Buffer Overflow

Schwachstellen und wie man sie Schließt

Jakob Stühn, John Meyerhoff, Sam Taheri

H-BRS

Inhaltsverzeichnis

1 Abstract **2**

2 Grundaufbau **3**

3 Geschichte **4**
3.1 Bekannte Buffer-overflows 4
3.2 Aktuelle Beispiele 4

4 Grundlegende Theorie **5**
4.1 Definition 5
4.2 Speicheraufbau 5
4.3 Stack Overflow 6
4.4 Heap Overflow 6

5 Shellcode **7**
5.1 Code 7
5.2 Erläuterung 7

6 Praktische Analyse **8**
6.1 Programmierfehler 8
6.2 Format-String-Schwachstelle 8
6.3 Code 9
6.4 Setup des Servers 9
6.5 Böswilliger Client 9
6.6 Erläuterung des Vorgangs 9

7 Gegenmaßnahmen **10**
7.1 Übersicht der Maßnahmen 10
7.2 Low-Level Probleme 10
7.3 Out Of Bounds Object 10
7.4 Statische Analyse 11
7.5 Canaries 11
7.6 Code-Beispiel 11
7.7 Testen 11

8 Quellen **12**

1 Abstract

Buffer Overflows gehören trotz ihres hohen Alters noch immer zu den relevantesten Schwachstellen in Computerprogrammen. Aus diesem Grund ist es unabdinglich für jeden, der sich im Feld der Software-Entwicklung oder IT-Sicherheit bewegt, ein grundlegendes Verständnis für Buffer Overflows aufzubauen. Ein Überblick über große und aktuelle Angriffe zeigt, wie verheerend die Auswirkungen einer solchen Schwachstelle sein können. Ein Buffer Overflow beschreibt im weitesten Sinne das "Überlaufen" eines Speicherbereiches durch unvorhergesehene Eingaben, wodurch Schadcode in einen laufenden Prozess injiziert und ausgeführt werden kann.

Der vorliegende Projektbericht beschäftigt sich deshalb mit einer theoretisch-technischen Einführung sowie der praktischen Analyse von Buffer-Overflow-Schwachstellen. Um eine Grundlage für praktische Tests zu schaffen, wurden zunächst die Struktur und der Ablauf eines Programms im Speicher analysiert und theoretische Angriffsmöglichkeiten konstruiert. Anschließend wurde ein fiktives Angriffsszenario und ein verwundbares C-Programm entwickelt. Hierbei zeigte sich schnell, dass bereits die Nutzung von vermeintlich harmlosen Funktionen, wie `fprint()` oder `gets()`, schwerwiegende Auswirkungen auf die Sicherheit einer Applikation haben kann. Um die zuvor konstruierten Angriffstechniken real zu erproben und die Sicht eines Angreifers möglichst realistisch zu analysieren, wurde das verwundbare Programm anschließend im GNU-Debugger durchleuchtet. Dabei ließ sich klar erkennen, dass der Angreifer von einer Kopie des anzugreifenden Programms, oder sogar des Source Codes, profitiert. Mit dem aus dem Debugger erlangten Wissen wurde nun ein Exploit gebaut und in einer simulierten Server-Umgebung ausgeführt. Unter Zuhilfenahme eines injizierten Assembler Programms, konnte auf dem Zielsystem erfolgreich eine privilegierte Shell geöffnet werden. Abschließend wurde sich mit den wichtigsten Abwehrmechanismen auseinandergesetzt und es wurden "cutting-edge" Präventions-Mechanismen, wie statische Codeanalyse oder Canaries, untersucht. Hier zeigte sich klar, dass einem Angreifer die Arbeit zwar erschwert werden kann, Buffer Overflows jedoch nie vollständig verhindert werden können.

Durch diese auf praktische Beispiele fokussierte Herangehensweise soll der Leser dieses Projektberichts die Thematik der Buffer Overflows besser verstehen und einen direkten Nutzen für seine Arbeit ziehen können.

2 Grundaufbau

- Eingabemöglichkeit
- Speichern der Eingabe
- Ablegen von Anweisungen durch übergroße Eingabe
- Ausführen der Anweisungen → Remote Code Execution

3 Geschichte

Nachdem der Begriff Buffer Overflow und die verschiedenen Typen erklärt worden sind, werden jetzt zwei ältere, aber sehr bekannte Buffer-Overflow-Attacken und zwei aktuell relevante Beispiele gezeigt, die dieselbe Schwachstelle ausnutzen.

