Scanning Ajax for XSS entry points

Shreeraj Shah
Founder, Net Square
shreeraj@net-square.com

Scanning Ajax component for XSS entry points

Introduction

The continuous adoption of Web 2.0 architecture for web applications is instrumental in Ajax, Web services and Flash, emerging as key components. Ajax is a combination of technologies such as JavaScript with the XMLHttpRequest object, DOM and XML streams. Cross site scripting (XSS) can make browsers vulnerable to critical information hijacking if exploited with malicious intent. XSS is already categorized as persistent [1], non-persistent [1] and DOM-based [2]. Ajax code loaded in browser can have entry points to XSS and it is the job of the security analyst to identify these entry points. It is difficult to decisively conclude that possible entry points to an application can be exploited. One may need to do a trace or debug to measure the risk of these entry points. This paper introduces you to a quick way to identify XSS entry points in an application.

Application scenario

An AJAX-enabled application is hosted on a Web server, with the Ajax functions residing in .js files (for example, prototype.js). These files get included when the HTML page is loaded in the browser. Target applications may have several functions in multiple .js files or libraries. An Ajax function would have the following key elements residing in a single function call or multiple function calls depending on how the application has been designed.

- An XMLHttpRequest call to communicate to the backend application as shown in Figure 1.0. The objective of this call is to fetch information by opening an asynchronous socket from the browser.
- In another scenario, the XMLHttpRequest object and browser policy do not support cross-domain or cross-site calls but such functionality is required. The application is designed a way that allows users to provide RSS addresses to fetch news, blogs, videos. At the same time, the application uses third-party Web Services or Open APIs to gain access to information. All this information cannot be accessed from the browser. As a solution, a cross-site proxy is opened up on the server side. (Figure 1.0)
- In yet another scenario, JavaScript can manipulate the DOM on-the-fly once a response is received by XMLHttpRequest or it can *eval()* certain parts of the information based on the streams being received. (Figure 1.0)

Possible XSS entry points

An Ajax function that fetches information from the backend may either use the DOM or have a small JavaScript routine to update the browser. This update process can have a possible XSS entry point. The chances of exposing the browser to a potential XSS attack are compounded if for example, `document.write()` can be used to initiate a DOM-based XSS [2] attack or the routine calls eval() to inject certain values to already defined variables. (Figure 1).

The Ajax routine that calls a cross-site proxy and fetches information from untrusted sources is highly vulnerable to an XSS attack. A user who uses an Ajax function to configure an RSS feed or uses Web Services APIs to fetch information without properly filtering the stream that originates from the proxy, is opening up the framework to the possibility of a successful XSS attack. Recommended measures include scanning the routine for XSS entry points and performing logic analysis prior to deploying the application on a production system.

