Systematic XSS exploitation

Aureliano Calvo
Core Security Technologies, ITBA phD Program

July 27, 2009

Abstract

The cross-site scripting (XSS) vulnerabilities are usually overlooked and their impact is typically underestimated because its analysis requires security skills that are often absent in testers and developers. In this paper I introduce a tool that enables the decoupling of the exploitation and post-exploitation. The tool provides the means to turn a XSS vulnerability into a machine that receives payloads with post-exploitations actions written against a generic API; therefore allowing to asses the full potential of the vulnerability. In particular, I show how can exploited pages be used as vantage points for other kinds of attacks, such as exploitation of binary vulnerabilities and malware distribution. I also include full details into how the tool works and code for its critical functions.

keywords: Software agents, penetration testing, network vulnerability assessment, cross site scripting, web application vulnerability assessment, hybrid penetration testing.

1 Introduction

Cross-site scripting (XSS) is a security vulnerability of dynamic Web pages generated from information supplied to the web server and replayed as part of the response to the browser. In an XSS attack, a malicious user can create a specially crafted link to inject unwanted executable script or code (usually JavaScript) into a Web site ([Sub06]).

There is a tendency to make penetration tests that use several attack vectors to achieve their success ([Ric08]). None of the tools available do not allow the operator to combine different attack vectors with cross-site scripting. For instance, [Mav] and [Wad06] do not incorporate binary exploits. This work explains the implementation of a framework that includes a systematic way to exploit cross-site scripting, allowing the penetration tester to combine several attack vectors.

In Section 2, I describe the browser as an execution environment and the same-origin policy and how it limits the communication between the attacked browser and the attacker. The Section 3 explains how the JavaScript agent works and why is it useful to homogenize attacks. The agent provides a stable base to work upon. In Section 4, different post-exploitation actions are described. These actions are greatly simplified by the API provided by the agent. At last, in Section 5 I provide a brief explanation of the agent implementation and a detailed explanation of the interaction between the owned browser and the attacker. The implementation builds upon the syscall-proxying agents ([Các02]) and uses the ideas to organize different kinds of agents presented in [RT07], allowing to combine different kinds of vectors in the same attack. In this paper there are shown two different ways on how to escalate a XSS exploitation to a full control of the host running the compromised browser.

2 Web browser security

Web browsers retrieve pages which may be hosted in different servers on different domains across one or many organizations. Pages can contain scripts that will be executed in the browser (usually written in JavaScript), other pages (through frames, iframes or popups), and images and scripts from other domains. How to prevent a document or script loaded from one "origin" from getting or set-

URL	Can be read?
http://foo.com/index.html	yes
http://foo.com/version2/webApp	yes
http://foo.com:80/bar/baz.html	yes (same port)
https://foo.com/bar/baz.html	no (protocol)
http://www.foo.com/bar/baz.html	no (host)
http://foo.com:8080/bar/baz.html	no (port)

Figure 1: Access restrictions from http://foo.com/bar/baz.html

ting properties of a document from a different "origin"? As an answer the *same domain/origin policy* is being applied since Netscape Navigator 2.0.

As described in [CDL08], the *same origin policy* means that code that runs in the context of a domain cannot read data fetched from another domain. But it does not restrict the data that can be sent to a domain (for instance, by issuing a HTTP POST with a form), the information that can be shown to the user or the domain of the scripts that can be ran in the page.

The scripts loaded in a web page run in the domain of the web page. For instance, the `urchin_tracker.js` script (that is the base for Google Analytics) runs in the domain of the page that includes it. A special note must be made. Any script can run a script from another domain and see the changes that this script makes on the page's object, but it cannot see its own source code. It is the equivalent of loading an image from a different domain (the image can be loaded from JavaScript code in the original domain, the image is shown to the user, but the image bytes cannot be read from JavaScript code). The ability to run code hosted in a different domain is crucial in the implementation of the JavaScript agent.

It also should be noted that the same origin policy is also known as "cross domain restrictions."

