Architektury systemów komputerowych

Pracownia 2: *„Atak hakerski”*

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`$login$», gdzie $login$ jest identyfikatorem Twojego użytkownika w serwisie `GitHub`.

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 11` 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. Umieszczają 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`

Poniżej znajdziesz naistotniejsze uwago o narzędziu «`hex2raw`»:

- 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*.

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*[1]"
Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell
Draft Version 0.99.7, November 2014.

[3] „*Return-oriented programming: Systems, languages, and applications.*[2]"
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.*[3]"
E. J. Schwartz, T. Avgerinos, and D. Brumley.
In *USENIX Security Symposium*, 2011.

[1] `https://uclibc.org/docs/psABI-x86_64.pdf`
[2] `https://hovav.net/ucsd/dist/rop.pdf`
[3] `https://www.usenix.org/legacy/event/sec11/tech/full_papers/Schwartz.pdf`

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 zdeasemblowanym 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 zdeasemblowanego 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.

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.

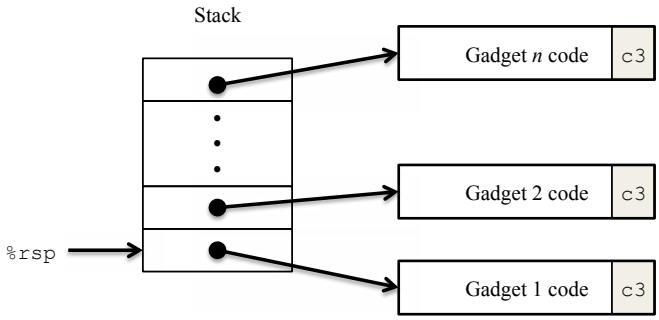
2 Atak przez konstrukcję łańcuchów powrotów z procedur

Przeprowadzanie ataku przez wstrzyknięcie kodu na programie «`rtarget`» jest dużo trudniejsze niż dla programu «`ctarget`». Przyczyną jest stosowanie przez system opracyjny dwóch technik mających za zadanie uniemożliwienie takich ataków:

- Uproszczony wariant **losowego układu przestrzeni adresowej** (ang. *adress space layout randomization*) polegający na umieszczaniu stosu pod inną pozycją przy każdym uruchomieniu programu. Dzięki temu nie można łatwo znaleźć adresu pod którym będzie umieszczony wstrzyknięty kod.
- Stosowanie przez system operacyjny sprzętowego wsparcia dla **ochrony pamięci** uniemożliwienia procesorowi wykonanie kodu, który leży w pamięci stosu. Nawet jeśli uda Ci ustawić licznik rozkazów procesora na chronioną pamięć stosu, to i tak proces zostanie zakończony w wyniku wysłania sygnału `SIGSEGV` (ang. *segmentation fault*) przez system operacyjny.

Całe szczęście, mądrzy ludzie wymyślili metodę wykonania przydatnych akcji przez program – nie przez wstrzyknięcie kodu, a poprzez zmyślne wykorzystanie fragmentów programu. Ogólnie nazywa się tą metodę **programowaniem zorientowanym na powroty** (ang. *return-oriented programming*) albo ROP [3, 4].

Metoda wykorzystująca ROP polega na zidentyfikowaniu w wykonywalnym segmencie programu użytecznych, z punktu widzenia atakującego, ciągów instrukcji zakończonych instrukcją powrotu z procedury «`ret`». Pojedynczy taki ciąg nazywamy **gadżetem**. Na rysunku 1 pokazano jak przygotować stos do wykonania sekwencji n gadżetów, tj. należy umieścić na stosie adresy kolejnych gadżetów. Kiedy program wróci z atakowanej procedury używając instrukcji «`ret`» to zainicjuje wykonanie szeregu gadżetów. Każdy kolejny gadżet, o ile nie modyfikuje wskaźnika stosu, po wykonaniu podniesie adres kolejnego gadżetu ze stosu i do niego skoczy.



