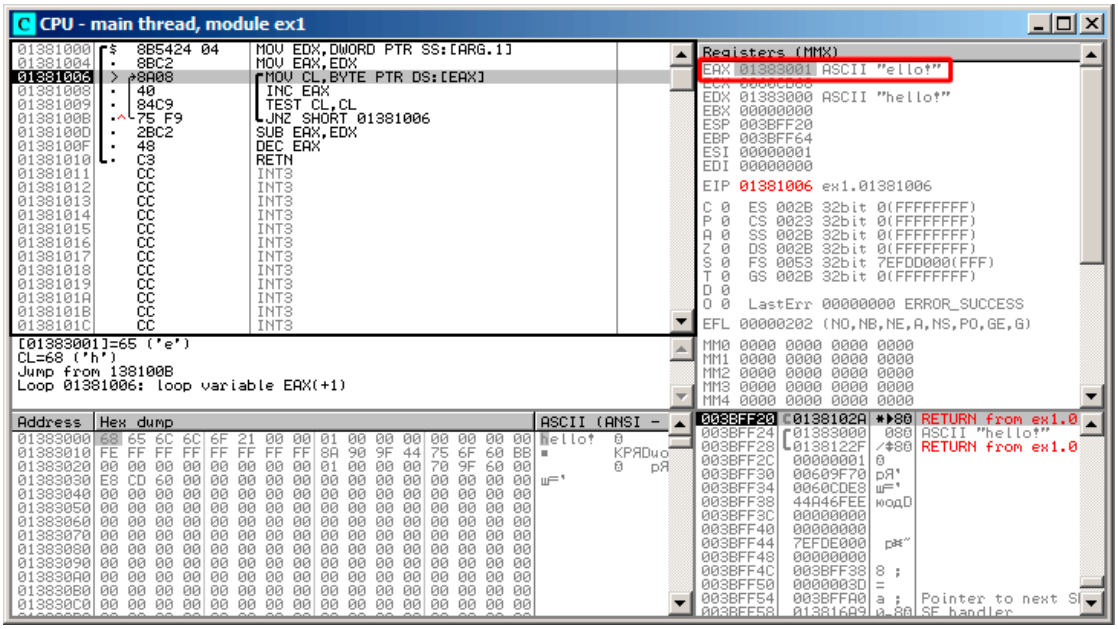


Figura 1.59: OllyDbg: inizio seconda iterazione

Notiamo che EAX contiene l'indirizzo del secondo carattere nella stringa.

Dobbiamo premere F8 un numero di volte sufficente per uscire dal ciclo:

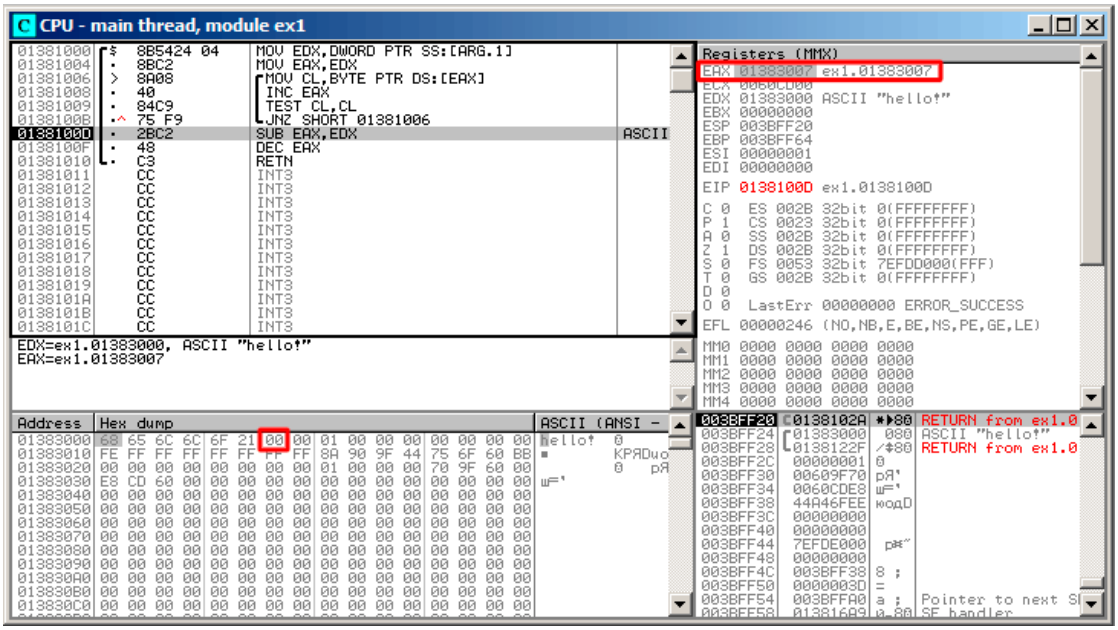


Figura 1.60: OllyDbg: differenza di puntatori da calcolare

Notiamo che ora EAX contiene l'indirizzo dello zero byte che si trova subito dopo la stringa più 1 (perché INC EAX è stato eseguito indipendentemente dal fatto che siamo usciti o meno dal ciclo). Nel frattempo, EDX non è cambiato, quindi sta ancora puntando all' inizio della stringa.

La differenza tra questi due indirizzi verrà calcolata ora.

L’ istruzione SUB è stata appena eseguita:

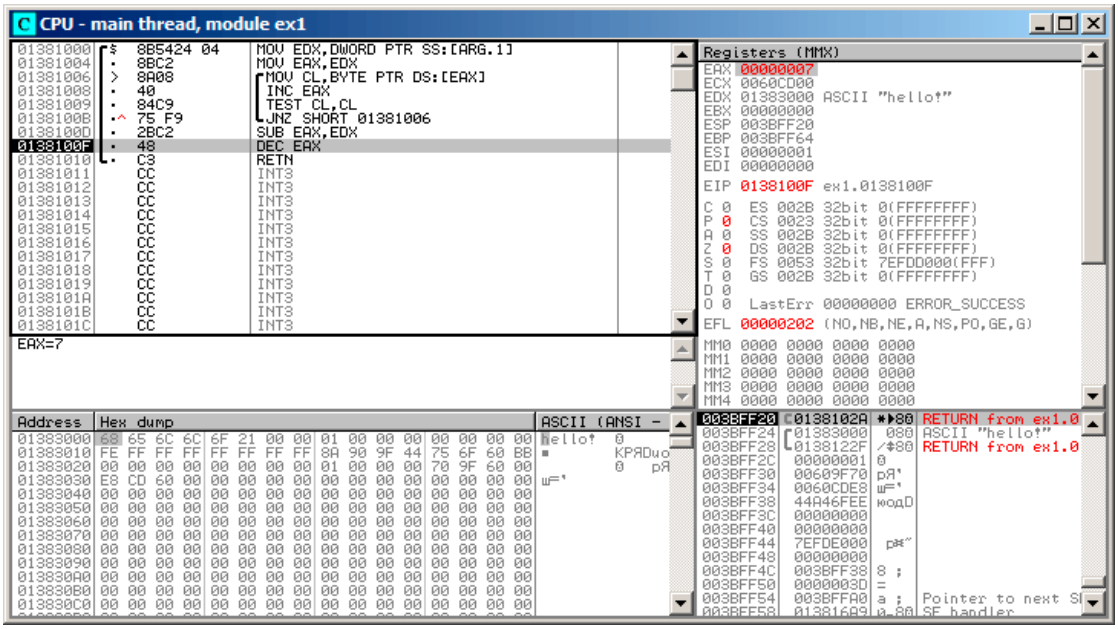


Figura 1.61: OllyDbg: EAX sta venendo decrementato

La differenza tra i puntatori, in questo momento, si trova nel registro EAX—7. In realtà, la lunghezza della stringa «hello!» è 6, ma con lo zero byte incluso—7. Ma `strlen()` deve ritornare il numero di caratteri nella stringa, diversi da zero. Quindi viene eseguito un decremento, dopodichè la funziona ritorna.

Con ottimizzazione GCC

Vediamo GCC 4.4.1 con le ottimizzazioni attivate (`-03` key):

```
                public strlen
strlen          proc near

arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                mov     ecx, [ebp+arg_0]
                mov     eax, ecx

loc_8048418:
                movzx   edx, byte ptr [eax]
                add     eax, 1
                test    dl, dl
                jnz     short loc_8048418
```

```
                not     ecx
                add     eax, ecx
                pop     ebp
                retn
strlen          endp
```

Qui GCC è quasi lo stesso di MSVC, eccetto per la presenza di `MOVZX`. Tuttavia, in questo caso `MOVZX` può essere rimpiazzato con
`mov dl, byte ptr [eax]`.

Forse è più semplice per il generatore di codice di GCC *ricordare* che l' intero registro EDX a 32-bit è stato allocato per una variabile *char* e quindi è sicuro che i bit più alti non contengono rumore in nessun momento.

Dopodichè vediamo una nuova istruzione—`NOT`. Questa istruzione inverte tutti i bit nell' operando.
Possiamo dire che è sinonimo dell' istruzione `XOR ECX, 0ffffffffh`. `NOT` e il seguente `ADD` calcolano la differenza di puntatori e sottraggono 1, solamente in maniera diversa. All' inizio ECX, dove è salvato il puntatore a *str*, viene invertito e gli viene sottratto 1.

In altre parole, alla fine della funzione, appena prima del corpo del ciclo, vengono eseguite queste istruzioni:

```
ecx=str;
eax=eos;
ecx=(-ecx)-1;
eax=eax+ecx
return eax
```

... che effettivamente è equivalente a:

```
ecx=str;
eax=eos;
eax=eax-ecx;
eax=eax-1;
return eax
```

Perchè GCC ha deciso che è meglio così? Difficile da dire. Ma forse entrambe le varianti hanno efficenza equivalente.

