fhwedel
UNIVERSITY OF APPLIED SCIENCES
Department of Computer Science

WS 2016/2017

Seminar
IT Sicherheit

Stackbased Overflow Linux

Autor: Meik Jejkal
Betreuer: Prof. Dr. Gerd Beuster
abgegeben am: 12. April 2017

Inhalt

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Speicherüberläufe stellen eine der häufigsten Sicherheitslücken in Software dar. Dies äußerte sich bereits in den Anfängen der Programmiersprachen. Jedes Programm muss für die zu verarbeitenden Daten, Speicher reservieren. Die so entstehenden Speicherbereiche haben eine fixe Größe. Wenn nun Daten in diese Bereiche geladen werden ohne deren Grenzen zu berücksichtigen, können sie überlaufen. Die Anwendung versucht dann in Speicherbereiche hineinzuschreiben, die nicht dafür reserviert worden sind.
Als prominente Beispiele seien an dieser Stelle die frühen C-Funktionen strcopy oder auch getstr genannt bei denen es jeweils keine Überprüfung der Eingabe auf ihre Grenzen gab. Ziele für derartige Angriffe können u.a. Bibliotheken, Programme, Scripte, Webanwendungen oder andere Formen von Software sein.
Die Auswirkungen eines Speicherüberlaufes können vielfältig sein. Die Palette reicht hier von DoS bis hin zur Eskalation von Rechten oder der Ausführung fremden Programmcodes. Die Angreifer können hierbei sowohl lokal als auch entfernt sein. Angriffsziele sind nicht nur klassische Computer, sondern auch eingebettete Systeme, Infrastruktur, Telefonanlagen, oder auch smarte Toaster, Kühlschränke, sowie sonstige IoT Geräte. Eingebettete Systeme sind hier ein besonderes Problem, da sie häufig mit einer Firmware laufen, die sich im Gegensatz zu vollständigen Betriebssystemen nur schwer aktualisieren lässt.

2 Grundlagen

Die Folgenden Kapitel behandeln die Grundlagen die notwendig sind, um einen Speicherüberlauf anschaulich zu machen.

2.1 Speicheraufbau

Darstellung 2.1 zeigt schematisch die Aufteilung des Arbeitsspeichers. Der Bereich *text* hält den auszuführenden Programmcode vor. Im *data* Bereich werden globale Variablen des Programms gespeichert. Dynamische Variablen sind im *heap* Bereich zu finden. Der Stack besitzt eine feste Größe und enthält alle Funktionen, Aufrufparameter sowie lokale Variablen. Er arbeitet nach dem LIFO-Prinzip. Datensätze die als letztes auf den Stapel gelegt wurden, werden als erstes bearbeitet. Je mehr Funktionen aufgerufen werden, desto größer wird der Stack und wächst Richtung *heap*.

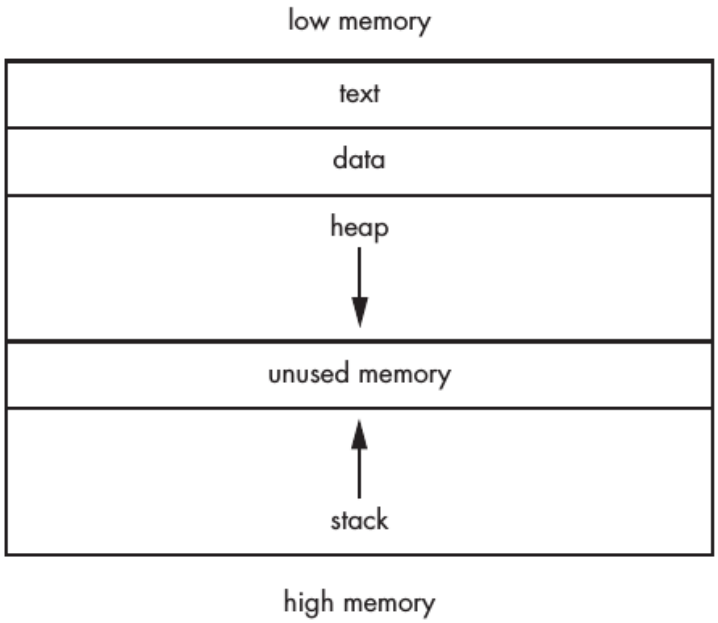

Abbildung 2.1: Speicheraufbau[Wei01]

