Buffer Overflows

Praktische Analyse von Schwachstellen

Jakob Stühn, John Meyerhoff, Sam Taheri

H-BRS

January 10, 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.

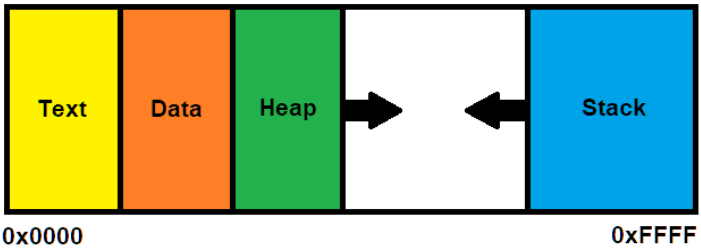

Figure: Prozess im Speicher

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

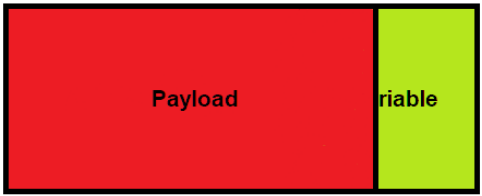

Figure: Buffer im Stack während eines Overflows

Während der Laufzeit eines Programms allozierter Speicher (z. B. durch `malloc()`) wird im Heap angelegt. Dabei setzt sich jeder Speicherblock aus einem Header und dem tatsächlich angeforderten Speicher zusammen. Der Header enthält hierbei, je nach Implementation, Informationen über den Block, wie z. B. seine Größe. Aus diesen Informationen kann dann abgeleitet werden, an welcher Stelle der nächste Block beginnt.

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)`
Ist eine gute Umsetzung,
`printf(‘‘%s’’, chars)`
hingegen nicht.

Praktische Analyse

Demonstration

[Screenshare]

Praktische Analyse

Demonstration

[Bild1]

Praktische Analyse

Demonstration

[Bild2]

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

Sowohl Hardwarelösungen als auch betriebssystembasierte Lösungen haben das grundlegende Problem, dass die Verhinderung von Buffer Overflows zu ungewünschten Nebeneffekten führen kann. Es ist auf jeden Fall möglich, jegliche Overflows zu verhindern - dabei werden jedoch auch (falls vorhanden) die vom Entwickler gewünschten Overflows blockiert, sodass Programme nicht mehr ordnungsgemäß funktionieren. Manchmal wird aus Gründen der Effizienz ein Buffer zum Überlaufen gebracht, ohne dass dieser Überlauf unkontrolliert geschieht. 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.

Ein Out Of Bounds Object ist eine vereinfachte Lösung, um Referenzen ungefährlich zu machen. Es wird verhindert, dass auf Speicher außerhalb des Programms zugegriffen wird, indem jede Adresse, welche nicht im spezifizierten Bereich liegt, auf ein bestimmtes Objekt, das sogenannte “Out Of Bounds Object” verweist. Diese Methode ist nicht gängig, da sie umgangen werden kann, sofern der Angreifer weiß, welcher Speicherbereich für das Programm vorgesehen ist.

Gegenmaßnahmen

Testen

Es gibt mehrere Möglichkeiten, um kompilierte Programme auf ihre Sicherheit und Robustheit zu testen. Wartbarkeit und Erweiterbarkeit sind langfristig ebenfalls zu beachten, da es bei Änderungen am Quellcode zu Fehlern kommen kann, welche Schwachstellen mit sich bringen. Mit Werkzeugen wie SonarLint können vor allem häufig auftretende Fehler entdeckt werden. Im Bereich der Overflow Payloads gibt es mehrere Werkzeuge, um Fuzzy Testing zu betreiben. Bei Fuzzy Tests wird gezielt-zufällig auf eine potentielle Schwachstelle getestet, wobei die tatsächlichen Aufrufe von einem sogenannten Fuzzer erstellt werden. Als Alternative zum Fuzzing gibt es spezifische Payloads und Escape-Sequenzen mit denen - auch automatisiert - getestet werden kann.

Gegenmaßnahmen

Statische Analyse

Wenn das Programm bereits vor der Ausführung analysiert wird, kann ein Tool bestimmen, an welchen Stellen Schwachstellen vorhanden sind, und ggf. vorschlagen, wie diese behoben werden können. Leider ist bei größeren Projekten die Abwesenheit von Schwachstellen unmöglich zu bestimmen. Daher ist dieser Ansatz zeitsparend, aber durch die fehlende finale Gewissheit alleine nicht ausreichend.

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.