ARM

32-bit ARM

Senza ottimizzazione Xcode 4.6.3 (LLVM) (Modalità ARM)

Listing 1.190: Senza ottimizzazione Xcode 4.6.3 (LLVM) (Modalità ARM)

```
_strlen
```

```
eos  = -8
str  = -4

     SUB    SP, SP, #8 ; alloca 8 byte per le variabili locali
     STR    R0, [SP,#8+str]
     LDR    R0, [SP,#8+str]
     STR    R0, [SP,#8+eos]

loc_2CB8 ; CODE XREF: _strlen+28
     LDR    R0, [SP,#8+eos]
     ADD    R1, R0, #1
     STR    R1, [SP,#8+eos]
     LDRSB  R0, [R0]
     CMP    R0, #0
     BEQ    loc_2CD4
     B      loc_2CB8
loc_2CD4 ; CODE XREF: _strlen+24
     LDR    R0, [SP,#8+eos]
     LDR    R1, [SP,#8+str]
     SUB    R0, R0, R1 ; R0=eos-str
     SUB    R0, R0, #1 ; R0=R0-1
     ADD    SP, SP, #8 ; libera gli 8 byte allocati
     BX     LR
```

LLVM non ottimizzata genera troppo codice, tuttavia, qui possiamo vedere come la funzione lavora con le variabili locali nello stack. Ci sono solo due variabili locali nella nostra funzione: *eos* e *str*. In questo listato , generato da IDA, abbiamo rinominato manualmente *var_8* e *var_4* in *eos* e *str*.

Le prime istruzioni, salvano solamente i valori di input in entrambe *str* e *eos*.

Il corpo del ciclo inizia a *loc_2CB8*.

Le prime tre istruzioni nel corpo del ciclo(`LDR`, `ADD`, `STR`) caricano il valore di *eos* in `R0`. Dopodichè il valore viene incrementato e risalvato in *eos*, che è situata nello stack.

L' istruzione successiva, `LDRSB R0, [R0]` («Load Register Signed Byte»), carica un byte dalla memoria all' indirizzo salvato in `R0` e lo extende con segno in 32-bit [106]. Ciò è simile all' istruzione `MOVSX` in x86.

Il compilatore tratta questo byte come con segno da quando il tipo *char* è con segno in accordo con lo standard C. Riguardo a ciò, se ne è già parlato nella sezione (1.23.1 on page 258), in relazione a x86.

E' da notare che in ARM è impossibile utilizzare una parte a 8 o 16 bit di un registro a 32-bit separatamente dall' intero registro, come in x86.

Apparentemente, è perché x86 ha un'enorme storia di retrocompatibilità con i suoi antenati fino all'8086 a 16 bit e persino all'8080 a 8 bit, ma ARM è stato sviluppato da zero come un processore RISC a 32 bit.

[106] Il compilatore Keil tratta il tipo *char* come con segno, proprio come MSVC e GCC.

Di conseguenza, per elaborare byte separati in ARM, è necessario utilizzare comunque i registri a 32 bit.

Quindi, LDRSB carica i byte dalla stringa a R0, uno alla volta. Le seguenti istruzioni CMP e BEQ, controllano se il byte caricato è 0. Se non è 0, il controllo passa all' inizio del corpo del ciclo. E se è 0, il ciclo termina.

Alla fine della funzione, la differenza tra *eos* e *str* è calcolata, viene sottrato 1 da esso, e il valore risultante iene ritornato attraverso R0.

N.B. I registri non sono stati salavati in questa funzione.

Questo perchè nella covenzione a chiamata ARM, i registri R0-R3 sono «scratch registers», utilizati per il passaggio di argomenti, e non ci è richiesto di ripristinare il loro valore quando la funzione esce, in quanto la funzione chiamante non gli userà più. Di conseguenza, possono essere utilizzati per ciò che vogliamo.

Nessun altro registro viene usato qui, quindi è per questo che non abbiamo nulla da salvare nello stack.

Così, il controllo può essere ritornato alla funzione chiamante con un semplice salto (BX), all' indirizzo nel registro LR.

Con ottimizzazione Xcode 4.6.3 (LLVM) (Modalità Thumb)

Listing 1.191: Con ottimizzazione Xcode 4.6.3 (LLVM) (Modalità Thumb)

```
_strlen
            MOV       R1, R0

loc_2DF6
            LDRB.W    R2, [R1],#1
            CMP       R2, #0
            BNE       loc_2DF6
            MVNS      R0, R0
            ADD       R0, R1
            BX        LR
```

Come conclude LLVM ottimizzata, *eos* e *str* non necessitano spazio nello stack, e possono sempre essere salvate nei registri.

Prima dell' inizio del corpo del ciclo, *str* è sempre in R0, e *eos*—in R1.

L' istruzione LDRB.W R2, [R1],#1, carica un byte dalla memoria all' indirizzo salvato in R1, in R2, estendendo con segno ad un valore 32-bit, ma non solo. #1 alla fine dell' istruzione è implicito «Post-indexed addressing», che significa che 1 verrà aggiunto a R1 dopo che il byte è caricato. Approfondimento: **??** on page ??.

Come potete vedere CMP e BNE[107] sono nel corpo del ciclo, queste istruzioni continueranno a ciclare fino a che 0 non verrà trovato nella stringa.

[107] (PowerPC, ARM) Branch if Not Equal

Le istruzioni MVNS[108] (inverte tutti i bit, come NOT in x86) e ADD calcolano $eos - str - 1$. Infatti, queste due istruzioni eseguono $R0 = str + eos$, che è effettivamente equivalente a quello che c' era nel codice sorgente, ed il motivo, è stato già spiegato qui (1.23.1 on page 265).

Apparentemente, LLVM, proprio come GCC, ha concluso che questo codice può essere più corto (o veloce).

Con ottimizzazione Keil 6/2013 (Modalità ARM)

Listing 1.192: Con ottimizzazione Keil 6/2013 (Modalità ARM)

```
_strlen
                MOV     R1, R0

loc_2C8
                LDRB    R2, [R1],#1
                CMP     R2, #0
                SUBEQ   R0, R1, R0
                SUBEQ   R0, R0, #1
                BNE     loc_2C8
                BX      LR
```

Quasi lo stesso di ciò che abbiamo visto prima, con la differenza che l' espressione $str - eos - 1$ può non essere calcolata alla fine della funzione, ma direttamente nel corpo del ciclo. Il suffisso -EQ, come possiamo ricordare, implica che l' istruzione viene eseguita solo se gli operandi in CMP, che sono stati eseguiti prima, erano uguali tra loro. Pertanto, se R0 contiene 0, entrambe le istruzioni SUBEQ vengono eseguite e il risultato viene lasciato nel registro R0.

ARM64

Con ottimizzazione GCC (Linaro) 4.9

```
my_strlen:
        mov     x1, x0
        ; X1 ora è un puntatore temporaneo (eos), si comporta come un cursore
.L58:
        ; carica un byte da X1 a W2, incrementa X1 (post-index)
        ldrb    w2, [x1],1
        ; "Compare and Branch if NonZero": controlla se W2 = 0,
        ; salta a .L58 se non lo è
        cbnz    w2, .L58
        ; calcola la differenza tra il puntatore iniziale
        ; in X0 e l' indirizzo corrente in X1
        sub     x0, x1, x0
        ; decrementa il 32° bit più basso del risultato (LSB)
```

[108] MoVe Not

```
        sub     w0, w0, #1
        ret
```

L' algoritmo è lo stesso di 1.23.1 on page 259: trova un byte zero, calcola la differenza tra i puntatori e decrementa il risultato di 1. Molti commenti sono stati aggiunti dall' autore di questo libro.

L' unica cosa degna di nota è che il nostro esempio è in qualche modo sbagliato: `my_strlen()` ritorna un *int* a 32 bit, mentre dovrebbe ritornare `size_t` o un altro tipo a 64 bit.

La ragione è che, teoricamente, `strlen()` potrebbe essere chiamata per un grosso blocco in memoria che eccede 4GB, quindi deve essere in grado di ritornare un valore a 64 bit su piattaforme a 64 bit.

Per via del mio errore, l' ultima istruzione SUB opera su una parte a 32-bit del registro, mentre la penultima istruzione SUB lavora sull' intero registro a 64 bit (calcola la differenza tra i puntatori).

E' un mio errore, è meglio lasciarlo com'è, come esempio di come il codice potrebbe apparire in questi casi.

Senza ottimizzazione GCC (Linaro) 4.9

```
my_strlen:
; prologo funzione
        sub     sp, sp, #32
; il primo argomento (str) verrà salvato in [sp,8]
        str     x0, [sp,8]
        ldr     x0, [sp,8]
; copia "str" nella variabile "eos"
        str     x0, [sp,24]
        nop
.L62:
; eos++
        ldr     x0, [sp,24]  ; carica "eos" in X0
        add     x1, x0, 1    ; incrementa X0
        str     x1, [sp,24]  ; salva X0 in "eos"
; carica un byte dall' indirizzo di memoria contenuto in X0 a W0
        ldrb    w0, [x0]
; vale zero? (WZR è il registro a 32 bit che contiene sempre zero)
        cmp     w0, wzr
; salta se non è zero (Branch Not Equal)
        bne     .L62
; zero byte trovato. ora calcola la differenza.
; carica "eos" in X1
        ldr     x1, [sp,24]
; carica "str" in X0
        ldr     x0, [sp,8]
; calcola la differenza
        sub     x0, x1, x0
; decrementa il risultato
```

```
        sub     w0, w0, #1
; epilogo funzione
        add     sp, sp, 32
        ret
```

E' più verboso. Le variabili vengono spesso lanciate qui da e verso la memoria (stack locale). Qui c'è lo stesso errore: l'operazione di decremento avviene su una parte del registro a 32 bit.

MIPS

Listing 1.193: Con ottimizzazione GCC 4.4.5 (IDA)

```
my_strlen:
; la variabile "eos" risiederà sempre in $v1:
                move    $v1, $a0

loc_4:
; carica il byte dall' indirizzo in "eos" in $a1:
                lb      $a1, 0($v1)
                or      $at, $zero ; carica delay slot, NOP
; se il valore caricato è diverso da zero, salta a loc_4:
                bnez    $a1, loc_4
; incrementa "eos" altrimenti:
                addiu   $v1, 1 ; branch delay slot
; ciclo terminato, inverti la variabile "str":
                nor     $v0, $zero, $a0
; $v0=-str-1
                jr      $ra
; ritorna il valore = $v1 + $v0 = eos + ( -str-1 ) = eos - str - 1
                addu    $v0, $v1, $v0 ; branch delay slot
```

MIPS non ha una istruzione NOT, ma ha NOR che è l'operazione OR + NOT.

Questa operazione è ampiamente utilizzata nell' elettronica digitale[109]. Per esempio, l' Apollo Guidance Computer utilizzato nel programma Apollo, fu costruito solamente utilizzando 5600 porte NOR : [Jens Eickhoff, *Onboard Computers, Onboard Software and Satellite Operations: An Introduction*, (2011)]. Ma l' elemento NOR non è molto popolare nella programmazione.