2.2 Programmausführung

Im folgenden wird von einer Intel CPU ausgegangen. Tabelle 2.1 zeigt eine Übersicht der relevantesten Register:

Register	Aufgabe
EIP	instruction pointer
ESP	stack pointer
EBP	base pointer
ESI	source index
EDI	destination index
EAX	accumulator
EBX	base
ECX	counter
EDX	data

Tabelle 2.1: CPU-Register

Die Register ESP, EBP, und EIP sind für Speicherüberläufe die relevantesten. ESP und EBP steuern zusammen die eigentliche Programmausführung. Wird eine Funktion aufgerufen, so wird ein sog. stack frame auf dem Stack gespeichert. Er enthät sowohl die Adresse der aufrufenden Funktion, als auch alle aktuellen Variablen des Kontextes. Ist der Aufruf abgeschlossen, kann mit Hilfe des stack frames der Zustand vor dem Aufruf widerhergestellt werden um die Programmausführung fortsetzen zu können. ESP zeigt dabei auf den Anfang des stack frames an der kleinsten Speicheradresse. EBP zeigt umgekehrt auf das Ende des stack frames. EIP enthält die Adresse der nächsten Anweisung die ausgeführt werden soll. Dieses Register ist read-only markiert, dadurch kann es nicht von außerhalb manipuliert werden.

2.3 Overflow

Das Ziel eines Speicherüberlaufes ist es den Inhalt des Registers EIP zu manipulieren und damit die Kontrolle über die Programmausführung zu gewinnen. Abbildung 2.1 zeigt C-Beispielcode:

```
1  #include <string.h>
2  #include <stdio.h>
3
4  void overflowed(){
5      printf("%s\n", "Hier könnte ihr Rootkit stehen!");}
6
7  void function1(char *str){
8      char buffer[5];
9      strcpy(buffer, str);}
10
11 void main(int argc, char *argv[]){
12     function1(argv[1]);
13     printf("%s\n", "nix passiert!");}
```

Codebeispiel 2.1: C-Programmcode backdoor

Bei normaler Ausführung des Programmes wird die Funktion overflowed nicht ausgeführt. Sie wird allerdings trotzdem in den Speicher geladen. Dies ist ein klassisches Beispiel für eine bereits im Code enthaltene backdoor. Bei closed source Software ist es aufwendig derartige Besonderheiten zu erkennen. Bei normaler Programmausführung wird die Benutzereingabe in einen Buffer geschrieben der 5 Zeichen, also 5 Bytes umfasst. Aufgrund der terminierenden 0 am Ende von Strings ist der Speicher sogar nur auf 4 Zeichen begrenzt. Solange die Eingabe innerhalb dieser Grenze bleibt, wird das Programm forgesetzt und endet mit der Ausgabe in Zeile 13. Die Codebeispiele 2.2 bis 2.4 zeigen das Verhalten bei anderen Eingaben:

```
raziel@Zion:~$ ./overflowtest AAAA
nix passiert!
```

Codebeispiel 2.2: Eingabe im Bereich 5 Bytes

In Beispiel 2.2 wird das Programm normal ausgeführt und endet mit Exit Code 0. Die Eingabe ist genau 5 Bytes lang, da das Ende von Strings immer mit einer 0 markiert wird.

```
raziel@Zion:~$ ./overflowtest AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
```