3 JavaScript agent

When a cross-site scripting vulnerability is exploited, the attacker's JavaScript is run in the victim's browser under the vulnerable page domain. Once it is possible to run JavaScript code in a browser, several question arise: how can connectivity be maintained, how can it be kept running,

```
var BASE = "ATTACKER SERVER URL"
window.onload = function(){ egg() }
function egg() {
  get(BASE + "/action?id=AN_ID")
}
function timer() {
  if(typeof timeout !== "undefined"
    && timeout !== null){
    window.clearTimeout(timeout)
  }
  var timeout = window.setTimeout(
    "egg()", 2000)
}
function get(src){
  var script =
    document.createElement('script')
  script.defer = true
  script.type = 'text/javascript'
  script.id = escape( Math.random()
    + '-' + Math.random())
  script.src = src + '&tag_id=' +
    script.id;
  document.body.appendChild(script)
}
```

Figure 2: JavaScript egg code

what code should it run? We came up with our JavaScript agent as a solution to these problems. The payload of the exploit is code that retrieves the code that requests regularly a server for actions and then executes them. This code also provides methods to return data to the server.

Giving names to the components mentioned above, the custom JavaScript code is called the "exploit" and the code that it loads is called the "egg" (see figure 2).

3.1 Agent persistence

Once the agent is running, it runs until the browser navigates to another page. But there are some methods to keep it running longer. These mechanisms represent 3 different points in the trade-off curve between persistence (ability to keep running) and stealthiness (ability to hide from the user).

- *Non persistent* Does not attempt to keep running after the user navigates away from the page.

- *Pop-up* It launches a pop-up with the agent's code. It may be disguised as advertising. A pop up blocker may disable this way to persist an agent.
- *Page rewriting* Looks for the page links and adds code that embeds the pointed page in an IFrame and removes the rest of the markup, keeping the old page but with the new page content. The user may notice this persistence mechanism because the URL in the browser does not change when he navigates. XssShell implements this mechanism ([Mav]).

3.2 Victim identification

From the attacker's server point of view there is another challenge. When a request comes from a compromised browser, the server has to identify it. These are some possible ways to identify the victim:

- *Each new egg request is a new browser* Each time an egg is requested, an unique identifier for the browser is generated, and then the identification is sent from the owned browser to the attacked server on each request. The same browser will have 2 different identifications if the egg is loaded twice on the browser.
- *Cookies* If a request comes with the identification cookie not set to the attacker's server, the cookie is set to a new value. The cookie value is used to identify the browser on all the requests. A browser may be identified many times if the cookies are erased.
- *IP* The browser is identified by the IP of the computer that runs it. It will identify as the same browser all the browsers running on a NATed network.
- *IP and user-agent* It also uses the user agent HTTP header to identify the browser. It has the same problems as the former browser identification mechanism.

4 Post exploitation

Once it is possible to run JavaScript in a browser in the context of a domain, several things are possible. It is trivial that a malicious user can steal the cookies of the browser for the page. But the attacker is in no way limited to that.

The attacker may also do the following:

- Trigger a binary vulnerability on the browser.
- Alter the web page to provide links to download malware.
- Deface the page.
- Crawl the page domain using the owned browser and its credentials.
- Interact with the attacked page using the owned browser and its credentials.
- Launch a cross site request forgery attack[Rob08] using the owned browser and its credentials.
- Launch a non-blind SQL injection using the owned browser and its credentials on the same domain of the hijacked page.
- Launch a blind SQL injection using the owned browser and its credentials on any domain [RT07]
- Gather information of the owned browser and the platform where it runs.
- Launch a port scan from the owned browser[GNU06].
- Find if a link was visited by the user[Jer06] [Mav].
- Log both the keyboard and mouse actions[Mav].
- Get the clipboard contents in the browser's computer [Mav].
- Run arbitrary JavaScript on the page.

5 Implementation

All the framework functionality and most of the proposed post-exploitation functionality are implemented as parts of Core Impact which proves its strength and commercial grade quality. Because