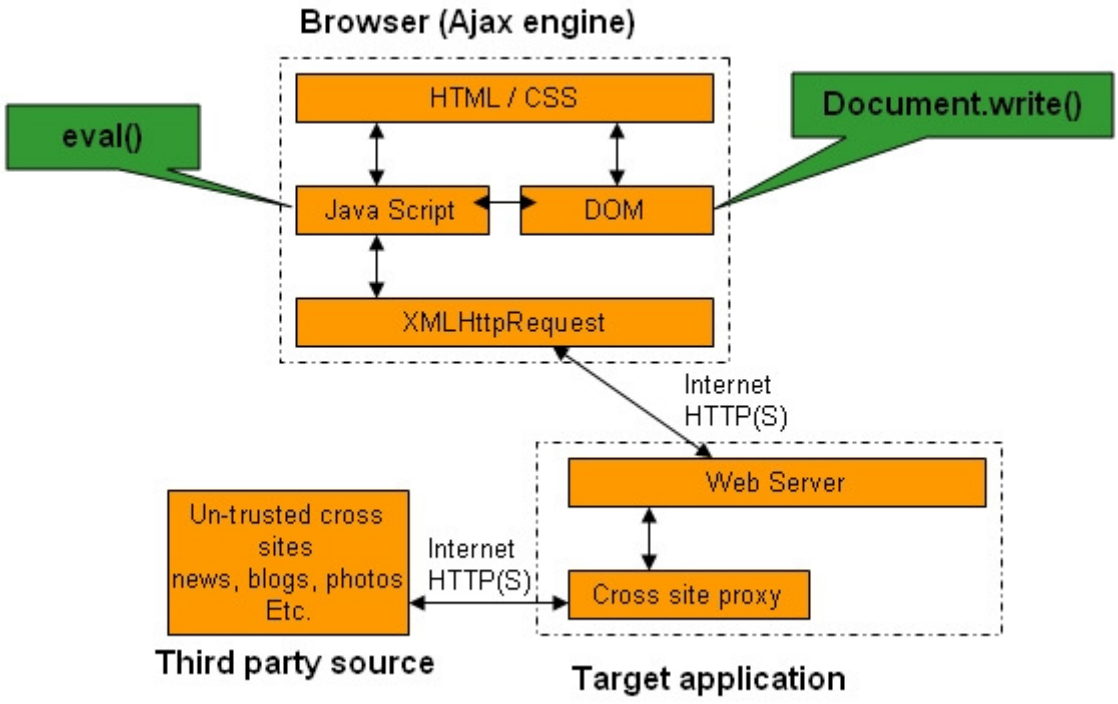

Figure 1.0 – Possible XSS entry points

Scanning XSS entry points

The objective here is to scan the target page or functionality for possible Ajax entry points. To do so one must perform the following tasks:

- Scanning the target page and collecting all dependent JavaScript (.js) files. These dependencies are a part of the <script> tag with "src"
- Fetching and scanning each of these JavaScript files from the server to identify functions.
- Looking for Ajax or XMLHttpRequest calls in each of these functions – of primary interest for security professionals.
- Grabbing certain *regex patterns* to help us identify potential XSS entry points.

This defines the scope and identifies high-value Ajax routines. With this information in place we can move to the next analysis phase. Exhibit 1 is a simple ruby script to achieve this. Running the script against a target such as http://example.com/custom.aspx provides the following output.

```
D:\scanajax> scanajax http://ajax.example.com/custom.aspx

---Scanning for scripts---
/src/process.js

---Enumerating javascripts---
http://ajax.example.com/src/process.js
-----------------------
[1]getnews <- function
-----------------------
[4]     if(window.XMLHttpRequest){
[5]         http = new XMLHttpRequest();
[31]                    eval(story);
-----------------------
[39]showblog <- function
-----------------------
[42]    if(window.XMLHttpRequest){
[43]        http = new XMLHttpRequest();
[69]                    document.write(blogdetail);
```

The "process.js" file that is fetched from the server contains two Ajax-based functions and uses an instance of the XMLHttpRequest object. Grab the two functions named "getnews" and "showblog". These functions may be fetching RSS feeds or APIs via cross-site proxy. Lines 31 and 69 are possible XSS entry points. One needs to analyze the script between lines 5 and 31 to check the *eval* call and how the "story" variable is constructed. At the same time the lines between 43 and 69 contain code to create the "blogdetail" variable that also needs to be analyzed. Assuming the stream originates from an untrusted source and the variables are constructed using this unsanitized content, it is possible to inject malicious content. This content may get executed in the browser and can lead to an XSS attack.

Analyzing XSS entry points

Analysis of XSS entry points can be done using the following two methods.

1. Debugging the JavaScript code – a JavaScript debugger such as Firebug or Venkman can be used. The complete explanation is available in the article titled "Hacking Web 2.0 Applications with Firefox" [3]
2. Tracing the call – Another way of analyzing xss entry points is to trace or reverse trace the JavaScript call to its source and in the process determine the possibility of XSS exploitation. The process is explained in the article titled "Detecting Web Application Security Vulnerabilities" [4]

The preceding process is difficult to automate completely because human intelligence is needed to analyze the call thoroughly. Developers have plenty of ways available for implementation. To

analyze each in automated fashion is a bit tricky. However, tools can be of help in reverse engineering via tracing or debugging.