Quindi, l' operazione NOT qui è implementata come NOR DST, $ZERO, SRC.

Dai fondamenti sappiamo che l' inversione di tutti i bit di un numero con segno, equivale a cambiargli il segno e sottrarre 1 dal risultato.

Quindi quello che NOT fa in questo caso, è prendere il valore di str e trasformarlo in $-str-1$. L' operazione di addizione che segue prepara il risusltato.

[109] NOR è chiamata «porta universale»

1.23.2 Delimitazione delle stringhe

E' interessante notare, come i parametri vengono passati alla funzione win32 *GetOpenFileName()*. Per chiamarlo, è necessario impostare un elenco di estensioni di file consentite:

```
        OPENFILENAME *LPOPENFILENAME;
        ...
        char * filter = "Text files (*.txt)\0*.txt\0MS Word files (*.doc)↙
    ↳ \0*.doc\0\0";
        ...
        LPOPENFILENAME = (OPENFILENAME *)malloc(sizeof(OPENFILENAME));
        ...
        LPOPENFILENAME->lpstrFilter = filter;
        ...

        if(GetOpenFileName(LPOPENFILENAME))
        {
                ...
```

Ciò che accade qui, è che la lista di stringhe viene passata a *GetOpenFileName()*. Non è un problema analizzarla: ogni volta che si incontra un singolo zero byte, questo è un elemento. Ogni qualvolta si incontrano due zero byte, si è alla fine della lista. Se passassimo la stringa a `printf()`, tratterebbe il primo oggetto come una singola stringa.

Quindi, questa è una stringa, oppure...? Sarebbe meglio dire che questo un buffer contenente diverse stringhe C zero terminate, le quali possono essere salvate e processate nel loro insieme.

Un altro esempio è la funzione *strtok()*. Essa prende una stringa e scrive dei byte zero al suo interno. Trasforma quindi la stringa di input in un qualche tipo di buffer, che ha diverse stringhe C con lo zero terminatore.

1.24 Sostituzione di istruzioni aritmetiche con altre

Nel ricercare l' ottimizzazione, un istruzione può essere rimpiazzata con un' altra, o anche con un gruppo di istruzioni. Per esempio, ADD e SUB possono rimpiazzarsi a vicenda: linea 18 in listato.**??**.

Per esempio, l' istruzione LEA è spesso utilizzata per semplici calcoli aritmetici: **??** on page ??.

1.24.1 Moltiplicazioni

Moltiplicare usando addizioni

Ecco un semplice esempio:

```
unsigned int f(unsigned int a)
```

```
{
        return a*8;
};
```

La moltiplicazione per 8 è stata rimpiazzata con 3 istruzioni di addizione, che fanno la stessa cosa. Apparentemente, l' ottimizzatore di MSVC ha deciso che questo codice potrebbe essere più veloce.

Listing 1.194: Con ottimizzazione MSVC 2010

```
_TEXT   SEGMENT
_a$ = 8         ; size = 4
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        add     eax, eax
        add     eax, eax
        add     eax, eax
        ret     0
_f      ENDP
_TEXT   ENDS
END
```

Moltiplicare usando "shift"

Le istruzioni di moltiplicazione e divisione con numeri che sono potenze di 2 sono spesso rimpiazzate con istruzioni "shift".

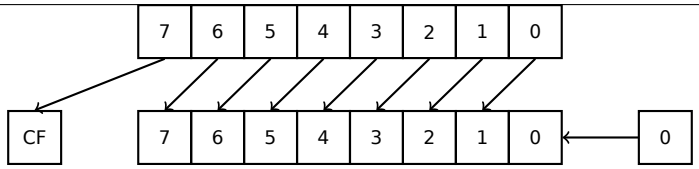
```
unsigned int f(unsigned int a)
{
        return a*4;
};
```

Listing 1.195: Senza ottimizzazione MSVC 2010

```
_a$ = 8         ; size = 4
_f      PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        shl     eax, 2
        pop     ebp
        ret     0
_f      ENDP
```

Moltiplicare per 4, significa semplicemente traslare il numero a sinistra di 2 bit e inserire due bit zero a destra (ultimi 2 bit). Esattamente come moltiplicare 3 per 100 —dobbiamo solo aggiungere due zeri a destra.

Questo è il funzionamento dell' istruzione "shift" a sinistra:



I bit aggiunti a destra sono sempre zeri.

Moltiplicazione per 4 in ARM:

Listing 1.196: Senza ottimizzazione Keil 6/2013 (Modalità ARM)

```
f PROC
        LSL      r0,r0,#2
        BX       lr
        ENDP
```

Moltiplicazione per 4 in MIPS:

Listing 1.197: Con ottimizzazione GCC 4.4.5 (IDA)

```
      jr      $ra
      sll     $v0, $a0, 2 ; branch delay slot
```

SLL corrisponde a «Shift Left Logical».

Moltiplicare usando shift, sottrazioni e addizioni

È anche possibile eliminare l'operazione di moltiplicazione quando si moltiplica per numeri come 7 o 17 utilizzando nuovamente lo shift. La matematica utilizzata è relativamente semplice.

32-bit

```
#include <stdint.h>

int f1(int a)
{
        return a*7;
};

int f2(int a)
{
        return a*28;
};

int f3(int a)
{
        return a*17;
};
```

x86

Listing 1.198: Con ottimizzazione MSVC 2012

```
; a*7
_a$ = 8
_f1     PROC
        mov     ecx, DWORD PTR _a$[esp-4]
; ECX=a
        lea     eax, DWORD PTR [ecx*8]
; EAX=ECX*8
        sub     eax, ecx
; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
        ret     0
_f1     ENDP

; a*28
_a$ = 8
_f2     PROC
        mov     ecx, DWORD PTR _a$[esp-4]
; ECX=a
        lea     eax, DWORD PTR [ecx*8]
; EAX=ECX*8
        sub     eax, ecx
; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
        shl     eax, 2
; EAX=EAX<<2=(a*7)*4=a*28
        ret     0
_f2     ENDP

; a*17
_a$ = 8
_f3     PROC
        mov     eax, DWORD PTR _a$[esp-4]
; EAX=a
        shl     eax, 4
; EAX=EAX<<4=EAX*16=a*16
        add     eax, DWORD PTR _a$[esp-4]
; EAX=EAX+a=a*16+a=a*17
        ret     0
_f3     ENDP
```

ARM

Keil in modalità ARM sfrutta i modificatori dello shift del secondo operando:

Listing 1.199: Con ottimizzazione Keil 6/2013 (Modalità ARM)

```
; a*7
||f1|| PROC
        RSB      r0,r0,r0,LSL #3
; R0=R0<<3-R0=R0*8-R0=a*8-a=a*7
```

```
        BX      lr
        ENDP

; a*28
||f2|| PROC
        RSB     r0,r0,r0,LSL #3
; R0=R0<<3-R0=R0*8-R0=a*8-a=a*7
        LSL     r0,r0,#2
; R0=R0<<2=R0*4=a*7*4=a*28
        BX      lr
        ENDP

; a*17
||f3|| PROC
        ADD     r0,r0,r0,LSL #4
; R0=R0+R0<<4=R0+R0*16=R0*17=a*17
        BX      lr
        ENDP
```

Tali modificatori non sono presenti in modalità Thumb. Non si può nemmeno ottimizzare `f2()`:

Listing 1.200: Con ottimizzazione Keil 6/2013 (Modalità Thumb)

```
; a*7
||f1|| PROC
        LSLS    r1,r0,#3
; R1=R0<<3=a<<3=a*8
        SUBS    r0,r1,r0
; R0=R1-R0=a*8-a=a*7
        BX      lr
        ENDP

; a*28
||f2|| PROC
        MOVS    r1,#0x1c ; 28
; R1=28
        MULS    r0,r1,r0
; R0=R1*R0=28*a
        BX      lr
        ENDP

; a*17
||f3|| PROC
        LSLS    r1,r0,#4
; R1=R0<<4=R0*16=a*16
        ADDS    r0,r0,r1
; R0=R0+R1=a+a*16=a*17
        BX      lr
        ENDP
```

MIPS

Listing 1.201: Con ottimizzazione GCC 4.4.5 (IDA)

```
_f1:
                sll     $v0, $a0, 3
; $v0 = $a0<<3 = $a0*8
                jr      $ra
                subu    $v0, $a0 ; branch delay slot
; $v0 = $v0-$a0 = $a0*8-$a0 = $a0*7

_f2:
                sll     $v0, $a0, 5
; $v0 = $a0<<5 = $a0*32
                sll     $a0, 2
; $a0 = $a0<<2 = $a0*4
                jr      $ra
                subu    $v0, $a0 ; branch delay slot
; $v0 = $a0*32-$a0*4 = $a0*28

_f3:
                sll     $v0, $a0, 4
; $v0 = $a0<<4 = $a0*16
                jr      $ra
                addu    $v0, $a0 ; branch delay slot
; $v0 = $a0*16+$a0 = $a0*17
```

64-bit

```
#include <stdint.h>

int64_t f1(int64_t a)
{
        return a*7;
};

int64_t f2(int64_t a)
{
        return a*28;
};

int64_t f3(int64_t a)
{
        return a*17;
};
```

x64

Listing 1.202: Con ottimizzazione MSVC 2012

```
; a*7
f1:
```

```
        lea     rax, [0+rdi*8]
; RAX=RDI*8=a*8
        sub     rax, rdi
; RAX=RAX-RDI=a*8-a=a*7
        ret

; a*28
f2:
        lea     rax, [0+rdi*4]
; RAX=RDI*4=a*4
        sal     rdi, 5
; RDI=RDI<<5=RDI*32=a*32
        sub     rdi, rax
; RDI=RDI-RAX=a*32-a*4=a*28
        mov     rax, rdi
        ret

; a*17
f3:
        mov     rax, rdi
        sal     rax, 4
; RAX=RAX<<4=a*16
        add     rax, rdi
; RAX=a*16+a=a*17
        ret
```

ARM64

Anche GCC 4.9 per ARM64 è conciso, grazie ai modificatori dello ”shift”:

Listing 1.203: Con ottimizzazione GCC (Linaro) 4.9 ARM64

```
; a*7
f1:
        lsl     x1, x0, 3
; X1=X0<<3=X0*8=a*8
        sub     x0, x1, x0
; X0=X1-X0=a*8-a=a*7
        ret

; a*28
f2:
        lsl     x1, x0, 5
; X1=X0<<5=a*32
        sub     x0, x1, x0, lsl 2
; X0=X1-X0<<2=a*32-a<<2=a*32-a*4=a*28
        ret

; a*17
f3:
        add     x0, x0, x0, lsl 4
; X0=X0+X0<<4=a+a*16=a*17
```

```
        ret
```

Algoritmo di moltiplicazione di Booth

Ci fu un tempo in cui i computer erano grossi e costosi, per molti di essi mancava il supporto hardware per l' operazione di moltiplicazione nella CPU, come Data General Nova. E quando serviva l' operazione di moltiplicazione, veniva fornita a livello software, per esempio, utilizzando l' algoritmo di moltiplicazione di Booth. Esso è un algoritmi di moltiplicazione che usa solamente addizioni e shift.