3.1 Bekannte Buffer-overflows

Beim ersten handelt es sich um „The Morris Worm“ der am 2. November 1988 ins damals noch junge Internet freigelassen worden ist und sich rasant verbreitete. Er verursachte hohen Schaden in Form von überlasteten Systemen bzw. Ausfällen von Systemen. Das Programm wurde vom amerikanischen Robert T. Morris in der Sprache C geschrieben, dessen Dateigröße ca. 3200 Programmierzeilen umfasst. Robert T. Morris war Student an der Cornell University und die eigentliche Idee hinter seinem Experiment war es, die angeschlossenen Rechner an einem Netz zu zählen. Stattdessen wurden alle 15 stunden 2000 Unix Systeme infiziert. Im Endeffekt war damals 10% des gesamten Internets betroffen und um den Schaden des Virus einzudämmen, wurden regionale Netzwerke vom Internet getrennt.

Der nächste von den älteren ist der „SQL-Slammer“. Dieser begann am 25. Januar 2003 und hatte schon innerhalb von 30 Minuten 75.000 Opfer. Der Computerwurm infizierte ungepatchte Microsoft SQL Server 2000 und nutze zwei Buffer Overflows. Das Besondere an diesem Wurm war seine Größe: Er bestand nur aus einem UDP-Paket mit lediglich 376 Bytes und befand sich nur auf dem Arbeitsspeicher eines Computers und nicht jedoch auf der Festplatte. Zusätzlich enthielt er keine Payload und seine einzige Aufgabe war es, sich selbst zu kopieren und auf so vielen zufälligen Computern wie möglich zu verbreiten. Bei den Angreifern handelt es ich um zwei Mitglieder der Virenschreibergruppe 29A, die im Jahr 2004 gefasst worden sind.

Der Erste der neuen Attacken geschah bei einem der größten Messenger, und zwar WhatsApp. im Jahr 2019 wurde eine Sicherheitslücke ausgenutzt, die in DoS-Attacken über manipulierten MP4-Dateien endete. Mit der Hilfe der Videos konnte man sich Zugriff auf Smartphones verschaffen und Malware einschleusen. Die Security Abteilung sprach von einem „Stack-based buffer Overflow“ der über korrupte MP4-Dateien ausgenutzt wird. Der Fehler wurde durch einen neuen Patch behoben und es wird empfohlen, immer die neueste Version zu haben.

Beim letzten Angriff handelt es sich um eine Sicherheitslücke bei HP-Druckern. Dabei handelt es sich konkret um zwei Sicherheitsprobleme in der Firmware von HP und viele Druckermodelle sind davon betroffen. Eine Liste dieser wurde bereits veröffentlicht. Die Sicherheitslücke hat zur Folge, dass der Angreifer eine korrupte Datei an den Drucker sendet und diese zu einem Buffer Overflow führt, dadurch haben die Angreifer dann Fernzugriff auf das Gerät und können infizierten Code ausführen. Auch hier wurde die Lücke durch ein Update geschlossen.

3.2 Aktuelle Beispiele

4 Grundlegende Theorie

4.1 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.

4.2 Speicheraufbau

Wird eine Binärdatei durch den Linker von der Festplatte entnommen, so wird der auszuführende Programmcode zunächst in den Arbeitsspeicher geladen. Im Speicher gliedert sich der Prozess dann in folgende Segmente:

- **Stack**: Wächst von oben nach unten und enthält lokale Daten sowie Funktionsparameter
- **Heap**: Wächst von unten nach oben und enthält dynamisch allozierten Speicher
- **Data**: Liegt unter dem Heap und enthält initialisierte statische Variablen
- **Text**: Liegt unter dem Data-Segment und enthält die Assembler-Instruktionen des Programms

(Segmente, die im weiteren Verlauf keine größere Rolle spielen, werden hier unterschlagen.) Wenn

