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:	31. August 2013
Betreuer:	Dr.-Ing. Mario Kubek

Inhaltsverzeichnis

1 Einleitung

Jede Entwicklerin und jeder Entwickler möchte neben der Funktionalität auch die Sicherheit ihrer/seiner Software garantieren.

Um die Aspekte der sicheren Programmierung zu betrachten, muss man sich im Vorfeld über die grundlegenden Risiken bei der Entwicklung von Softwarelösungen informieren. Im Kontext der Informationstechnik haben sich drei primäre Schwachpunkte herauskristallisiert:

- **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 Softwareentwicklung

Die Softwareentwicklung selbst ist so alt wie das Computerzeitalter. Mit den ersten Computern wurden auch die ersten Programme entwickelt, diese waren noch lange nicht so umfangreich wie die heutigen, jedoch wurden auch schon in den frühen Anfängen einfache mathematische Operationen durchgeführt.

Aufgrund der immer leistungsfähiger werdenden Hardware werden Softwareentwicklungen, die einst als sicher galten heute als unsicher eingestuft. Allein die stetig verbesserte Hardware schafft es, mehr Operationen in kürzerer Zeit zu berechnen. Was zur Folge hat, dass auch komplexe Algorithmen in immer kürzerer Zeit gelöst werden. Was eigentlich als technischer Fortschritt und Verbesserung angesehen werden kann, führt aber auch zu neuen Problemen. Um vertrauliche Informationen unleserlich zu machen, reichte es Julius Cäsar, ein Stück Papyrus um einen zylindrischen Holzstab [Wika] zu wickeln. Im zweiten Weltkrieg wurde die Chiffriermaschine Enigma [Wikc] lange Zeit eingesetzt, um eine sichere Kommunikation zu gewährleisten. Erst in den 1970er-Jahren gab es einen Wechsel von symmetrischer auf asymmetrische Verschlüsselung, wobei Sender und Empfänger nicht mehr auf den gleichen Schlüssel angewiesen waren. Seitdem wird der als sicher geltende Schlüssel gefühlt jährlich verdoppelt. Dazu kommen immer neuere Verschlüsselungsalgorithmen, die auf mathematischen Grundlagen basieren. So wird die 1993 entwickelte Blowfish-Verschlüsselung [Sch] heute bevorzugt verwendet und ersetzt damit den SHA-Algorithmus [Nat] In PHP5.5 wurde Blowfish als erste Passwortverschlüsselung in die neue Hash-Funktion aufgenommen.

Es findet ein ständiger Wettkampf im Hardware-Bereich statt und auch die immer komplexer werdenden Software-Anwendungen konkurrieren. Ein Programm, das im Jahr 2000 nach den damaligen "Sicherheitsrichtlininen" entwickelt wurde, gilt heute meist als risikobehaftet. So entstehen neue Herausforderungen, die es zu meistern gilt. Softwareentwicklungen müssen oft in einem zeitlich vorgeschriebenen Rahmen fertiggestellt werden. Daher wird oft geschaut, welche Lebensdauer die Software hat und für welchen Einsatzzweck sie bestimmt ist. Bei einigen Projekten wird dann ein Risikomanagement durchgeführt, das den Schaden schon vor der eigentlichen Entwicklung abschätzen soll.

2.1 Programmiersprachen

Im 21. Jahrhundert bewegt man sich, in mehrerlei Hinsicht, in einem multikulturellen Umfeld. So gab es schon in den Anfängen des Computerzeitalters gleich Dutzende von Programmiersprachen, die aber jeweils an spezielle Bedürfnisse angepasst waren. Mit Basic und C entstanden die ersten universellen Programmiersprachen, die die Entwicklung rasch vorantrieben und heute noch eingesetzt werden. Dann kam in den 1990er-Jahren das Internet, welches neue Anforderungen an die Programmiersprachen stellte und durch die rasche Entwicklung auch viele Programmiersprachen sterben ließ, da sie mit dem Fortschritt nicht mithalten konnten. Das war die Geburtsstunde der objektorientierten und skriptbasierten Programmiersprachen. Durch die Globalisierung und die damit einhergehenden schnelleren Kommunikationswege sowie die Verwendung von Englisch als Weltsprache kann man sich heute auch über große Distanzen austauschen und Wissen teilen. Das aktuell verbreitetste Programmierparadigma ist das Objektorientierte. Dabei wird versucht, die Daten als Objekt zu betrachten und so die Komplexität zu vereinfachen, indem Redundanzen nutzbar gemacht werden.

