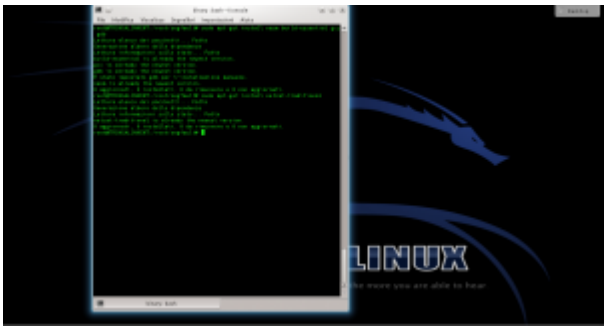
Hacking&Cracking: Buffer overflow, un tutorial passo passo

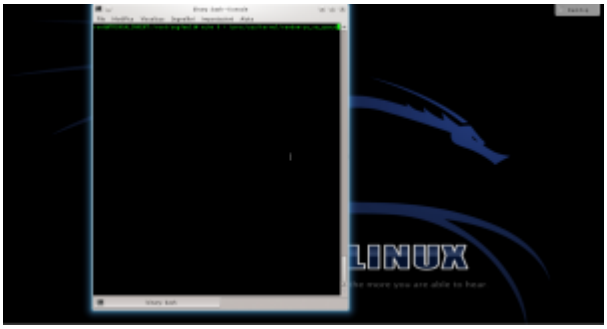
Nella puntata precedente di questa mini-serie (https://www.codice-sorgente.it/2019/06/buffer-overflow-e-errori-di-segmentazione-della-memoria/) abbiamo descritto il funzionamento della memoria di un computer, e in particolare gli overflow nello stack. In questo breve articolo presentiamo un tutorial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

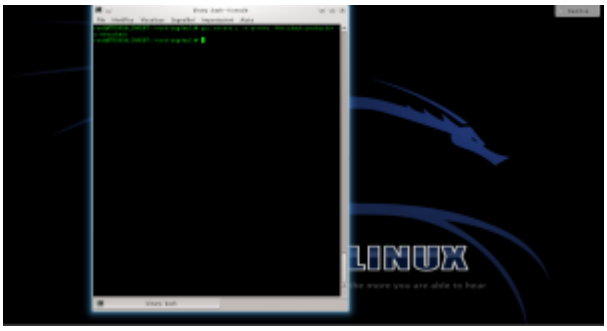
Per testare questi esempi bisogna innanzitutto avere a disposizione un sistema operativo a 32 bit, possibilmente su una macchina virtuale per mantenere stabile il proprio sistema host. Bisogna poi disabilitare alcune norme di sicurezza di Linux, altrimenti l'analisi della vulnerabilità e l'esecuzione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



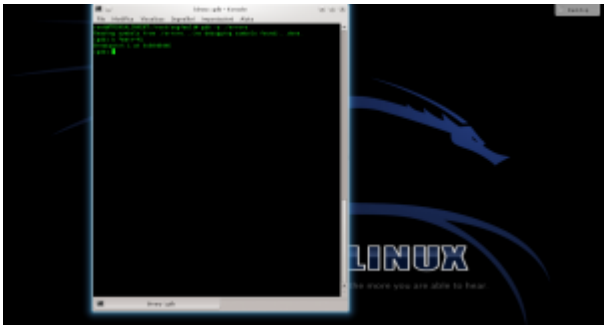
Bisogna ora procurarsi il programma buggato: per esempio, si può scaricare il file errore.c (https://pastebin.com/8DZQZzqx). Il programma va compilato con il comando
In questo modo, il programma viene compilato senza le protezioni per lo stack inserite automaticamente da GCC. Naturalmente, si potrebbe 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.