Ciò che i moderni compilatori ottimizzati fanno, è diverso, ma lo scopo (moltiplicare) e le risorse (operazioni veloci) sono le stesse.

1.24.2 Divisioni

Dividere usando gli shift

Esempio di divisione per 4:

```
unsigned int f(unsigned int a)
{
        return a/4;
};
```

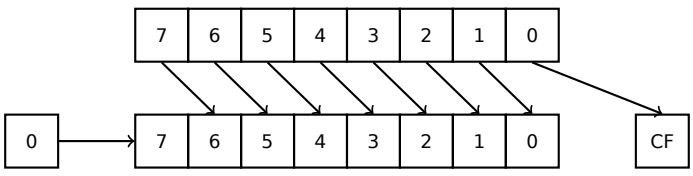
Otteniamo (MSVC 2010):

Listing 1.204: MSVC 2010

```
_a$ = 8         ; size = 4
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        shr     eax, 2
        ret     0
_f      ENDP
```

L' istruzione SHR (*SHift Right*) in questo esempio fa scorrere un numero di 2 bit a destra. I due bit liberati a sinistra (e.g., i due bit più significativi) sono impostati a zero. I due bit meno significativi sono scartati. Infatti, questi due bit scartati sono il resto della divisione.

L' istruzione SHR funziona proprio come SHL, ma nella direzione opposta.



E' semplice da capire se immaginiamo il numero 23 nel sistema numerico decimale. 23 può essere facilmente diviso per 10, semplicemente scartando l' ultima cifra (3—resto divisione). 2 rimane dopo l' operazione come quoziente.

Quindi il resto viene scartato, ma questo è OK, lavoriamo comunque su valori interi, che non sono numeri reali!

Divisione per 4 in ARM:

Listing 1.205: Senza ottimizzazione Keil 6/2013 (Modalità ARM)

```
f PROC
        LSR      r0,r0,#2
        BX       lr
        ENDP
```

Divisione per 4 in MIPS:

Listing 1.206: Con ottimizzazione GCC 4.4.5 (IDA)

```
        jr      $ra
        srl     $v0, $a0, 2 ; branch delay slot
```

L' istruzione SRL corrisponde a «Shift Right Logical».

1.24.3 Esercizio

- http://challenges.re/59

1.25 Array

yy [110]

1.25.1

```
#include <stdio.h>

int main()
{
    int a[20];
    int i;

    for (i=0; i<20; i++)
        a[i]=i*2;

    for (i=0; i<20; i++)
        printf ("a[%d]=%d\n", i, a[i]);

    return 0;
};
```

[110] AKA «homogener Container».

1.25.2

1.25.3 Esercizi

- http://challenges.re/62
- http://challenges.re/63
- http://challenges.re/64
- http://challenges.re/65
- http://challenges.re/66

1.26 Strutture

1.26.1 UNIX: struct tm

1.26.2

1.26.3 Esercizi

- http://challenges.re/71
- http://challenges.re/72

1.27

1.27.1

Capitolo 2

Italian text placeholder

2.1 Endianness

Con il termine endianness si intende un modo di rappresentare i dati in memoria.

2.1.1 Big-endian

Nelle architetture big-endian il valore `0x12345678` viene rappresentato in memoria come:

Indirizzo in memoria	Valore in byte
+0	0x12
+1	0x34
+2	0x56
+3	0x78

Tra le CPU big-endian ricordiamo Motorola 68k e IBM POWER.

2.1.2 Little-endian

Nelle architetture little-endian il valore `0x12345678` viene rappresentato in memoria come:

Indirizzo in memoria	Valore in byte
+0	0x78
+1	0x56
+2	0x34
+3	0x12

Tra le CPU aventi architettura little-endian ricordiamo l'Intel x86. Un importante esempio di utilizzo di endianness little-endian si trova in questo libro: **??** on page ??.

2.1.3 Esempio

Si consideri un sistema Linux installato su un'architettura MIPS big-endian e disponibile in QEMU [1].

E si compili il seguente semplice esempio:

```
#include <stdio.h>

int main()
{
        int v;

        v=123;

        printf ("%02X %02X %02X %02X\n",
                *(char*)&v,
                *(((char*)&v)+1),
                *(((char*)&v)+2),
                *(((char*)&v)+3));
};
```

Eseguendolo si ottiene:

```
root@debian-mips:~# ./a.out
00 00 00 7B
```

Il risultato è 0x7B, ovvero 123 in decimale. Nelle architetture little-endian il valore 7B verrà memorizzato nel primo byte di memoria (è possibile verificarlo su x86 o x86-64), mentre in questo caso viene memorizzato nell'ultimo byte, poiché il byte più alto viene memorizzato per primo.

Ecco perché esistono diverse distribuzioni Linux per architetture MIPS («mips» (big-endian) e «mipsel» (little-endian)). Non è possibile che un file binario compilato per una specifica endianness possa essere eseguito su un OS avente diversa endianness.

Si può trovare un altro esempio di architettura MIPS big-endian in questo libro: **??** on page ??.

2.1.4 Bi-endian

Esistono poi alcune CPU che possono cambiare tra diverse endianness. Tra queste vi sono ARM, PowerPC, SPARC, MIPS, IA64[2], etc.

2.1.5 Converting data

L'istruzione BSWAP può essere usata per la conversione di endianness.

I pacchetti di rete TCP/IP usano la convenzione big-endian quindi un programma che lavora su un'architettura little-endian deve convertire i dati ricevuti. Per questo motivo le funzioni `htonl()` e `htons()` sono tipicamente usate per questo scopo.

[1] Scaricabile qui: https://people.debian.org/~aurel32/qemu/mips/
[2] Intel Architecture 64 (Itanium)

Nel mondo TCP/IP il big-endian viene anche chiamato «network byte order», mentre il byte order sul computer viene chiamato «host byte order». L'«host byte order» è little-endian sulle architetture Intel x86 e altre architetture little-endian, ma è big-endian su architettura IBM POWER, quindi le funzioni `htonl()` e `htons()` non cambiano l'ordinamento dei byte in quest'ultima.

2.2 Memoria

Esistono 3 tipi principali di memoria:

- Memoria globale AKA «allocazione statica di memoria». L'allocazione di memoria non avviene esplicitamente ma semplicemente dichiarando variabili e/o arrays globalmente. Queste sono variabili globali e risiedono nei segmenti di dato o delle costanti. Essendo variabili globali sono considerate un anti-pattern. L'allocazione statica non è conveniente per buffer o arrays poiché devono avere una dimensione fissa e nota a priori. I buffer overflows che possono verificarsi in questa porzione di memoria solitamente sovrascrivono variabili o buffer che risiedono in locazioni contigue a questi in memoria. Un possibile esempio viene riportato in questo libro: 1.12.3 on page 102.

- Stack AKA «allocazione sullo stack». L'allocazione avviene semplicemente dichiarando variabili e/o arrays localmente all'interno delle funzioni. Infatti si tratta solitamente di variabili locali ad una funzione. Alcune volte queste variabili locali sono disponibili in seguito ad una catena di chiamate a funzione (alle funzioni chiamate se la funzione chiamante passa un puntatore ad una variabile ad una funzione chiamata che deve essere eseguita). L'allocazione e la deallocazione sono molto veloci poiché lo **SP!** viene semplicemente riposizionato.

  Nonostante questo, l'allocazione e la deallocazione non sono convenienti per buffer e/o arrays dal momento che la dimensione del buffer deve essere fissa, a meno che venga utilizzata `alloca()` (1.9.2 on page 48) o un array di lunghezza variabile. I buffer overflows solitamente sovrascrivono importanti strutture dati sullo stack: 1.25.2 on page 280.

- Heap AKA «allocazione dinamica di memoria». L'allocazione e la deallocazione avvengono chiamando
  `malloc()/free()` oppure `new/delete` in C++. Questo è il metodo più conveniente poiché la dimensione dell'area di memoria può essere decisa a runtime.

  Il resizing della memoria è possibile (utilizzando `realloc()`), ma può rivelarsi lento. Questo è il metodo più lento per allocare memoria: l'allocatore di memoria deve supportare e aggiornare tutte le strutture di controllo mentre esegue l'allocazione e la deallocazione. I buffer overflows solitamente sovrascrivono queste strutture. L'allocazione dinamica di memoria è anche fonte di possibili memory leak poiché ogni blocco di memoria deve essere deallocato esplicitamente. Tuttavia gli sviluppatori potrebbero dimenticarsi di farlo o eseguirlo in modo errato.

Un altro problema è il cosiddetto «use after free»—ovvero usare un blocco di memoria dopo che la `free()` è stata chiamata per rilasciarlo, uno scenario che può rivelarsi molto pericoloso. Alcuni esempi di riferimento possono essere trovati in questo libro: **??** on page ??.

2.3 CPU

2.3.1 Branch predictors

Alcuni compilatori più recenti provano a non utilizzare le istruzioni di salto condizionato. Questo avviene poiché il branch predictor non è sempre perfetto quindi il compilatore prova ad evitare l'uso di salti condizionali se è possibile. Gli esempi possono essere trovati in questi libri: 1.18.1 on page 174, 1.18.3 on page 184, **??** on page ??.

I processori ARM e x86 forniscono alcune istruzioni condizionali nel loro instruction set (ADRcc nel caso di ARM e CMOVcc nel caso di x86).

2.3.2 Data dependencies

Le attuali CPU sono in grado di eseguire istruzioni simultaneamente (OOE[3]), a patto che i risultati di un'istruzione in un gruppo non influenzino l'esecuzione delle altre. Per questo il compilatore prova ad utilizzare istruzioni che influenzino al minimo lo stato della CPU.