Auch heute gibt es noch zahlreiche Programmiersprachen, die ihre Anhänger haben, die sich in sogenannte Communitys zusammenschließen, um die Programmiersprachen weiter zu entwickeln und das Wissen auszutauschen.

2.2 Frameworks

Ein Framework stellt kein fertiges Programm dar, vielmehr dient es Entwicklerinnen und Entwicklern als Grundgerüst in der objektorientierten Programmierung. Es bringt dabei für gewöhnlich neben der Struktur auch die Anwendungsarchitektur mit. Insbesondere wird aber der Kontrollfluss der Anwendung und die Schnittstellen darüber in architektonischen Mustern bereitgestellt. Heute gibt es zahlreiche Frameworks zu fast jeder Programmiersprache und jedem Anwendungsfall, daher gibt es keine allgemeingültige Definition.

Framework Typen

- Application Frameworks
  bilden das Programmiergerüst für Anwendungen.
- Domain Frameworks
  bilden das Programmiergerüst für Problembereich.
- Class Frameworks
  bilden über Klassen und Methoden die Abstraktionsebenen für ein breites Anwendungsfeld ab.
- Komponenten-Frameworks
  bilden die Basis für Software-Komponenten, indem sie die objektorientierte Ebenen abstrahieren.
- Coordination-Frameworks
  bilden die Basis für Geräte-Interaktion.
- Tests Frameworks
  dienen als Programmiergerüst für (automatisierte) Softwaretests.
- Webframeworks
  sind speziell für Webanwendungen ausgelegt.

2.3 Entwicklungsumgebungen

Eine Entwicklungsumgebung oder kurz IDE (Integrierte Entwicklungsumgebung) ist eine Sammlung verschiedener Anwendungsprogramme. Damit kann man Medienbrüche in der Softwareentwicklung vermeiden, was gerade bei größeren Entwicklungen zu Effizienz in der Arbeit führt. Es gibt für fast jede Programmiersprache eine große Vielzahl von Entwicklungsumgebungen. Sowohl Open-Source als auch proprietäre Entwicklungsumgebungen. So ist die wohl bekannteste und verbreitetste Entwicklungsumgebung für Softwareentwicklungen die in und ursprünglich für Java geschriebene Umgebung Eclipse[1]. Die Entwicklungsumgebung erfreute sich nicht zuletzt durch den 2001 von IBM freigebenden Quellcode großer Beliebtheit. Auch die Weiterentwicklung ist zur Zeit gesichert, da IBM weiterhin Entwickler für die Arbeit an der Software finanziert. Durch Erweiterungen werden weitere Programmiersprachen von Eclipse unterstützt. Dadurch ist jeder, der mit Softwareentwicklung zu tun hat schon mal mit Eclipse in Berührung gekommen. Dieses ist aber lange nicht die einzige Erfolgsgeschichte auf dem Markt der Entwicklungsumgebungen. So gibt es zahlreiche Produkte, die um die Gunst der Softwareentwickler buhlen.

[1] `http://www.eclipse.org`

2.4 Versionskontrollsystem

Bei einem Versionskontrollsystem werden Änderungen dokumentiert und festgehalten. So lassen sich Veränderungen an einem Dokument auch über viele folgende Änderungen nachschlagen und rückgängig machen bzw. nachverfolgen, warum eine Änderung vorgenommen wurde. Es gibt verschiedene Funktionsweisen:

- Lokale Versionsverwaltung
- Zentrale Versionsverwaltung
- Verteilte Versionsverwaltung.

Die wohl bekanntesten Vertreter solcher Software sind Subversion (SVN)[2] und git[3], die zudem noch unter freien Lizenzen stehen. Dabei zählt SVN zu den zentralen Versionsverwaltung und git zur verteilten Versionsverwaltung. SVN ist wohl die bekannteste Versionsverwaltung und mit Version 1.1 schaffte sie bereits 2004 den Durchbruch, als Änderungen nicht mehr in der Datenbank, sondern im Dateisystem abgelegt werden konnten. git ist erst 2005 durch Linus Torvalds entwickelt worden, nachdem die Lizenz für das bisher von ihm eingesetzte Versionskontrollsystem geändert wurde. Der große Durchbruch gelang aber erst durch github, einer Plattform auf der man seine Projekte teilen und mit anderen bearbeiten kann.

