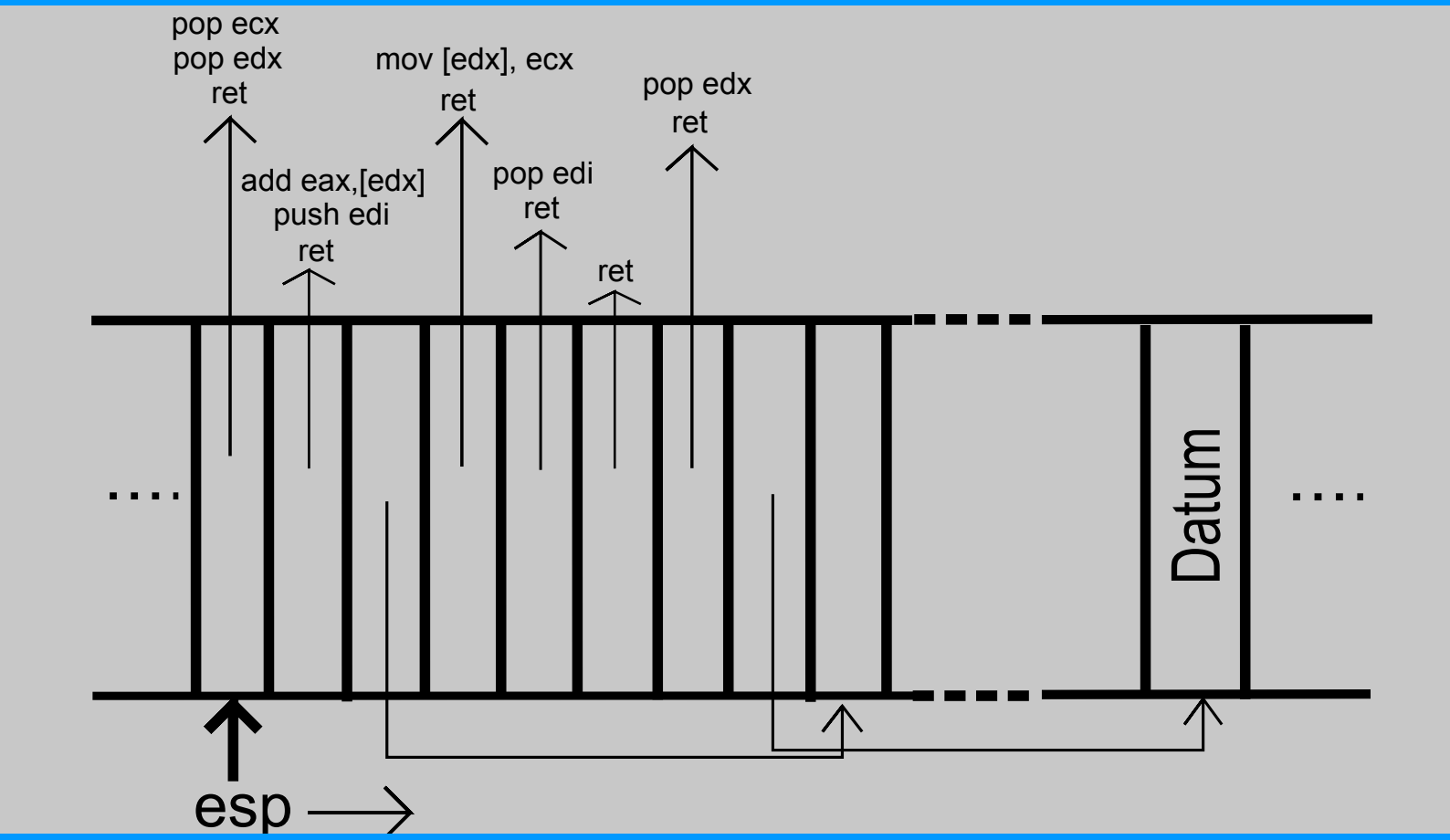
FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Open C3S
Open Competence Center for Cyber Security



Zertifikatsprogramm

Systemnahe Aspekte der Softwaresicherheit [MM-104]

Autoren:

Dr. rer. nat. Werner Massonne

Prof. Dr.-Ing. Felix C. Freiling

Friedrich-Alexander-Universität Erlangen-Nürnberg

MM-104

Systemnahe Aspekte der Softwaresicherheit

Autoren:

Dr. rer. nat. Werner Massonne

Prof. Dr.-Ing. Felix C. Freiling

---

1. Auflage

Friedrich-Alexander-Universität Erlangen-Nürnberg

GEFÖRDERT VOM
Bundesministerium
für Bildung
und Forschung

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Martensstr. 3
91058 Erlangen

1. Auflage (12. Dezember 2016)

Um die Lesbarkeit zu vereinfachen, wird auf die zusätzliche Formulierung der weiblichen Form bei Personenbezeichnungen verzichtet. Wir weisen deshalb darauf hin, dass die Verwendung der männlichen Form explizit als geschlechtsunabhängig verstanden werden soll.

Das diesem Bericht zugrundeliegende Vorhaben wurde mit Mitteln des Bundesministeriums für Bildung, und Forschung unter dem Förderkennzeichen 16OH12022 gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt beim Autor.

Inhaltsverzeichnis

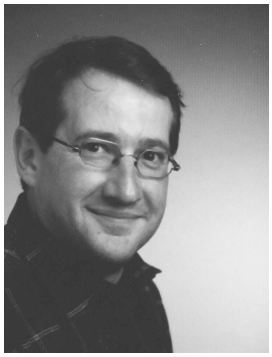
Einleitung

I. Abkürzungen der Randsymbole und Farbkodierungen

Quelltext	Q
Übung	Ü

II. Zu den Autoren

Felix Freiling ist seit Dezember 2010 Inhaber des Lehrstuhls für IT-Sicherheitsinfrastrukturen an der Friedrich-Alexander-Universität Erlangen-Nürnberg. Zuvor war er bereits als Professor für Informatik an der RWTH Aachen (2003-2005) und der Universität Mannheim (2005-2010) tätig. Schwerpunkte seiner Arbeitsgruppe in Forschung und Lehre sind offensive Methoden der IT-Sicherheit, technische Aspekte der Cyberkriminalität sowie digitale Forensik. In den Verfahren zur Online-Durchsuchung und zur Vorratsdatenspeicherung vor dem Bundesverfassungsgericht diente Felix Freiling als sachverständige Auskunftsperson.



Werner Massonne erwarb sein Diplom in Informatik an der Universität des Saarlandes in Saarbrücken. Er promovierte anschließend im Bereich Rechnerarchitektur mit dem Thema „Leistung und Güte von Datenflussrechnern“. Nach einem längeren Abstecher in die freie Wirtschaft arbeitet er inzwischen als Postdoktorand bei Professor Freiling an der Friedrich-Alexander-Universität.

III. Lehrziele

Große Teile der heute weit verbreiteten Betriebssysteme wie Windows und Linux und auch viele normale Anwendungsprogramme sind in C programmiert. Die Programmiersprache C ist eine systemnahe Programmiersprache. Das wesentliche Merkmal der systemnahen Programmierung ist die direkte Kommunikation mit der Hardware und dem Betriebssystem eines Rechners.

Dieses Mikromodul behandelt das Thema „Systemnahe Programmierung“ unter dem Aspekt der Sicherheit. Die Verwendung einer systemnahen Programmiersprache wie C gibt dem Programmierer zwar viele Freiheiten, birgt aber auch viele Risiken. Durch den unbedarften Umgang mit C können leicht sogenannte Sicherheitslücken in Programmen entstehen.

Sicherheitslücken bergen die Gefahr, dass ein Angreifer die Kontrolle über einen Rechner übernimmt. Es ist daher sehr wichtig, dass ein Programmierer diese Gefahren und ihre Entstehungsweisen kennt. Der erste Teil des Mikromoduls beschäftigt sich mit Buffer Overflows. Buffer Overflows führen letztendlich zu unkontrollierbaren Speicherzugriffen, die ein Angreifer ausnutzen kann.

Die beste Methode zur Vermeidung von Buffer Overflows ist natürlich eine aus Sicherheitssicht fehlerfreie Programmierung. Sollte es aber nicht möglich sein, aus einem fertigen Programm Sicherheitslücken zu entfernen, weil bspw. kein Quellcode vorliegt, so gibt es dennoch diverse etablierte Verfahren, welche die Ausnutzung dieser Lücken zu unterbinden versuchen. Mit solchen Gegenmaßnahmen beschäftigt sich der zweite Teil dieses Mikromoduls.

Die Gegenmaßnahmen sind allerdings kein Allheilmittel und in in ihrer universellen Wirksamkeit begrenzt. Aus Sicht eines Angreifers gilt es also, Gegen-Gegenmaßnahmen zu treffen, um weiterhin erfolgreiche Angriffe lancieren zu können. Damit beschäftigt sich der dritte Teil dieses Mikromoduls.

Auch wenn die in diesem Mikromodul vorgestellten Techniken weitgehend universeller Natur sind, so wird für konkrete Bezüge meist die 32-Bit-Architektur IA-32 von Intel verwendet.

Systemnahe Aspekte der Softwaresicherheit

1 Lernergebnisse

Sie können durch Buffer Overflows verursachte Sicherheitsrisiken erklären und klassifizieren. Entstehungsweise und Gründe von Sicherheitslücken können Sie benennen. Die durch Ausnutzung (Exploit) von Sicherheitslücken entstehenden Gefahren können Sie darlegen. Sie können erklären, wie ein Exploit durch das Einbringen von Shellcode praktisch durchgeführt werden kann.