Codebeispiel 2.3: Eingabe über 5 Bytes

Bei der Eingabe in Beispiel 2.3 wurde die Rücksprungadresse im EIP durch AAAA überschrieben. Bei hexadezimaler Betrachtungsweise soll versucht werden auf die Speicheradresse 41414141 zuzugreifen. Diese Adresse liegt außerhalb des zugewiesenen Speichers für diese Anwendung und löst so einen Speicherzugriffsfehler aus, da keine Berechtigung für diesen Speicherbereich existiert. Zugriffe auf den Speicher werden von der Memory-Management-Unit verwaltet, die diesen Fehler in diesem Fall auslöst.

```
raziel@Zion:~$ ./overflowtest $(perl -e 'print "A" x 9 . "\xf4\x83\x04\x08"')
Hier könnte ihr Rootkit stehen!
Segmentation fault
```

Codebeispiel 2.4: aktivierte backdoor

Im letzten Beispiel 2.4 wurde mit Hilfe eines Perl Scriptes der Buchstabe A genau 9x widerholt. Darauf folgt nun die hexadezimale Adresse der enthaltenen Funktion overflowed im Speicher. Diese wurde zuvor mittels eine Debuggers ausgelesen. Diesmal wird die Funktion *overflowed* aufgerufen die sonst nicht Teil der normalen Programmausführung ist. Das EIP Register wurde erfolgreich mit Hilfe eines gezielten Speicherüberlaufes manipuliert. Nach Aufruf dieser Funktion enthält das EIP

Register erneut eine unzulässige Adresse die einen Zugriffsfehler auslöst. Dies lässt sich verhindern indem die weiterführenden Adressen ebenfalls mit existierenden belegt werden.

Derartige Angriffe wurden durch diverse Schutzmaßnahmen stark erschwert. Das Feststellen der Adresse mittels eines Debuggers ist unter Berücksichtigung dieser Maßnahmen nicht mehr trivial. Nähere Informationen sind in Kapitel 4 zu finden.

Bei geläufigeren Angriffsformen wird der Code mit Hilfe der Eingabe in den Speicher geladen. In der Regel wird hier Shellcode[Han04] verwendet. Dabei wird die gewünschte Funktion zunächst in einer höheren Programmiersprache, häufig C, geschrieben und compiliert. Der Maschinencode wird dann, unter Berücksichtigung eines Regelsatzes, konvertiert. Viele Instruktionen können weggelassen oder verkürzt werden. Dies ist wichtig, da der nutzbare Speicherbereich bei einem Angriff sehr begrenzt sein kann. Shellcode darf häufig auch kein 0-Byte enthalten, dieses markiert das Ende innerhalb eines Strings. Manchmal müssen noch weitere Filter umgangen werden, beispielsweise werden nur Buchstaben und Zahlen erlaubt oder Groß- und Kleinschreibung alterniert. Auftretende Speicherlücken können mit Nops gefüllt werden. NOP steht für No-Op und ist eine CPU-Anweisung bei der weder Berechnungen durchgeführt, noch Speicherstellen manipuliert werden. NOP's können auch per XOR-Operation mit dem eigentlichen Shellcode verknüpft werden. Dies verändert die Signatur des Shellcodes weiter, um einer frühzeitigen Erkennung vorzubeugen. Das Werkzeug Metasploit[Rap16] enthält bereits vorgefertige Shellcodes, bei denen diese Schritte schon durchgeführt wurden, sodass sich diese sofort nutzen lassen.

3 Angriffsformen

Nach erfolgreicher Übernahme der Programmausführung gibt es verschiedene Möglichkeiten diese zu nutzen. Der auszuführende Teil des Speicherüberlaufes wird auch als Payload bezeichnet. Hierfür gibt es verschiedene Möglichkeiten:

Reverse-Shell: Eine klassische Remote-Shell geht immer vom Client selbst aus der sich aktiv mit dem Zielsystem verbindet und authentifiziert. Liegt das Zielsystem allerdings hinter einer Firewall oder einem NAT ist es sinnvoller die Verbindung vom Zielsystem aus aufbauen zu lassen. Die Reverse-Shell bedient sich dieser Technik und stellt einem Client eine Shellsession zur Verfügung, die dieser annehmen kann, um die Verbindung herzustellen. Dieser Dienst eignet sich perfekt als Payload für einen kontrollierten Speicherüberlauf, da er auch entfernten Angreifern Kontrolle über das System zur Verfügung stellen kann.