2.5 Ticketsystem

In einem Ticketsystem werden Fehler oder aber neue Funktionswünsche durch Mitglieder/Kunden/... gemeldet, die dann von einem Entwickler oder einer Entwicklerin betrachtet, nachvollzogen und klassifiziert werden können. Voraussetzung für eine Fehlerbehebung ist eine genaue Fehlerbeschreibung und ein reproduzierbarer Weg, um den Fehler zu erzeugen/hervorzurufen. Durch verschiedene Status kann dann dem Nutzer mitgeteilt werden, wann oder in welcher Version der Fehler behoben ist. In Open-Source-Projekten sind solche Ticketsysteme zudem offen, sodass andere die Problemstellung einsehen können und ggf. einen besseren oder anderen Lösungsansatz vorab diskutieren können.

[2] `http://subversion.apache.org`
[3] `http://git-scm.com`

2.6 Softwarearten

Mit Softwarearten ist in diesem Kapitel die Art der Entstehung/Entwicklung der Software gemeint. Weithin gibt es zwei Arten der Softwarentwicklung:

- kommerzielle Softwareentwicklung
- freie Softwareentwicklung.

Mit "frei" ist in diesem Fall Software gemeint, wo der Quellcode der Software frei von jedem Nutzer eingesehen werden kann. Sehr häufig wird "frei" mit Open Source gleichgesetzt, was aber de facto nicht der Fall ist.

Kommerzielle Softwareentwicklung

Bei kommerzieller Softwareentwicklung handelt es sich häufig um Auftragsprojekte durch einen Kunden bzw. Eigenentwicklungen, um ein Softwareprodukt auf dem Markt zu platzieren. Bei kommerziellen Softwareentwicklungen kennt oft nur ein erlesener Kreis an Personen den Quellcode und ist auch für dessen Qualität verantwortlich. Zwar wird hier versucht, hochwertige Arbeit abzuliefern, doch leider ist das nicht immer der Fall. Teilweise wird der Code an externe Firmen zur Sicherheitsprüfung herausgegeben, da die Gewinnoptimierung stets eine große Rolle spielt, wird leider aber genauso oft auf solche Tests verzichtet und darauf vertraut, dass die eigenen Entwickler/Entwicklerinnen den Code geprüft haben und an alle Sicherheitsmerkmale gedacht haben.

Freie Softwareentwicklung

Quellcodefreie Software kann jeder selbst prüfen - vorausgesetzt, er/sie verfügt über das erforderliche Wissen - oder aber jemanden mit der Prüfung beauftragen. In Open-Source-Communitys passiert dies häufig automatisch. Da hier mehrere Entwickler/-Entwicklerinnen an einem Projekt arbeiten, wird ein Versionskontrollsystem eingesetzt. Hier kann jede Veränderung nachverfolgt werden. Durch die Verknüpfung mit einem Ticketsystem sind Nutzer in der Lage, Wünsche und Fehler zu melden, die dann behoben oder aufgenommen werden können. Anschließend schauen mehrere unabhängige Entwickler/Entwicklerinnen über den neu hinzugefügten Code und geben ihr Urteil darüber ab. Durch diese Art der Entwicklung findet gleichzeitig eine Prüfung der Qualität und Sicherheit der Software statt.

2.7 Softwareentwicklung durch *Security by Design*

Der Frauenhofer Verlag hat auch in diesem Jahr eine aktualisierte Ausgabe seines Trend- und Strategieberichts zur "Entwicklung sicherer Software durch Security by Design" herausgegeben. Dabei geht es darum, bei der Softwareentwicklung bestimmte Sicherheitsaspekte wie Lebenszyklus und die Integration von Softwarekomponenten anderer Hersteller frühzeitig zu analysieren. Dieser Trend- und Strategiebericht soll nicht als Richtlinie verstanden werden, eher soll er als Leitfaden dienen, um sich mit der Problemstellung zu befassen. Das 65-seitige Werk geht dabei auf die Bedeutung von "Security By Design" ein und zeigt, wie durch Automatisierung und Reduktion menschlicher Fehler Code sicherer gemacht werden kann. Weiterhin bietet es einen Einblick in Probleme, die Legacy-Software mit sich bringt und zeigt an verteilter Entwicklung und Interaktion, dass kaum noch ein Softwareprojekt von einem einzigen Entwicklerteam realisiert wird und wo hier die Gefahren liegen. Den Abschluss bildet ein Blick in die Zukunft: Man sei schon einen Schritt in die richtige Richtung gegangen, doch es stünde noch ein weiter Weg bevor, bis es keine Meldungen mehr über Sicherheitslücken und Angriffe gäbe.

