Blatt 4

Aufgabe 1

(1)

```
returnValue fctName(arg1,... ,argN)
            returnValue : Datentyp der Rückgabe
            fctName     : Funktionsname zum aufrufen
            arg1 - argN : Argumente mit festen Datentypen welche beim
                          Aufruf der Funktion mit übergeben werden müssen

int funk(int x, char y)
            returnValue : int
            fctName     : funk
            arg1 - argN : (int , char) in genau dieser Reihenfolge
```

Source: http://www.lab4inf.fh-muenster.de/lab4inf/docs/Prog-in-C/04-Funktionen.pdf Folie 5

(2)

Der Typ ist ein Funktionspointer auf eine Funktion, die einen int und einen char bekommt.

```
int main(){
            int (*fptr) (int, char);
            fptr = &funk;
            printf("Adr von func: %p\n", fptr); return 0;
        }
```

(3)

```
mov rdi, 0
mov rax, 60
syscall
```

Ausgabe objdump: Dateiformat der Eingabe (also tunix: Dateiformat elf64-x86-64). Dann wird mitgeiteilt das das Ergebniss vom "Disassembler" ausgegeben wird, woraufhin gesagt wird auf welche Adresse sich der erste Befehl (start) befindet. Dann wird in einer Tabelle (3 Spalten) die Adresse, der Maschinencode und das Disassemblierte Ergebnis von diesem Code ausgegeben.

(4)

```
Zeilen 2-12:
0:  b8 01 00 00 00          mov    eax,0x1
5:  bf 01 00 00 00          mov    edi,0x1
a:  48                      dec    eax
b:  be 48 61 6c 6c          mov    esi,0x6c6c6148
10: 6f                      outs   dx,DWORD PTR ds:[esi]
11: 20 44 75 56             and    BYTE PTR [ebp+esi*2+0x56],al
15: 54                      push   esp
16: 5e                      pop    esi
17: ba 08 00 00 00          mov    edx,0x8
1c: 0f 05                   syscall
```

```
1e: bf 00 00 00 00          mov    edi,0x0
23: b8 3c 00 00 00          mov    eax,0x3c
28: 0f 05                   syscall
```

Es wird eine Maschinencode Varaible “maschinencode_als_funk” erstellt (Typ: Funktionspointer auf Funktion die keine Paramter enthält, keine Ausgabe liefert). Danach wird die getypecastet und mit dem Wert von “maschinencode_als_string” initialisiert. Dann wird der code auf den “maschinencode_als_funk” zeigt ohne parameter ausgeführt. Es sorgt dafür dass die Adresse auf die rsp zeigt ausgegeben wird.
Aufgabe 2

gets() schreibt ALLES was sie bekommt in den übergebenen Buffer, auch darüber hinaus, wenn die Eingabe größer ist als der Buffer. Hier liegt der Buffer im Speicher vor den drei Funktionen. Heißt man kann diese überschreiben, indem man den buf[32] voll spammt, wobei der erste Buchstabe ein “J” sein muss, damit die Funktion auch ausgeführt wird! “Jdhdhdhdhdhdhdhdhdhdhdhdhdhdhd” Füllt buf[32] komplett. Nun muss man noch die ersten beiden Funktionen so überschreiben, dass sie nicht ausgeführt werden. Da man selber entscheidet welche Funktion ausgeführt wird ist es man einfachsten die erste zu nehmen und dort die Adresse der Funktion 3 hinein zu schreiben. Oder 0x0000 0000 FF10 bis 0x0000 0000 FF20 mit “0” zu überschrieben. (?)

Aufgabe 4

ASLR steht für Address Space Layout Randomization. Wie der nahme schon sagt, sorgt es dafür dass die Adressen für Programme zufällig zugewiesen werden und nicht mehr vorhersehbar sind. Dies soll Angriffe über Bufferoverflow erschweren. (Es gibt natürlich Angriffstechniken die diesen Schutz umgehen).

Eine temporäre Abschaltung lässt sich durch das Interface /proc/sys/kernel/randomize_va_space erreichen. Hier bei ist 0 keine randomizierung, 1 eine teilweise randomizierung (Shared libraries, stack, heap) und 2 eine volle randomizeirung. Um ASLR abzuschaltung führt man also folgenden Befehl aus:

echo 0 | sudo tee /proc/sys/kernel/randomize_va_space

und um es wieder zu enablen mit echo 2.

(2)

```
1 #include <stdio.h>
2 #include <stdint.h>
3
4 int main () {
5 uint64_t rip , rbp , rsp;
6 asm ("lea (%% rip ),%% rax" : "=r" (rip ));
7 asm ("lea (%% rbp ),%% rax" : "=r" (rbp ));
8 asm ("lea (%% rsp ),%% rax" : "=r" (rsp ));
9 printf("rip: %p\n", rip );
10 printf("rbp: %p\n", rbp );
11 printf("rsp: %p\n", rsp );
12 return 0;
13 }
```

Zeile 5: unsigned long long (also 64 Bit) deklaration von drei variablen (rip, rbp, rsp)

Zeile 6: asm zeigt an dass es inline-assembler code ist, wobei der assembler code der erste string ist. also “l
“=r” (rip) ist ein Parameter der asm übergeben wird (erkennbar an den :). Dies ist für die Outputoperation zuständig, also wird das ergebnis in dem in Zeile 5 definierten rip abgelegt werden soll (also das was zurvor in rax stand). =r sagt hierbei dass nur schreibend auf ein Beliebiges Genereal Purpose

Register zugegriffen werden soll. Effektiv wird also in die in Zeile 5 definierte Variable rip die Adresse reingeschrieben auf die das Register rip zeigt.

Zeile 7-8: analog für rbp und rsp Zeile 9-12: Hier werden dann die Adressen ausgegeben.

Diese Programm gibt also die Adressen von den Speicherzellen aus die zu dem Zeitpunkt von den Registern rip, rbp und rsp referenziert werden.

(3)

jmpto muss den Wert der Adresse von buf enthalten. Um diese rauszufinden kann man gbd nutzen: "‘ "gdb –args victim test" (Einmal victim "debuggen" mit belibiger Eingabe) Victim mit "run" ausführen "p &buf" (adresse von buf auslesen)