Rysunek 1: Zlepienie sekwencji gadżetów do wykonania. Bajt o wartości `0xc3` koduje instrukcję «`ret`».

Gadżet może się składać z instrukcji wygenerowanych przez kompilator języka C. Chodzi o kod znajdujący się blisko **epilogu procedury**. W praktyce, w programach możemy znaleźć dużo gadżetów o takiej strukturze, ale niewystarczająco dużo, żeby wykonać użyteczny dla hakera kod. Na przykład, jest mało prawdopodobne, żeby kod wygenerowany z procedury w języku C zawierał instrukcję «`popq %rdi`» zaraz przed instrukcją «`ret`». Ponieważ zestaw instrukcji `x86-64` jest kodowany bajtowo, to początek gadżetu może leżeć wewnątrz instrukcji wygenerowanej przez kompilator.

Poniżej widnieje procedura z przykładowego programu «`rtarget`» zapisana w języku C, która wygląda na nieatrakcyjną z punktu widzenia hakera. W zdeasemblowanej postaci tej procedury można znaleźć interesującą nas sekwencję bajtów «`48 89 c7`», która koduje instrukcję «`movq %rax, %rdi`»!

```
1 void setval_210(unsigned *p) {
2   *p = 3347663060U;
3 }
```

```
1 0000000000400f15 <setval_210>:
2   400f15:        c7 07 d4 48 89 c7           movl   $0xc78948d4,(%rdi)
3   400f1b:        c3                          retq
```

Za znalezioną instrukcją «movq» występuje bajt «c3» kodujący instrukcję «ret». Procedura «setval_210» zaczyna się pod adresem 0x400f15, a interesujący ciąg bajtów rozpoczyna się na czwartym bajcie względem początku procedury. Zatem nasz gadżet leży pod adresem 0x400f18 i potrafi skopiować 64-bitową wartość z rejestru %rax do rejestru %rdi.

Każdy student otrzymuje w programie «rtarget» szereg procedur podobnych do «setval_210» w przedziale adresów, który określamy **hodowlą gadżetów**. Twoim zadaniem będzie zidentyfikowanie użytecznych gadżetów w hodowli i użycie ich do przeprowadzeniu ataków podobnych do tych z zadania 2 i 3.

UWAGA! Początek i koniec hodowli gadżetów jest oznaczony odpowiednio «start_farm» i «end_farm». Nie wolno używać gadżetów znalezionych w pozostałej części dostarczonego programu!

Source S	Destination D							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
%rax	48 89 c0	48 89 c1	48 89 c2	48 89 c3	48 89 c4	48 89 c5	48 89 c6	48 89 c7
%rcx	48 89 c8	48 89 c9	48 89 ca	48 89 cb	48 89 cc	48 89 cd	48 89 ce	48 89 cf
%rdx	48 89 d0	48 89 d1	48 89 d2	48 89 d3	48 89 d4	48 89 d5	48 89 d6	48 89 d7
%rbx	48 89 d8	48 89 d9	48 89 da	48 89 db	48 89 dc	48 89 dd	48 89 de	48 89 df
%rsp	48 89 e0	48 89 e1	48 89 e2	48 89 e3	48 89 e4	48 89 e5	48 89 e6	48 89 e7
%rbp	48 89 e8	48 89 e9	48 89 ea	48 89 eb	48 89 ec	48 89 ed	48 89 ee	48 89 ef
%rsi	48 89 f0	48 89 f1	48 89 f2	48 89 f3	48 89 f4	48 89 f5	48 89 f6	48 89 f7
%rdi	48 89 f8	48 89 f9	48 89 fa	48 89 fb	48 89 fc	48 89 fd	48 89 fe	48 89 ff

Tablica 1: Bajtowe kodowanie instrukcji movq S, D.

Operation	Register R							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
popq R	58	59	5a	5b	5c	5d	5e	5f

Tablica 2: Bajtowe kodowanie instrukcji popq.