3 Schwachstellen

In den folgenden Kapiteln werden die Top 4 der am häufigsten anzutreffenden Schwachstellen aus Sicht des Common Weakness Enumeration [Chr] Projekts 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 werden mangelnde Maskierung oder eine fehlende Überprüfung von Metazeichen in Benutzereingaben durch Angreifer ausgenutzt.

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

Typische Injection- und Cross-Site-Scripting-Schwachstellen treten vermehrt bei webbasierten Anwendungen auf.

Bei Buffer-Overflow-Schwachstellen werden die durch einen Anwender erzeugten Datenmengen in einen reservierten Speicherbereich geschrieben. Der reservierte Bereich ist dabei zu klein für die erzeugte Datenmenge. Die Daten, die darüber hinaus in dem Bereich Platz bräuchten, überschreiben dann. Buffer-Overflow-Schwachstellen, in Binäranwendungen oder Runtime-basierenden Anwendungen, werden unter anderem durch das Internet ausgenutzt.

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

3.1 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) unzureichend prüfen.

3.1.1 SQL-Injection-Schwachstellen

Die SQL-Injection-Technik wurde 1998 im Phrack Magazin [RFP98] einem breiten Publikum vorstellt. Mithilfe einer SQL-Injection-Schwachstelle kann ein Angreifer über einen serverseitig unzureichend validierten Eingabeparameter beliebige Datenbankbefehle an eine der Webanwendung nachgelagerte 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 einem 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 potenzielle SQL-Injection-Schwachstellen innerhalb einer Anwendung zu finden, existieren eine Vielzahl von kommerziellen Anwendungen (z.B. Havij[4]) als auch Open-Source-Tools (z.B. Sqlmap[5]).

[4] `http://www.itsecteam.com/products/havij-v116-advanced-sql-injection/`
[5] `http://sqlmap.org/`

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 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 darunterliegen Betriebssystem, der eingesetzten Datenbank sowie Details zur der 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. provozierte Veränderungen der Laufzeiten durch einen Angreifer oder kleinste 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-Sheet [6] verwiesen.

3.1.2 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.

3.2 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 Semikola, bereits im Entwicklungsprozess regelmäßig eine statische Quellcode-Analysen durchzuführen.

3.2.1 Maßnahmen zur Behebung SQL-Injection-Schwachstellen

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

[6] `http://ferruh.mavituna.com/sql-injection-cheatsheet-oku`

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 Beispielquelltext 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 API[7] zurückzugreifen.

Verwendung von Prepared Statements

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

```
1 prepStatement 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.

3.2.2 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 es hierbei keine Datenbank-Metazeichen sondern die Shell-Metazeichen.

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

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.

3.3 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 Arten unterscheiden:

3.3.1 Persistentes Cross-Site-Scripting

Bei persistentem Cross-Site-Scripting wird der applikationsfremde JavaScript-Code dauerhaft (z.B. in ein 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.

3.3.2 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 Beispielcode 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>`

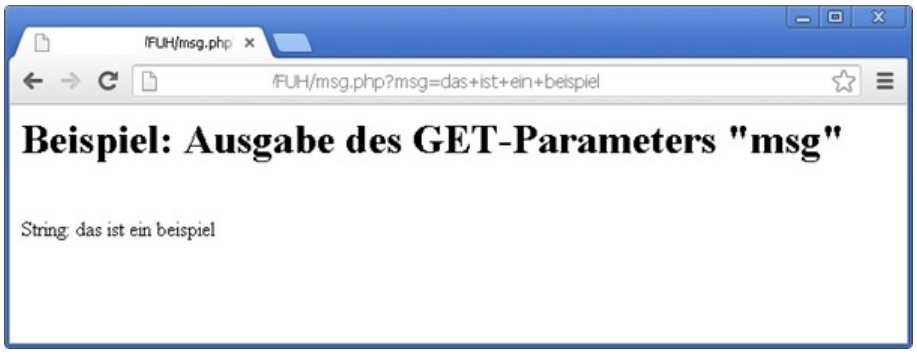

Abbildung 1: Ausgabe des Beispiel-Strings

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
6     <script>alert('XSS')</script>
7   </body>
8 </html>
```

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

In diesem Abschnitt werden mögliche Maßnahmen zur Behebung von Cross-Site-Scripting-Schwachstellen vorgestellt.

