Architektury systemów komputerowych

Pracownia 2: „Atak hakerski (cz. 1)"

Termin oddawania 7 maja 2019

Wprowadzenie

Dane są dwa programy, przygotowane indywidualnie dla każdego studenta, wykazujące podatność na atak z przepełnieniem bufora. Analizując je nauczysz się odkrywać i wykorzystywać takie podatności. Zrozumiesz, jak pisać bardziej bezpieczne programy oraz jak pomaga Ci w tym system operacyjny i kompilator. Przećwiczysz binarne kodowanie instrukcji procesora o architekturze `x86-64` oraz konwencję wołania procedur [2].

UWAGA! Przed przystąpieniem do rozwiązywania zadań należy zapoznać się z [1, §3.10.3] i [1, §3.10.4].

Pliki

Na stronie przedmioty w systemie SKOS znajduje się plik «`targets.tar.xz`». Ściągnij go i rozpakuj! Zadania przeznaczone dla Ciebie znajdują się w katalogu «`target`*indeks*», gdzie *indeks* jest numerem Twojego indeksu.

W katalogu tym znajduje się pięć plików:

- «`ctarget`» Plik wykonywalny podatny na atak przez **wstrzyknięcie kodu** (ang. *code injection*).
- «`rtarget`» Plik wykonywalny podatny na atak przez tworzenie **łańcuchów powrotów z procedur** (ang. *return oriented programming*).
- «`cookie.txt`» Unikatowy kod (osiem cyfr szesnastkowych), zwany dalej **ciasteczkiem**, który wykorzystasz w trakcie przeprowadzania ataków.
- «`farm.c`» Kod źródłowy zawierający **gadżety** (ang. *gadget*), które należy wykorzystać w tworzeniu łańcuchów powrotów z procedur.
- «`hex2raw`» Narzędzie do generowania kodu przeznaczonego do wstrzyknięcia.

Wytyczne

Poniżej wymieniono podstawowe zasady dotyczące prawidłowego rozwiązania zadań z tej listy.

1. Zadania należy rozwiązywać pod systemem `Linux` działającym pod kontrolą procesora o architekturze `x86-64`. Dostarczone pliki wykonywalne zostały skompilowane pod systemem `Debian GNU/Linux 9` z wyłączoną opcją `PIE` (ang. *position independent executable*).
2. Dane pobierane przez instrukcje «`ret`» muszą być:
   - adresami procedur «`touch1`», «`touch2`» lub «`touch3`»,
   - lub adresami we wstrzykniętym przez Ciebie kodzie,
   - lub adresami wskazującymi na kod wewnątrz Twojej **hodowli gadżetów** (ang. *gadget farm*).
3. Możesz konstruować gadżety wyłącznie z bajtów pliku «`rtarget`» zawartych między adresami symboli «`start_farm`» i «`end_farm`».

Programy `ctarget` i `rtarget`

Programy «ctarget» and «rtarget» czytają łańcuchy znaków ze standardowego wejścia «stdin». Używają do tego następującej dziurawej procedury:

```
1 unsigned getbuf(void) {
2   char buf[BUFFER_SIZE];
3   Gets(buf);
4   return 1;
5 }
```

Procedura Gets realizuje podobne zadanie co biblioteczna procedura «gets», tj. czyta ona ciąg znaków ze standardowego wejścia (zakończonego znakiem '\n' lub EOF) i zapisuje go pod adres «buf» dodając terminator '\0' na końcu ciągu. Bufor ma stały rozmiar wynoszący «BUFFER_SIZE» bajtów. Wartość tej stałej jest ustalona indywidualnie w trakcie kompilacji programów dla studentów.

Procedury «Gets()» i «gets()» nie mają możliwości określenia, czy zmienna «buf» ma rozmiar wystarczający do pomieszczenia odczytanego ciągu znaków. Umieszczczają one kolejne znaki odczytane ze standardowego wejścia w buforze. Oczekują naiwnie, że w buforze jest wystarczająco dużo miejsca, zatem mogą nadpisać sąsiadującą z nim pamięć.

Jeśli podany na wejściu ciąg znaków jest odpowiednio krótki, to procedura «getbuf» zakończy się powodzeniem, jak na poniższym przykładzie:

```
$ ./ctarget
Cookie: 0x1a7dd803
Type string: Keep it short!
No exploit.  Getbuf returned 0x1
Normal return
```

Gdy wpisany zostanie odpowiednio długi ciąg znaków, zwykle wystąpi błąd wykonania programu:

```
$ ./ctarget
Cookie: 0x1a7dd803
Type string: This is not a very interesting string, but it has the property ...
Ouch!: You caused a segmentation fault!
Better luck next time
```

UWAGA! Wartość ciasteczka na powyższych wydrukach jest inna niż w Twojej wersji zadania.