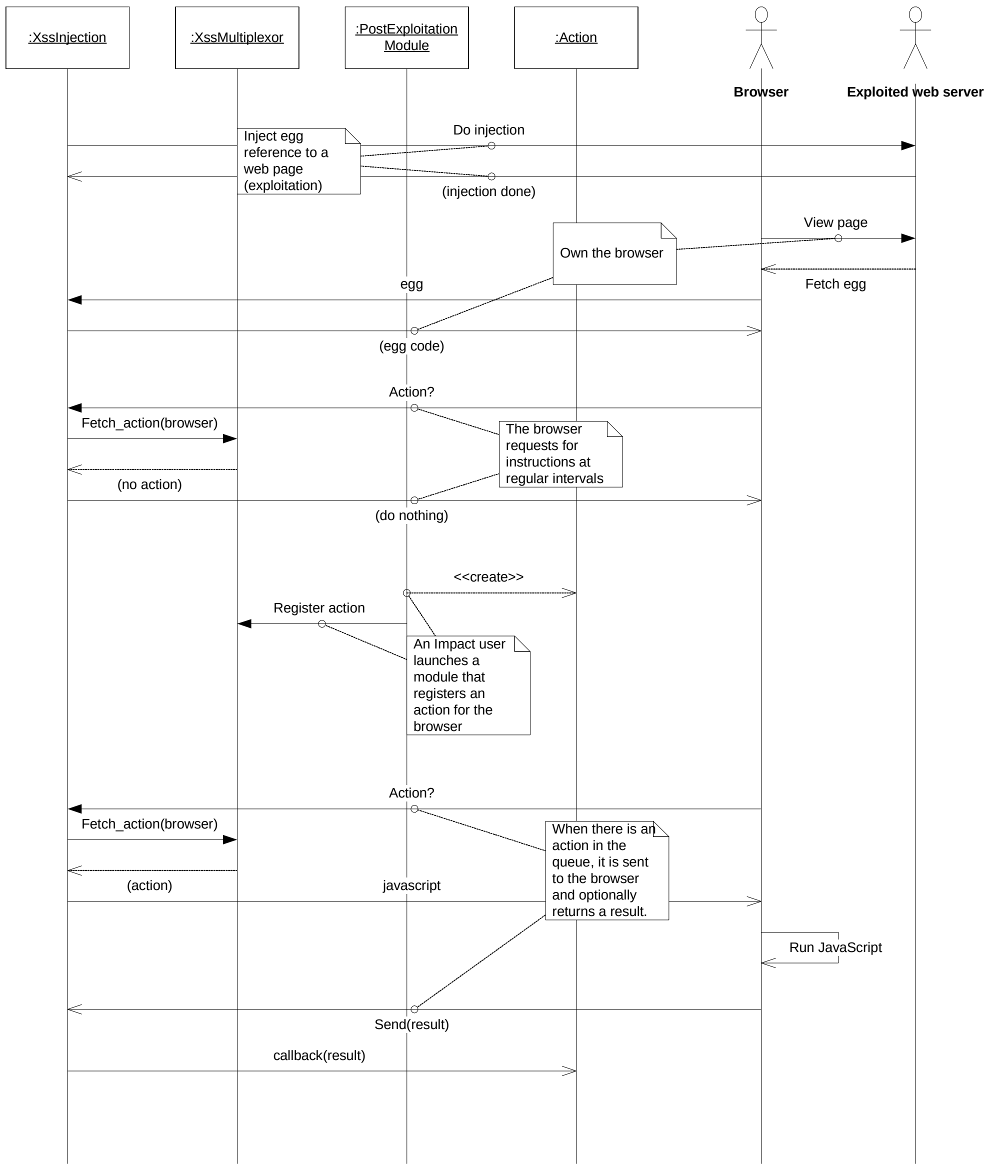

Figure 3: Detailed steps in the exploitation of a persistent XSS vulnerability

Impact already provides functionality for remote-code execution exploits, it includes the ability to escalate to full control of the host running the owned browser. In this section I'll explain the technical details of this implementation.

5.1 Injection

Injections are detected and exploited using the techniques explained in [BM09]. Page parameters are fuzzed looking for errors in the escaping and then this information is used to generate request that renders a page with a valid HTML DOM that includes a XSS agent. Manual injections can also be programmed as modules.

5.2 Browser execution

When a browser navigates to a compromised web page, it runs JavaScript code that sends a request to the attacker's server every 10 seconds asking for instructions. These instructions are normal JavaScript but a primitive is provided to communicate back to the server some findings. This primitive is called `answer()`.

The communication is established by creating a `<script>` tag with its source attribute pointing to the attacker's server (using DOM manipulation via JavaScript) and when the code sent by the browser is ran, the script is removed from the DOM.

All this low level handling is abstracted away from the post-exploitation code.

5.3 Persistence and browser identification

By default, it has no persistence implemented as defined in the section 3.1. But two different kinds of persistence where implemented. The "Extend web browser agent life" module changes the owned web page to open a `<iframe>` with the content of the linked page, keeping the old page but showing all the content of the new page. The "Launch web browser agent in popup" module opens a popup from the current page that runs another agent.