3.4.1 Encoding von HTML-Metazeichen

Durch HTML-Encoding [Wikb] 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>
2   <body>
3     <h1>Beispiel: Ausgabe des GET-Parameters "msg"</h1>
4     <br>
5     String: das ist ein beispiel
6     &lt;script&gt;alert(&#39;XSS&#39;)&lt;/script&gt;
7   </body>
8 </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.

3.4.2 HTML-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.

Das Vorgehen bei der Verwendung von Tag-Filtern ist ähnlich der empfohlenen Maßnahmen zur Behebung von Injection-Schwachstellen (siehe Abschnitt 3.2.1), 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')</sytle>
```

An dieser Stelle sei darauf hingewiesen, dass bei der Verwendung von Tag-Filtern (bzw. Blacklist-Filtern) nach Möglichkeit auf eine eigenständige Implementierung verzichtet werden 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-Sheet[8] zu finden.

3.4.3 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 (wie z.B. ihre Darstellung) beinhalten zu müssen.

Aus diesem Grund kann man bereits bei der Entwicklung von Anwendungen nach dem Model-View-Controler (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.

[8] `http://html5sec.org/`

3.5 Buffer-Overflow-Schwachstellen

Buffer-Overflow-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 potenziell für Buffer-Schwachstellen anfällig ist, gilt die Programmiersprache C. Diese 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 }
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 Debugger[9] (GDB-Kurzreferenz [Hal]). Für einen ersten Überblick werden die drei Funktionen disassembliert.

`main()`-Funktion

```
1  [user@audit exploit]$ gdb -q poc
2  Reading symbols from /home/user/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
15    0x000000000040060b <+59>:    retq
16 End of assembler dump.
```

[9] http://www.gnu.org/software/gdb

`pwnd()`-Funktion

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

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-Codes 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 Haltepunkt an der Speicheradresse `0x40060b` gesetzt (siehe letzte Zeile der disassemblierten `go()` -Funktion), um die Überlegungen verifizieren zu können.

```
gdb) break *0x40060b
Breakpoint 6 at 0x40060b: file poc.c, line 13.
(gdb) run AAAAAAAA

String: AAAAAAAA

Breakpoint 6, 0x000000000040060b in go
(input=0x7fffffffecdb "AAAAAAAA") at poc.c:13}

(gdb) p &data
$20 = (char (*)[64]) 0x7fffffffe950
(gdb) x/12xg 0x7fffffffe950
0x7fffffffe950: 0x4141414141414141      0x00007ffff7ff9100
0x7fffffffe960: 0x00007ffff7ffe190      0x0000000000f0b2ff
0x7fffffffe970: 0x0000000000000001      0x000000000040069d
0x7fffffffe980: 0x00007fffffffe9be      0x0000000000000000
0x7fffffffe990: 0x00007fffffffe9b0      0x000000000040064c
0x7fffffffe9a0: 0x00007fffffffea98      0x0000000200000000
(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:

```
0x7fffffffe990: 0x00007fffffffe9b0      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 den Startwert der `go()`-Funktion festzustellen. Im Anschluss werden die 12-mal 8 Byte großen Speicheradressen ausgegeben und 2 Byte der Rücksprungadresse `0x40064c` modifiziert. Danach wird das Programm weiter ausgeführt und springt in die `pwnd()`-Funktion.

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

End of assembler dump.
(gdb) x/12xg 0x7fffffffe950
0x7fffffffe950: 0x4141414141414141      0x00007ffff7ff9100
0x7fffffffe960: 0x00007ffff7ffe190      0x0000000000f0b2ff
0x7fffffffe970: 0x0000000000000001      0x000000000040069d
0x7fffffffe980: 0x00007fffffffe9be      0x0000000000000000
0x7fffffffe990: 0x00007fffffffe9b0      0x000000000040064c
0x7fffffffe9a0: 0x00007ffffffffea98      0x0000000200000000
(gdb) set {char}0x7fffffffe998 = 0x0c
(gdb) set {char}0x7fffffffe999 = 0x06
(gdb) x/12xg 0x7fffffffe950
0x7fffffffe950: 0x4141414141414141      0x00007ffff7ff9100
0x7fffffffe960: 0x00007ffff7ffe190      0x0000000000f0b2ff
0x7fffffffe970: 0x0000000000000001      0x000000000040069d
0x7fffffffe980: 0x00007fffffffe9be      0x0000000000000000
0x7fffffffe990: 0x00007fffffffe9b0      0x000000000040060c
0x7fffffffe9a0: 0x00007ffffffffea98      0x0000000200000000
(gdb) c
Continuing.
PWND!
[Inferior 1 (process 1190) exited normally]
(gdb)
```