Przepełnienie bufora zwykle uszkadza działający program na tyle, że dalsze jego wykonanie nie jest możliwe. Twoim zadaniem jest wymuszenie innego zachowania, tj. przechwycenie kontroli działania programu przez skonstruowanie ciągów znaków zwanych **nadużyciami** (ang. *exploit*).

Programy «ctarget» i «rtarget» pobierają następujące parametry wiersza poleceń:

- «-h» Wydrukuj listę wszystkich dopuszczalnych parametrów programu.
- «-q» Nie wysyłaj rezultatów do serwera oceniającego.
- «-i PLIK» Zamiast wczytywać dane ze standardowego wejścia wczytaj je z pliku «PLIK».

UWAGA! W bieżącym semestrze serwer oceniający jest nieaktywny. Należy zawsze używać opcji «-q»!

Nadużycia będą zawierały znaki o kodach, które niekoniecznie odpowiadają drukowalnym symbolom ASCII. Pomocniczy program «hex2raw» przekształca ciąg cyfr szesnastkowych kodujących wartości bajtów na ich reprezentację binarną generując **surowe dane** (ang. *raw strings*).

Kiedy rozwiążesz już jedno z zadań program wydrukuje odpowiedni komunikat:

```
$ ./hex2raw < ctarget.l2.txt | ./ctarget -q
Cookie: 0x1a7dd803
Type string:Touch2!: You called touch2(0x1a7dd803)
Valid solution for level 2 with target ctarget
PASS: Would have posted the following:
  ...
```

Należy wtedy zawartość pliku «`ctarget.l2.txt`» umieścić w odpowiedniej rubryce w systemie SKOS.

Narzędzie `hex2raw`

Więcej informacji o narzędziu «`hex2raw`» znajdziesz w sekcji **??**, tutaj znajdziesz tylko najistotniejsze uwagi:

- Wewnątrz ciągu znaków stanowiących *exploit* nie może wystąpić bajt o wartości 10 zapisywany symbolicznie jako '`\n`'. Procedura «`Gets`» kończy wczytywanie ciągu znaków wraz z pojawieniem się znaku końca linii.
- Narzędzie «`hex2raw`» pobiera ze standardowego wejścia tekst kodujący surowe dane. Tekst składa się z dwucyfrowych wartości szesnastkowych oddzielonych od siebie przynajmniej jednym białym znakiem. Zatem, jeśli zamierzasz wytworzyć znak o kodzie 0, należy go zapisać jako «`00`». Aby zapisać słowo czterobajtowe `0xDEADBEEF` należy na standardowe wejście przekazać ciąg «`EF BE AD DE`». Zauważ, że w grę wchodzi tu porządek bajtów w słowie – w tym przypadku *little-endian*.

1 Atak przez wstrzyknięcie kodu

Trzy pierwsze zadnia polegają na stworzeniu ciągów znaków dla programu «ctarget». Program ten jest celowo skonstruowany tak, że stos jest wykonywalny i przy każdym uruchomieniu programu zawsze znajduje się pod tą samą ustaloną pozycją. To powoduje, że program jest podatny na atak, gdzie surowe dane nadużycia kodują instrukcje wstrzykiwanego kodu.

1.1 Zadanie 1 (2 punkty)

W tym zadaniu stworzysz nadużycie, który spowoduje wykonanie procedury znajdującej się w obrazie programu «ctarget». Procedura «getbuf» jest wywołana w programie «ctarget» przez procedurę «test»:

```
1 void test() {
2   int val;
3   val = getbuf();
4   printf("No exploit.  Getbuf returned 0x%x\n", val);
5 }
```

Po powrocie z «getbuf» program kontynuuje wykonanie procedury «test». Chcemy to zmienić! W programie «ctarget» zdefiniowano procedurę «touch1» o następującym kodzie:

```
1 void touch1() {
2   vlevel = 1; /* Part of validation protocol */
3   printf("Touch1!: You called touch1()\n");
4   validate(1);
5   exit(0);
6 }
```

Twoim zadaniem jest zmuszenie programu «ctarget» do wywołania procedury «touch1» w momencie powrotu z procedury getbuf. To znaczy, że powrót z wywołania «getbuf» zamiast przekazać sterowanie do «test», powinien spowodować rozpoczęcie wykonania «touch1».

Twoje nadużycie może w dowolny sposób nadpisać stos programu – i tak jego zawartość już Ci się nie przyda jako, że procedura «touch1» wymusza zakończenie programu przez wywołanie systemowe «exit».