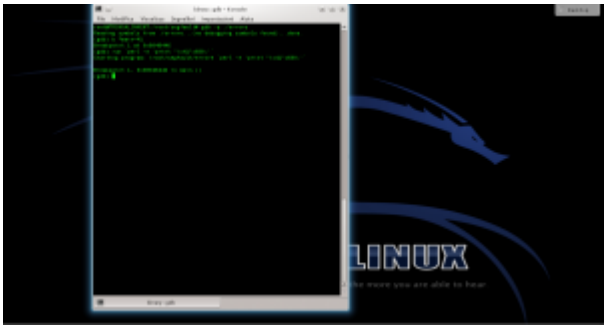
In una situazione reale il codice sorgente potrebbe non essere

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.

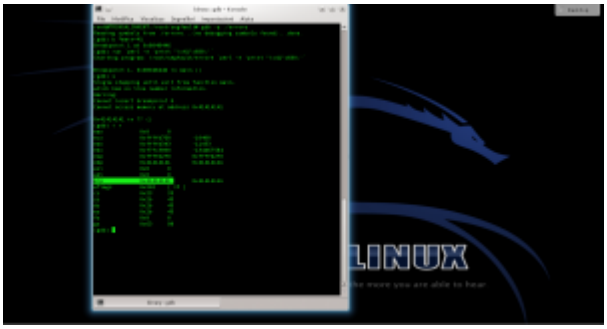


Ora, dobbiamo studiare il programma vulnerabile per capire quali indirizzi di memoria possiamo utilizzare. Serve un debugger quindi, supponendo di voler utilizzare il programma "errore" precedentemente compilato, il pirata da il comando
Aperto il debugger, possiamo disassemblare il programma per leggere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

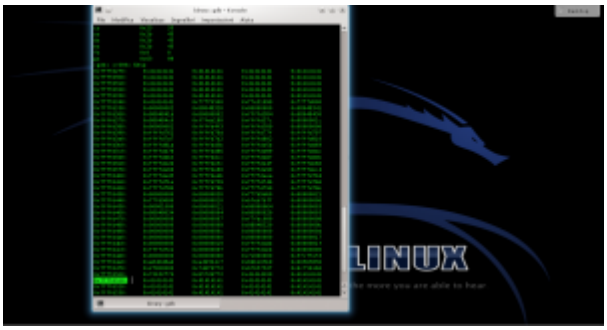
Dal listato si capisce che l'istruzione di ritorno della funzione (**leave**) è nel punto **+41**.



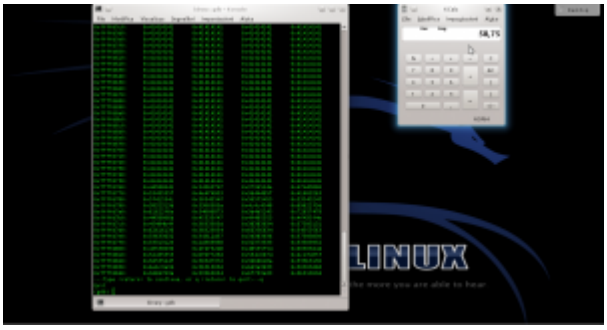
Impostiamo quindi un breakpoit per il controllo del programma prima dell'istruzione di ritorno della funzione buggata, scrivendo Poi proviamo a far crashare il programma fornendogli una stringa di 600 caratteri con il comando



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**.

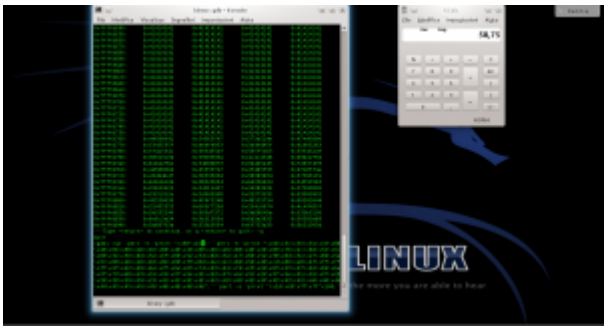


Questo è l'indirizzo in cui sarà inserita la nop sled. Una buona dimensione potrebbe essere 100 byte. Però, lo shellcode è lungo 135 byte, e la somma (235) non è divisibile per 4. Il numero 236, però, lo è. Quindi la nop sled dovrà contenere 101