Denial-of-Service: Durch die Manipulation von Boot-Dateien in Verbindung mit einem Neustart des Systems kann dies einen schweren DoS-Zustand hervorrufen, der manuelles Eingreifen erfordert. Weitere Möglichkeiten wären eine Umkonfiguration der Netzwerkschnittstellen, oder auch eine Überlastung des Systems mittels forkbombs oder ähnlichen Scripten.

Malware: Beliebiges Nachladen von Programmcode um das System einem Botnetz hinzuzufügen, den Inhalt der Datenträger zu verschlüsseln (Ransomware), oder weitere Informationen auszuspähen (Keylogger, Screenshots, Kamera). Die Installation von backdoors um den weiteren Zugriff zu erleichtern wäre hier ebenfalls denkbar. Hierfür kann auch ein Dienst mit Sicherheitslücken gestartet werden für die bereits Exploits existieren.

Zertifikate: Durch die Installation beliebiger Zertifikate baut das Zielsystem vermeintlich sichere Verbindungen ohne Warnung auf. Der Angreifer besizt den privaten Schlüssel und kann damit den Datenverkehr von TLS Verbindungen entschlüsseln. Gleichzeitig können über Einträge in der .hosts Datei Verbindungen auf eigene Server zwecks Phishing umgeleitet werden.

3.1 Beispiele

Im Folgenden werden Beispiele für stackbasierte Angriffe inklusive ihrer Folgen beschrieben:

CVE-2015-7547: Die weit verbreitete Bibliothek *glibc* enthält bis Version 2.6 einen Fehler in der System-Funktion *getaddrinfo()*, die für das Auflösen von Netzwerknamen via DNS zuständig ist. Mit Hilfe von zwei präparierten DNS-Paketen lässt sich ein Stackoverflow mit anschließender Programmcode Ausführung auslösen. Dabei muss vom Angreifer nicht zwingend ein eigener DNS-Server betrieben werden. Mittels eines Man-in-the-Middle Angriffs können auch reguläre DNS-Antworten manipuliert werden. Prinzipiell kommt jede Software in Frage die DNS-Namen auflöst, also auch *curl* oder *sudo*. Die betroffene Bibliothek ist ein Grundbestandteil vieler Distributionen wie Debian, Fedora oder RedHat.[CVE16b]

CVE-2015-5681: Bei bestimmten D-Link Routern kann es bei der Verifizierung von Session Cookies zu einem Speicherüberlauf kommen, der die Ausführung beliebigen Programmcodes erlaubt. Das Gerät lässt sich so auch in einen DoS-Zustand versetzen. Der betroffene Dienst lauscht auf Port 8181 und kann damit entfernt angegriffen werden. Router sind wichtiger Bestandteil der Netzwerkinfrastruktur, Angreifer haben hier weitreichende Möglichkeiten Netzwerkverkehr zu manipulieren oder zu blockieren.[CVE16a]

4 Schutzmechanismen

Der klassische Speicherüberlauf wie er zur Veranschaulichung in Kapitel 2.3 verwendet wurde ist auf aktuellen Systemen in dieser Form nicht mehr durchführbar. Dafür sorgen diverse Schutzmechanismen die seither eingeführt wurden. Die folgenden Unterkapitel werden auf die wichtigsten Systeme eingehen:

4.1 Canary Values

Canary Values[Edg14] sind zufällig erzeugte Integer Werte die direkt nach der Rücksprungadresse auf den Stack gelegt werden. Beim Versuch den Inhalt des EIP mittels eines Speicherüberlaufes zu verändern wird unweigerlich auch der Canary Value überschrieben. Vor dem Rücksprung auf die im EIP angegebene Speicherstelle wird der Wert überprüft. Stimmt er nicht mit dem zuvor generierten Wert überein, wird die Programmausführung sofort unterbrochen um eine Übernahme zu verhindern. Diese Schutzmaßnahmen kann ein Compiler bereits bei der Erstellung von Software automatisch einfügen. Jede Überprüfung eines Canary Values kostet jedoch Ressourcen und damit Performance. GCC[GCC16] unterstützt aus diesem Grund momentan drei verschiedene Ansätze die jeweils durch ein Compiler Flag ausgewählt werden können. Die Ansätze unterscheiden sich in der Menge der ausgewählten Funktionen wie folgt:

fstack-protector-all: Schützt jede Funktion im gesamten Code unabhängig der verwendeten Variablen.

fstack-protector: Schützt in der Standardeinstellung alle Funktionen die Arrays mit mehr als 8 Byte verwenden. Der Schwellenwert kann durch das Flag *param=ssp-buffer-size=N* angepasst werden.

fstack-protector-strong: Schützt Funktionen mit Arrays jeder Art auch in structs oder unions. Zusätzlich werden noch Funktionen einbezogen die Adressen lokaler Variablen als Funktionsargument, oder für Zuweisungen verwenden.

4.2 Address Space Layout Randomization

Die ASLR[Mic16b] sorgt dafür, dass bei jeder Programmausführung ein anderer zufälliger Speicherbereich allokiert wird. Dies erschwert es platzierten Programmcode im Speicher wiederzufinden. ASLR bezieht sich auf den EBP, Bibliotheken sowie den *heap*-, *text*- und *dataBereich*. ASLR lässt sich durch *Spraying* umgehen. Dabei wird der Programmcode großflächig im Speicher dupliziert. Dadurch steigt die Wahrscheinlichkeit, dass dieser bei einem Aufruf ausgeführt wird. In iOS 4.3 wurde ASLR erstmals auch für Mobilgeräte eingesetzt[Gie11]. Die Implementierung im mobilen Browser Safari, führte allerdings zu einer neuen Sicherheitslücke. Charlie Miller gelang es nur drei Tage nach Erscheinen der Firmware, über ASLR ins System einzudringen. Das PaX-Projekt[PaX16] hatte 2002 mittels eines Kernel-Patches ASLR für Linux eingeführt. 2003 führte OpenBSD als erstes Betriebssystem ASLR standardmäßig ein. PaX ist mittlerweile Teil des Grsecurity-Projekts, das einen Kernel-Patch mit zahlreichen zusätzlichen Sicherheitsfunktionen bereitstellt der aber nie Teil des offiziellen Linux-Kernels geworden ist. Mit der Version 2.6.12 führte Linux eine eigene Implementierung von ASLR ein. Es wird unter Linux allerdings nur mangelhaft genutzt. Nicht jedes Programm kann automatisch an beliebige Speicherbereiche geladen werden. Sprungbefehle in Software können auf feste Adressen verweisen. Damit der Code an beliebige Speicherbereiche geladen werden kann, muss dies bereits beim kompilieren beachtet werden. GCC bietet hierfür das Flag *fpic* (Position Independent Code"). Der Linker muss die Option *pie* (Position Independent Executable") übergeben bekommen. Viele große Distributionen (SuSe, RedHat) nutzen standardmäßig Programme, die nicht positionsunabhängig kompiliert wurden und für die ASLR nicht greift. Stack- und Heap-Speicher landen trotzdem in zufälligen Speicherbereichen und Bibliotheken werden generell positionsunabhängig kompiliert.

