

Towards Automated Malicious Code Detection and Removal on the Web

Dabirsiaghi, Arshan

Open Web Application Security Project
Aspect Security, Inc., 2007

Abstract

The most common vulnerability in web applications is Cross-Site Scripting, popularly abbreviated as “XSS”. Cross-site scripting results when a threat agent (an attacker or unwitting victim) supplies malicious content that is delivered by an unwitting server to a victim’s HTML-enabled agent (such as a web browser or email client) where it is interpreted as executable code instead of data [1].

Defusing XSS attacks from the web application’s perspective is easy. Any “data” to be delivered to a user’s browser can be *HTML entity encoded*. This encoding will render any XSS attacks inert because the control characters used in the attack will be translated into harmless display characters by the victim’s user agent [2]. Although the attack is easy to defend against in practice, the lack of awareness surrounding XSS and the growing catalogue of high-profile bugs [3] give the problem a mythical standing in the security community.

Incidentally, many sites today would enjoy the ability to allow users to provide their own HTML in order to customize their page layout and themes. User generated content in general appears to be a popular and growing trend. Unfortunately, because of the concerns regarding XSS, it is generally thought of as ‘too dangerous’ to allow users who are not strongly authenticated to provide any HTML at all [4]. Sites like MySpace that have been brave enough to provide this functionality have no standardized, proven solution to validate or clean user provided code to prevent the insertion of possible malicious code while retaining all formatting code. Not coincidentally, those sites have been the targets of complex and successful attacks [5].

Introduction

There is a well-established need for web applications to provide users with the ability to format their profile or postings using Hypertext Markup Language / Cascading Style Sheet (HTML/CSS). To attain that functionality, developers must allow users to provide their own source code directly or give the user an intermediate language with which the user can work.

Intermediate Languages

An intermediate language provides a simple façade for the target language. In this type of situation, the user is provided a markup language that provides a largely reduced subset of HTML/CSS functionality. An example intermediate language code for rendering green text can be seen in Figure 1.

```
[color=green]
This is green text.
[/color]
```

Figure 1.1

After translation the code in Figure 1.1 would be rendered to the user’s browser in the target language, HTML/CSS as seen in Figure 1.2.

```
<font color="green">
This is green text.
</font>
```

Figure 1.2

This is a safe approach in general because it does not allow users to specify arbitrary target language code which can be obfuscated and disguised using various encoding and fragmenting techniques. By providing an intermediate language and interpreting it in a top-down fashion the application can only render the subset of HTML functionality that they wish to interpret. This type of intermediate sandbox language was made popular by Wikipedia [6] but has been present on other sites for some time.

There are two practical problems with this approach. The first is that the user will be fairly limited in what formatting code they can provide since whatever façade instruction set provided by the web application is unlikely to ever be as complete as the HTML/CSS specifications [7].

The second major problem is one of energy misuse. An enormous amount of energy could be invested in growing a façade language and its accompanying translation logic which will result in a complex language that more and more resembles the capability (although not necessarily the syntax) of HTML/CSS. Also, the translation logic must be kept up to date so that it produces a syntax that is compliant with the current HTML/CSS specifications.

A fraction of the effort required to implement and maintain such a complex intermediate language could be used to

help build and refine a high assurance validation mechanism for HTML/CSS.

Validating / Sanitizing

The other option when providing formatting capability is to allow users to input HTML/CSS directly. If users' input cannot be trusted, it is imperative that the application be able to detect and remove any malicious code.

User Cooperation

Much like other contemporary information security mechanisms, the standard counter-measures for XSS detection do not engage the user in a cooperative manner. In the spirit of engaging with the user to create a positive experience we must acknowledge the fact that the application may unintentionally be provided executable code that is not malicious.

Given that fact, it should then be the responsibility of the application to clean the code as much as possible while still retaining all the formatting elements of the input.

The flow of information between the user and current validation mechanisms is virtually one-way. Very little if any feedback is provided to the user after validation takes place. This is largely because possible attackers and regular users are treated the same way (as possible attackers) in order to reduce the risk of leaking information about the validation mechanism.

However, the workings of a high assurance security mechanism should be available for public consumption without increasing the risk of vulnerability in accordance with Kerchoff's principle [8].

Malicious Code

Often the term "malicious code" (in the context of the web) is interpreted as simply JavaScript. Although JavaScript provides attackers with many interesting exploitation capabilities, it should also be noted how dangerous the injection of nefarious HTML/CSS can be for a web application.

For the purpose of argument, let's say that a user could only inject HTML/CSS into a portion of a web page. What could they do?

- Invoke JavaScript through the use of URL lookups (*list-style-image*, *background-image*, etc).
- Specify a *div* with a *z-index* of 1 that overlays the existing page, making the original content essentially invisible, affecting attack vectors for phishing
- A *base* tag that made the base URL for all links in the page point to a malicious site, effectively hijacking all the out-of-page resources in that HTML
- A *meta* tag that redirects users to a phishing site with the same look and feel as the victim's site

All of these things should concern a web application that is allowing users to provide their own HTML/CSS.

Related Work

There are many free tools that relate substantially to this work. Their purposes fall into two domains; HTML sanitization and XSS filters.

Because of the ease of web authoring and the fact that HTML is interpreted rather than compiled, there are no assurances that the HTML user agents try to process will be syntactically correct.

Although it is possible to use a user agent's display engine to validate that HTML code is written correctly, there is no assurance that it will actually render correctly in any other user agents because broken code can result in a correct view in one user-agent and an incorrect view in another, due to the fact that user-agent parsing engines are written differently.

Because of these discrepancies, there was a need for HTML sanitization tools which "clean up" any "broken" HTML. A number of tools and libraries were created by the community in response to this need. The communal evolution of the tools reached impressive heights [9] [10]. Unbalanced tags would be balanced (by addition or subtraction) and any dangling control characters would be HTML-entity encoded.

Though these sanitization tools "clean" up HTML they do not validate or sanitize the input in terms of XSS. They validate the well-formedness and legality of the document's structure, but they are not generally concerned with the content of the elements within the document. In response to this gap, a few XSS-aware "cleaners" have been created.

What follows is an abbreviated survey of related work and how it affects our original work.

NekoHTML

NekoHTML [10] is an Apache project for advanced HTML sanitization. It has the rare ability to validate HTML "fragments" as well as entire documents. It validates the legality of HTML elements, e.g., ensuring that anchor tags don't have nested anchor tags and making sure that heading tags don't have illegal tags within them.

DeXSS

DeXSS [11] is an XSS validation tool that parses and cleans tainted HTML. The tool uses TagSoup, an SGML parsing library, to create well formed HTML that it can parse according to a set of hard-coded rules. The tool, though visionary, suffers from a few major drawbacks.

First, the rules are hard-coded into the validation tool and are impossible to change without source code modification, making deployment and configuration very difficult. Also, the rules that are hard-coded are not adequate according to the standards of today's attackers.

JavaScriptWhiteListValidator

This research maps very closely to this project. At Eurosec 2004 [12] Stefan Fünfroeken presented a framework for validating HTML against XSS according to a policy file. However, no reference implementation or proof-of-concept code was ever released. Although the validation framework used a positive security model, it did not have the customization ability, CSS support or feedback mechanism available in our original approach.

Original Approach

The primary focus of this work was to create a XSS filter that worked on a positive and customizable security model. The name of the project is "AntiSamy", in reference to Samy Kamkar's now infamous MySpace XSS worm. The secondary focus was to make this tool as user friendly as possible so as to allow applications using it to communicate to the user how their input was filtered or how they could tune it themselves in order to accommodate a more successful filter.

HTML Sanitization (Pre-Processing)

The approach to engineering the filter included leveraging an existing HTML sanitization tool for validation pre-processing. This was required because it was thought that good validation software required code that it could safely assume was syntactically well-formed. This way no fragmentation attacks involving unbalanced start or end markers (tags or brackets, in the world of the web) could prevent accurate filtering.

An HTML validation tool must also have some idea of context because W3C specifications dictate constraints that require contextual awareness. For instance, an anchor tag is not legally allowed to have another anchor tag nested inside of it.

To take care of the HTML sanitization (including contextual validation), the open source project named *NekoHTML* was utilized. *NekoHTML* is a Java API (and standalone program) that transforms unbroken of any version into clean XHTML 1.0.

Tag/CSS Validation Rules (Processing)

The main validation processing takes place in a depth-first fashion. Starting with the root, each node is processed according to the specifications inside the security model XML file given the node name (e.g., *html* or *input*). There

are three modes of validation (also called *processing actions*): *filter*, *truncate* and *validate* and they are each described in the following section.

Filter

The *filter* processing action performs no validation per se, but only removes the start and end tags, promoting the tag's contents. This sanitization is useful in many cases. For example, if you decided you wouldn't like users to input *meta* tags that could mess with your robot indexing, setting *filter* would have the effect demonstrated in Figure 2.

```
<meta name="expires;-
1;www.phishingsite.com">This is some
text.</meta>
```

Figure 2.1

```
This is some text.
```

Figure 2.2

Truncate

When the *truncate* processing action is set, no actual validation takes place. The *truncate* action simply removes all the attributes and child nodes of a tag, making validation of its attributes unnecessary. A number of tags should be set to *truncate*.

A *truncate* would turn the text from Figure 3.1 into the text seen in Figure 3.2:

```
<br unknownAttributeAttack="lattack"
onClick="alert(document.cookie)">
```

Figure 3.1

```
<br />
```

Figure 3.2

Many formatting tags are set to *truncate* in the default policy file, including *em*, *small*, *big*, *i*, *b*, *u*, *center*, *pre* and more.

Validate

The *validate* processing action is where the meat of the filtering logic resides. If there are no attributes defined for a tag by the policy file, the *validate* processing action will

act the same as the *truncate* processing action, except the child nodes will be validated instead of removed.

The *validate* action steps through each of the attributes in the tag to be filtered and checks if there is a corresponding entry for that tag and attribute combination in the policy file. If no entry is found, the attribute is simply removed. If there is an entry, the filter tries to validate its value against the rules in the entry.

There are two ways for an attribute value to be validated; by being equal to a literal string value or by the matching of a regular expression. Accordingly, each attribute's definition in the policy can have a list of valid literal strings and a list of regular expressions to match. This is a departure from other XSS filters (and other security tools, in general) that don't allow for multiple ways to specify valid values, which force the user into writing overly complex (and likely incomplete or unpredictable) regular expressions.

Failed Validation (Remediation)

When an attribute does not pass a validation check, one of a few *onInvalid* actions is taken. The possible *onInvalid* actions (which are detailed in the following section and are also set in the policy file) dictate what to do with the tag and its contents. The set of *onInvalid* actions includes *removeTag*, *filterTag* and *removeAttribute*. The default action is *removeAttribute*.

If an attribute with the *removeTag* set for its *onInvalid* action fails validation, the tag holding the attribute being checked and its contents will be removed entirely. This *onInvalid* action is reserved for those attributes, which when removed, make the presence of the tag meaningless. An example usage of this setting is displayed in Figure 4.

```
Welcome, my name is
<script>
var cke = document.cookie;
var url= 'http://evil.rt/cookie.cgi'+cke;
document.location = url;
</script>
and I'm 25 years old!
```

Figure 4.1

```
Welcome, my name is
and I'm 25 years old!
```

Figure 4.2

If an attribute with an *onInvalid* action set to *filterTag* fails validation, the start and end tag of the node will be removed while the contents are promoted. This is exactly what happens in the *filter* processing action. The process can be seen in Figure 5.

```
<a href="javascript:alert('xss')">Click
on this!</a>
```

Figure 5.1

```
Click on this!
```

Figure 5.2

The default *onInvalid* action is *removeAttribute*. When this *onInvalid* action is set (or if none is set) on an attribute that fails validation, the attribute itself is removed from the tag, but the tag and its contents will remain. The process can be seen in action in Figure 6.

```
<input src="javascript:alert('xss')"
type="button" value="Hi!">
```

Figure 6.1

```
<input type="button" value="Hi!">
```

Figure 6.2

Policy File (Knowledge Base)

The knowledge base for the filter's engine is an XML file called *antisamy.xml*. The same policy file can be used across multiple implementations (.Net, J2EE, etc.).

The default policy file was tailored to W3C's HTML 4.0 and CSS 2.0 specifications [7]. Thus any official attributes which is dictated by the specifications can be used. If a user agent supports an attribute not specified, it can be added to the policy file, though some effort has already been put into integrated those non-standard attributes which are being used and honored in the wild.

Results

Security mechanisms cannot be comprehensively tested because it's impossible to prove a negative. Another way of saying that is, there is no way of knowing if the set of all publicly known attacks, which can be incorporated into test cases, is equal to the set of all possible attacks. A large subset of all publicly known XSS attacks gathered from many recognized knowledge bases [13] [14] [15] have been tested with 100% effectiveness. There is one class of

attacks that the validation tool will not be able to detect with high assurance without further customization, and that is the class of attacks regular modulated character set attacks [16]. In such cases where the browser is essentially translating the validated input into another character set, any assurance provided by the validation tool is removed.

Running time was also a very important consideration given the importance of availability and response time for enterprise applications. The results from our timing tests can be seen in Table 1.

Size of HTML(b)	Average Running Time(s)
1k-11k	.238357
12k-25k	.341
32k-53k	0.5483

Conclusions

In order for the project to be considered a success, the filter must display strong degree of resistance to XSS attacks while retaining nearly all formatting and style information. Also, the filter must execute in reasonably short time.

References

1. *Cross-Site Scripting (XSS)* (n.d.). Retrieved November 19, 2007, from OWASP: http://www.owasp.org/index.php/Cross_Site_Scripting
2. *Cross Site Scripting Info: Encoding Examples.* (n.d.). Retrieved November 19, 2007, from Apache HTTP Server Project: http://httpd.apache.org/info/css-security/encoding_examples.html
3. (n.d.). Retrieved November 19, 2007, from `</xssed>_:` [http://www.xssed.com/archive/special=1/\(n.d.\)](http://www.xssed.com/archive/special=1/(n.d.)).
4. Shiflett, C. (n.d.). *Allowing HTML and Preventing XSS.* Retrieved November 19, 2007, from PHP And Web Application Security: <http://shiflett.org/blog/2007/mar/allowing-html-and-preventing-xss>
5. *How to Hack MySpace.* (2007, March 20). Retrieved November 19, 2007, from Forbes.com: http://www.forbes.com/security/2007/03/20/myspace-hackers-virus-tech-security-cx_ll_0329myspace.html
6. *How to Edit a Page.* (n.d.). Retrieved November 19, 2007, from Wikipedia, The Free Encyclopedia: http://en.wikipedia.org/wiki/WP:MARKUP#Wiki_markup
7. *XHTML2 Working Group Home Page.* (n.d.). Retrieved November 19, 2007, from W3C Interaction Domain: <http://www.w3.org/MarkUp/>
8. Kerchoffs, A. (1883). *La Cryptographie Militaire. Journal des Sciences Militaires* , IX, 5-83, 161-191.
9. Raggett, D. (n.d.). *Clean up your Web pages with HTML TIDY.* Retrieved November 19, 2007, from W3C World Wide Web Consortium: <http://www.w3.org/People/Raggett/tidy/>
10. Clark, A. (n.d.). *CyberNeko HTML Parser.* Retrieved November 19, 2007, from The Apache Software Foundation: Committers: <http://people.apache.org/~andyc/neko/doc/html/>
11. Retrieved November 19, 2007, from DeXSS -- Java program for removing JavaScript from HTML: <http://software.graflex.org/dexss/>
12. *Filtering JavaScript to Prevent Cross-Site Scripting.* (2005, January 7). Retrieved November 19, 2007, from http://www.secollogic.de/downloads/web/051207_EUROSEC_Draft_Whitepaper_Filtering_JavaScript.pdf
13. *DOM Based Cross Site Scripting or XSS of the Third Kind.* (n.d.). Retrieved November 19, 2007, from Web Application Security Consortium: <http://www.webappsec.org/projects/articles/071105.shtml>
14. *XSS (Cross Site Scripting) Cheat Sheet Esp: for Filter Evasion.* (n.d.). Retrieved November 19, 2007, from ha.ckers.org: <http://ha.ckers.org/xss.html>
15. *XSSDB.* (n.d.). Retrieved November 19, 2007, from GNUCITIZEN: <http://www.gnucitizen.org/xssdb/application.htm>
16. *US-ASCII part 2.* (n.d.). Retrieved November 19, 2007, from ha.ckers.org: <http://ha.ckers.org/blog/20060621/us-ascii-xss-part-2/>