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

```
(gdb) run ‘perl -e ’print "A"x72 . "\x0c\x06\x40"’‘
String: AAAAAAAAAAAAAAAAAAAAAAAA ... AAAAAA@
PWND!
[Inferior 1 (process 1624) exited normally]
(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.x (vgl. [Kle]) verwendet, ist es möglich, dass dieses Beispiel nicht funktioniert!

3.6 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 beschriebe Quellcodebeispiel.

3.6.1 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 24731-1 [Inta]
- TR 24731-2 [Intb]

Die beiden Dokumente werden in Foren und Fachkreisen kontrovers diskutiert, dennoch ist das Dokument TR 24731-1 in die Entwicklung der Microsoft-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 hergenommen 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.

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

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

Effiziente Quellcodescanner reduzieren die Rate der gefundenen "False-Postives" beispielsweise durch die Verwendung interner Datenbanken, die potenziell unsichere Quellcodefragmente mit den 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-Postives" noch "False-Negatives" produzieren. Dabei sollten "False Negatives" nach Möglichkeit nie vorkommen, da diese im Gegensatz zu "False Positives" zu Sicherheitsproblemen führen können.

Nachfolgend soll der frei verfügbare Quellcodescanner RATS[10] (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.

```
[user@audit exploit]$ rats -i --resultsonly  *.c
poc.c:8: High: fixed size local buffer
Extra care should be taken to ensure that character arrays that
are allocated on the stack are used safely.  They are prime
targets for buffer overflow attacks.

poc.c:9: High: strcpy
Check to be sure that argument 2 passed to this function
call will not copy 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 potenzielle 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.

[10] `https://www.fortify.com/downloads2/public/rats-2.3-2.tar.gz`

3.6.2 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.

Grundlegende Überprüfungen mit dem Compiler

Ein Compiler verfügt bereits meist über grundlegende Techniken, um mindestens eine lexikalische und zusätzlich 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 Compilers offensichtlicher zu gestalten, wird die `main()`-Funktion durch die folgenden 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 [user@audit exploit]$ gcc -Wall poc.c -o poc
2 poc.c: In function main:
3 poc.c:29:2: warning: gets is deprecated
4 (declared at /usr/include/stdio.h:638) [-Wdeprecated-declarations]
5   gets(array);
6 poc.c:31:1: warning: control reaches end of non-void
7 function [-Wreturn-type]
8 }
```

Der GNU C Compiler macht den Entwickler darauf aufmerksam, dass er zum einen die unsichere und veraltete `gets()`-Funktion verwendet und zum anderen, 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 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

Splint[11] ist ein statischer Quellcodescanner, der in der Lage ist, eine weitaus detaillierte semantische Analyse als der GNU C Compiler vorzunehmen.

Splint ist imstande, sogenannte LINT [Wikd] durchzuführen. Zu diesen Überprüfungen gehören beispielsweise die Suche nach Endlosschleifen, falschen Deklarationen oder ignorierten Rückgabewerten.

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

```
1  [user@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
10     (input @ poc.c:9:14)
11      derived from strcpy precondition: requires maxSet
12      (<parameter 1>) >=
13     maxRead(<parameter 2>)
14   A memory write may write to an address beyond the
15   allocated buffer. (Use -boundswrite to inhibit warning)
16 poc.c: (in function main)
17 poc.c:25:2: Return value (type int) ignored: go(argv[1])
18   Result returned by function call is not used. If this is
19   intended, can cast result to (void) to eliminate message.
20   (Use -retvalint to inhibit warning)
21   [...]
```

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.

3.6.3 Programmanalyse zur Programmlaufzeit

Neben den beschreiben Möglichkeiten zur statischen Quellcodeanalyse besteht darüber hinaus die Möglichkeit, ein Programm zur Laufzeit auf Schwachstellen zu überwachen

[11] `http://www.splint.org/`

bzw. zu untersuchen, um gezielt Overflow-Schwachstellen, die erst zur Programmlaufzeit entstehen ausfindig zu machen.

Ein mögliches Beispiel für eine potenzielle Overflow-Schwachstelle, die erst während der Laufzeit eines Programmes auftreten kann, ist der Aufruf der C-Funktion `malloc()`, die 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 Fachexpertise eines Entwicklers ab. Für die 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 Programmlaufzeit dar, die dynamischen Fehleranalyse durchführt. Dabei wird ein zu analysierendes Programm nicht auf der nativen Host-CPU, sondern innerhalb einer virtuellen Umgebung ausgeführt. Vagrind 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 2 `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 vorhergehen Beispielen bereits bekannt, zu einem Buffer-Overflow:

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

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