Darüber hinaus können Sie verschiedenste Präventivmaßnahmen zur Verhinderung von Exploits beschreiben. Sie können die Vorteile und Schwächen dieser Verfahren detailliert erklären. Sie können Verfahren benennen und erklären, mit denen einige der Präventivmaßnahmen umgangen werden können.

Sie können die Notwendigkeit einer sicheren Programmierweise begründen, um einem Angreifer die Ausnutzung von Sicherheitslücken und auch die Umgehung von Präventivmaßnahmen unmöglich zu machen.

2 Buffer Overflow

Sicherheitsprobleme durch Benutzung systemnaher Programmiersprachen

Systemnahe Programmiersprachen wie C oder C++ erlauben einen direkten Zugriff auf die Speicherstrukturen eines Rechners. Aus dieser Tatsache ergeben sich unter Umständen Sicherheitsprobleme, die Programme und damit den Rechner selbst angreifbar machen. Viele der heute üblichen System- und Anwenderprogramme sind in C bzw. C++ geschrieben. Wenn der Programmierer nicht aufgepasst hat oder nachlässig bei der Implementierung von Programmteilen war, so können dadurch sogenannte Sicherheitslücken entstehen, die von einem Angreifer analysiert und ausgebeutet werden können. Die potenzielle Ausbeutung einer Sicherheitslücke wird als *Exploit* bezeichnet.

Sicherheitslücken

Eine Sicherheitslücke in unserem Sinne liegt dann vor, wenn ein Programm es gestattet, „Daten“ an eine dafür nicht vorgesehene Stelle des Speichers zu schreiben. Die Mechanismen und dadurch geschaffenen Möglichkeiten zur Durchführung eines Exploit sind vielfältig. In diesem Mikromodul werden einige der bekanntesten Sicherheitsprobleme und die Möglichkeiten ihrer Ausbeutung vorgestellt. Letztendlich schafft der Programmierer zwar die Sicherheitslücken selbst und ist damit auch der beste Ansatzpunkt für ihre Beseitigung, in der Praxis ist aber kein Programmierer so perfekt, dass er Sicherheitslücken gänzlich ausschließen kann. In den letzten Jahren wurden viele Verfahren etabliert, die die Ausbeutung von Sicherheitslücken unterbinden oder zumindest erschweren sollen. Diese Verfahren werden durch Compiler, Bibliotheken, Betriebssystem und Hardware oder Kombinationen davon getragen. Einige dieser Verfahren werden hier vorgestellt, aber auch Gegenverfahren der Angreifer, um diese auszuhebeln.

Shellcode

Eine Sicherheitslücke kann im einfachsten Fall zu einem Programmabsturz führen. Die gezielte Ausnutzung einer Sicherheitslücke besteht aber i. Allg. darin, einem Programm eine andere Funktionalität unterzuschieben als die, für die es eigentlich geschrieben wurde. Dies wird durch eine Injektion von Schadcode erreicht, den das Programm dann ausführt. Solcher Schadcode wird historisch bedingt als *Shellcode* bezeichnet. Gemeint ist damit eine kurze Programmsequenz, deren Ausführung die Kontrolle über das infizierte Programm und damit im weitesten Sinne über den Rechner selbst übernimmt. Die Ausführung des Shellcode kann insbesondere die Ausführung weiterer Schad-Software initiieren.[1]

[1] Historisch leitet sich der Begriff Shellcode von dem Angriffsziel ab, eine Kommando-*Shell* zu öffnen, also eine Eingabemöglichkeit für beliebige Betriebssystembefehle zu schaffen.

Buffer Overflow

Wie gelangen nun „Daten“ bzw. Shellcode in einen nicht dafür vorgesehenen Speicherbereich? Das Zauberwort dazu lautet *Buffer Overflow*. Ein Buffer ist ein *begrenzter* Speicherbereich zur Zwischenablage von Daten. Gelingt es, dort mehr Daten abzulegen als die Begrenzung es vorgibt, so läuft der Buffer schlichtweg über. Warum kann das überhaupt passieren? Der Programmierer hat in seinem Programm keine Vorkehrungen getroffen, die einen Überlauf des Buffer verhindern. Eine solche Vorkehrung ist insbesondere die Überprüfung der maximalen Datenlängen. Dies kann schlichtweg wegen Schlampigkeit des Programmierers passieren, oder auch dadurch, dass er sich darauf verlässt, dass entsprechende Überprüfungen „automatisch“ stattfinden. C-Programmierer verwenden ausgiebig die C-Standardbibliothek. In dieser sind bspw. viele Funktionen zur String-Manipulation enthalten, und deren Implementierung ist zum Teil nicht sicher. Betrachten wir das folgende kleine C-Programm:

Q

Quelltext 1

```
1 #include <stdio.h>
2
3 int main ( int argc , char **argv ) {
4    char buf [10] ;
5    strcpy(buf , argv[1]) ;
6    printf("%s \n",buf);
7    return 0 ;
8 }
```

Das Programm kopiert einen Kommandozeilen-Parameter mittels `strcpy` in den Buffer `buf` und druckt diesen aus. Der Buffer hat lediglich eine Länge von 10 Byte, aber es wird nicht überprüft, ob der Parameter dort auch hineinpasst. Die Funktion `strcpy` überprüft die Buffer-Größe auch nicht und verursacht einen Buffer Overflow, falls der Parameter mehr als 10 Zeichen enthält. In diesem Fall schreibt das Programm die überzähligen Daten in einen dafür nicht vorgesehenen, kritischen Speicherbereich und stürzt ab. In den Bibliotheken von C und C++ existieren viele solche unsicheren Funktionen. Dies ist einerseits historisch bedingt, hat aber auch Performance-Gründe. Der verantwortungsvolle Umgang mit diesen unsicheren Funktionen ist allein dem Programmierer überlassen. In Sprachen wie Java und Visual Basic existieren solche unsicheren Funktionen nicht; hier finden automatische Längenüberprüfungen statt, die allerdings einen Overhead verursachen, der bei einer systemnahen Programmierung unter Umständen unerwünscht ist.

In den folgenden Abschnitten werden die prominentesten Beispiele gezeigt, wie durch die Ausnutzung eines Buffer Overflow die Kontrolle über ein Programm gewonnen werden kann. Zunächst müssen wir aber noch anschauen, wie Shellcode prinzipiell aufgebaut ist. Dieser soll schließlich durch den Exploit eines Buffer Overflow zur Ausführung gebracht werden.

3 Shellcode

Shellcode macht sich in der Regel[2] die Tatsache zu Nutze, dass bei einer Von-Neumann-Architektur nicht strikt zwischen Daten und Programmen unterschie-

[2] Diese Aussage ist nicht allgemeingültig. Später in diesem Mikromodul werden wir das Verfahren des *Return Oriented Programming* kennenlernen, das eine Art Shellcode benutzt, die auch auf Architekturmodellen lauffähig ist, die Daten- und Programmbereiche strikt trennen.

den wird. Shellcode wird in Form von Daten angelegt, die dann zur Ausführung gebracht werden. Betrachten wir das folgende Beispielprogramm:

Q

Quelltext 2

```
1 #include <stdio.h>
2
3 signed char cmdshell[] = {
4  0x55, 0x89, 0xe5, 0x31, 0xff, 0x57, 0x81, 0xec, 0x04, 0x00,
5  0x00, 0x00, 0xc6, 0x45, 0xf8, 0x63, 0xc6, 0x45, 0xf9, 0x6d,
6  0xc6, 0x45, 0xfa, 0x64, 0xc6, 0x45, 0xfb, 0x2e, 0xc6, 0x45,
7  0xfc, 0x65, 0xc6, 0x45, 0xfd, 0x78, 0xc6, 0x45, 0xfe, 0x65,
8  0xb8, 0xc7, 0x93, 0xc2, 0x77, 0x50, 0x8d, 0x45, 0xf8, 0x50,
9  0xff, 0x55, 0xf4, 0x55, 0x89, 0xe5, 0xba, 0x7e, 0x9e, 0xc3,
10 0x77, 0x52, 0xff, 0x55, 0xfc
11 };
12
13 int main(int argc, char *argv[]) {
14   void (*sc)() = (void *)cmdshell;
15   printf("\n Shellcode-Groesse ist: %d Byte \n",sizeof(
        cmdshell));
16   printf("\n Wird gestartet . . .\n\n");
17   sc();
18   return 0;
19 }
```

Das Feld `cmdshell` beinhaltet die hexadezimalen Opcodes eines kleinen Maschinenprogramms, das die Windows-Eingabeaufforderung `cmd.exe` startet.[3] Der Funktions-Pointer `sc` zeigt auf den Shellcode, der schließlich zur Ausführung gebracht wird.