4.3 Data Execution Prevention

DEP[Mic16b] markiert Speicherbereiche als nicht ausführbar. Wird versucht Programmcode aus einem entsprechend markierten Bereich auszuführen wird der entsprechende Prozess beendet. Dies kann sowohl in Software, als auch auf Hardwarebene passieren. Bei x86-Prozessoren der Intel Familie wird die Markierung mit einem NX-Bit[Mic16a] realisiert. Die Technik wird von AMD als Enhanced Virus Protection (EVP) bezeichnet. DEP wurde 2004 innerhalb von Linux (kernel 2.6.8) sowie im selben Jahr von Microsoft in Windows-XP SP2 eingeführt. Apple führte die Technik beim Update auf die x86-Architektur 2006 ein. Um den Mechanismus zu umgehen wird eine Technik namens Return-Oriented Programming (ROP)[She16] gebraucht. Hierbei wird in legitimen Modulen nach kleinen Codeschnipseln gesucht den ROP gadgets. Die Besonderheit dieser Gadgets ist eine Kombination aus einem oder mehreren Anweisungen auf die ein *return* folgt. Werden mehrere dieser Aufrufe hintereinander ausgeführt erlaubt dies beliebigen Programmcode auszuführen. Eine weitere Möglichkeit die DEP zu umgehen, ist *return into libc*[ElS16]. Dabei werden bereits geladene Funktionen aufgerufen die in ausführbaren Speicherbereichen liegen. Typischerweise wird hier die C-Bibliothek genutzt, da diese in Linux Systemen standardmäßig geladen wird. Sie stellt u.a. eine *system()* Funktion bereit die beliebige Systemprogramme ausführen kann. Eine aktivierte ASLR 4.2 erschwert das Finden derartiger Bibliotheken im Speicher.

4.4 Sicheres Programmieren

Bereits während der Programmentwicklung können Speicherüberläufe zur Laufzeit vermieden werden. Kritische Stellen sind hier die Verarbeitung von Benutzereingaben, oder anderen externen Prozessen. Diese müssen sowohl auf ihre Länge, als auch ihren Inhalt geprüft werden.

Literaturverzeichnis

[CVE16a] CVE20155681. *D-Link is patching CVE-2016-5681*. 20.12.2016. 2016. URL: http://bestsecuritysearch.com/d-link-patching-cve-2016-5681-critical-flaws-routers/.

[CVE16b] CVE20157547. *glibc getaddrinfo stack-based buffer overflow*. 20.12.2016. 2016. URL: https://security.googleblog.com/2016/02/cve-2015-7547-glibc-getaddrinfo-stack.html.

[Edg14] Jake Edge. *Strong stack protection for GCC*. 20.12.2016. 2014. URL: https://lwn.net/Articles/584225/.

[ElS16] Saif El-Sherei. *Return into libc*. 20.12.2016. 2016. URL: https://www.exploit-db.com/docs/28553.pdf.

[GCC16] GCC. *GCC, the GNU Compiler Collection*. 20.12.2016. 2016. URL: https://gcc.gnu.org/.

[Gie11] Katia Giese. *iOS 4.3 Sicherheitslücke erfordert Update*. 20.12.2016. 2011. URL: http://www.giga.de/apps/ios/news/ios-4-3-sicherheitslucke-erfordert-update/.

[Han04] Steve Hanna. *Shellcoding for Linux and Windows Tutorial*. 20.12.2016. 2004. URL: http://www.vividmachines.com/shellcode/shellcode.html.

[Mic16a] Microsoft. *Data Executive Prevention*. 20.12.2016. 2016. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/aa366553(v=vs.85).aspx.

[Mic16b] Microsoft. *Protecting Your Software*. 20.12.2016. 2016. URL: https://www.microsoft.com/security/sir/strategy/default.aspx#!section_3_3.

[PaX16] PaX. *Homepage of The PaX Team*. 20.12.2016. 2016. URL: https://pax.grsecurity.net/.

[Rap16] Rapid7. *Shellcode*. 20.12.2016. 2016. URL: https://www.metasploit.com/.

[She16] Saif El Sherei. *Return Oriented Programming(ROP FTW)*. 20.12.2016. 2016. URL: https://www.exploit-db.com/docs/28479.pdf.

[Wei01] Georgia Weidman. *Penetration Testing - A hands-on introduction to Hacking*. San Francisco: no starch press, 2001.