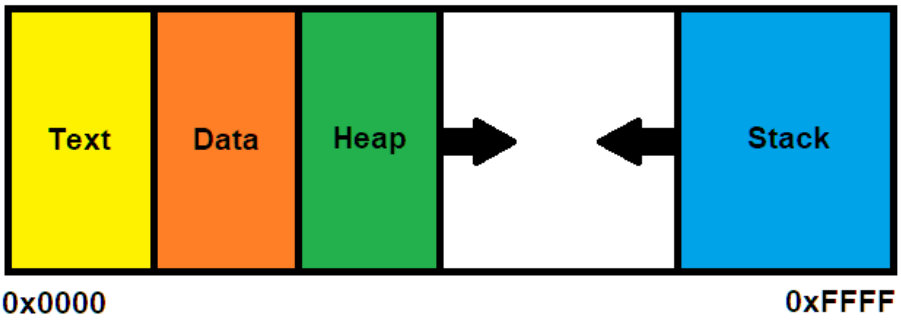


Abbildung 1: Prozess im Speicher

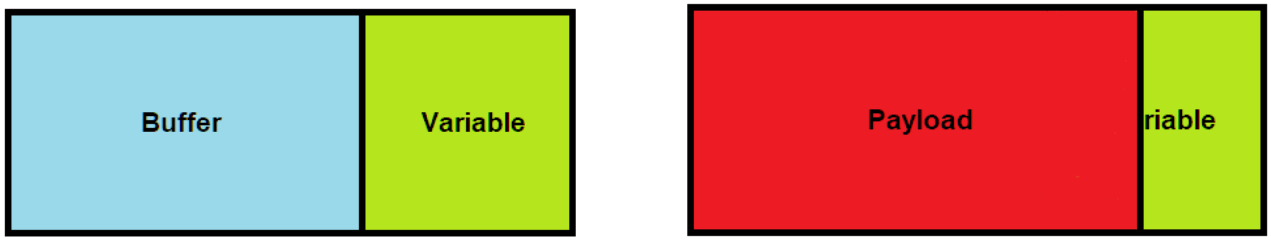
eine neue Funktion aufgerufen wird, legt diese zunächst ihre Funktionsparameter auf den Stack, gefolgt von einer Return-Adresse, die angibt, zu welcher Stelle im Programm im Anschluss an die Ausführung der Funktion gesprungen wird, und einem Base Pointer. Darauf folgen lokale Daten, die von der Funktion verwendet werden, wie z. B. ein Char Array.

4.3 Stack Overflow

Der zuvor beschriebene Aufbau des Stacks lässt sich nun durch gezieltes Einfügen von Daten in eine Funktion ausnutzen. Wenn beispielsweise ein Char Array mit einer Größe von 64 Bytes auf den Stack gelegt wird und es dem Angreifer gelingt, als Folge von fehlerhafter Programmierung eine Zeichenkette mit mehr als 64 Bytes in das Array zu laden, so können die überschüssigen Zeichen andere Daten im Stack überschreiben. Durch diese Methode kann der Prozess auf folgende Weisen beeinflusst werden:

- Es kann der Wert einer Variable verändert werden, um den Prozess zu manipulieren.
- Function Pointer können manipuliert werden, um den Programmfluss umzuleiten und zuvor präparierten Shellcode auszuführen.
- Auch durch das Überschreiben von Return Pointern kann auf Shellcode umgeleitet werden.

Ausgeführter Shellcode läuft dann immer unter denselben Privilegien wie der Prozess.



(a) Vorher (b) Nachher

Abbildung 2: Buffer im Stack während eines Overflows

4.4 Heap Overflow

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.

Wenn ein Angreifer nun Manipulationen im Heap-Speicher vornehmen möchte, muss er die verwendete Implementation kennen und kann in der Regel nur in Richtung der neu allozierten Speicherblöcke überschreiben. Da er aus dem Heap keine Möglichkeit hat, Sprungadressen direkt zu manipulieren und den Programmfluss so umzuleiten, muss er versuchen, einen bestimmten Speicherblock zu überschreiben, zu dem im weiteren Programmverlauf noch einmal gesprungen wird. Dies macht Heap Overflows in der Praxis um einiges komplexer und schwieriger als Stack Overflows.

5 Shellcode

5.1 Code

5.2 Erläuterung

6 Praktische Analyse

6.1 Programmierfehler

Grundsätzlich kann jede Software von Buffer Overflows betroffen sein, die in einer Programmiersprache geschrieben ist, welche direkte Zugriffe auf die Speicherstrukturen des Systems ermöglicht. Beispiele hierfür wären: Assembler, C/C++ oder Fortran. Prinzipiell nicht betroffen sind Programme, die in einer interpretierten Sprache wie Python oder Java geschrieben sind. Bei diesen Sprachen wäre nur ein Overflow im Interpreter selber möglich, da dieser in der Regel auf einer der zuerst genannten Sprachen basiert.