Shellcode unterliegt Restriktionen

In diesem Beispiel haben wir den Shellcode selbst zur Ausführung gebracht. Soll er einem anderen Programm per Buffer Overflow „untergejubelt“ werden, so müssen einige grundsätzliche Restriktionen beachtet werden:

1. Der Shellcode muss in der Regel klein sein, weil seine Maximalgröße aufgrund von vielerlei Randbedingungen beschränkt ist.
2. Der Shellcode ist architektur- und betriebssystemabhängig. Shellcode für Windows wird in der Regel nicht unter Linux funktionieren.
3. Die Lage des Shellcode im Speicher kann in der Regel nicht vorausgesagt werden. Deswegen dürfen im Shellcode selbst keine Sprünge zu absoluten Adressen enthalten sein; Shellcode ist lageunabhängig.
4. Ruft der Shellcode selbst Bibliotheks- oder API-Funktionen auf, so gibt es hierfür zwei Möglichkeiten:
   - Die Adressen sind hart codiert. In diesem Fall wird der Shellcode bestenfalls unter einer bestimmten Betriebssystemversion universell funktionsfähig sein.
   - Der Shellcode errechnet die Adressen durch Verwendung geeigneter Funktionen (unter Windows z. B. über *LoadLibrary* und *GetProcAd-*

[3] Das Programm ist unter aktuellen Windows-Versionen aufgrund dort bereits ergriffener Standardsicherheitsverfahren nicht lauffähig. Dies spielt an dieser Stelle jedoch keine Rolle, da nur das Prinzip gezeigt werden soll.

dress) selbst. Dies vergrößert den Umfang des Shellcode allerdings nicht unerheblich.

5. In der Regel wird der Shellcode durch Übergabe eines String-Parameters an eine unsichere String-Funktion platziert. C-Strings enden mit einem `0x00`-Byte. Damit die Übergabe des Parameters nicht vorzeitig abgebrochen wird, darf der Shellcode keine `0x00`-Bytes enthalten. Auch verschiedene andere Sonderzeichen wie z. B. `0x0a` (Linefeed) oder `0x0d` (Carriage Return) sind meist nicht erlaubt.

Es existieren in der Literatur und im Internet viele „Anleitungen“ zur Produktion von Shellcode inklusive Listen mit verwendbaren Opcodes. Die Produktion von korrektem Shellcode ist allerdings nicht Ziel dieses Mikromoduls. Wenden wir uns lieber der Aufgabe zu, einen korrekt formulierten Shellcode in eine konkrete Sicherheitslücke zu injizieren.

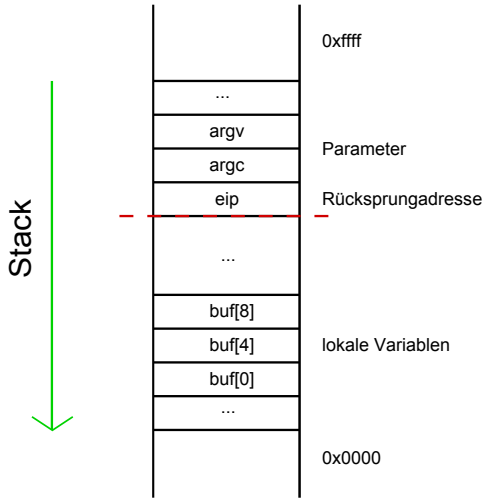
4 Arten des Buffer Overflow

Nachdem wir gesehen haben, was ein Buffer Overflow prinzipiell ist und wie Shellcode aufgebaut ist, wollen wir in diesem Abschnitt die häufigsten konkreten Erscheinungsformen des Buffer Overflow und ihre Benutzbarkeit zur Durchführung eines Exploit näher betrachten.

4.1 Stack Overflow

Die klassische Form des Buffer Overflow ist der Stack Overflow. Das einführende Beispiel dieses Mikromoduls zeigt einen Stack Overflow. Um zu verstehen, was im Fall eines Stack Overflow passiert, muss man sich den Aufbau eines Stack Frame bei einem Funktionsaufruf anschauen (Abb. 1 zum Beispielprogramm).

Abb. 1: Stack Frame der *main*-Funktion



Der Stack wächst bekanntermaßen von hohen Adressen zu niedrigen hin. Auf dem Stack werden Parameter, Rücksprungadresse und lokale Variablen in dieser Reihenfolge abgelegt. Der Buffer `buf` liegt unterhalb der Rücksprungadresse, beginnt mit `buf[0]` und seine nachfolgenden Elemente (`buf[4]`[4] usw.) liegen oberhalb von `buf[0]` in Richtung Rücksprungadresse. Durch einen Buffer Overflow in `strcpy` kann nun der Stack Frame von *main* zerstört werden. Insbesondere kann die Rücksprungadresse überschrieben werden. Wird diese mit der Startadresse eines Shellcode überschrieben, so verzweigt der Kontrollfluss durch die Ausführung des

[4] Es handelt sich hier um einen Stack mit 32-Bit-Daten, daher belegen `buf[0]` bis `buf[3]` eine Stack-Position.

Befehls `ret` am Ende von *main* zum Shellcode. Der Shellcode selbst kann in den Speicherzellen ab `buf[0]` platziert werden. Abb. 2 zeigt das mögliche Szenario.

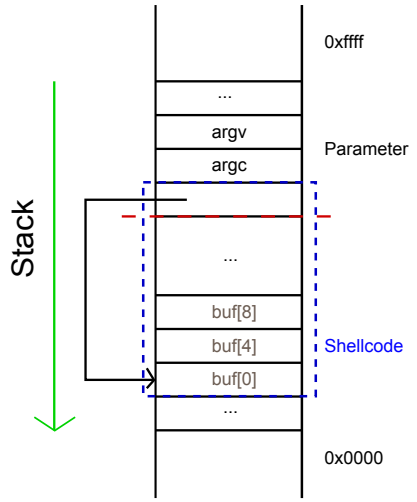


Abb. 2: Shellcode-Platzierung durch Stack Overflow

4.2 Heap Overflow

Ein Heap Overflow findet auf ähnliche Weise wie ein Stack Overflow statt, allerdings im Bereich der dynamisch (z. B. durch `malloc`) allokierten Daten, dem Heap. Durch einen Heap Overflow können dort abgelegte Daten manipuliert werden. Es ist bspw. denkbar, auf diese Weise im Heap gespeicherte Passwörter zu überschreiben. Neben den eigentlichen Daten können aber auch die Verwaltungsstrukturen des Heap selbst durch einen Heap Overflow überschrieben und manipuliert werden.

Exploit von Heap Overflows

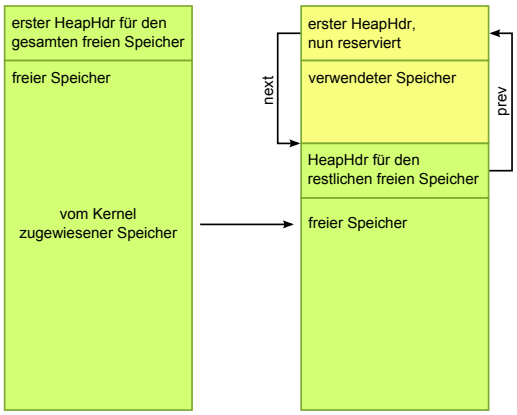
Die Verfahren zur Ausnutzung von Heap Overflows sind sehr vielfältig und auch sehr umgebungsspezifisch. Die Einschleusung und Ausführung von Shellcode ist allerdings nicht so naheliegend wie bei einem Stack Overflow, weil im Heap i. Allg. keine Rücksprungadressen abgelegt werden, über deren Manipulation der Kontrollfluss direkt verändert werden kann. Ein Exploit ist hier immer mehrstufig. Leicht vorstellbar, aber in der Praxis schwer realisierbar (weil dazu sehr detaillierte Kenntnisse über den Aufbau und den Ablauf eines angegriffenen Programms nötig sind) ist das direkte Überschreiben von Funktions-Pointern, sodass beim nächsten Funktionsaufruf über diesen Pointer die Kontrolle an den Shellcode übergeht.

Angriff auf die Verwaltungsstrukturen

Subtilere Methoden greifen die Verwaltungsstrukturen des Heap an. Der Heap besteht aus Speicherblöcken. Diese beinhalten die eigentlichen Daten und einen Header. Der Header enthält einige Metadaten wie die Größe des Speicherblocks und ein Flag, das angibt, ob ein Speicherblock in Benutzung oder frei ist. Darüber hinaus befinden sich im Header auch Pointer auf den nachfolgenden (`next`) und vorangehenden (`prev`) Speicherblock. Insgesamt sind die Speicherblöcke des Heap durch diese Pointer als doppelt verkettete Liste organisiert. Auf dieser Datenstruktur arbeiten die Funktionen zur Allokierung und Freigabe von Speicherblöcken (also `malloc` und `free` in C oder die Windows API-Funktionen *Heapalloc* und *Heapfree*). Abb. 3 zeigt einen Heap vor und nach der Allokierung eines Speicherblocks. Aus dem anfänglich großen, freien Speicherblock werden wiederholt Speicherblöcke entnommen, als belegt markiert und entsprechend verkettet.

Der Heap selbst wächst (bei IA-32 unter Windows) zu höheren Speicheradressen hin. Durch einen Heap Overflow kann demnach ein nachfolgender Speicherblock bzw. dessen Verwaltungsstruktur überschrieben werden, insbesondere auch die

Abb. 3: Allokierung eines Speicherblocks im Heap

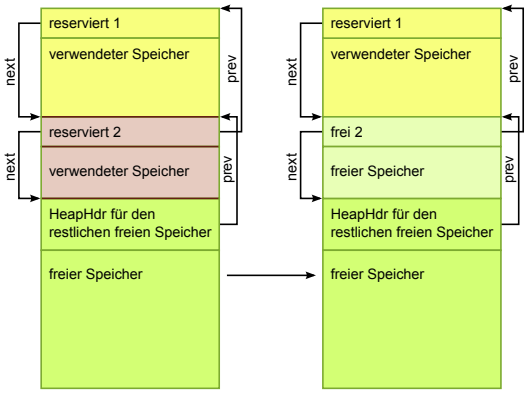


beiden Pointer. Wir werden im Folgenden einen Exploit entwickeln, der die Speicherfreigabe im Heap benutzt.[5]

Exploit bei unsicherer Speicherfreigabe

Wird ein Speicherblock freigegeben, so können „Löcher“ im Heap entstehen, wie dies in Abb. 4 zu sehen ist, weil die Freigabereihenfolge der Speicherblöcke nicht festgelegt ist.