```
[...]
==10902== Invalid write of size 1
==10902== at 0x4C2CBB2: __GI_strcpy
          (in /usr/lib/valgrind/vgpreload_memcheck-
          amd64-linux.so)
==10902== by 0x40069A: go (poc2.c:14)
==10902== by 0x400703: main (poc2.c:31)
==10902== Address 0x51e00e4 is 20 bytes after a block
          of size 64 alloc'd
==10902== at 0x4C2C04B: malloc
          (in /usr/lib/valgrind/vgpreload_memcheck-
          amd64-linux.so)
[...]
==10902== HEAP SUMMARY:
==10902== in use at exit: 8 bytes in 1 blocks
==10902== total heap usage: 2 allocs, 1 frees, 72 bytes
          allocated
==10902==
==10902== 8 bytes in 1 blocks are definitely lost in loss
                  record 1 of 1
==10902== at 0x4C2C04B: malloc
          (in /usr/lib/valgrind/vgpreload_memcheck-
          amd64-linux.so)
==10902== by 0x400675: go (poc2.c:9)
==10902== by 0x400703: main (poc2.c:31)
[...]
```

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

4 Umgang mit Sicherheitslücken

Wie geht man mit dem Bekanntwerden von Sicherheitslücken in Software um?

Bisher wurde beschrieben, welche Arten von Sicherheitslücken es gibt und wie man Code aktuell wirkungsvoll davor schützt. Desweiteren wurden aktuelle Lösungswege gezeigt, um sie zu vermeiden.

In den meisten Fällen kann man bei Bekanntwerden von Sicherheitslücken diese beseitigen und ein Update bereitstellen. Was aber passiert mit Code, der eine Sicherheitslücke hat, die kritisch ist, aber nicht zeitnah aktualisiert werden kann?

Aktuell gibt es zahlreiche solcher bekannten Sicherheitslücken. Der wohl populärste Hacker im Dienste des "Guten" war der Sicherheitsexperte Barnaby Jack, der zu den sogenannten *white hats* zählte. Er starb vergangene Woche an bislang ungeklärter Ursache im Alter von 35 Jahren [SPIa]. Kommende Woche wollte er auf einer Konferenz über die Sicherheit von Herzschrittmachern sprechen. In den Medien sorgt aktuell noch ein weiterer Fall für Schlagzeilen. Dabei geht es um geknackte Wegfahrsperrcodes [SPIb] bei den Luxusmarken der Volkswagen Gruppe (VW). Hier wurde von VW kurzfristig eine Verfügung veranlasst, die den Wissenschaftlern eine Veröffentlichung untersagt.

Beide Beispielfälle zeigen Sicherheitslücken in Systemen, die nur schwer aktualisiert werden können. Die Softwareentwicklung bei einem Herzschrittmacher benötigt viele Tests und einen Langzeittest um sicherzustellen, dass durch ein Update nicht eine Fehlfunktion zum Herzstillstand des Patienten führt. Bei VW muss jetzt jeder betroffene Wagen in eine Werkstatt, um mit neuer Software ausgestattet zu werden, da man nicht einfach das Update über eine Datenleitung einspielen kann. Zwar gibt es immer mehr Systeme, die eine Anbindung an das Internet haben [N-T] doch bringt dieses nicht nur Vorteile, da gerade die globale Erreichbarkeit neue Sicherheitsprobleme hervorruft. Für Hersteller sind nicht nur die Fehlerbehebung und die Tests mit Kosten verbunden, sondern auch eine eventuelle Rufschädigung. So versuchen gerade große Konzerne, die Hacker ihrer Systeme als Sicherheitsexperten zu gewinnen.

5 Fazit

Durch die globalisierte Vernetzung nimmt die Sicherheit in der Softwareentwicklung einen immer größeren Stellenwert ein. In den vergangenen Jahren ist die Komplexität mit jeder neu entstandenen Software gestiegen.