Am problematischsten sind hierbei Funktionen, die es ermöglichen Nutzereingaben zu lesen und zu speichern, die jedoch nicht die Länge der eingegebenen Daten überprüfen können. Zwei der bekanntesten Vertreter für Funktionen dieser Art sind die C Funktionen `gets()` und `strcopy()`:

- `gets(buffer)` Fragt nach Input und kopiert die Eingaben in den angegebenen Speicher
- `strcopy(buffer, input)` Kopiert den Input (z. B. ein Kommandozeilenargument) in den angegebenen Speicher

Da keine Kontrolle auf die Länge des Inputs durchgeführt wird, kann nicht sichergestellt werden, dass der angegebene Speicherbereich ausreichend groß ist oder ob der Input in andere Bereiche überläuft.

6.2 Format-String-Schwachstelle

Bei Format-String-Schwachstellen handelt es sich zwar nicht direkt um eine Art von Buffer Overflow, jedoch können diese oft in ähnlichen Kontexten aufkommen und ermöglichen es Angreifern, Informationen über die Interna eines Programms zu gewinnen.

Problematisch ist hierbei die unvorsichtige Verwendung von Formatierungsfunktionen wie `fprint()`. Soll beispielsweise eine Zeichenkette ausgegeben werden, sollte korrekterweise ein Formatierungsparameter wie `%s` verwendet werden: `printf(“%s”, chars)`. Die Unterschlagung dieses Parameters scheint zwar auf den ersten Blick dasselbe Ergebnis zu liefern: `printf(chars)`. Die zweite Variante ermöglicht es dem Angreifer jedoch, eigene Parameter einzusetzen, um Informationen auszulesen oder zu manipulieren:

- `%x` Liest Daten vom Stack
- `%s` Liest Strings aus dem Prozess
- `%n` Schreibt einen Integer in den Prozess
- `%p` Gibt Pointer auf void aus

6.3 Code

6.4 Setup des Servers

6.5 Böswilliger Client

6.6 Erläuterung des Vorgangs

7 Gegenmaßnahmen

Wie in Abschnitt 6.6 bereits aufgeführt, gehören zu einem Buffer-Overflow-Angriff mehrere kombinierte Teile. Wenn man nun verhindern möchte, dass ein Programm über diesen Angriff gestürzt werden kann, so hat man mehrere Möglichkeiten diese Teile aufzuhalten oder die Kombination zu blockieren. Es folgen mehrere Möglichkeiten, grob nach Aufwand (für den Entwickler) sortiert.

7.1 Übersicht der Maßnahmen

Low-Level:

- Hardware-basierte Lösungen
- Betriebssystembasierte Ansätze

Passive Härtung der Programme:

- C Range Error Detector und Out Of Bounds Object
- Safe Pointer Instrumentalisierung
- Manuelles Buffer-Overflow Blocken (Input-Bereinigung)

Aktive (Analysierende) Lösungen:

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

7.2 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 vom Entwickler gewünschte Overflows verhindert, sodass Programme nicht mehr ordentlich funktionieren. Manchmal wird aus gründen der Effizienz ein Buffer zum Überlaufen gebracht, ohne dass dieser Überlauf unkontrolliert ist. Leider ist es Praktisch nicht umsetzbar, in einer Hardwarelösung zu entscheiden, welcher Buffer-Overflow böswillig ist.

7.3 Out Of Bounds Object

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 Öut Of Bounds Object"verweist. Dadurch kann innerhalb des laufenden Programms erkannt werden, das etwas falsch gelaufen ist. Diese Methode ist nicht gängig, da sie technisch gesehen umgangen werden kann, solange der Angreifer weiß, welcher Speicherbereich für das Programm vorgesehen ist.

7.4 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 Aussage, dass keine Schwachstellen mehr vorhanden seien unmöglich zu treffen.

7.5 Canaries

Die Bezeichnung Canary (Engl. Kanarienvogel) stammt aus der Verwendung der Kanarienvögel als Indikator für Gas in Mienen. Die Canaries im Code werden als Stack-Schutz verwendet. Das bedeutet, dass beispielsweise Zufallszahlen im Programm auf dem Stack sind und bei einem Buffer-Overflow-Angriff überschrieben werden. Ein Tool wie StackGuard kann dann anhand der Änderung einen Fehler feststellen und die Ausführung des Programms abbrechen.