Abb. 4: Freigabe eines Speicherblocks



Zwischen belegten Speicherblöcken entsteht ein freier Speicherblock. Durch wiederholte Allokierung und Freigabe kann so eine Fragmentierung des Speichers erfolgen, wenn keine geeigneten Gegenmaßnahmen getroffen werden. Die Lösung des Problems besteht in der Zusammenführung (kleiner) freier Speicherblöcke zu großen. Dies ist in Abb. 5 dargestellt.

Der untere, zweite freie Speicherblock muss dazu aus der doppelt verketteten Liste entfernt werden (gepunktete Linien). Zum Ausklinken des zweiten freien Speicherblocks sind diese beiden, in C-Syntax angegebenen, Zuweisungen auszuführen, wenn `header1` auf den ersten reservierten Speicherblock zeigt:

```
1. header1->next->next->next->prev = header1->next->next->prev
2. header1->next->next = header1->next->next->next
```

Insbesondere bei

[5] Das verwendete Freigabeverfahren entstammt keiner konkreten Heap-Implementierungen, funktioniert aber meist ähnlich. Im Prinzip bleibt bei unterschiedlichen Implementierungen der nachfolgend beschriebene Exploit gleich, wenn auch seine Realisierung komplizierter werden kann. Das Beispiel stammt aus http://www.heise.de/security/artikel/Der-Heap-271456.html

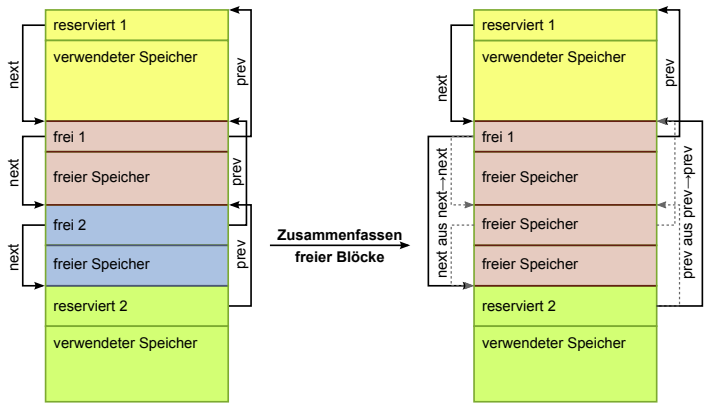


Abb. 5: Zusammenfassung freier Speicherblöcke

```
1. header1->next->next->next->prev = header1->next->next->prev
```

kann der Angreifer sowohl den zu schreibenden Wert als auch die Zieladresse bestimmen, wenn es ihm durch einen Überlauf gelingt, `header1->next->next` auf einen von ihm fingierten Header zeigen zu lassen. Durch passende Werte in `header1->next->next->prev` und `header1->next->next->next->prev` kann somit ein beliebiges 32-Bit-Datum an eine beliebige Speicherstelle geschrieben werden. Dadurch kann bspw. gezielt eine Rücksprungadresse im Stack oder ein Funktions-Pointer überschrieben werden.

Das Beispiel zeigt, dass der Exploit eines Heap Overflow nicht trivial, aber durchaus möglich ist. Die eigentliche „Dreckarbeit“ nach dem Overflow wird hier automatisch durch eine Funktion der Speicherverwaltung erledigt.

4.3 Format-String-Angriff

Betrachten wir das folgende Programm.[6] Es scheint nichts weiter zu tun als einen Kommandozeilenparameter einzulesen und wieder auszugeben. Geben Sie bspw. „hallo“ als Kommandozeilenparameter ein, so druckt es auch „hallo“ aus.

Quelltext 3

```
1 #include <stdio.h>
2
3 int main (int argc, char** argv) {
4
5   long int i = 0xdeadbeef;
6   printf(argv[1]);
7   return 0 ;
8 }
```

Q

Wählen Sie nun als Kommandozeilenparameter den String “%x-%x“. Das Programm gibt jetzt zwei Hexadezimalzahlen aus, z. B. `401910-28ff94` . Was sind das für Zahlen und was ist hier passiert? `printf` erwartet als ersten Parameter einen Format-String. Es interpretiert also unsere Eingabe “%x-%x“ als Format-String,

[6] Am besten kompilieren Sie das Programm und testen es danach aus. Falls das Programm den Namen *fs* hat, so rufen Sie das Programm mit einem Kommandozeilenparameter auf, also z. B. `fs Hallo!`.

parst diesen und gibt die nachfolgenden Parameter entsprechend der Formatvorgabe aus. Leider hat `printf` in diesem Fall keine weiteren Parameter. Dies interessiert `printf` aber nicht weiter. Vielmehr werden die Werte vom Stack als Parameter interpretiert, die an der passenden Stelle liegen, wie wenn es passende Parameter gäbe. Durch die Eingabe eines geschickten Format-String gelingt es so, den Stack oberhalb des Zeigers auf den Format-String beliebig auszulesen. Zum Zeitpunkt der Ausführung von *printf* liegt der Stack Frame von *printf* auf dem Stack, und oberhalb davon der Stack Frame der *main*-Funktion. Durch einen verlängerten Format-String ist es nun möglich, beispielsweise die Rücksprungadresse von *main*[7] oder auch den Wert der lokalen Variablen `i` vom Stack auszulesen. In der Übersetzungsvariante des Autors[8] erfolgt auf die Eingabe "%x-%x-%x-%x-%x-%x-%x" die Ausgabe:

```
401910-28ff94-40196e-401910-4b4838-27-deadbeef
```

Wir können auch destruktiv tätig werden, indem wir durch die Eingabe "%s%s%s%s%s%s%s" einen Programmabsturz verursachen. „%s" erwartet einen Zeiger auf einen String auf dem Stack. Zeigt ein solcher „Zeiger" auf eine ungültige Adresse, so führt das zu einer Exception.

Allgemein funktioniert ein Format-String-Angriff dann, wenn es gelingt, die Kontrolle über einen Format-String zu gewinnen. Dies geschieht entweder, weil der Programmierer es versäumt hat, einen Format-String explizit anzugeben, oder weil der Format-String selbst eine Variable innerhalb eines Programms ist, die manipuliert werden kann. Format-String-Angriffe sind prinzipiell mit allen C-Befehlen möglich, die Format-Strings verwenden, also z.B̈ auch mit `scanf`, `fprintf` usw. Neben der Manipulation von Rücksprungadressen können mit dem Verfahren beispielsweise auch Funktions-Pointer manipuliert werden.

Wir können mit einem Format-String-Angriff offensichtlich Teile des Stack auslesen, aber gelingt es auch, beliebige Werte gezielt in den Stack zu schreiben?
Antwort: Ja!
Dazu dient der Format-String „%n". Durch ihn schreibt `printf` die Anzahl der

[7] Die Rücksprungadresse von *printf* kann nicht ausgelesen werden, weil diese unterhalb der Parameter von *printf* liegt.
[8] Die konkrete Ausgabe ist abhängig vom Betriebssystem und vom verwendeten Compiler.

bis dahin ausgegebenen Zeichen an eine durch einen Pointer bestimmte Adresse. Betrachten wir das folgende Beispiel:[9]

Q

Quelltext 4

```
1  #define __USE_MINGW_ANSI_STDIO 1
2  #include <stdio.h>
3
4  int main (int argc, char** argv) {
5
6    long int branch = 0;
7    long int * brptr = & branch;
8    printf(argv[1]);
9    if (branch) {
10     printf("\nSecret! \n");
11     printf("Zeichen: %d \n",branch);
12   }
13   return 0 ;
14 }
```

Wenn es gelingt, die Variable `branch` mit einem Wert ungleich Null zu überschreiben, so gibt das Programm den String „Secret!“ und die Anzahl der ausgegebenen Zeichen aus. Durch Auslesen des Stack nach dem oben genannten Verfahren kann die Adresse der Variablen `brptr` bestimmt werden, die ein Pointer auf die Variable `branch` ist. Der Format-String zum Überschreiben von `branch` muss nun so aufgebaut sein, dass der Stack vor dem Pointer `brptr` ausgelesen und der Pointer `brptr` der Parameter ist, der zum abschließenden „%n“ passt. In der Übersetzungsvariante des Autors gelingt dies mit folgender Eingabe:

%x-%x-%x-%x-%x-%x-%n

Die Ausgabe des Programms lautet:

```
401970-28ff94-4019ce-101970-864790-0-
Secret!
Zeichen: 37
```

Die Variable `branch` wurde mit dem Wert `37` überschrieben, was der Anzahl der ausgegebenen Zeichen in der ersten Zeile entspricht.[10]

Wir können jetzt einen Wert in eine beliebige Stelle im Stack oder allgemein im Speicher ablegen, auf die ein Pointer verweist, der auf dem Stack liegt. Wir wollen aber mehr, nämlich einen vorgegebenen Wert in den Speicher schreiben. Dazu müssen wir den Format-String geschickt formulieren und die Ausgabe aufpolstern. Standardmäßig bedient man sich dabei Formatvorgaben der Form „%mx“. Der Wert `m` gibt an, mit wie vielen Stellen die Ausgabe des zu x korrespondierenden Hexadezimalwerts erfolgt. Die (vielen) ausgedruckten Leerzeichen werden einfach mitgezählt.