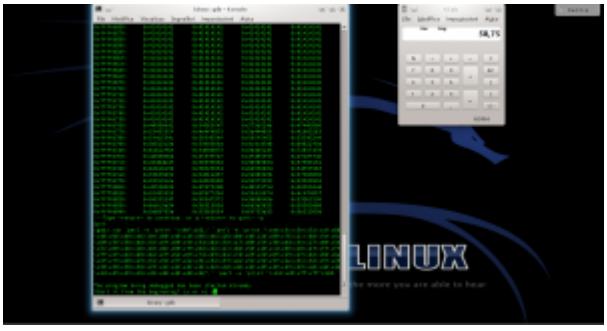
byte, per evitare sfasamenti.

Il payload

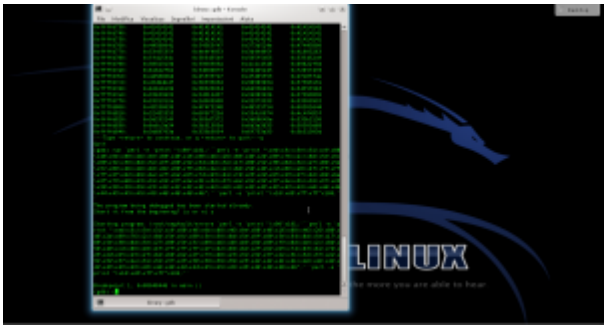
Ormai abbiamo la dimensione della NOP sled e anche l'indirizzo di ritorno. Ci manca soltanto lo shellcode, che possiamo recuperare da un elenco online (come quelli pubblicati su exploit-db.com.



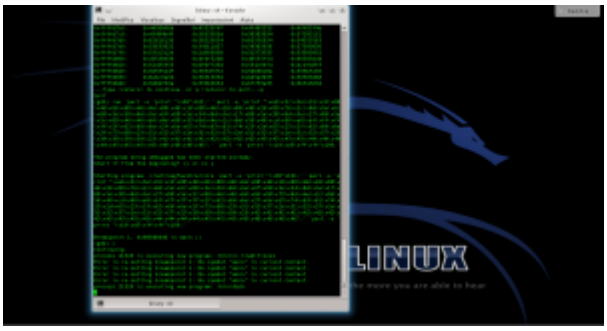
Possiamo quindi scrivere la stringa completa (https://pastebin.com/biSxHhRT): 101 byte del carattere NOP (90), seguiti dallo shellcode, e poi dall'indirizzo di ritorno scritto al contrario per mantenere la codifica little endian, ripetuto almeno un centinaio di volte.



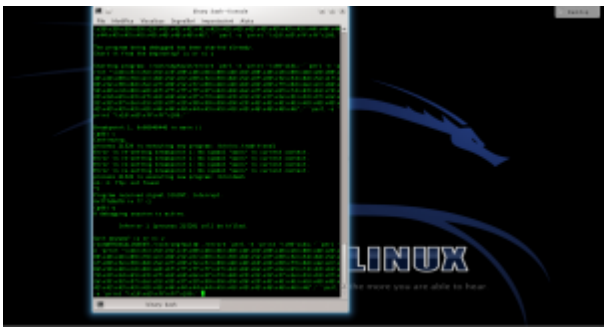
Basta eseguire il programma con il comando seguito dalla stringa completa: ovviamente, GDB chiederà conferma, visto che si deve riavviare il programma attualmente fermo al breakpoint. Digitiamo e il programma viene lanciato di nuovo ma con l'argomento costruito dai vari comandi Perl.



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.