7.6 Code-Beispiel

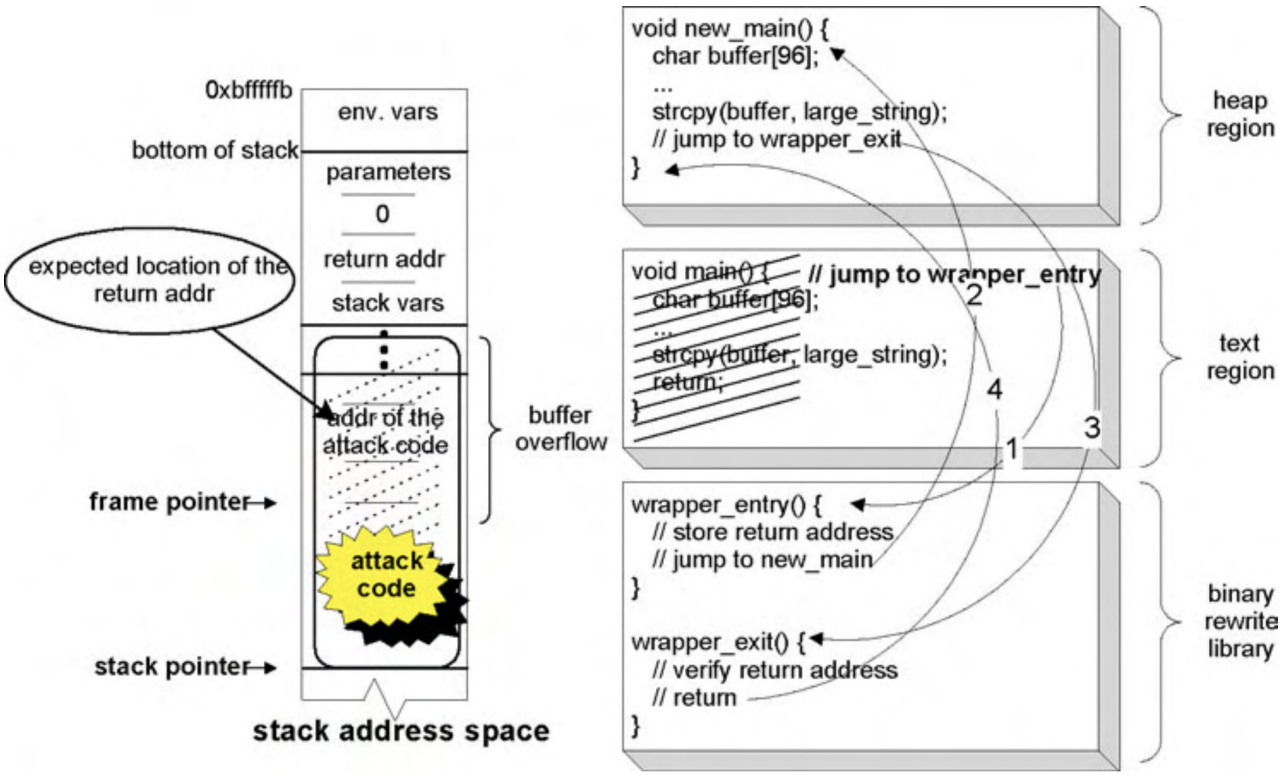


Figure 9: Libverify funtion call and stack layout

7.7 Testen

Es gibt mehrere Möglichkeiten um kompilierte Programme auf ihre Sicherheit und Robustheit zu testen. Diese beiden Qualitätsmerkmale sind vor Allem im Bezug zu Buffer-Overflows am wichtigsten. Wartbarkeit und Erweiterbarkeit sind langfristig auch zu beachten, da es bei Änderungen am Quellcode zu Fehlern kommen kann, welche Schwachstellen herbeibringen. Mit Werkzeugen wie Sonar lint

können vor allem häufig auftretende Fehler entdeckt werden. Im Bereich der Overflow Payloads gibt es mehrere Werkzeuge um Fuzzy Testing zu betreiben. Beim Fuzzy Test wird strukturiert 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 welche - auch automatisiert - getestet werden können.

8 Quellen

- https://www.nds.ruhr-uni-bochum.de/media/nds/attachments/files/2010/11/Survey.on.Buffer.Overflo