[9] Die erste Programmzeile ist unter Windows bei Verwendung von MinGW erforderlich. `#define __USE_MINGW_ANSI_STDIO 1` veranlasst den Compiler, den ANSI-I/O-Standard statt der Konventionen von Windows anzuwenden. Unter Windows ist ansonsten die Formatvorgabe „%n“ wirkungslos.

[10] Anzumerken ist hier, dass der Pointer `brptr` im Stack oberhalb der Variablen `branch` liegt, obwohl er im Programmtext nach der Variablen deklariert ist. Dies ist ein Werk des eingesetzten Compilers. Die `0` am Ende der ersten Zeile der Ausgabe ist der ursprüngliche Wert der Variablen `branch`.

Im folgenden – gegenüber dem vorangegangenen leicht abgewandelten – Beispiel muss die Variable `branch` genau mit dem Wert `256` überschrieben werden, damit `Secret!` ausgegeben wird. Das gelingt mit der folgenden Eingabe:

%50x-%50x-%50x-%50x-%30x-%20x-%n

Q

Quelltext 5

```
1  #define __USE_MINGW_ANSI_STDIO 1
2  #include <stdio.h>
3
4  int main (int argc, char** argv) {
5
6    long int branch = 0;
7    long int *brptr = &branch;
8    printf(argv[1]);
9    if (branch == 256) {
10     printf("\nSecret! \n");
11     printf("Zeichen: %d \n",branch);
12   }
13   return 0 ;
14 }
```

Wieder wird der Stack bis zur gewünschten Stelle ausgelesen. Durch das Aufpolstern der Ausgabe wurden bis dahin genau `256` Zeichen ausgegeben. Damit wird die Variable `branch` überschrieben.

Bisher können wir einen beliebigen Wert an eine Stelle im Speicher schreiben, auf die ein Pointer verweist, der auf dem Stack liegt. Wenn jedoch der Format-String selbst auf dem Stack liegt, so können wir durch einen Format-String-Angriff sogar Shellcode auf dem Stack ausführen. Beim folgenden Programmfragment wird die Eingabe zu `fgets` im Feld `buf` im Bereich der lokalen Variablen, also auf dem Stack, abgelegt:

Q

Quelltext 6

```
1 int main ( void ) {
2   char buf [512];
3   ...
4   fgets(buf,sizeof(buf),stdin);
5   printf(buf);
6   return 0;
7 }
```

One-Shot-Methode

Abb. 6[11] zeigt das prinzipielle Szenario zur Einbringung und Ausführung von Shellcode. Die Eingabe zu `fgets`, also der Format-String, besteht aus einem Zeiger auf eine Rücksprungadresse[12], gefolgt von den üblichen Formatvorgaben, gefolgt vom Shellcode. Zu beachten ist, dass der Format-String mit aufsteigenden Adressen

[11] Diese und die folgende Abbildung sind [Müller, 2015] entnommen.

[12] Das kann beispielsweise die Rücksprungadresse von *main* oder die von *printf* sein. Wo genau die Rücksprungadresse im Stack liegt, ist für das prinzipielle Verfahren irrelevant. Die Position der Rücksprungadresse auf dem Stack könnte durch eine vorangegangene Stackanalyse mittels eines Debuggers gewonnen worden sein.

im Stack abgelegt wird, `buf[0]` liegt also an der niedrigsten Adresse. Wenn alles geschickt gewählt ist, kann jetzt die Rücksprungadresse mit der Startadresse des Shellcode überschrieben, und somit der Shellcode zur Ausführung gebracht werden. Da die Rücksprungadresse hier mit einem einzigen „Schuss“ überschrieben wird, wird die gezeigte Methode auch als One-Shot-Methode bezeichnet.

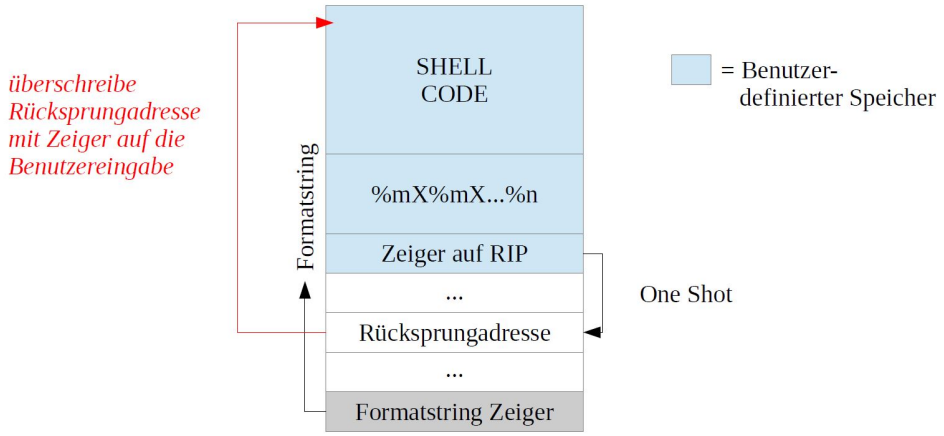


Abb. 6: Ausführung von Shellcode

Die One-Shot-Methode birgt Schwierigkeiten, falls sehr große Werte in den Speicher geschrieben werden müssen, beispielsweise reale 32-Bit-Adressen. Der zu einem „%n“ korrespondierende Zeiger zeigt zwar auf einen 32-Bit-Wert, aber es sind unter Umständen gewaltige Aufpolsterungen erforderlich, um einen gewünschten Wert zu erzeugen. Glücklicherweise gibt es aber zwei weitere Varianten von „%n“, nämlich „%hn“ und „%hhn“. Ersterer korrespondiert mit einem Zeiger auf ein Wort (16 Bit), letzterer mit einem Zeiger auf ein Byte. Statt also einen sehr großen Wert mit einem „Schuss“ zu schreiben, kann man ihn in mehreren kleineren „Portionen“ im Speicher erzeugen. Diese Methode wird als Short-Write-Methode bezeichnet. Das Prinzip zeigt Abb. 7 bei Verwendung von „%hn“. Hier wird die Rücksprungadresse in zwei Teilen überschrieben, ansonsten entspricht das Szenario dem der vorangegangenen Abbildung.

Short-Write-Methode

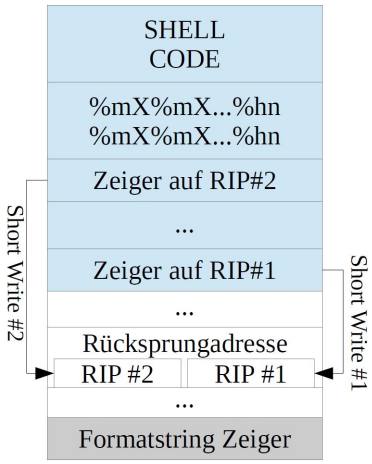


Abb. 7: Short-Write-Methode

5 Gegenmaßnahmen

Sicherheitslücken der gezeigten Art sind letztendlich immer die Folge von Programmierfehlern. Die beste Gegenmaßnahme ist daher ohne jeden Zweifel die Vermeidung von Programmierfehlern, sofern die Schwächen einer Programmiersprache bekannt sind. Im Fall von C sollte der Programmierer über unsichere

Vermeidung unsicherer Bibliotheksfunktionen

Bibliotheksfunktionen wie `strcpy` oder `puts` Bescheid wissen, und diese entweder vermeiden oder ihren Gebrauch durch Längenüberprüfungen absichern. Für die meisten unsicheren Funktionen existieren in der Standardbibliothek sichere Äquivalente. Andererseits sind unsicheren Funktionen aus verschiedenen Gründen nicht so ohne Weiteres streichbar, ihre Benutzung kann also nicht a priori ausgeschlossen werden.

Verschiedene Unterstützung kann der Programmierer durch Verwendung weiterentwickelter Compiler mit vielerlei Kontrolloptionen erhalten, die den fehleranfälligen Gebrauch von Programmkonstrukten anmahnen oder unterbinden. So kann bspw. von den meisten Compilern prinzipiell das Fehlen eines Format-String erkannt werden.

Sicherheitslücken können selbstverständlich auch im wahrsten Sinne des Wortes importiert sein, bspw. in Form unsicherer API-Funktionen. Diese liegen außerhalb des eigentlichen Verantwortungsbereichs des Programmierers und sind auch nur schwer zu erfassen. Hier kann man eigentlich nur auf das Verantwortungsbewusstsein des Zulieferers hoffen und darauf, dass Fehler durch Updates eliminiert werden.

Die bisher genannte Gegenmaßnahmen setzten ausdrücklich Wissen und Willen des Programmierers zur Vermeidung von Sicherheitslücken voraus. Andere Gegenmaßnahmen sind allgemeiner Art. Sie können innerhalb des Compilers angesiedelt, aber auch Bestandteil von Betriebssystem und Hardware sein. Es sind hierbei zwei Klassen zu unterscheiden:

Compileroptionen

1. Sicherheitsmechanismen bei der Codeerzeugung. Diese sollen keineswegs dazu verleiten, unsichere Programme mit dem Vertrauen darauf zu verfassen, dass der Compiler die Fehler schon eliminieren wird. Vielmehr sollen sie als Rückversicherung dienen, falls dem Programmierer doch ein unbeabsichtigter Fehler unterläuft. Diese Sicherheitsmechanismen müssen in der Regel durch Compileroptionen aktiviert werden.