Each new egg request is a new browser, as defined in section 3.2.

5.4 Actions

Once a XSS vulnerability is exploited, the attacker can run actions on the browser. An action is composed of a JavaScript code that runs in the browser and, optionally, a callback that runs in the attacker server after information is sent (using `answer`) back.

It is important to notice that actions can be "chained". If the callback returns another action, this action will be the next action executed in this browser.

Except for the chained actions, the actions are queued and when a browser requests a new action, the first action available on the queue for this browser is sent (or the "null" action if the queue is empty).

5.5 Back-end

The back-end has a queue for each browser attacked. Each queue contains the actions to be executed for each browser. When a request for an action arrives to the back-end web server, it responds with JavaScript code that contains the next action to be executed and when the actions responds it executes the action callback in the server. The action may queue other actions for the browser (or other browsers).

The back-end also has a database that is used to persist the information gathered in the browser.

5.6 Post exploitation modules

Several post-exploitation modules were implemented for XSS vulnerabilities, including cookie stealing, fetching other pages of the domain using the browser credentials, web page defacement and a JavaScript console that runs in the context of the owned web page.

And there are two modules that enable a penetration tester to do hybrid penetration testing. The "Launch Client-Side exploit using Web Browser agent" enables the penetration tester to use all the shipped web-browser exploits using the compromised browser as a vantage point and the "Replace executable links with OS agent executable" changes all the links pointing to .exe files with links pointing to executable files that run a system-call proxying agent. The last module allows an attacker

```
try {
  function params( msg ) {
    result = ''
    for( var key in msg ) {
      result += '&' + key + '=' +
        encodeURIComponent( msg[key] )
    }
    return result.substr(1)
  }

  function answer(data) {
    # current action id
    data.action_id = 'XXXXXX'
    get(BASE + "/answer?" + params(data))
  }
  eval('<INSTRUCTIONS>')
} catch (e) {
    answer( { error: ('' + e) } )
} finally {
  #current tag id
  document.body.removeChild(
    document.getElementById('XXXXXX'))
  if(typeof timeout !== "undefined" &&
    timeout !== null) {
      window.clearTimeout(timeout)
    }
  var timeout = window.setTimeout(
    "egg()", 2000)
}
```

Figure 4: JavaScript sent to the browser when a non chained action is executed.

to get full control of a host using just a cross-site scripting vulnerability.

6 Acknowledgments

This paper couldn't have been written without the help of: Diego Tiscornia, Ariel Waissbein, Ivan Arce, Ariel Futoransky, Gerardo Richarte, Jose Orlicki, Damian Saura, Ernesto Alvarez and Fernando Russ.

Matías Blanco, Federico Mutti and myself have successfully implemented a XSS agent on CORE IMPACT.

References

[BM09] Matías Blanco and Federico Muttis. User input piercing for cross-site scripting. *OWASP Appsec 2009. Washington DC.*, 2009.

[Các02] Max Cáceres. Syscall proxying - simulating remote execution. *CoreLabs Technical Report*, 2002.

[CDL08] Rich Cannings, Himanshu Dwivedi, and Zane Lackey. *Hacking exposed. Web 2.0.* McGraw Hill, 2008.

[GNU06] GNU Citizen. JavaScript Port Scanner, 2006. `http://www.gnucitizen.org/projects/javascript-port-scanner/`.

[Jer06] Jeremiah Grossman. I know where you've been, 2006. `http://jeremiahgrossman.blogspot.com/2006/08/i-know-where-youve-been.html`.

[Mav] Ferruh Mavituna. XSS tunnelling. tunnelling http traffic through xss channels.

[Ric08] Gerardo Richarte. The evolution of penetration testing, 2005 to 2013 (keynote address). *SANS 2008. Las Vegas*, 2008.

[Rob08] Robert Auger. The Cross-Site Request Forgery (CSRF/XSRF) FAQ, 2008. `http://www.cgisecurity.com/articles/csrf-faq.shtml` [Online; accessed 14-February-2008].

[RT07] Fernando Russ and Diego Tiscornia. Zombie 2.0. *HackLu. October 18-20, 2007. Luxembourg.*, 2007.

[Sub06] Subratam Biswas. Browser Security: Concepts and Terms, 2006. `http://technet.microsoft.com/en-us/library/cc512657.aspx`.

[Wad06] Wade Alcorn. BeEF, 2006. `http://www.bindshell.net/tools/beef`.