Obwohl Sicherheitslücken wie SQL-Injection und Cross-Site Scripting (XSS), wie in dieser Hausarbeit beschrieben, schon seit Jahren bekannt sind und effektiv bekämpft werden können, liest man noch heute immer wieder über diese Schwachstellen. Inzwischen gibt es eigene Firmen, die bezahlt werden, um Sicherheitslücken zu finden und damit das Softwareprodukt sicherer zu machen. Doch durch die stetige Weiterentwicklung und neuen Möglichkeiten entstehen wiederum neue Sicherheitslücken. Auch wenn eine Schwachstelle behoben wurde bedeutet dieses nicht gleich, dass alle betroffenen Systeme von der Behebung profitieren. Ein bekanntes Beispiel sind Content Managment Systeme, die von Administratoren oft nur schlecht gewartet werden. So gibt es Schätzungen, das heute nur ca. 50% aller Wordpress-Installationen [Mü] aktuell sind. Sicherheitsexperten gehen sogar aufgrund der stetigen Entwicklung von Hardware und Softwarekomplexitäten davon aus, dass nach einem Jahr ohne Aktualisierungen eine Software hochgradig anfällig für Schwachstellen ist. Auch der gerne benutzte Punkt "Sicherheit durch Unbekanntheit" kann nicht genutzt werden. Gerade in Zeiten von Spionage untersuchen Hacker Webseiten auf Auffälligkeiten.

Daher müssen sich nicht nur Entwickler mit der ständigen Weiterentwicklung auseinandersetzen, sondern auch die Nutzer dieser Software selbst. Auch wenn gleich es inzwischen viele Hilfsprogramme gibt, die nachschauen ob es eine neuere Version gibt, so muss schlussendlich doch auch der Nutzer tätig werden, um das Update durchzuführen.

Literaturverzeichnis

[Chr] Christey, Steve: *2011 CWE/SANS Top 25 Most Dangerous Software Errors*. http://cwe.mitre.org/top25/archive/2011/2011_cwe_sans_top25.pdf

[Hal] Hall, Brian: *Beej's Quick Guide to GDB*. http://beej.us/guide/bggdb/

[Inta] International Organization for Standardization (ISO): *ISO/IEC TR 24731-1*. http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1225.pdf

[Intb] International Organization for Standardization (ISO): *ISO/IEC WDTR 24731-2*. http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1337.pdf

[Kle] Klein, Tobias: *GCC 3.x Stack Layout - Auswirkungen auf Stack-basierte Exploit-Techniken*. http://www.trapkit.de/papers/gcc_stack_layout_v1_20030830.pdf

[Mü] Müller, Sergej: *Verteilung der WordPress-Versionen im deutschsprachigen Raum*. http://playground.ebiene.de/wordpress-versionen-verteilung

[N-T] N-TV: *Das Internet der Dinge: Wenn nicht nur der Kühlschrank online ist*. http://www.n-tv.de/ratgeber/Sendungen/Wenn-nicht-nur-der-Kuehlschrank-online-ist-article10431986.html

[Nat] National Institute of Standards and Technology (NIST): *FIPS 180-2, Secure Hash Standard*. http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf

[RFP98] RFP: NT Web Technology Vulnerabilities. In: *Phrack Magazine* Volume 8, Issue 54 (1998), Dezember. http://www.phrack.org/issues.html?issue=54&id=8#article

[Sch] Schneier, Bruce: *The Blowfish Encryption Algorithm*. http://www.schneier.com/blowfish.html

[SPIa] SPIEGEL ONLINE: *Hacker Barnaby Jack in San Francisco gestorben*. http://www.spiegel.de/netzwelt/web/hacker-barnaby-jack-in-san-francisco-gestorben-a-913380.html

[SPIb] SPIEGEL ONLINE: *Volkswagen erwirkt Verfügung gegen akademische Codeknacker*. http://www.spiegel.de/auto/aktuell/volkswagen-erwirkt-verfuegung-gegen-akademische-codeknacker-a-913462.html

[Wika] Wikipedia: *Caesar-Verschlüsselung*. http://de.wikipedia.org/wiki/Caesar-Verschlüsselung

[Wikb] Wikipedia: *Character encodings in HTML*. http://en.wikipedia.org/wiki/Character_encodings_in_HTML

[Wikc] Wikipedia: *Enigma*. http://de.wikipedia.org/wiki/Enigma_(Maschine)

[Wikd] Wikipedia: *Lint*. http://en.wikipedia.org/wiki/Lint_(software)

Abbildungsverzeichnis

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 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.

$Unterschrift$: $Ort, Datum$: