Code-Injection Attacks in Browsers Supporting Policies

Elias Athanasopoulos, Vasilis Pappas, and Evangelos P. Markatos
Institute of Computer Science
Foundation for Research and Technology - Hellas
N. Plastira 100, Vassilika Vouton, GR-700 13 Heraklion, Crete, Greece
{elathan, vpappas, markatos}@ics.forth.gr

ABSTRACT

Code-injection attacks can take place in a large variety of layers, from native code to databases and web applications. The latter case involves mainly client-side code injection in the browser environment, also known as Cross-Site Scripting (XSS). There are numerous ways to defeat XSS attacks, from static and taint analysis to policy enforcement in the web browser. In this paper, we enlist new forms of XSS attacks that seek to bypass browser enforced policies. The attacks outlined in this paper resemble the classic *return-to-libc* attack in native code. We propose a new form of code isolation, based on browser actions, in order to mitigate the problem.

1. INTRODUCTION

Code injection is traditionally considered as a major threat. A significant fraction of host compromising is carried out using buffer overflow attacks [8]. In the same fashion an adversary can compromise a database using a `SQL` injection attack [1] or the web browser's environment using a Cross-Site Scripting (XSS) attack.

An XSS attack is typically carried out as follows. An attacker injects some client-side code, usually JavaScript, in a web document. The injection may performed, but is not limited to, in a content submission. For example, a user posts a comment in a blog story, which embeds some JavaScript. The result is that every web browser that renders the comment of the blog story will, also, execute the attacker's JavaScript. The attacker's code can steal the user's cookies and, thus, hijack her session.

Our contribution. There are numerous proposals for XSS attack mitigation. In this paper we explore BEEP [6], which tries to defend against XSS attacks using a policy enforcement framework in web browsers. We spot limitations in the approach and develop new XSS attacks that succeed to bypass the policy framework. Finally, we propose our own framework which is based on policies expressed as *browser actions*. Our framework guarantees that all trusted client-side code can be successfully isolated from possibly untrusted.

This paper is organized as follows. We review the state of the art in current approaches for browser enforced policies in Section 2 and we enlist new forms of XSS attacks in Section 3. We outline our XSS mitigation proposal in Section 4. We conclude and present our future steps in Section 5.

2. BROWSER POLICIES

In this section we present the state-of-the-art approach for embedding policies in the web browser [6] and we highlight the weaknesses of the methodology. We then proceed, in the next section, and present XSS attacks that cannot be captured by the current scheme.

2.1 Overview

Enforcing policies in the web browser aims at handling and, possibly, aborting execution of untrusted client-side code. Implementing policies in the browser, according to [6], is based on the following assumptions:

1. Web browsers have all the required complexity in order to detect (parse) and render a script.

2. The web application developer knows exactly which scripts are trusted to be executed in the web browser.

We totally agree, as far as assumption (1) is concerned. The complexity and the plethora of client-side technologies have transformed a web document's rendering to a very sophisticated process, which can be carried out, correctly, only by modern full-featured web browsers. Consider, also, that it is a common practice for browsers to render a page in a *best-effort* approach, meaning that even grammatically ill constructs (e.g. misplacement of HTML tags), sometimes, are executed. These arguments conclude our thesis: script detection should be carried out in the web browser and not in the web server.

On the contrary, we do not fully support assumption (2). Considering that (a) modern web applications are composed by thousands, or even millions, lines of code (for example take into account applications like GMail or Google Documents), (b) most web applications contain code for server-side tasks, database access, layout formatting (CSS), interaction with XML data and multiple client-side technologies and, finally, (c) most web applications are developed by groups of programmers, we feel that spotting all trusted scripts is *not*, by any means, a trivial task. More precisely, as we explore in detail in Section 4, we believe that an explicit code separation scheme should be applied in web development.

BEEP suggests that policies should be enforced in the browser using a *Whitelisting* or a *DOM Sandboxing* approach. We, shortly, review each one of them.

Whitelisting. Script whitelisting works as follows. The web application includes a list of cryptographic hashes of valid (trusted) client-side scripts. The browser, using a *hook*, checks, upon execution of a JavaScript snippet, if there is a cryptographic hash for the script in the white-list. If the hash is found the script is considered trusted and the browser executes it. If not, the script is considered non-trusted and the policy defines if the script will be rendered or not.

Notice that there is no checking for the location of the script inside the web document. Consider, for example, the simple case where an attacker places a trusted script, initially configured to run upon a user's click (using the `onclick` action), to be rendered upon document loading (using the `onload`[1] action).

DOM Sandboxing. DOM sandboxing works as follows. The web server places trusted scripts inside `div` or `span` HTML elements that are attributed as *trusted*. For example, consider the construct:

```
<div class='trusted'> ... script ... </div>
```

The web browser, upon rendering, parses the DOM tree and executes client-side scripts only when they are contained in *trusted* DOM elements. This method is vulnerable to the *node-splitting* attack, in which a malicious script is surrounded, on purpose, by misplaced HTML tags in order to escape from a DOM node. Consider for example the construct:

```
<i>{ message }<\i>
```

which, denotes that a message should be rendered in *italic* style. If the message variable is filled in with:

```
</i><b> bold message </b><i>
```

then the carefully placed `<i>` and `<b>` tags should result the message to be displayed in **bold** style, rather than *italic*.

The authors of BEEP suggest a workaround for dealing with node-splitting, but we consider it rather inefficient, since some client-side code should be emitted using a special coding idiom. We rather agree with a more elegant approach suggested in [5] (some very similar ideas have also been proposed in [7]).

Notice that DOM sandboxing requires the code injection to take place in an existing DOM tree. However, as it was recently shown, this is not always the case; a simple file upload and rendering is enough [2]. We further investigate these attacks in Section 3.

3. ATTACK EXAMPLES

Based on the analysis we conducted in the previous section, we proceed and present a form of XSS attacks that defeats policies enforced in the web browser. We first present attacks that escape whitelisting and then attacks that defeat DOM sandboxing.

3.1 Defeating Whitelisting

Most XSS attacks are considered to happen by injecting arbitrary client-side code in a web document. This code is assumed to be foreign, i.e. not generated by the web server. However, it is possible to perform an XSS attack by placing code that *is* generated by the web server in different regions of the web page. This attack resembles the classic *return-to-libc* attack [3] in native code applications. Return oriented programming suggests that an exploit may simply transfer execution to a place in `libc`[2], which may cause again execution of arbitrary code on behalf of the attacker. The difference with the traditional buffer overflow attack [8] is that the attacker has not injected any *foreign* code in the program. Instead, she transfers execution to a point that already hosts code that can assist her goal.

A similar approach can be used by an attacker in order to escape whitelisting. Instead of injecting her code, she can take advantage of existing *white-listed* code available in the web document. Note that, typically, a large fraction of client-side code is not executed upon document loading, but it is triggered during user events, such as mouse clicks or mouse movements. Below we enumerate some possible scenarios.

Forcing Logout. Assume a blog site that has a JavaScript function `logout()`, which is executed when the user clicks *Logout* from the web site's menu. An attacker could perform an XSS attack by placing a script that calls `logout()` when a new blog entry is rendered. A user reading a blog story will be forced to logout.

Redirecting. In a similar fashion, a web site that uses JavaScript code to perform redirection (for example using `location.href`) can be also attacked by plac-

[1] One can argue that the `onload` action is limited and usually associated with the `<body>` tag. The latter is considered hard to be altered through a code-injection attack. However, note that the `onload` event is available for all elements (images) included in the web document using the `<img>` tag.

[2] This can also happen with other libraries as well, but `libc` seems ideal since (a) it is linked to every program and (b) it supports operations like `system(), exec(), adduser()`, etc., which can be (ab)used accordingly. More interestingly, the attack can happen with no function calls but using available combinations of existing code [9].

ing this white-listed code in an `onload` event.

Data Erasing. In a similar fashion, a portal which places user content that can be deleted using client-side code (AJAX [4] interfaces are popular in social networks like `Facebook` and `MySpace`) can be attacked by injecting the white-listed deletion code in an `onload` event. This can be considered similar to a `SQL` injection attack [1], since the attacker implicitly grants access to the web site's database.

Complete Takeover. Theoretically, a web site that has a full featured AJAX interface can be completely taken over, since the attacker has all she needs to use already white-listed by the web server. For example, a bank web site that uses a JavaScript `transact()` function for user transactions is vulnerable to XSS attacks that perform arbitrary transactions.

A quick workaround to mitigate the attacks presented above, is to include the event type, during the whitelisting process. For example, upon trying to execute script `S1`, which was triggered by an `onclick` event, the browser should check the white-list for finding a hash key for `S1` *associated with an* `onclick` *event.* However, this can only mitigate attacks which are based on using existing code with a different event type than the one used initially by the web programmer. Attacks may still happen. Consider the *Data Erasing* attack described above and an attacker that places the deletion code in `onclick` events associated with new web document's regions, not initially designed by the web programmer.

Finally, attacks that are based on injecting malicious data in white-listed scripts have been described in [7].

3.2 Defeating DOM Sandboxing

DOM sanbdoxing marks regions defined by `div` and `span` tags as trusted or non-trusted. JavaScript code is executed only if it is contained in a trusted region. We assume that a technique like Noncespaces [5] is used to prevent node-splitting.

We have two arguments against DOM sandboxing. First, we believe that marking a region as trusted or non-trusted may not always be that trivial. Especially, if we take into account the complexity of modern web sites, which are typically composed by hundreds of different `div` elements and thousands of JavaScript code. But, even if the marking is carried out correctly, there is no guarantee that a trusted `div` element will never host code from an XSS attack. The site designer should take care of this issue, programmatically. More precisely, the site designer should, somehow, provide guarantees that a trusted `<div>` element will never host user input. Second, XSS attacks do not always need a DOM tree in order to take place. For example, consider an XSS attack which is *carried in* a PostScript file [2]. The attack will be launched when the file is previewed. There is high probability that upon previewing there will be no DOM tree to surround the injected code.

4. PROPOSAL

This section presents a proposal to mitigate attacks outlined in this paper. This proposal summarizes our next steps in this work. We propose a framework that is composed by three different elements: *client-side code separation*, *client-side code isolation* and *action based policy enforcement.* Below we present each element in details.

Client-Side Code Separation. We propose that client-side code should be separated from the rest of the code base during development. Currently, this is the case with every server-side technology, such as PHP, ASP, Python (Django), Ruby (Ruby on Rails), etc. For example, in PHP all server-side code is enclosed in: `<?php` and `?>`. If client-side code separation is applied, then the second assumption (see Section 2) for policy enforcement in web browsers is valid. Moreover, using this type of separation assists in client-side code isolation, as we will explain below. The reason is that code separation assists in applying *isolation operators* to the client-side code corpus.

Client-Side Code Isolation. We propose web servers to apply *isolation operators* in all client-side code. An isolation operator transposes all code in a new *isolated* domain. For an example, consider the isolation operator $I_{S \times A}$ which denotes that the separated code is multiplied with a matrix, A. All client-side code, produced by the web server, is isolated from any other client-side code, which was injected in the web document. Another example of operator is a cryptographic function or a simple transformation function based on the `XOR` operation. Client-side code isolation suggests that all legitimate code should be completely isolated from, possibly, injected code. The concept of isolation is not new. An extensive study for various approaches in isolating hypertext has been conducted in [10].

Action Based Policy Enforcement. We propose that policies should be expressed using *actions.* More precisely, we propose that the browser should handle trusted code as *apply* I^{-1} *(de-isolation) and execute* and non-trusted code as *no-execute* or *execute* according to the defined policies. The de-isolation operator should be a matrix multiplication with the reverse matrix, a decrypt function, or a second `XOR` operation, respectively, for the examples of the previous section. In all cases, the web browser and web server have to share a key (the matrix, the cryptographic key or the value which is used in `XOR`). This information should be refreshed frequently (for example, every time there is a new session) and transmitted using an `HTTP` header. Using a policy enforcement scheme expressed as actions it is guaranteed that the web browser executes *only* code that is produced by the web server and the attacker, as long as she misses the operator's *key*, can not inject any code, *even existing code produced by the web server.*

This proposal requires modifications in both server and client, as it is also the case for BEEP.

Currently, we are at the stage of implementing the client-side part of the proposal in Firefox (we are still investigating Safari and Chromium). The server-side part is almost complete for the Apache web server.

4.1 Challenges

One significant challenge in implementing the approach outlined in this section is support for *mashups.* A mashup is a web site that collects information from third parties and presents it to the user. One could argue that a mashup is essentially a code injection process. The main site will fetch code from third party sites and inject it to the web documents it generates. There is a number of possible things that may produce confusion or even have negative result.

A fraction of the third party web sites have implemented the framework. This case will produce a mixed up of client-side code in the web browser. Parts will be isolated and parts, trusted or not, will be in plain text. There will be confusion and the final web document will be probably no functional. This issue can be addressed if each web server reports, using `HTTP` headers, if the framework is supported or not. This is also convenient for an incremental deployment. However, the security of the mashup, as far as XSS attacks are concerned, is not fully guaranteed.

None of the third party web sites have implemented the framework, but the main site. This case will produce a negative result. The mashup will isolate all collected, third party, client-side code in the final web document and thus will advertise *all* generated client-side code as trusted. The code will possibly include code injections performed in any of the third party web sites. Thus, the framework must *not* be, in any case, applied in proxy environments on behalf of third party sites.

All third party web sites have implemented the framework, but not the main site. This case is considered healthy, since all authentic sources perform the isolation. The isolated code is trusted, even if the main site does not implement the framework. However, the main site must also transfer the keys in order the browser to be able to perform de-isolation. As long as the main site is considered trusted, then the framework guarantees no code injection incidents in the final web document.

5. CONCLUSION

In this paper we have investigated how current approaches for policy enforcement in web browsers, namely BEEP [6], fail to defeat XSS attacks under specific circumstances. We have presented a series of web attacks, based on injection of existing white-listed (trusted) code, which resemble the well known *return-to-libc* attack. Based on our findings, we propose a new policy scheme for web browsers, which uses actions, in order to promote policies. More precisely, we propose (a) *client-side code separation* during development, so that client-side code is easily separated from the rest of the source, (b) *client-side code isolation* using isolation operators, so that trusted code is easily isolated from code injections at browsing time and (c) *action based policy enforcement*, so that browsers *de-isolate and execute* trusted code. This framework guarantees that the web browser executes *only* code that is produced by the web server and the attacker, as long as she misses the key for the isolation operation, can not inject any code, *even existing code produced by the web server.*

Currently, we are at the stage of implementing the client-side part of the proposal in Firefox (we are still investigating Safari and Chromium). The server-side part is almost complete for the Apache web server.

6. ACKNOWLEDGEMENTS

Elias Athanasopoulos, Vasilis Pappas and Evangelos P. Markatos are also with the University of Crete. Elias Athanasopoulos is also funded by the Microsoft Research PhD Scholarship project, which is provided by Microsoft Research Cambridge.

7. REFERENCES

[1] C. Anley. Advanced SQL injection in SQL server applications. *White paper, Next Generation Security Software Ltd*, 2002.

[2] A. Barth, J. Caballero, and D. Song. Secure Content Sniffing for Web Browsers or How to Stop Papers from Reviewing Themselves. In *Proceedings of the 30th IEEE Symposium on Security & Privacy*, Oakland, CA, May 2009.

[3] S. Designer. Return-to-libc Attack. *Bugtraq, Aug*, 1997.

[4] J. Garrett et al. Ajax: A new approach to web applications. *Adaptive path*, 18, 2005.

[5] M. V. Gundy and H. Chen. Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 8-11, 2009.

[6] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 601–610, New York, NY, USA, 2007. ACM.

[7] Y. Nadji, P. Saxena, and D. Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, 2009.

[8] A. One. Smashing the stack for fun and profit. *Phrack magazine*, 49(7), 1996.

[9] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, New York, NY, USA, 2007. ACM.

[10] M. Ter Louw, P. Bisht, and V. Venkatakrishnan. Analysis of hypertext isolation techniques for XSS prevention. In *Web 2.0 Security and Privacy 2008*, May 2008.