Kilka wskazówek:

- Zadanie to możesz rozwiązać posługując się jedynie zdezasemblowanym kodem programu «ctarget». Użyj polecenia «objdump -d».
- Skonstruuj surowe dane tak, by instrukcja «ret» procedury «test» przekazała sterowanie do «touch1».
- Pamiętaj o kolejności bajtów w słowie (ang. *endianess*)!
- Możesz używać gdb do krokowego wykonania kilku ostatnich instrukcji «getbuf» i odpluskwiania swojego nadużycia.
- Położenie bufora «buf» w ramce stosu funkcji «getbuf» zostało ustalone podczas kompilacji i może być ustalone poprzez zbadanie zdezasemblowanego kodu.

1.2 Zadanie 2 (2 punkty)

W obrazie programu «ctarget» znajduje się kod następującej procedury:

```
1 void touch2(unsigned val) {
2   vlevel = 2; /* Part of validation protocol */
3   if (val == cookie) {
4     printf("Touch2!: You called touch2(0x%.8x)\n", val);
5     validate(2);
6   } else {
7     printf("Misfire: You called touch2(0x%.8x)\n", val);
8     fail(2);
9   }
10  exit(0);
11 }
```

Twoim zadaniem jest zmuszenie programu «ctarget» do wywołania procedury «touch2» zamiast powrotu do «test». Jednak tym razem wołana procedura oczekuje argumentu «val» o wartości z dostarczonego pliku «cookie.txt».

Kilka wskazówek:

- Skonstruuj nadużycie tak, by instrukcja «ret» procedury «test» przekazała sterowanie do «touch2».
- Pamiętaj, że pierwszy argument funkcji przekazywany jest zwykle w rejestrze %rdi.
- Skonstruowane surowe dane powinny zawierać kod wpisujący w powyższy rejestr wartość ciasteczka.
- Nie używaj instrukcji «jmp» lub «call» w swoim nadużyciu. Wykonywanie skoków z użyciem instrukcji «ret» jest prostsze.
- W sekcji **??** znajdziesz informacje, jak generować binarne kodowanie ciągów instrukcji.

1.3 Zadanie 3 (2 punkty)

Tym razem należy przekazać napis jako argument procedury, na którą należy przekierować działanie programu. W obrazie pliku wykonywalnego «ctarget» znajdują się poniższe procedury:

```
1 int hexmatch(unsigned val, char *sval) {
2   char cbuf[110];
3   /* Make position of check string unpredictable */
4   char *s = cbuf + random() % 100;
5   sprintf(s, "%.8x", val);
6   return strncmp(sval, s, 9) == 0;
7 }
8
9 void touch3(char *sval) {
10  vlevel = 3;       /* Part of validation protocol */
11  if (hexmatch(cookie, sval)) {
12    printf("Touch3!: You called touch3(\"%s\")\n", sval);
13    validate(3);
14  } else {
15    printf("Misfire: You called touch3(\"%s\")\n", sval);
16    fail(3);
17  }
18  exit(0);
19 }
```

Twoim zadaniem jest zmuszenie programu «ctarget» do wywołania procedury «touch3» zamiast powrotu do «test». Jednak tym razem procedura wołana oczekuje argumentu «sval» będącego ciągiem znaków reprezentującym wartość ciasteczka w postaci liczby szesnastkowej.

Kilka wskazówek:

- Twoje surowe dane muszą zawierać znakową reprezentację wartości ciasteczka. Napis ten składa się wyłącznie z ośmiu szesnastkowych cyfr, od najbardziej znaczącej do najmniej znaczącej cyfry.
- Zauważ, że ciągi znaków w języku C są reprezentowane jako sekwencja bajtów zakończona bajtem o wartości 0. Zapoznaj się z kodami ASCII przywołując podręcznik systemowy poleceniem `man ascii`. Znajdziesz tam liczby reprezentujące znaki, które należy umieścić w surowych danych.
- Należy ustawić rejestr %rdi na adres przygotowanego ciągu znaków, którego adres należy do obszaru na stosie, w którym leży Twoje nadużycie.
- Pamiętaj, że procedury «hexmatch» and «strncmp» wołane z wnętrza «touch3» używają stosu, zatem nadpisują pamięć przydzieloną poprzednio buforowi z funkcji «getbuf». Musisz zatem dokładnie przemyśleć, gdzie umieścić znakową reprezentację wartości ciasteczka.

Literatura

[1] „Computer Systems A Programmer's Perspective"
Randal E. Bryant, David R. O'Hallaron,
Pearson Education, 3rd edition, 2016.

[2] „System V Application Binary Interface: AMD64 Architecture Processor Supplement"
Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell
Draft Version 0.99.7, November 2014.

[3] „Return-oriented programming: Systems, languages, and applications."
R. Roemer, E. Buchanan, H. Shacham, and S. Savage.
ACM Transactions on Information System Security, 15(1):2:1–2:34, March 2012.

[4] „Q: Exploit hardening made easy."
E. J. Schwartz, T. Avgerinos, and D. Brumley.
In *USENIX Security Symposium*, 2011.