Questo è il motivo per cui l'istruzione LEA è così popolare, poiché non modifica i flag della CPU a differenza di altre istruzioni aritmetiche.

[3] Out-of-Order Execution

Capitolo 3

Capitolo 4

Java

4.1 Java

4.1.1 Introduzione

Esistono alcuni noti decompilatori per Java (o bytecode acJVM in generale). [1].

Il motivo è che la decompilazione del bytecode-JVM[2] è più semplice che per un codice di più basso livello per architetture x86:

- C'è molta più informazione riguardo i tipi di dato.
- Il modello di memoria della acJVM è molto più rigoroso e delineato.
- Il compilatore Java non esegue alcuna ottimizzazione (la JVM JIT[3] la esegue in fase di runtime), quindi i bytecode nei file relativi alle classi sono solitamente molto più leggibili.

Quando può essere utile la conoscenza della acJVM?

- Patching approssimativo di classi senza la necessità di ricompilare il risultato del decompilatore.
- Analisi del codice nascosto.
- Realizzazione del proprio obfuscator.
- Realizzazione di un compilatore generatore di codice (back-end) orientata alla JVM (come Scala, Clojure, etc. [4]).

[1] Ad esempio, JAD: http://varaneckas.com/jad/
[2] Java Virtual Machine
[3] Just-In-Time compilation
[4] Elenco completo: http://en.wikipedia.org/wiki/List_of_JVM_languages

Partiamo con alcuni semplici frammenti di codice. JDK 1.7 è usato ovunque, se non diversamente indicato.

Questo è il comando usato per decompilare le classi ovunque:
`javap -c -verbose`.

Questo è il libro che ho usato mentre preparavo tutti gli esempi: [Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, *The Java(R) Virtual Machine Specification / Java SE 7 Edition*] [5].

4.1.2 Ritornare un valore

Le funzioni che semplicemente ritornano indietro un qualche valore sono probabilmente tra le più semplici in Java.

Naturalmente, quando parliamo di funzioni in Java facciamo riferimento implicito ai metodi di una classe, non essendo possibile definire funzioni «libere», nel senso comune del termine.

Infatti, in Java ogni metodo (statico o dinamico che sia) è sempre definito in relazione ad una classe, poiché non è possibile fare diversamente.

Ad ogni modo, per semplicità, faremo uso del termine «funzione».

```
public class ret
{
        public static int main(String[] args)
        {
                return 0;
        }
}
```

Per compilare il file si esegue:

```
javac ret.java
```

...e possiamo decompilarne il risultato utilizzando una «utility» standard di Java:

```
javap -c -verbose ret.class
```

Ecco ciò che si ottiene dalla decompilazione:

Listing 4.1: JDK 1.7 (excerpt)

```
  public static int main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: iconst_0
         1: ireturn
```

[5] Italian text placeholderhttps://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf; http://docs.oracle.com/javase/specs/jvms/se7/html/

L'utilizzo della costante 0 è particolarmente frequente nella programmazione, per questo esiste un'istruzione apposita `iconst_0`, di un byte, che carica la costante nello «stack».

[6].

Similarmente, esistono altre istruzioni apposite per caricare costanti nello stack: `iconst_1` (che imposta 1), `iconst_2`, etc., fino a `iconst_5`.

E c'è un'istruzione `iconst_m1` apposita per caricare la costante -1.

Nella JVM, lo «stack» viene impiegato per il passaggio dei dati alle funzioni al momento della loro invocazione, e per consentire a queste ultime di ritornare a loro volta dei dati indietro. Quindi, osservando il codice dell'esempio precedente, la funzione `iconst_0` carica il valore 0 nello «stack». `ireturn` ritorna indietro un intero (*i* come prefisso nel nome della istruzione significa *integer*), prelevandolo dalla testa dello stack TOS[7].

Riscriviamo l'esempio precedente in modo da far ritornare indietro il valore 1234 alla funzione:

```
public class ret
{
        public static int main(String[] args)
        {
                return 1234;
        }
}
```

...quello che otteniamo ora dalla decompilazione è:

Listing 4.2: JDK 1.7 (excerpt)

```
  public static int main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: sipush        1234
         3: ireturn
```

`sipush` (*short integer*) imposta il numero 1234 nello «stack». *short* come prefisso nel nome della istruzione indica il fatto di lavorare con dati a 16-bit e infatti, il numero 1234 può essere rappresentato con 16-bit.

Cosa accade per valori più grandi?

```
public class ret
{
        public static int main(String[] args)
        {
                return 12345678;
        }
}
```

[6] Come in in MIPS, dove esiste un registro separato per la costante zero: 1.5.4 on page 35.

[7] Top of Stack

Listing 4.3: Constant pool

```
...
   #2 = Integer            12345678
...
```

```
  public static int main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: ldc           #2                  // int 12345678
         2: ireturn
```

Nella JVM non è possibile codificare numeri a 32-bit in un'istruzione «opcode», i progettisti non hanno lasciato questa possibilità.

Di conseguenza, il numero 12345678 che deve essere rappresentato con 32-bit viene registrato in uno spazio specifico detto «constant pool», diciamo che questo spazio sia una sorta di libreria delle costanti più frequenti (inclusi i tipi di dati stringa, oggetti, etc.).

Questa modalità di gestire le costanti non si riscontra solo nella JVM.

Anche MIPS, ARM ed altre CPUs RISC non permettono di codificare numeri di 32-bit in una istruzione «opcode» di 32-bit, di conseguenza il codice della CPU RISC (incluso MIPS e ARM) si trova a dover codificare il valore di una costante in più passi, a meno di poter utilizzare un «segmento dati»: **??** on page ??, **??** on page ??.

Tradizionalmente il codice MIPS dispone anche di un «constant pool» chiamato «literal pool», dove i segmenti sono chiamati «.lit4» (per i valori costanti a virgola mobile indirizzati con 32-bit in precisione singola) e «.lit8» (per i valori costanti a virgola mobile indirizzati con 64-bit in precisione doppia).

Proviamo ora con altri tipi di dati!

Booleani:

```
public class ret
{
        public static boolean main(String[] args)
        {
                return true;
        }
}
```

```
  public static boolean main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: iconst_1
         1: ireturn
```

Il bytecode prodotto non è diverso da quello che si otterrebbe per una funzione che semplicemente ritorna indietro il valore numerico 1.

Gli spazi da 32-bit per i dati nello stack sono utilizzati anche per i valori booleani, come nel C/C++.

Tuttavia il valore booleano che la funzione ritorna non può essere utilizzato come valore intero o viceversa — le informazioni sui tipi di dati dichiarati sono registrati nel file della classe e verificati a runtime.

La stessa storia vale per indirizzare valori numerici con 16-bit *short*:

```
public class ret
{
        public static short main(String[] args)
        {
                return 1234;
        }
}
```

```
  public static short main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: sipush        1234
         3: ireturn
```

...e *char*!

```
public class ret
{
        public static char main(String[] args)
        {
                return 'A';
        }
}
```

```
  public static char main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: bipush        65
         2: ireturn
```

`bipush` significa «caricare un byte nello stack». In Java un *char* viene indirizzato con la codifica UTF-16 a 16-bit, ed è equivalente a *short*. Il codice ASCII del carattere «A» è 65, per cui l'istruzione può caricarlo nello stack come singolo byte.

Proviamo ora con un *byte*:

```
public class retc
{
        public static byte main(String[] args)
        {
                return 123;
        }
}
```

```
  public static byte main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: bipush        123
         2: ireturn
```

Ci si può chiedere: perché scegliere il tipo dati a 16-bit *short*, quando poi internamente viene gestito con numeri interi a 32-bit?

Perché usare il tipo di dato specifico *char*, quando quest'ultimo è equivalente al tipo *short*?

La ragione è nella necessità di controllo sul tipo di dati dichiarati e per una questione di leggibilità del codice.

Se da una parte il tipo di dato *char* è essenzialmente lo stesso di *short*, dall'altra ci permette di intuire facilmente che si tratta di uno spazio che ospita un carattere UTF-16 e non un qualsiasi valore intero.

Quando usiamo *short* per una variabile, rendiamo esplicito a chiunque come i possibili valori siano limitati da 16 bit.

Così è una buona prassi usare il tipo di dato *boolean* quando necessario, piuttosto che *int* come in stile C.

Esiste in Java anche il tipo di dato intero a 64-bit:

```
public class ret3
{
        public static long main(String[] args)
        {
                return 1234567890123456789L;
        }
}
```

Listing 4.4: Constant pool

```
...
   #2 = Long               1234567890123456789l
...
```

```
  public static long main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: ldc2_w        #2                  // long 1234567890123456789l
         3: lreturn
```

Anche il tipo intero a 64-bit viene registrato nel («constant pool»): `ldc2_w` è l'istruzione che carica il valore nello stack e `lreturn` (*long return*) lo preleva dallo stack per ritornarlo indietro.

L'istruzione `ldc2_w` è anche impiegata per caricare nello stack numeri in virgola mobile con precisione doppia (appunto indirizzati con 64 bit) dal «constant pool»:

```
public class ret
{
        public static double main(String[] args)
        {
                return 123.456d;
        }
}
```

Listing 4.5: Constant pool

```
...
   #2 = Double             123.456d
...
```

```
  public static double main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: ldc2_w        #2                  // double 123.456d
         3: dreturn
```

`dreturn` significa «return double».

Infine abbiamo i numeri con virgola mobile e precisione singola:

```
public class ret
{
        public static float main(String[] args)
        {
                return 123.456f;
        }
}
```

Listing 4.6: Constant pool

```
...
   #2 = Float              123.456f
...
```

```
  public static float main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: ldc           #2                  // float 123.456f
         2: freturn
```

Notare come l'istruzione `ldc` è la stessa usata per caricare nello stack numeri interi a 32-bit dal «constant pool».

`freturn` significa «return float».

Cosa accadrebbe nel caso di una funzione che non ritorna indietro alcun dato?

```
public class ret
{
        public static void main(String[] args)
        {
                return;
        }
}
```

```
  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=0, locals=1, args_size=1
         0: return
```

Notare come l'istruzione `return` sia impiegata per restituire il controllo al termine dell'esecuzione di una funzione che non ritorna indietro alcun dato.

Quanto detto fino ad ora rende facile dedurre il tipo di dato che una funzione (o metodo) ritorna, semplicemente osservando l'ultima istruzione.