Laufzeitmechanismen

2. Laufzeitmechanismen. Die Umsetzung der Sicherheitsmechanismen der ersten Klasse setzen eine (Neu-)Kompilierung voraus. Dadurch können alle die Programme nicht davon profitieren, deren Sourcecode nicht vorliegt. Sollen auch eventuell ältere Programme, die Sicherheitslücken enthalten, ohne böse Folgen weiter verwendet werden, so müssen die Sicherheitsmaßnahmen außerhalb des Programms stattfinden.

Die wichtigsten und geläufigsten Verfahren dieser beiden Klassen werden in den folgenden Abschnitten vorgestellt.

5.1 Stack Smashing Protection

C sieht keine automatische Längenüberprüfungen für Arrays vor, weswegen ein Buffer Overflow nicht generell ausgeschlossen werden kann. Es besteht aber die Möglichkeit einen Buffer Overflow zu erkennen, bevor es zur Codeausführung kommt, also vor Ausführung eines `ret` mit einer manipulierten Rücksprungadresse. Dieses Verfahren wird *Stack Smashing Protection* genannt und ist in Abb. 8 dargestellt.

Canary

Bei jedem Unterprogrammaufruf wird ein sogenannter *Canary* auf den Stack gelegt, und zwar an den Anfang des neuen Stack Frame. Dieser Canary[13] wird beim

[13] Die Verwendung des Begriffs „Canary“ (Kanarienvogel) wurde beim Bergbau entlehnt. In frühen Zeiten wurde ein Käfig mit einem Kanarienvogel mit in die Grube genommen. Fiel der (empfindliche)

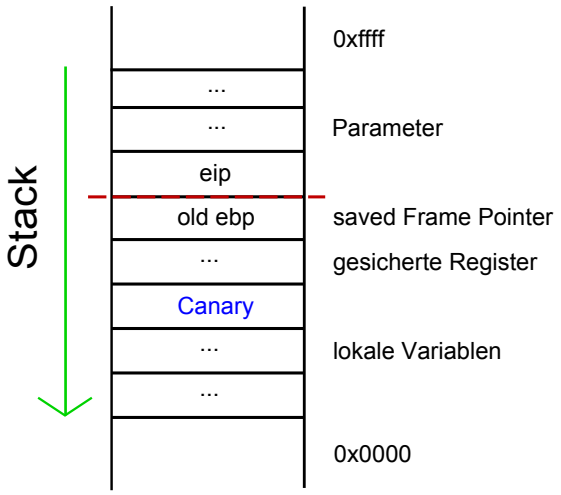


Abb. 8: Schutz durch Einfügen eines Canary

Programmstart pseudozufällig gewählt und dann bei allen Unterprogrammaufrufen verwendet. Tritt ein Buffer Overflow auf, der über den aktuellen Stack Frame hinausreicht, so wird der Canary überschrieben. Vor dem Rücksprung aus einem Unterprogramm wird die Gültigkeit des Canary überprüft. Wurde er verändert, so muss ein Buffer Overflow stattgefunden haben, und das Programm wird an dieser Stelle unterbrochen, also vor der mutmaßlichen Ausführung von injiziertem Code. Um neben der Rücksprungadresse auch den alten Frame Pointer und sonstige gesicherte Register zu schützen, wird der Canary nach diesen platziert.

Ein Buffer Overflow, der den Canary nicht erreicht, wird logischerweise durch dieses Schutzverfahren nicht erkannt. Dadurch ist ein potentieller Angriff, der nur auf der Manipulation der lokalen Variablen einer Funktion beruht, nicht erkennbar. Des Weiteren ist ein Angriff denkbar, der auf den Parametern einer Funktion beruht. Diese können weiterhin durch einen Buffer Overflow manipuliert werden, wobei der Canary überschrieben wird. Da der Canary allerdings erst beim Verlassen der Funktion überprüft wird, kann es in diesem Augenblick schon zu spät sein.

Canaries bieten durchaus einen sinnvollen Schutz, aber keine perfekte Sicherheit. Zudem sinkt die Performance eines Programms etwas durch das Einfügen und Überprüfen der Canaries. Die Verwendung von Canaries wird über eine Compileroption gesteuert.

5.2 Maßnahmen gegen Heap Overflow

Da die Angriffe durch Heap Overflow vielfältig sind, ist es schwer, generelle Gegenmaßnahmen zu ergreifen. Die Absichten eines Heap Overflow innerhalb eines Speicherblocks sind ebenso schwer erfassbar wie die innerhalb eines Stack Frame.

Konsistenz der Datenstrukturen

Betreffen die Angriffe jedoch die Verwaltungsstrukturen des Heap, überläuft also ein Heap Overflow die Grenzen eines Speicherblocks, so sind zwei Schutzmaßnahmen wirksam. Das erste Verfahren betrifft die Implementierung der Speicherverwaltung. Zentralpunkt ist hierbei die Überprüfung der Konsistenz der Datenstrukturen, also der Pointer, vor jeder Umstrukturierung. Dazu werden die Verkettungen in beiden Richtungen überprüft. Es ist leicht nachvollziehbar, dass bei einem solchen Schutz der Angriff in Abs. 4.2 nicht funktionieren könnte. Das

Kanarienvogel tot von der Stange, so war das ein sicheres Anzeichen dafür, dass giftiges Gas in der Grube war.

zweite Verfahren funktioniert ganz ähnlich wie das der Canaries. Hierzu wird im Header eines Speicherblocks ein Canary eingefügt.

Beide Verfahren wurden mit Service Pack 2 in Windows XP eingeführt. Die Canaries heißen hier allerdings *Cookies*, werden im Header eines Speicherblocks eingefügt und bestehen nur aus jeweils einem Byte. Da es somit nur 256 unterschiedliche Möglichkeiten zur Wahl der Cookies gibt, können diese potentiell durch eine Brute-force-Attacke erraten werden.

5.3 Verhinderung von Format-String-Angriffen

Zur Verhinderung von Format-String-Angriffen bieten weiterentwickelte Compiler einige Optionen an. Neben dem Fehlen des Format-String kann dadurch auch die Konsistenz von Format-String und den sich anschließenden Parametern automatisch überprüft werden. Ausgenommen von der automatischen Überprüfung zur Kompilierungszeit sind allerdings Befehle mit variabler Parameterliste (z. B. `vprintf`) und Format-Strings in Form von Variablen.

5.4 Address Space Layout Randomization

ASLR

Das Verfahren der *Address Space Layout Randomization* (ASLR) wurde unter Linux im Jahr 2001 und unter Windows mit der Version Vista eingeführt. Ziel von ASLR ist es, die Lage von Code-und Datenbereichen sowie Stack und Heap bei jedem Programmstart im Adressraum des Prozesses zu randomisieren. Dies verhindert zwar keine Buffer Overflows, erschwert aber den Exploit, weil der Shellcode nicht mehr mit absoluten Adressen arbeiten kann. Insbesondere unter Linux ist ohne ASLR das Adress-Layout eines Programms leicht voraussagbar.

Position Independent Executable

Zur Durchführung von ASLR müssen zwei Komponenten zusammenarbeiten. Zum einen muss das Betriebssystem, also letztendlich der Loader, eine Randomisierung durchführen. Dazu wurde unter Windows Vista das PE-Format um ein Flag erweitert, das signalisiert, ob eine Adressrandomisierung möglich ist. Zum zweiten ist eine Neukompilierung mit den entsprechenden Optionen erforderlich, um Programme für ein randomisierbares Adress-Layout zu erzeugen. Der Programmcode darf keine absoluten Adressen oder Sprünge mehr enthalten und wird zum *Position Independent Executable* (PIE). Auch ohne Neukompilierung bietet ASLR einen teilweisen Schutz, da auch dann noch die benutzten Bibliotheken sowie Stack und Heap im Adressraum randomisiert werden können. Lediglich Code- und Datensegmente des Programms bleiben statisch.

ASLR ist ein wirksamer Schutz, um die meisten Angriffe schwerer oder ganz unmöglich zu machen, da die Anzahl der dem Angreifer bekannten Adressen deutlich sinkt. Perfekt ist ASLR jedoch nicht, wenn die Randomisierung nur blockweise erfolgt. Beginnt bspw. ein Codesegment immer an einem Vielfachen von Adresse `0x1000`, so bleiben Teile der Adressen bei jedem Programmlauf konstant. Methoden zur Ausnutzung dieser Schwäche sind bspw. in [Shacham and andere, 2004] zu finden.

5.5 Data Execution Prevention

Wir gehen bisher davon aus, dass Shellcode in den Datenbereich (insbesondere Stack und Heap) eingeschleust und dort zur Ausführung gebracht wird. Ein adäquates Mittel zur Unterbindung eines solchen Exploit ist das Verbot einer Codeausführung im Datenbereich. Um dies zu realisieren, müssen Hardware und Betriebssystem eines Rechners zusammenarbeiten. Moderne Prozessoren verfügen über eine Hardware-Erweiterung mit dem allgemeinen Namen NX-Bit in der

MMU. Das NX-Bit wurde in der x86-Welt von AMD unter diesem Namen mit dem Athlon 64 eingeführt, bei Intel heißt es XD-Bit und hielt mit den späteren Version des Pentium 4 Einzug. NX steht für „No eXecute“ und bietet die Möglichkeit, Speicherseiten als ausführbar oder nicht-ausführbar (*non-executable*) zu markieren.

