„Śmieciowe” wartości zmiennych w C/CPP (sheadovas/artykuly/programowanie/smieciowe-wartosci-zmiennych-w-ccpp/)

Paź 28, 2017 / Programowanie (sheadovas/category/artykuly/programowanie/)

Jak się okazuje nie są do końca ani losowe, ani śmieciowe. Spróbujemy nad nimi zapanować...

Hej, dzisiaj chcę wspomnieć parę słów o tzw. „śmieciowych”wartościach zmiennychw C/C++ , które mają zmienne na start w takiej sytuacji:

Listing 1 C

```c
int variable;
printf("%s\n", variable);
```

W każdym tutorialu z C ostrzega się przed użyciem zmiennej, która jest utworzona ale nie ma przypisanej wartości. Często mówi się, że są tam „śmieci” lub co jest akurat błędne: wartość pod

`variable`

jest „losowa”.

Dzisiaj chciałbym pokazać, że można zapanować nad takimi wartościami, tzn. jak można przewidzieć ich wartość.

> Uwaga: ten artykuł należy traktować jako ciekawostkę, a nie użyteczny trick w programowaniu. Działanie tego tricku jest bardzo łatwo zaburzyć, co też sobie pokażemy.

PoC

Na wstępie warto zaznaczyć z jakiego środowiska korzystam, a więc:

- Linux x86_64
- GCC 5.4.0

Kod do testów:

Listing 2 C

```c
#include <stdio.h>

void fun1() {
  unsigned long foo = 0xc0ffebabe;
}

void fun2() {
  unsigned long bar;
  printf("%lx\n", bar);
}

int main() {
  fun1();
  fun2();

  return 0;
}
```

Widzimy, że powyższy listing jest oczywiście „niepoprawny”, bo korzysta z śmieciowej (nieprzypisanej) wartości zmiennej *a*. Zobaczmy sobie co się stanie gdy skompilujemy sobie powyższy listing:

Listing 4 Shell

```
$ gcc foo.c -o foo
$ ./foo
c0ffebabe
```

Dla pewności możemy uruchomić sobie powyższy kod kilka razy:

Listing 5 Shell

```
$ for i in {1..10}; do ./foo; done
c0ffebabe
c0ffebabe
c0ffebabe
c0ffebabe
c0ffebabe
c0ffebabe
c0ffebabe
c0ffebabe
c0ffebabe
c0ffebabe
```

Widzimy, więc że powyższa wartość nie jest przypadkowa, a na pewno nie „losowa” jak twierdzą niektórzy prowadzący zajęcia z C/C++. Co więcej: ta wartość jest równa dokładnie tyle ile sami ustaliliśmy w funkcji

`foo1()`

.

Zatem mamy kontrolę nad tym co się znajduje w zmiennej

`bar`

, mimo że nic do niej nie przypisujemy. Jak to się dzieje?

Wyjaśnienie

Prześledźmy *Listing2* jeszcze raz, ale z dodatkowym komentarzem:

Listing 6 C

```c
#include <stdio.h>

void fun1() {
  unsigned long foo = 0xc0ffebabe; // alokacja zmiennej "foo", przypisanie wartości
} // "usunięcie" zmiennej foo

void fun2() {
  unsigned long bar; // alokacja zmiennej "bar"
  printf("%lx\n", bar); // wydrukowanie jej zawartości
}

int main() {
  fun1(); // wywołanie funkcji
  fun2(); //

  return 0;
}
```

Aby zrozumieć co tu się stało należy najpierw zejść nieco niżej, a więc do tego jak to wygląda z poziomu assemblera.

W funkcji *fun1* alokacja zmiennej *foo* to nic innego jak odjęcie 8 bajtów od wskaźnika stosu (dla niewtajemniczonych: stos rośnie „w dół” – stąd odejmowanie od wskaźnika; wskaźnik stosu kryje się pod rejestrem RSP).

Następnie w to miejsce przypisywana jest wartość *0xc0ffebabe*, po czym zwalniamy pamięć, a więc tym razem dodajemy 8 bajtów to RSP. Należy zauważyć, że nigdzie nie zerujemy tej wartości przed usunięciem, ona wciąż jest w tym miejscu.

W pseudo-kodzie assembly, gdzie stos jest tablicą funkcja *fun1* może wyglądać następująco:

Listing 8 Assembly (x86)

```
fun1:
rsp = rsp - 8 ; alokacja zmiennej 'foo'
stack[rsp] = 0xc0ffebabe ; przypisanie wartości
rsp = rsp + 8 ; zwolnienie pamięci po zmiennej 'foo'

ret ; powrót do funkcji 'main'
```

Stos obrazkowo:

Listing 9

```
Wejście w fun1, jeszcze przed alokacją:
| AAA | ... | XXX | YYY | ... | ZZZ |  <---- stos
   ^           ^
 jakieś       RSP
wartości


Alokacja:
| AAA | ... | XXX | YYY | ... | ZZZ |
                    ^
                   RSP

Przypisanie wartości:
| AAA | ... | XXX | 0xc0ffebabe | ... | ZZZ |
                       ^
                      RSP

Zwolnienie pamięci:
| AAA | ... | XXX | 0xc0ffebabe | ... | ZZZ |
                ^
               RSP
```

Z kolei *fun2* to alokacja zmiennej (odjęcie wartości o 8), wydrukowanie zmiennej spod adresu w którym za alokowana jest zmienna i zwolnienie pamięci:

Listing 10 — Assembly (x86)

```
fun2:
rsp = rsp - 8 ; alokacja 'bar'
call printf, stack[rsp] ; wydrukowanie zmiennej spod miejsca gdzie wskazuje rsp
rsp = rsp + 8 ; zwolnienie pamięci

ret ; powrót do main
```

Znowu obrazki:

Listing 11

```
W fun2, przed alokacją:
| AAA | ... | XXX | 0xc0ffebabe | ... | ZZZ |
                ^
               RSP

Po alokacji:
| AAA | ... | XXX | 0xc0ffebabe | ... | ZZZ |
                       ^
                      RSP

Wywołanie printf używa wartości spod RSP!

Zwolnienie:
| AAA | ... | XXX | 0xc0ffebabe | ... | ZZZ |
                ^
               RSP
```

Dla upewnienia można zobaczyć do samo przy zżucie kodu assembly:

Listing 12 — Assembly (x86)

```
(gdb) disas main
Dump of assembler code for function main:
   0x000000000040055c <+0>:     push   rbp
   0x000000000040055d <+1>:     mov    rbp,rsp
   0x0000000000400560 <+4>:     mov    eax,0x0
   0x0000000000400565 <+9>:     call   0x400526 <fun1>
   0x000000000040056a <+14>:    mov    eax,0x0
   0x000000000040056f <+19>:    call   0x40053b <fun2>
   0x0000000000400574 <+24>:    mov    eax,0x0
   0x0000000000400579 <+29>:    pop    rbp
   0x000000000040057a <+30>:    ret
End of assembler dump.
(gdb) disas fun1
Dump of assembler code for function fun1:
   0x0000000000400526 <+0>:     push   rbp
   0x0000000000400527 <+1>:     mov    rbp,rsp
   0x000000000040052a <+4>:     movabs rax,0xc0ffebabe
   0x0000000000400534 <+14>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000000000400538 <+18>:    nop
   0x0000000000400539 <+19>:    pop    rbp
   0x000000000040053a <+20>:    ret
End of assembler dump.
(gdb) disas fun2
Dump of assembler code for function fun2:
   0x000000000040053b <+0>:     push   rbp
   0x000000000040053c <+1>:     mov    rbp,rsp
   0x000000000040053f <+4>:     sub    rsp,0x10
   0x0000000000400543 <+8>:     mov    rax,QWORD PTR [rbp-0x8]
   0x0000000000400547 <+12>:    mov    rsi,rax
   0x000000000040054a <+15>:    mov    edi,0x400604
   0x000000000040054f <+20>:    mov    eax,0x0
   0x0000000000400554 <+25>:    call   0x400400 <printf@plt>
   0x0000000000400559 <+30>:    nop
```

Jak widzimy jeden z mechanizmów, który tu zachodzi to „przypadek” dzięki, któremu nowa zmienna (bar) wskazuje na to samo miejsce na stosie co stara zmienna (foo). Na ten przypadek składa się kilka rzeczy, m.in:

1. Typ (wielkość) zmiennych,
2. Ilość zmiennych zarówno wewnątrz funkcji, jak i ilości argumentów,
3. Sposób wywoła (t.j. wywołanie *fun2* z funkcji *fun1* da nam zupełnie inny rezultat),
4. Proces kompilacji.

Kiedy to nie działa?

Jak widzimy takie „zgadnięcie”, czy też kontrola nad wartością nieprzypisanej zmiennej wymaga dość specyficznego środowiska wynikającego głównie z tego jak zmienne ułożone są na stosie. Nie zadziała to także, gdy zmienne są różnej długości, tzn zadziała, ale wartości mogą być różnie interpretowane zależnie od typu.

Inną ciekawą sytuacją jest moment gdy skompilujemy powyższy kod z inną opcją:

Listing 13 Shell

```
$ gcc -O3 foo.c -o foo
$ ./foo
0
```

Zadaniem domowym jest dowiedzenie się co się właściwie stało i dlaczego w wyniku jest ;)

Podsumowanie

Podsumowując: istnieją specyficzne sytuacje kiedy jesteśmy w stanie przewidzieć „śmieciowe” wartości zmiennych. To co możemy na pewno powiedzieć, to to że te wartości nie są losowe.

Jeżeli chodzi o wykorzystanie tej własności, to korzysta się z niej od czas do czasu w bezpieczeństwie, a konkretniej eksploitacji kodu. Trzeba mieć też świadomość, że ja pokazałem jedynie najprostszą z możliwych wersji takiego programistycznego błędu (bo tym właśnie jest korzystanie z niezainicjalizowanych zmiennych), osobiście spotkałem się z takim mechanizmem na kilku CTF’ach ;)

Zachęcam do eksperymentów z bogatszymi wersjami kodu powyżej (np. mamy zmienne ale różnych typów, więcej różnych zmiennych, ...).

Dajcie znać czy nie chcielibyście więcej ciekawostek i tricków tego typu. Na koniec tradycyjnie zapraszam do komentarzy, a także do śledzenia bloga przez social-media.

Code ON!