4.1.3 Semplici funzioni di calcolo

Continuiamo con delle semplici funzioni di calcolo.

```
public class calc
{
        public static int half(int a)
        {
                return a/2;
        }
}
```

Ecco il risultato, quando ad essere impiegata è l'istruzione `iconst_2`:

```
  public static int half(int);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: iload_0
         1: iconst_2
         2: idiv
         3: ireturn
```

`iload_0` prende il primo argomento passato alla funzione (il dato nella posizione 0) e lo carica nello stack.

`iconst_2` carica nello stack il valore 2. Al termine dell'esecuzione di queste due istruzioni, lo stato dello stack è il seguente:

```
      +---+
TOS ->| 2 |
      +---+
```

```
      | a |
      +---+
```

`idiv` prende i primi due valori in testa allo stack TOS, divide uno per l'altro e carica il risultato in testa allo stack TOS:

```
      +--------+
TOS ->| result |
      +--------+
```

`ireturn` ritorna indietro il primo valore trovato in testa allo stack.

Procediamo ora con numeri in virgola mobile e precisione doppia:

```
public class calc
{
        public static double half_double(double a)
        {
                return a/2.0;
        }
}
```

Listing 4.7: Constant pool

```
...
   #2 = Double             2.0d
...
```

```
  public static double half_double(double);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=4, locals=2, args_size=1
         0: dload_0
         1: ldc2_w        #2                  // double 2.0d
         4: ddiv
         5: dreturn
```

Il risultato delle decompilazione è simile al caso precedente, ma qui viene utilizzata l'istruzione `ldc2_w` per caricare nello stack il valore 2.0, a sua volta preso dal «constant pool».

Inoltre, le altre tre istruzioni hanno il prefisso *d*, ad indicare come esse lavorino con il tipo *double*.

Vediamo adesso delle funzioni con due argomenti:

```
public class calc
{
        public static int sum(int a, int b)
        {
                return a+b;
        }
}
```

```
  public static int sum(int, int);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=2, args_size=2
         0: iload_0
         1: iload_1
         2: iadd
         3: ireturn
```

`iload_0` carica il primo argomento (a), `iload_1`—il secondo (b).

Di seguito lo stato dello stack al termine di entrambe le istruzioni:

```
      +---+
TOS ->| b |
      +---+
      | a |
      +---+
```

`iadd` somma i due valori e carica il risultato in testa allo stack TOS:

```
      +--------+
TOS ->| result |
      +--------+
```

Estendiamo l'esempio al tipo di dato *long*:

```
        public static long lsum(long a, long b)
        {
                return a+b;
        }
```

...abbiamo:

```
  public static long lsum(long, long);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=4, locals=4, args_size=2
         0: lload_0
         1: lload_2
         2: ladd
         3: lreturn
```

La seconda istruzione `lload` prende l'argomento che si trova nello spazio dati in posizione 2.

Ciò accade poiché il tipo *long* occupa 64-bit, ossia due spazi da 32-bit, per cui la posizione del secondo argomento è 2 e non 1.

Ecco un esempio leggermente più complesso:

```
public class calc
{
```

```
        public static int mult_add(int a, int b, int c)
        {
                return a*b+c;
        }
}
```

```
  public static int mult_add(int, int, int);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=3, args_size=3
         0: iload_0
         1: iload_1
         2: imul
         3: iload_2
         4: iadd
         5: ireturn
```

Il primo passo è una moltiplicazione. Il prodotto viene caricato in testa allo stack TOS:

```
      +---------+
TOS ->| product |
      +---------+
```

`iload_2` carica il terzo argomento (c) nello stack:

```
      +---------+
TOS ->|    c    |
      +---------+
      | product |
      +---------+
```

L'istruzione `iadd` somma i due valori prelevati dalla testa dello stack.

4.1.4

4.1.5

4.1.6

Capitolo 5

5.1 Linux

5.1.1 *LD_PRELOAD* hack in Linux

Questo ci permette di caricare le nostre librerie dinamiche prima delle altre, anche quelle del sistema, come libc.so.6.

Questo a sua volta ci permette di «sostituire» la funzione che abbiamo scritto con quella nelle librerie del sistema. Ad esempio, è facile intercettare tutte le chiamate da time(), read(), write(), etc.

Proviamo ad ingannare l'utility *uptime*. Come sappiamo, essa ci dice da quanto tempo il computer sta lavorando. Con l'aiuto di strace(6.2.3 on page 303), è possibile osservare che l'utility prende questa informazione dal file `/proc/uptime`:

```
$ strace uptime
...
open("/proc/uptime", O_RDONLY)          = 3
lseek(3, 0, SEEK_SET)                   = 0
read(3, "416166.86 414629.38\n", 2047)  = 20
...
```

Questo non è un file presente su disco ma uno virtuale, il suo contenuto è generato al volo nel Linux kernel. Contiene due numeri:

```
$ cat /proc/uptime
416690.91 415152.03
```

Da Wikipedia possiamo imparare che [1]:

> Il primo numero è il numero totale di secondi che il sistema è acceso. Il secondo numero è la quantità di tempo che la macchina è rimasta in attesa (idle), in secondi.

[1] https://en.wikipedia.org/wiki/Uptime

Proviamo a scrivere la nostra libreria dinamica con le funzioni open(), read(), e close().

Come prima cosa, la funzione open() confronterà il nome del file da aprire con con quello che ci serve, se l'esito è positivo, scriverà il descrittore del file aperto.

In secondo luogo, la funzione read(), se chiamata per tale descrittore del file, sostituirà l'output, altrimenti chiamerà la funzione read() originale dalla libreria libc.so.6. E quindi la funzione close(), chiuderà il file che abbiamo utilizzato.

Useremo le funzioni dlopen() e dlsym() per determinare l'indirizzo della funzione originale in libc.so.6. Dobbiamo usare queste funzioni per passare il controllo alla «vera» funzione.

D'altra parte, se intercettiamo la chiamata a strcmp() e monitoriamo ogni confronto tra stringhe nel programma, dovremmo implementare una nostra versione di strcmp(), senza usare la funzione originale. [2]

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <dlfcn.h>
#include <string.h>

void *libc_handle = NULL;
int (*open_ptr)(const char *, int) = NULL;
int (*close_ptr)(int) = NULL;
ssize_t (*read_ptr)(int, void*, size_t) = NULL;

bool inited = false;

_Noreturn void die (const char * fmt, ...)
{
        va_list va;
        va_start (va, fmt);

        vprintf (fmt, va);
        exit(0);
};

static void find_original_functions ()
{
        if (inited)
                return;

        libc_handle = dlopen ("libc.so.6", RTLD_LAZY);
        if (libc_handle==NULL)
                die ("can't open libc.so.6\n");

        open_ptr = dlsym (libc_handle, "open");
```

[2] Ad esempio, in questo articolo trovi quanto facilmente strcmp() riesce ad intercettare le chiamate [3]
written by Yong Huang

```
        if (open_ptr==NULL)
                die ("can't find open()\n");

        close_ptr = dlsym (libc_handle, "close");
        if (close_ptr==NULL)
                die ("can't find close()\n");

        read_ptr = dlsym (libc_handle, "read");
        if (read_ptr==NULL)
                die ("can't find read()\n");

        inited = true;
}

static int opened_fd=0;

int open(const char *pathname, int flags)
{
        find_original_functions();

        int fd=(*open_ptr)(pathname, flags);
        if (strcmp(pathname, "/proc/uptime")==0)
                opened_fd=fd; // that's our file! record its file descriptor
        else
                opened_fd=0;
        return fd;
};

int close(int fd)
{
        find_original_functions();

        if (fd==opened_fd)
                opened_fd=0; // the file is not opened anymore
        return (*close_ptr)(fd);
};

ssize_t read(int fd, void *buf, size_t count)
{
        find_original_functions();

        if (opened_fd!=0 && fd==opened_fd)
        {
                // that's our file!
                return snprintf (buf, count, "%d %d", 0x7fffffff, 0↙
    ↳ x7fffffff)+1;
        };
        // not our file, go to real read() function
        return (*read_ptr)(fd, buf, count);
};
```

(Source code)

Compiliamolo con librerie dinamiche comuni:

```
gcc -fpic -shared -Wall -o fool_uptime.so fool_uptime.c -ldl
```

Avviamo *uptime* caricando prima le nostre librerie:

```
LD_PRELOAD=`pwd`/fool_uptime.so uptime
```

Osserviamo che:

```
 01:23:02 up 24855 days,  3:14,  3 users,  load average: 0.00, 0.01, 0.05
```

Se la variabile d'ambiente *LD_PRELOAD* punta sempre al nome del file ed al percorso della nostra libreria, deve essere per forza avviato per tutti i programmi che andremo ad avviare.

Altri esempi:

- Semplice intercettazione della funzione strcmp() (Yong Huang) https://yurichev.com/mirrors/LD_PRELOAD/Yong%20Huang%20LD_PRELOAD.txt
- Kevin Pulo—Fun with LD_PRELOAD. Molti esempio ed idee. yurichev.com
- Funzioni che intercettano file, per la compressione/decompressione di file al volo (zlibc). ftp://metalab.unc.edu/pub/Linux/libs/compression

5.2 Windows NT

5.2.1 Windows SEH

SEH

[Matt Pietrek, *A Crash Course on the Depths of Win32™ Structured Exception Handling*, (1997)][4], [Igor Skochinsky, *Compiler Internals: Exceptions and RTTI*, (2012)] [5].

[4] Italian text placeholderhttp://www.microsoft.com/msj/0197/Exception/Exception.aspx

[5] Italian text placeholderhttp://yurichev.com/mirrors/RE/Recon-2012-Skochinsky-Compiler-Internals.pdf

Capitolo 6

Strumenti

> Ora che Dennis Yurichev ha reso questo libro free (libre), è un contributo a tutto il mondo della libera informazione ed educazione. Comunque, per il bene della nostra libertà, abbiamo bisogno di strumenti free (libre) per il reverse engineering in modo da rimpiazzare quelli proprietari descritti in questo libro.
>
> Richard M. Stallman

6.1 Analisi di Binari

Strumenti da utilizzare senza eseguire nessun processo:

- (Free, open-source) *ent*[1]: strumento per analizzare l'entropia. Leggi di più riguardo l'entropia: **??** on page ??.
- *Hiew*[2]: per piccole modifiche del codice dei file binari.
- (Free, open-source) *xxd* e *od*: standard UNIX utility per effettuare il dump nel formato desiderato.
- (Free, open-source) *strings*: strumento *NIX per cercare stringhe ASCII all'interno di file binari, eseguibili compresi. Sysinternals ha un'alternativa[3] che supporta la stringhe con caratteri di tipo "wide" (UTF-16, ampiamente utilizzati in Windows).
- (Free, open-source) *Binwalk*[4]: analisi di immagini firmware.

[1] http://www.fourmilab.ch/random/
[2] hiew.ru
[3] https://technet.microsoft.com/en-us/sysinternals/strings
[4] http://binwalk.org/

- (Free, open-source) *binary grep*: piccola utility per cercare una sequenza di byte in molti file, incluso file non eseguibili: GitHub. C'è anche rafind2 in rada.re allo stesso scopo.

6.1.1 Disassemblers

- *IDA*. Una versione più vecchia è liberamente disponibile per lo scaricamento [5]. Elenco delle scorciatoie da tastiera: .3.1 on page 316
- *Binary Ninja*[6]
- (Free, open-source) *zynamics BinNavi*[7]
- (Free, open-source) *objdump*: semplice utility command-line per effettuare il dump e il disassembling.
- (Free, open-ssource) *readelf*[8]: dump delle informazioni dei file ELF.

6.1.2 Decompilers

C'è solo un decompiler conosciuto, pubblicamente disponibile e di elevata qualità per decompilare in C: *Hex-Rays*: hex-rays.com/products/decompiler/

Più informazioni su: **??** on page ??.

6.1.3 Comparazione Patch/diffing

Potresti voler utilizzare questi strumenti quando devi comparare la versione originale di un eseguibile con quella patchata, in modo da trovare cos'è stato patchato e perchè.

- (Free) *zynamics BinDiff*[9]
- (Free, open-source) *Diaphora*[10]

6.2 Analisi live

Strumenti da utilizzare per effettuare un'analisi live del sistema o di un processo in esecuzione.

6.2.1 Debuggers

- (Free) *OllyDbg*. Popolare win32 debugger[11]. Elenco delle scorciatoie da tastiera: .3.2 on page 317

[5] hex-rays.com/products/ida/support/download_freeware.shtml
[6] http://binary.ninja/
[7] https://www.zynamics.com/binnavi.html
[8] https://sourceware.org/binutils/docs/binutils/readelf.html
[9] https://www.zynamics.com/software.html
[10] https://github.com/joxeankoret/diaphora
[11] ollydbg.de

- (Free, open-source) *GDB*. Strumento non molto popolare tra reverse engineers perchè è per lo più inteso per programmatori. Alcuni comandi: .3.4 on page 318. C'è anche un'interfaccia per GDB, "GDB dashboard"[12].
- (Free, open-source) *LLDB*[13].
- *WinDbg*[14]: kernel debugger per Windows.
- (Free, open-source) *Radare* AKA rada.re AKA r2[15]. Esiste anche una GUI: *ragui*[16].
- (Free, open-source) *tracer*. L'autore usa spesso *tracer* [17] invece di un debugger.

  L'autore di queste righe ha smesso di utilizzare un debugger dato che l'unica cosa di cui aveva bisogno era di trovare gli argomenti delle funzioni durante l'esecuzione o lo stato dei registri ad un determinato punto. Caricare un debugger ogni volta risultava essere non ottimale, perciò è nacque una nuova utility chiamata *tracer*. Funziona dalla linea di comando e permette di intercettare l'esecuzione di funzioni, impostare breakpoint in posizioni arbitrarie, leggere e modificare lo stato dei registri, ecc.

  N.B.: *tracer* non sta evolvendo perchè è nato principalmente come strumento di dimostrazione per questo libro, non come strumento di ogni giorno.

6.2.2 Tracciare chiamate alle librerie

ltrace[18].

6.2.3 Tracciare chiamate di sistema

strace / dtruss

Mostra quali chiamate di sistema sono chiamate da un processo. (syscalls(**??** on page ??))

Per esempio:

```
# strace df -h

...

access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or ↙
    ↳ directory)
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF↙
    ↳ \1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\220\232\1\0004\0\0\0"..., ↙
    ↳ 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1770984, ...}) = 0
```

[12] https://github.com/cyrus-and/gdb-dashboard
[13] http://lldb.llvm.org/
[14] https://developer.microsoft.com/en-us/windows/hardware/windows-driver-kit
[15] http://rada.re/r/
[16] http://radare.org/ragui/
[17] yurichev.com
[18] http://www.ltrace.org/

```
mmap2(NULL, 1780508, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) ↙
    ↳ = 0xb75b3000
```

Mac OS X ha dtruss per lo stesso compito.

Cygwin ha strace ma, per quanto ne so, funziona solo per file .exe compilati all'interno dell'ambiente cygwin.

6.2.4 Network sniffing

Sniffing significa intercettare informazioni di cui si potrebbe essere interessati.

(Free, open-source) *Wireshark*[19] per lo sniffing di rete. Puà sniffare anche USB[20].

Wireshark ha un fratello chiamato *tcpdump*[21], un semplice strumento a linea di comando.

6.2.5 Sysinternals

(Free) Sysinternals (developed by Mark Russinovich) [22]. Questi strumenti sono importanti e vale la pena studiarli: Process Explorer, Handle, VMMap, TCPView, Process Monitor.

6.2.6 Valgrind

(Free, open-source) strumento per rilevare memory leak: http://valgrind.org/. A causa del suo potente meccanismo JIT, Valgrind è utilizzato come framework per altri strumenti.

6.2.7 Emulatori

- (Free, open-source) *QEMU*[23]: emulatore per differenti tipi di CPU e architetture.
- (Free, open-source) *DosBox*[24]: emulatore MS-DOS, usato soprattutto per il retrogaming.
- (Free, open-source) *SimH*[25]: emulatore di antichi computer, mainframe, ecc.

6.3 Altri strumenti

Microsoft Visual Studio Express [26]: Versione gratuita di Visual Studio, conveniente per semplici esperimenti.

[19] https://www.wireshark.org/
[20] https://wiki.wireshark.org/CaptureSetup/USB
[21] http://www.tcpdump.org/
[22] https://technet.microsoft.com/en-us/sysinternals/bb842062
[23] http://qemu.org
[24] https://www.dosbox.com/
[25] http://simh.trailing-edge.com/
[26] visualstudio.com/en-US/products/visual-studio-express-vs

Alcune opzioni utili: .3.3 on page 317.

C'è un sito chiamato "Compiler Explorer", che permette di compilare piccoli pezzi di codice e vederne l'output in varie versioni di GCC ed architetture differenti (almeno x86, ARM, MIPS): http://godbolt.org/—lo avrei utilizzato io stesso per il libro se lo avessi saputo!

6.3.1 Calcolatrici

Una buona calcolatrice, per le esigenze del reverse engineer, dovrebbe supportare almeno le basi decimale, esadecimale e binaria, ed operazioni importanti come XOR e gli shift.

- IDA ha una calcolatrice integrata ("?").
- rada.re ha *rax2*.
- https://yurichev.com/progcalc/
- Come ultima opzione, la calcolatrice standard di Windows ha una modalità programmatore.

6.4 Manca qualcosa qui?

Se conosci qualche strumento non elencato qui, per favore segnalamelo tramite e-mail al seguente indirizzo:
my emails.

6.5

6.6

Pierre Capillon – Black-box cryptanalysis of home-made encryption algorithms: a practical case study.

How to Hack an Expensive Camera and Not Get Killed by Your Wife.

Capitolo 7

Capitolo 8

Libri/blog da leggere

8.1 Libri ed altro materiale

8.1.1 Reverse Engineering

- Eldad Eilam, *Reversing: Secrets of Reverse Engineering*, (2005)
- Bruce Dang, Alexandre Gazet, Elias Bachaalany, Sebastien Josse, *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*, (2014)
- Michael Sikorski, Andrew Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, (2012)
- Chris Eagle, *IDA Pro Book*, (2011)
- Reginald Wong, *Mastering Reverse Engineering: Re-engineer your ethical hacking skills*, (2018)

Inoltre, i libri di Kris Kaspersky.

8.1.2 Windows

- Mark Russinovich, *Microsoft Windows Internals*
- Peter Ferrie – The “Ultimate” Anti-Debugging Reference[1]

:

- Microsoft: Raymond Chen
- nynaeve.net

[1] http://pferrie.host22.com/papers/antidebug.pdf

8.1.3 C/C++

- Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2ed, (1988)
- *ISO/IEC 9899:TC3 (C C99 standard)*, (2007)[2]
- Bjarne Stroustrup, *The C++ Programming Language, 4th Edition*, (2013)
- C++11 standard[3]
- Agner Fog, *Optimizing software in C++* (2015)[4]
- Marshall Cline, *C++ FAQ*[5]
- Dennis Yurichev, *C/C++ programming language notes*[6]
- JPL Institutional Coding Standard for the C Programming Language[7]

8.1.4 x86 / x86-64

- Manuali Intel[8]
- Manuali AMD[9]
- Agner Fog, *The microarchitecture of Intel, AMD and VIA CPUs*, (2016)[10]
- Agner Fog, *Calling conventions* (2015)[11]
- *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, (2014)
- *Software Optimization Guide for AMD Family 16h Processors*, (2013)

Un po' datati ma sempre interessanti:

Michael Abrash, *Graphics Programming Black Book*, 1997[12] (è conosciuto per i suoi lavori di ottimizzazione a basso livello su progetti come Windows NT 3.1 e id Quake).

8.1.5 ARM

- Manuali ARM[13]
- *ARM(R) Architecture Reference Manual, ARMv7-A and ARMv7-R edition*, (2012)

[2] Italian text placeholder http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf
[3] Italian text placeholder http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf.
[4] Italian text placeholder http://agner.org/optimize/optimizing_cpp.pdf.
[5] Italian text placeholder http://www.parashift.com/c++-faq-lite/index.html
[6] Italian text placeholder http://yurichev.com/C-book.html
[7] Italian text placeholder https://yurichev.com/mirrors/C/JPL_Coding_Standard_C.pdf
[8] Italian text placeholder http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
[9] Italian text placeholder http://developer.amd.com/resources/developer-guides-manuals/
[10] Italian text placeholder http://agner.org/optimize/microarchitecture.pdf
[11] Italian text placeholder http://www.agner.org/optimize/calling_conventions.pdf
[12] Italian text placeholder https://github.com/jagregory/abrash-black-book
[13] Italian text placeholder http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html

- [*ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, (2013)][14]
- Advanced RISC Machines Ltd, *The ARM Cookbook*, (1994)[15]

8.1.6 Assembly

Richard Blum — Professional Assembly Language.

8.1.7 Java

[Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, *The Java(R) Virtual Machine Specification / Java SE 7 Edition*] [16].

8.1.8 UNIX

Eric S. Raymond, *The Art of UNIX Programming*, (2003)

8.1.9 Programmazione in generale

- Brian W. Kernighan, Rob Pike, *Practice of Programming*, (1999)
- Henry S. Warren, *Hacker's Delight*, (2002). Alcune persone sostengono che i trucchi e gli hack di questo libro non siano più attuali adesso perchè erano validi solo per le CPU RISC, dove le istruzioni di branching sono costose. Ad ogni modo, possono aiutare enormemente a comprendere l'algebra booleana e tutta la matematica coinvolta.

8.1.10 Crittografia

- Bruce Schneier, *Applied Cryptography*, (John Wiley & Sons, 1994)
- (Free) lvh, *Crypto 101*[17]
- (Free) Dan Boneh, Victor Shoup, *A Graduate Course in Applied Cryptography*[18].

[14] Italian text placeholder http://yurichev.com/mirrors/ARMv8-A_Architecture_Reference_Manual_(Issue_A.a).pdf
[15] Italian text placeholder https://yurichev.com/ref/ARM%20Cookbook%20(1994)/
[16] Italian text placeholder https://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf; http://docs.oracle.com/javase/specs/jvms/se7/html/
[17] Italian text placeholder https://www.crypto101.io/
[18] Italian text placeholder https://crypto.stanford.edu/~dabo/cryptobook/

Capitolo 9

Community

Esistono due eccellenti subreddit riguardo il RE[1] su reddit.com: reddit.com/r/ReverseEngineering/ e reddit.com/r/remath (Sugli argomenti di intersezione fra RE e matematica).

Esiste anche una sezione relativa a RE sul sito Stack Exchange: reverseengineering.stackexchange.com.

In IRC c'è un canale ##re su Libera.

[1] Reverse Engineering

Afterword

9.1 Domande?

Non esitate a inviare una e-mail all'autore in caso abbiate una domanda o un dubbio o altro:
my emails. Hai dei suggerimenti su dei contenuti da aggiungere al libro? Per favore, non esitare ad inviare qualsiasi correzione (compresse quelle grammaticali), ecc.

L'autore sta lavorando molto sul libro quindi i numeri delle pagine e gli elenchi numerati, etc., cambiano molto rapidamente. Per favore non riferitevi a numeri di pagine quando mi scrivere un'e-mail. C'è un metodo molto più semplice: fai uno screenshot della pagina e, tramite un editor per le foto, sottolinea il posto in cui hai visto l'errore e inviamelo. Lo sistemerò molto più velocemente! E se hai familiarità con git e LaTeX allora puoi correggere l'errore direttamente nei sorgenti:

https://beginners.re/src/.

Non farti scrupoli nell'inviarmi gli errori che hai trovato anche nel caso non fossi sicuro(a). Sto scrivendo per principianti e quindi la vostra opinione è estremamente importante per me.

Appendice

.1 x86

.1.1 Terminologia

Comune per 16-bit (8086/80286), 32-bit (80386, etc.), 64-bit.

byte 8-bit. La direttiva DB in assembly è utilizzata per definire variabili ed array di byte. I byte sono passati nella parte ad 8-bit dei registri: AL/BL/CL/DL/AH/BH/CH/DH/SIL/DIL/R*L

word 16-bit. La direttiva DW in assembly —"—. Le Word sono passate nella parte a 16-bit dei registri:
AX/BX/CX/DX/SI/DI/R*W.

double word («dword») 32-bit. La direttiva DD in assembly —"—. Le Double words sono passate nei registri (x86) o nelle parti a 32-bit dei registri (x64). Nel codice a 16-bit, le double words sono passate in coppie di registri a 16-bit.

quad word («qword») 64-bit. La direttiva DQ in assembly —"—. Negli ambienti a 32-bit, le quad words sono passate in coppie di registri a 32-bit.

tbyte (10 bytes) 80-bit o 10 bytes (usati per i registri FPU IEEE 754).

paragraph (16 bytes)—termine popolare nell'ambiente MS-DOS.

Tipi di dati della stessa dimensione (BYTE, WORD, DWORD) sono gli stessi nelle Windows API[2].

.1.2 npad

`listing.inc` (MSVC):