Conclusion

Determining Ajax calls in Web 2.0 applications is important for tracking some of the vulnerabilities. The approach explained in this article can be taken as a starting point. Other methods can be followed to measure browser-side security for end clients. We are witnessing an increasing number of browser-side attacks in the form of worms and viruses. It is important to identify entry points and perform a thorough analysis for better threat modeling for next generation web applications.

References:

[1] Cross-site scripting (http://www.webappsec.org/projects/threat/classes/cross-site_scripting.shtml)

[2] XSS of third kind (http://www.webappsec.org/projects/articles/071105.html)

[3] Hacking Web 2.0 Applications with Firefox (http://www.securityfocus.com/infocus/1879)

[4] Detecting Web Application Security Vulnerabilities (http://www.oreillynet.com/pub/a/sysadmin/2006/11/02/webapp_security_scans.html)

Exhibit 1 – scanajax.rb

Note: Download the most recent version of this script from http://net-square.com/

```
# Ajax scanning script
# Author: Shreeraj Shah (shreeraj@net-square.com)

require 'open-uri'
require 'rexml/document'
require 'uri'
include REXML

if (ARGV.length != 1)
  puts "\nUsage:\n"
  puts "scanajax.rb <Target URL>\n"
  puts "Example:\n"
  puts "scanajax.rb http://digg.com\n"
  Kernel.exit(1)
end

# Grabbing the target
url = ARGV[0]
html = open(url)
page = html.read

# Path manipulation - Supporting absolute and starting with "/"
# More work to be done on this conversion

b_uri=html.base_uri
abspath=b_uri.scheme+"://"+b_uri.host
relpath =abspath

# Variables for  processing
all_path = ""
scriptname = []
scriptcontent = []
sn=0

# Scanning for script in the target
puts "\n---Scanning for scripts---"
a_script=page.scan(/<script.*?>/)
a_script.each do |temp|
  if(temp.scan("src").length > 0)
    temp += "</script>"
    doc=Document.new temp
    root = doc.root
    all_path += root.attributes["src"]+"|"
    puts root.attributes["src"]
  end
end

# Collecting all src files
puts "\n---Enumerating javascripts---"
a_path=all_path.split("|")
a_path.each do |temp|
```

```
      uri=URI.parse(temp)
      if(uri.absolute)
       tpage=open(temp)
       scriptname.push(temp)
       scriptcontent.push(tpage.read)
      else
       if(/^\//.match(temp))
         turi=abspath+temp
         tpage=open(turi)
         scriptname.push(turi)
         scriptcontent.push(tpage.read)
       else
         turi=relpath+"/"+temp # More on this later
         tpage=open(turi)
         scriptname.push(temp)
         scriptcontent.push(tpage.read)
       end
      end
    end

  # Scanning for functions, Ajax calls and XSS entry points
  scriptname.each do |sname|
    puts sname
    p=scriptcontent[sn].split("\n")
    i=1
    p.each do |temp|
      # Grab function
      reg=Regexp.new(/function\s(.*?)\(/i)
      match=reg.match(temp)
      if match != nil
        puts "------------------------"
        puts "["+i.to_s+"]"+match[1]+" <- function"
        puts "------------------------"
      end
      # Grab XMLHttpRequest call (Ajax)
      reg=Regexp.new(/XMLHttpRequest/i)
      match=reg.match(temp)
      if match != nil
        puts "["+i.to_s+"]"+temp
      end
      # Grab eval entry point
      reg=Regexp.new(/eval\(.*?\)/i)
      match=reg.match(temp)
      if match != nil
        puts "["+i.to_s+"]"+temp
      end
      # Grab document.write entry point
      reg=Regexp.new(/document.write\(.*?\)/i)
      match=reg.match(temp)
      if match != nil
        puts "["+i.to_s+"]"+temp
      end
      i+=1
    end
    sn += 1
  end
```