NX-Bit

Unterstützt das Betriebssystem die Auswertung des NX-Bit, so kann es die Ausführung von Code in Datenbereichen unterbinden. Dieses Verfahren wird als *Data Execution Prevention* (DEP) bezeichnet. DEP hielt mit Service Pack 2 von XP Einzug in die Windows-Welt. Bei den 64-Bit-Varianten der aktuellen Windows-Versionen ist DEP immer aktiviert, bei den 32-Bit-Versionen ist es zwar standardmäßig aktiviert, kann aber pauschal abgeschaltet werden. Darüber hinaus ist die Möglichkeit gegeben, durch Setzten entsprechender Compileroptionen DEP individuell abzuschalten. Manche Programme benötigen eine Codeausführung im Datenbereich und wären somit bei aktiviertem DEP nicht mehr lauffähig. Gerade diese Programme sind deswegen besonders anfällig für Angriffe.

DEP ist ein gutes Sicherheitsverfahren, das auch bei Programmen wirkungsvoll ist, die nicht neu kompiliert werden können. Es erfordert allerdings moderne Hardware und geht davon aus, dass Shellcode tatsächlich im Datenbereich zur Ausführung gebracht wird. Im folgenden Abschnitt werden wir ein Verfahren kennenlernen, bei dem Shellcode **nicht** im Datenbereich eines Programms ausgeführt wird.

6 Gegen-Gegenmaßnahmen

Angriffe und Abwehrmaßnahmen beim Versuch, Exploits bei Buffer Overflows zu vermeiden, gleichen einem Katz-und-Maus-Spiel. Kaum ist eine Gegenmaßnahme ergriffen worden, entwickelt sich von Seiten der Angreifer ein großer Ehrgeiz, diese durch Gegen-Gegenmaßnahmen zu untergraben. Im Folgenden werden wir zwei prominente Gegen-Gegenmaßnahmen kennenlernen, die der aktuellen IT-Sicherheit große Kopfschmerzen bereiten.

6.1 Return-to-libc

Gehen wir von dem realistischen Szenario aus, dass ein (älteres) Programm mit Sicherheitslücken vorliegt, das nicht mehr nachgebessert und neu kompiliert werden kann. Dieses Programm bietet die Möglichkeit, einen Stack Overflow zu erzeugen. Das Programm wird auf einem modernen Rechner mit DEP ausgeführt. Das Einbringen und Ausführen von Shellcode auf dem Stack ist also ausgeschlossen.

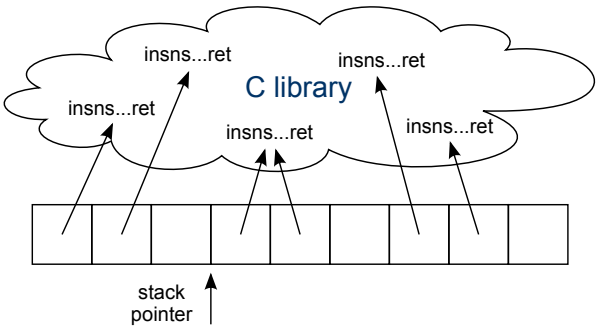
Der Trick besteht nun darin, die Rücksprungadresse nicht mit der Startadresse eines auf dem Stack abgelegten Shellcode zu überschreiben, sondern mit der Startadresse einer bereits im Codebereich des Programms liegenden Funktion. Damit wird auf dem Stack kein Code ausgeführt, und DEP ist unwirksam. Ein geeigneter Kandidat ist die Funktion `system` der C-Standardbibliothek *stdlib.h*, die vielfach in Programmen eingebunden wird. Mit `system` können Systemkommandos abgesetzt werden. Es ist also im Prinzip nichts weiter zu tun, als geeignete Parameter für `system` auf dem Stack zu generieren und eine Rücksprungadresse mit der Startadresse von `system` zu überschreiben. Beim nächsten `ret` wird das Systemkommando ausgeführt. Eine andere Umsetzungsmöglichkeit bietet die Funktion `memcpy`. Mit Hilfe von `memcpy` wird hierbei Shellcode vom Stack in den Codebereich eines Programms kopiert und dort ausgeführt. Wegen der Verwendung der C-Bibliothek und der darauf beruhenden Veröffentlichung in [Designer, 1997] wird das Verfahren *Return-to-libc* genannt. Prinzipiell ist es allerdings universell und nicht auf die C-Bibliothek beschränkt.

6.2 Return Oriented Programming

Gadget

Die Weiterentwicklung von Return-to-libc nennt sich *Return Oriented Programming* (ROP) und wurde erstmals in [Designer, 1997] vorgestellt. Hierbei werden nicht mehr ganze Funktionen der C-Bibliothek aufgerufen, sondern nur noch Codefragmente, sogenannte *Gadgets*. Ein Gadget besteht nur aus wenigen Instruktionen, gefolgt von einem `ret`-Befehl und bildet das Ende einer Funktion der C-Bibliothek. Bei einem Stack Overflow wird eine Rücksprungadresse hinterlegt, die kurz vor das Ende einer Bibliotheksfunktion zeigt. Bei einem Sprung dorthin mittels `ret` wird der Stack Pointer inkrementiert und die Instruktionen des Gadget werden ausgeführt. Da das Gadget seinerseits mit einem `ret` endet, wird das Datum, auf das der bereits inkrementierte Stack Pointer nunmehr zeigt, wieder als Rücksprungadresse interpretiert. Diese Rücksprungadresse ist logischerweise der Einstiegspunkt eines weiteren Gadget. Abb.9 zeigt nochmal das prinzipielle Verfahren.[14] Durch die fortlaufende Inkrementierung des Stack Pointer werden nacheinander die angewählten Gadgets ausgeführt. Der Stack Pointer übernimmt sozusagen die Rolle eines Instruction Pointer, wobei die Instruktionen in Form der Gadgets vorliegen.

Abb. 9: Linearer „Programmablauf“ durch ROP



Zur Umsetzung von ROP wird durch einen Stack Overflow lediglich eine Folge von Rücksprungadressen auf den Stack gelegt, die der Reihe nach die Gadgets zur Ausführung bringen. Da auf dem Stack kein Code ausgeführt wird, ist DEP damit ausgehebelt.

Um mit Hilfe von ROP aus Sicht des Angreifers sinnvolle Funktionalität zu gewinnen, müssen zwei Voraussetzungen geschaffen werden:

1. Die einzelnen Gadgets müssen so geartet sein, dass sie bei der „Programmausführung“ zusammen sinnvolle Arbeit leisten können.
2. Die lineare Abfolge der Gadgets führt unter Umständen zu sehr langen „Programmen“. Zudem sind solche Programme nicht universell in dem Sinne, dass durch sie eine beliebige Funktionalität abgebildet werden könnte.

Zusammengefasst lässt sich sagen, dass ein Ensemble geeigneter Gadgets in der C-Bibliothek für IA-32 und andere Architekturen gefunden wurde, mit dem die gewünschten Ziele erreichbar sind. Insbesondere die Umsetzung des zweiten Ziels ist interessant. Wenn wir eine sinnvolle Zusammenfassung von Gadgets als Makro bezeichnen, dann kann mit diesen Makros ein Befehlssatz zusammengestellt werden, der Turing-Mächtigkeit besitzt. Damit ist das Schreiben beliebiger Programme durch ROP möglich. Es existieren sogar Compiler, die in einer Hochsprache geschriebene Programme in ROP übersetzten [Roemer et al., 2008].

[14] In dieser und den folgenden Abbildungen ist die „Leserichtung“ der Programme von links nach rechts. Durch Inkrementierung des Stack Pointer wandert dieser also um eine Zelle nach rechts, weil dies eher der intuitiven Vorstellung eines Programmablaufs entspricht.

ROP-Makros

Die Vorstellung der kompletten Makros für ROP würde hier zu weit führen. Insbesondere die Umsetzung arithmetischer und logischer Operationen sowie die Generierung von bedingten Sprüngen und Unterprogrammaufrufen ist komplex. Der interessierte Leser sei hierzu bspw. an [Roemer et al., 2012] verwiesen. An dieser Stelle sollen lediglich beispielhaft Makros für das Laden von Konstanten, sowie für das Lesen und Schreiben von Registern vorgestellt werden.

1. **Laden von Konstanten:**
   Es besteht keineswegs die Notwendigkeit, den Stack bei ROP lediglich mit Rücksprungadressen zu füllen. Der Stack kann ebenso zur Speicherung von Konstanten und Variablen verwendet werden. Abb. 10 zeigt ein Makro zum Laden einer Immediate-Konstanten. Die Konstante liegt nach dem Pointer auf Gadget `pop edx; ret` im Stack. Nach dem (Rück-)Sprung[15] zum Gadget zeigt `esp` auf diese Konstante, die mittels `pop` nach `edx` geladen wird. `pop` inkrementiert den Stack Pointer, wodurch dieser auf die Adresse des nächsten Gadget zeigt.