Source S	Destination D							
	%eax	%ecx	%edx	%ebx	%esp	%ebp	%esi	%edi
%eax	89 c0	89 c1	89 c2	89 c3	89 c4	89 c5	89 c6	89 c7
%ecx	89 c8	89 c9	89 ca	89 cb	89 cc	89 cd	89 ce	89 cf
%edx	89 d0	89 d1	89 d2	89 d3	89 d4	89 d5	89 d6	89 d7
%ebx	89 d8	89 d9	89 da	89 db	89 dc	89 dd	89 de	89 df
%esp	89 e0	89 e1	89 e2	89 e3	89 e4	89 e5	89 e6	89 e7
%ebp	89 e8	89 e9	89 ea	89 eb	89 ec	89 ed	89 ee	89 ef
%esi	89 f0	89 f1	89 f2	89 f3	89 f4	89 f5	89 f6	89 f7
%edi	89 f8	89 f9	89 fa	89 fb	89 fc	89 fd	89 fe	89 ff

Tablica 3: Bajtowe kodowanie instrukcji movl S, D.

Operation		Register R			
		%al	%cl	%dl	%bl
andb	R, R	20 c0	20 c9	20 d2	20 db
orb	R, R	08 c0	08 c9	08 d2	08 db
cmpb	R, R	38 c0	38 c9	38 d2	38 db
testb	R, R	84 c0	84 c9	84 d2	84 db

Tablica 4: Kodowanie instrukcji 2-bajtowych, które pełnią rolę instrukcji nop.

2.1 Zadanie 4 (2 punkty)

W tym zadaniu należy powtórzyć atak z zadania 2 na programie RTARGET używając hodowli gadżetów. W rozwiązaniu potrzebne będą gadżety składające się z poniższych instrukcji, które używają pierwszych ośmiu rejestrów procesora `x86-64` (tj. `%rax` – `%rdi`).

- «`movq`» Kody widnieją w tabeli 1.
- «`popq`» Kody widnieją w tabeli 2.
- «`ret`» Instrukcja kodowana jest jednym bajtem «`0xc3`».
- «`nop`» Mnemonik to skrót od **brak operacji** (ang. *no operation*). Jedynym efektem działania tej instrukcji jest zwiększenie licznika rozkazów o 1. Instrukcja kodowana jest jednym bajtem «`0x90`».

Kilka wskazówek:

- Jedyne gadżety, których potrzebujesz, są zawarte w pliku wykonywalnym «`rtarget`» między adresami wyznaczonymi przez symbole «`start_farm`» i «`mid_farm`».
- Do wykonania zadania wystarczą Ci dwa gadżety.
- Jeśli gadżet zawiera instrukcję «`popq`», to zdejmie ona element ze stosu. Zatem nadużycie będzie składało się z adresów gadżetów jak i danych.

2.2 Zadanie 5 (4 punkty)

W tym zadaniu należy przeprowadzić atak typu ROP na program «`rtarget`» celem wywołania procedury «`touch3`» ze wskaźnikiem na tekstową reprezentację indywidualnie przygotowanego ciasteczka.

W rozwiązaniu tego zadania należy używać gadżetów zawartych między adresami wyznaczonymi przez symbole «`start_farm`» i «`end_farm`». Hodowla gadżetów, w porównaniu do zadania 4, zawiera teraz bajtowe kodowania różnych instrukcji «`movl`», pokazanych w tabeli 3, oraz 2-bajtowe instrukcje niezmieniające istotnej części stanu procesora, np. pamięci lub rejestru. Listę takich instrukcji pokazano w tabeli 4. Na przykład «`andb %al,%al`» operuje na dolnym bajcie rejestru `%rax`, ale nie zmienia jego wartości.

Kilka wskazówek:

- Pamiętaj o efekcie działania instrukcji «`movl`» na górne 32-bity rejestrów 64-bitowych!
- Oficjalne rozwiązanie wymaga użycia ośmiu gadżetów, z których niektóre mogą się powtarzać.