```
;; LISTING.INC
;;
;; This file contains assembler macros and is included by the files created
;; with the -FA compiler switch to be assembled by MASM (Microsoft Macro
;; Assembler).
;;
;; Copyright (c) 1993-2003, Microsoft Corporation. All rights reserved.

;; non destructive nops
npad macro size
if size eq 1
  nop
else
 if size eq 2
   mov edi, edi
 else
  if size eq 3
    ; lea ecx, [ecx+00]
    DB 8DH, 49H, 00H
  else
   if size eq 4
     ; lea esp, [esp+00]
```

[2] Application Programming Interface

```
    DB 8DH, 64H, 24H, 00H
  else
   if size eq 5
     add eax, DWORD PTR 0
   else
    if size eq 6
      ; lea ebx, [ebx+00000000]
     DB 8DH, 9BH, 00H, 00H, 00H, 00H
    else
     if size eq 7
       ; lea esp, [esp+00000000]
      DB 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H
     else
      if size eq 8
       ; jmp .+8; .npad 6
       DB 0EBH, 06H, 8DH, 9BH, 00H, 00H, 00H, 00H
      else
       if size eq 9
        ; jmp .+9; .npad 7
        DB 0EBH, 07H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H
       else
        if size eq 10
         ; jmp .+A; .npad 7; .npad 1
         DB 0EBH, 08H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 90H
        else
         if size eq 11
          ; jmp .+B; .npad 7; .npad 2
          DB 0EBH, 09H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8BH, 0FFH
         else
          if size eq 12
           ; jmp .+C; .npad 7; .npad 3
           DB 0EBH, 0AH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8DH, 49H, 00H
          else
           if size eq 13
            ; jmp .+D; .npad 7; .npad 4
            DB 0EBH, 0BH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8DH, 64H, 24↙
↳ H, 00H
           else
            if size eq 14
             ; jmp .+E; .npad 7; .npad 5
             DB 0EBH, 0CH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 05H, 00H, ↙
↳ 00H, 00H, 00H
            else
             if size eq 15
              ; jmp .+F; .npad 7; .npad 6
              DB 0EBH, 0DH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8DH, 9BH, ↙
↳ 00H, 00H, 00H, 00H
             else
              %out error: unsupported npad size
              .err
             endif
            endif
           endif
```

```
           endif
          endif
         endif
        endif
       endif
      endif
     endif
    endif
   endif
  endif
 endif
endif
endm
```

.2

`ll` «long long», .

__alldiv	
__allmul	
__allrem	
__allshl	
__allshr	
__aulldiv	
__aullrem	
__aullshr	

.

`VC/crt/src/intel/*.asm`.

.3 Cheatsheets

.3.1 IDA

Elenco delle scorciatoie da tastiera:

Space C D A * U O H R B Q N ? G : Ctrl-X X Alt-I Ctrl-I Alt-B Ctrl-B Alt-T Ctrl-T Alt-P Enter Esc Num - Num +	

. .

.3.2 OllyDbg

Elenco delle scorciatoie da tastiera:

F7 F8 F9 Ctrl-F2	step over

.3.3 MSVC

. .

/O1 /Ob0 /Ox /GS- /Fa(file) /Zi /Zp(n) /MD	 MSVCR*.DLL

: **??** on page ??.

.3.4 GDB

:

break filename.c:number	
break function	
break *address	
b	—”—
p variable	
run	
r	—”—
cont	
c	—”—
bt	
set disassembly-flavor intel	
disas	disassemble current function
disas function	
disas function,+50	disassemble portion
disas $eip,+0x10	—”—
disas/r	
info registers	
info float	
info locals	
x/w ...	
x/w $rdi	
x/10w ...	
x/s ...	
x/i ...	
x/10c ...	
x/b ...	
x/h ...	
x/g ...	
finish	
next	
step	
set step-mode on	
frame n	
info break	
del n	
set args ...	

Acronimi utilizzati

Glossario

Italian text placeholder Italian text placeholder. 242–244

anti-pattern Generalmente considerata una cattiva pratica. 44, 102, 283

chiamante Una funzione chiamante. 8–11, 14, 40, 63, 115, 128–130, 133, 143, 202

chiamata Una funzione chiamata. 44, 45, 63, 90, 115, 128, 131, 134, 283

decrementa Decrementa di 1. 25, 236, 260

endianness L'ordine dei byte. 29, 105

funzione foglia Una funzione che non chiama nessun' altra funzione. 38, 44

Funzione thunk Piccola funzione con un solo scopo: chiamare un' altra funzione. 30, 57

GiB Gibibyte: 2^{30} o 1024 megabyte o 1073741824 byte. 21

heap di solito, una grossa locazione di memoria fornito da ac OS in modo che le applicazioni possano dividerla da sole come desiderano. malloc () / free () lavorano con l'heap. 42

incrementa Incrementa di 1. 22, 26, 236, 241, 260, 266

ingegneria inversa L' atto di comprendere come una cosa funziona, a volte per clonarla. iv

numero reale Italian text placeholder. 279

offset di salto Italian text placeholder. 123, 170, 171

prodotto Risultato di una moltiplicazione. 129

quoziente Risultato di una divisione. 278

registro allocatore La parte del compilatore che assegna i registri della CPU alle variabili locali. 258

registro link (RISC) Un registro in cui generalmente l'indirizzo di ritorno viene salvato. Ciò rende possibile chiamare funzioni foglia più velocemente, ovvero senza l'utilizzo dello stack. 44

srotolamente del ciclo E' quando un compilatore, anzichè generare il codice di un ciclo per n iterazioni, genera n copie del corpo del ciclo, al fine di sbarazzarsi delle istruzioni per la manutenzione del ciclo. 239

stack frame Italian text placeholder. 91, 92, 129

stack pointer Un registro che punta nello stack. 13, 15, 26, 41, 47, 58, 73, 75, 98, 131

stdout Standard output. 28, 48, 202

Indice analitico