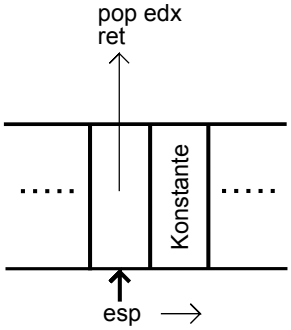


Abb. 10: Laden einer Konstanten in ROP

2. **Laden von Registern:**
   Abb. 11 zeigt, wie ein Register aus dem Speicher geladen werden kann. Nach dem Pointer auf das erste Gadget ist die Base der Zieladresse abgelegt. Diese wird – wie im ersten Makro gezeigt – nach `eax` geladen. Im zweiten Gadget wird der Speicher mit Base+Offset adressiert, wobei das Gadget den Offset 64 fest vorgibt. Der dort gespeicherte Wert wird nach `eax` kopiert.

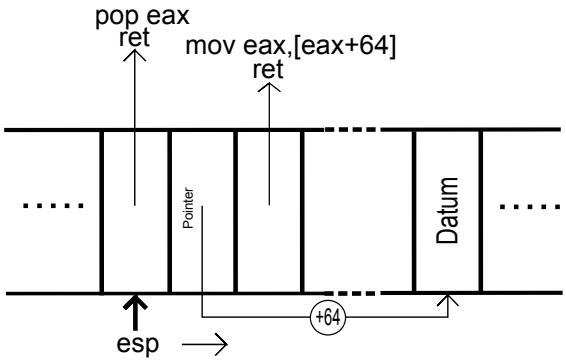


Abb. 11: Laden eines Registers in ROP

3. **Speichern von Registern:**
   Das Ablegen eines Registers in den Speicher funktioniert ganz ähnlich wie das Laden (Abb. 12), jedoch erfolgt hier die Adressierung über `edx`. Das zweite Gadget gibt hier den konstanten Offset 24 vor.

[15] In dieser und in den folgenden Abbildungen ist immer die Situation vor dem Rücksprung in das Start-Gadget dargestellt. Nach dem Rücksprung durch ein vorausgegangenes `ret` zeigt damit `esp` eine Position weiter rechts, in diesem Fall auf die Konstante.

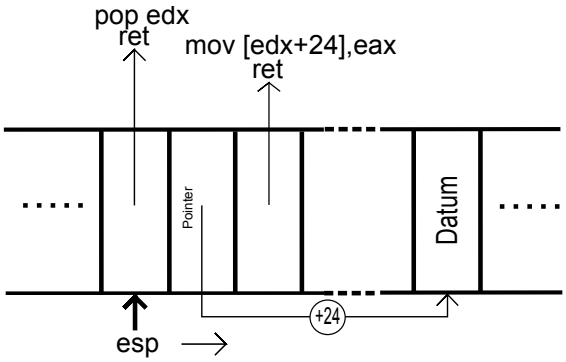


Abb. 12: Speichern eines Registers in ROP

7 Zusammenfassung

In diesem Mikromodul wurden einige Sicherheitsaspekte systemnaher Programmierung besprochen. Am Beispiel von C wurde gezeigt, dass durch die Freiheiten dieser Programmiersprache Sicherheitslücken entstehen können, die für Angriffe ausgenutzt werden. Viele Programme und die meisten Betriebssysteme und Betriebssystembibliotheken sind in C geschrieben. Demzufolge verwundert es nicht, dass immer wieder neue Sicherheitslücken gefunden und veröffentlicht werden und so zu der ständigen Notwenigkeit von Programm- und Betriebssystem-Updates führen.

Kenntnisse in systemnaher Programmierung ermöglichen es, diese Sicherheitslücken und ihre Ausnutzung zur Kompromittierung eines Rechners zu erkennen und zu verstehen. Sicherheitslücken beruhen zu einem großen Teil auf Buffer Overflows, also auf der Möglichkeit, Daten und Programme an unerwünschten Positionen im Speicher zu platzieren. Wir haben Stack Overflows, Heap Overflows und Format-String-Angriffe kennengelernt.

Von Seiten der Verteidigung wurde eine Reihe von Verfahren entwickelt, die die Ausbeutung von Sicherheitslücken verhindern oder zumindest stark behindern sollen. Keines dieser Verfahren ist perfekt, denn das eigentliche Grundproblem beim Entstehen der Sicherheitslücken bleibt immer bestehen: Der Faktor Mensch und seine Programmierfehler. Zudem wurden durch die Ergreifung von Gegenmaßnahmen auch die Methoden der Angreifer immer raffinierter.

8 Übungen

Übung 1

Ü

Beschreiben Sie detailliert einen Angriff durch einen Heap Overflow nach dem in Abs. 4.2 gezeigten Verfahren der Pointer-Manipulation. Konkret soll die Adresse `0x77352904` mit dem Wert `0xdeadbeef` überschrieben werden, wobei davon ausgegangen wird, dass unter dieser Adresse irgendeine Rücksprungadresse abgelegt ist. Beziehen Sie sich bei der Beschreibung auf Abb. 5.

Übung 2

Ü

Kompilieren Sie das folgende Programm. Untersuchen Sie zunächst den Stack-Aufbau durch geeignete Format-String-Angriffe. Formulieren Sie anschließend einen Format-String für einen Angriff nach der One-Shot-Methode, der an die Adresse `0x77352904` den Wert `1000` schreibt. Beachten Sie hierbei die Little-Endian-Konvention.

Tipp: Beliebige Bytes bzw. „Sonderzeichen“ – außer der NULL – können bei gedrückt gehaltener ALT-Taste und gleichzeitiger Eingabe eines Dezimalwertes über den NUM-Block erzeugt werden. Beispielsweise erzeugt `[ALT 33]` ein Ausrufezeichen. Eine NULL darf in einem Format-String ohnehin nicht vorkommen, denn sie würde den Format-String beenden. In unserem Format-String-Angriff darf daher auch die Adresse kein Byte `00` enthalten.

Quelltext 7

Q

```
1 #define __USE_MINGW_ANSI_STDIO 1
2 #include <stdio.h>
3 int main ( void ) {
4   char buf[512];
5   fgets(buf,sizeof(buf),stdin);
6   printf(buf);
7   return 0;
8 }
```

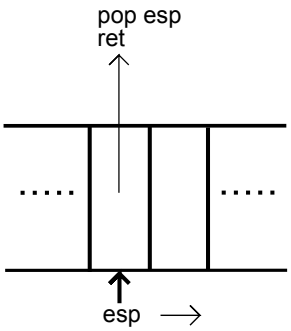
Formulieren Sie in einem zweiten Schritt einen Format-String für einen Angriff nach der Short-Write-Methode unter Verwendung von „%hhn“, der an die Adresse `0x77352904` den Wert `0xdeadbeef` schreibt.

Übung 3

Ü

Welche Funktionalität realisiert das einfache ROP-Makro in Abb. 13 ?

Abb. 13: ROP-Makro

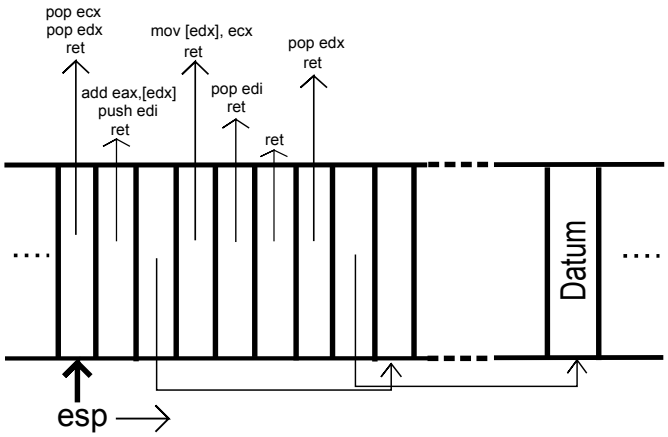


Ü

Übung 4

Welche Funktionalität entwickelt das ROP-Makro in Abb. 14 ?

Abb. 14: ROP-Makro



Stichwörter

Verzeichnisse

I. Abbildungen

II. Literatur

Parvez Anwar. Buffer overflows in the microsoft windows environment. *Technical Report, RHUL-MA2009-06, University of London*, 2009.

Solar Designer. Getting around non-executable stack (and fix). *Bugtraq*, 1997.

Intel. *Intel 80386, Programmers Reference Manual*. http://css.csail.mit.edu/6.858/2011/readings/i386.pdf, 1987.

Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html, 2012.

Kip R. Irvine. *Assembly Language for Intel-Based Computers (5th Edition)*. Prentice Hall, 2006.

Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1978.

Brian W. Kernighan and Dennis M. Ritchie. *Programmieren in C*. Hanser Fachbuch, 1990.

Tilo Müller. *Software reverse engineering*. Vorlesung, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2015.

Ryan Roemer, Erik Buchanen, Hovav Shacham, and Steve Savage. Return-oriented programming: Exploitation without code injection. *University of California, San Diego*, 2008.

Ryan Roemer, Erik Buchanen, Hovav Shacham, and Steve Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Info. & Sys. Security 15(1):2*, 2012.

Joachim Rohde. *Assembler GE-PACKT, 2. Auflage*. Redline GmbH, Heidelberg, 2007.

Hovav Shacham and andere. On the effectiveness of addressspace randomization. *ACM Conference on Computer and Communications Security (CCS)*, 2004.

Axel Stutz and Peter Klingebiel. *Übersicht über die C Standard-Bibliothek*. http://www2.hs-fulda.de/~klingebiel/c-stdlib/index.htm, 1999.