Crawling Ajax-driven Web 2.0 Applications

Introduction

Crawling web applications is one of the key phases of automated web application scanning. The objective of crawling is to collect all possible resources from the server in order to automate vulnerability detection on each of these resources. A resource that is overlooked during this discovery phase can mean a failure to detect some vulnerabilities. The introduction of Ajax throws up new challenges [1] for the crawling engine. New ways of handling the crawling process are required as a result of these challenges. The objective of this paper is to use a practical approach to address this issue using rbNarcissus, Watir and Ruby .

Problem domain and new approach

Usually crawling engines are "protocol-driven" and open a socket connection on the target host or IP address and port. Once a connection is in place the crawler sends HTTP requests and tries to interpret responses. All these responses are parsed and resources are collected for future access. The resource parsing process is crucial and the crawler tries to collect possible sets of resources by fetching links, scripts, flash components and other significant data.

The "protocol-driven" approach does not work when the crawler comes across an Ajax embedded page. This is because all target resources are part of JavaScript code and are embedded in the DOM context. It is important to both understand and trigger this DOM-based activity. In the process, this has lead to another approach called "event-driven" crawling. It has following three key components

1. Javascript analysis and interpretation with linking to Ajax
2. DOM event handling and dispatching
3. Dynamic DOM content extraction

Solution with event-driven crawling

To perform event-driven crawling we need to use the browser context to both understand the DOM and fire possible events. Several tools and plugins can be used to automate browser access. *Watir* [2] will be used to demonstrate this concept. *Watir* is a Ruby port for the IE COM driver. It is possible to drive IE instance using Watir.

Let's take a target – http://ajax.example.com

Accessing this page produces the following HTML stream from the server,

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
<title>Dynamic site</title>
<script src="./src/master.js"></script>
<script>loadhtml()</script>
<div id='main'></div>
<div id='myarea'></div>
</body>
</html>
```

It is clear that "protocol-driven" crawlers would fail to fetch much information out of this page because the loading pages are loaded from within JavaScript code. As a first step, this JavaScript code needs to be grabbed and the methods that are using the XHR object need to be collected. **X**ML**H**ttp**R**equest is used by Ajax.

Analyzing JavaScript Code

As first step one needs to analyze JavaScript; in this case it is *master.js*. Parse this script to identify all possible functions along with XHR calls. Here is a simple way of doing it in Ruby – using rbNarcissus[3], the ruby port for Narcissus. Running a script against *master.js* using this library, produces the following information.

```
D:\crawl-ajax> jsparser.rb master.js
---- XHR call mapping ----
http.onreadystatechange
getQuote[XHR found]
getPrice
loadmyarea[XHR found]
loadhtml[XHR found]
--------------------------

---- Function mapping ----
http.onreadystatechange
getQuote
[+]http.onreadystatechange
getPrice
[+]getQuote
loadmyarea
[+]http.onreadystatechange
loadhtml
[+]http.onreadystatechange
--------------------------
```

Note: The *rbNarcissus* web site hosts the open source code that is available for download, along with an example script.

The preceding script clearly helps discover functions and their inter-relationship. *getQuote*, *loadmyarea* and *loadhtml* use the XHR object whereas *gePrice* calls getQuote as indicated in "Function mapping". With this information in place, events which call this function using the browser need to be triggered, so that the corresponding part of script gets executed in the browser.

Automating IE with Watir

The next step is to automate the browser with *Watir*. Several other tools can also be used to automate the browser. Ensure that they are capable enough to trigger events. To understand the process clearly let's use *irb* – the *interactive ruby prompt*. This process can be scripted too; however, using an interactive ruby prompt would aid in better understanding of the process. HTTP traffic also needs to be tapped to capture resources after events. This can be done by activating a proxy, sniffing the wire or through a browser plug-in. In this example, a proxy is used to tap HTTP traffic. With the *burp* proxy running and a simple Ruby *call,* generated traffic for a particular event can be grabbed from its log file. Let's see all the action.

```
D:\crawl-ajax> irb --simple-prompt
>> require 'watir'
=> true
>> include Watir
=> Object
>> require 'webtraffic'
=> true
>>
```

Watir is loaded and *include*d in the code. At the same time *webtraffic functions* are loaded to "see" the traffic going from the browser. Now let's start the browser from Watir.

```
>> ie=IE.new
```

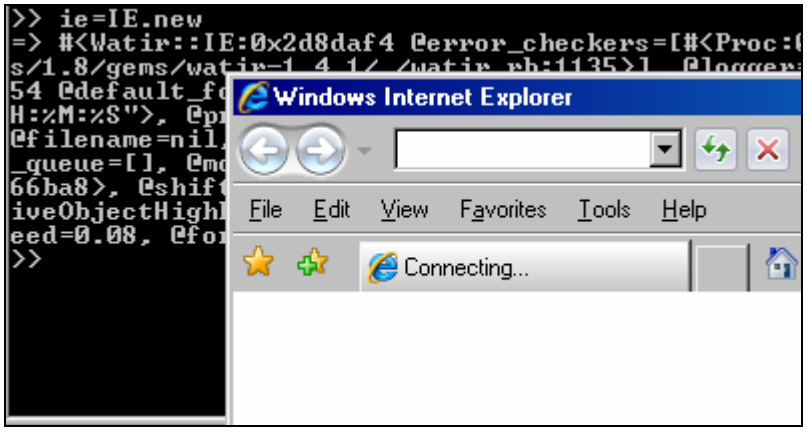

Figure 1.0 – Watir starts an IE instance

Here, *Watir's* IE class has started a fresh instance of IE. Let us start playing around with the target. The following command will load *http://ajax.example.com*

```
>> ie.goto("http://ajax.example.com")
=> 4.847
```

The following page is loaded in the browser,

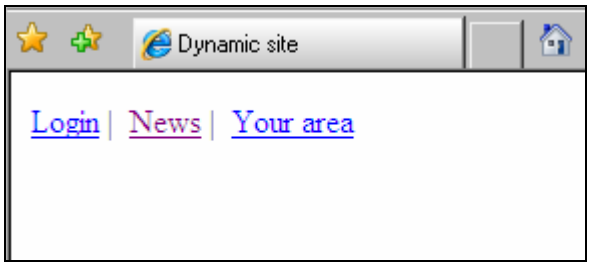

Figure 2 – The first page is loaded

Now "view" the traffic generated from the browser using "*showreq*". Alternatively, traffic can also be collected from a sniffer. The advantage of collecting traffic using this program is clear: it can be linked to the event.

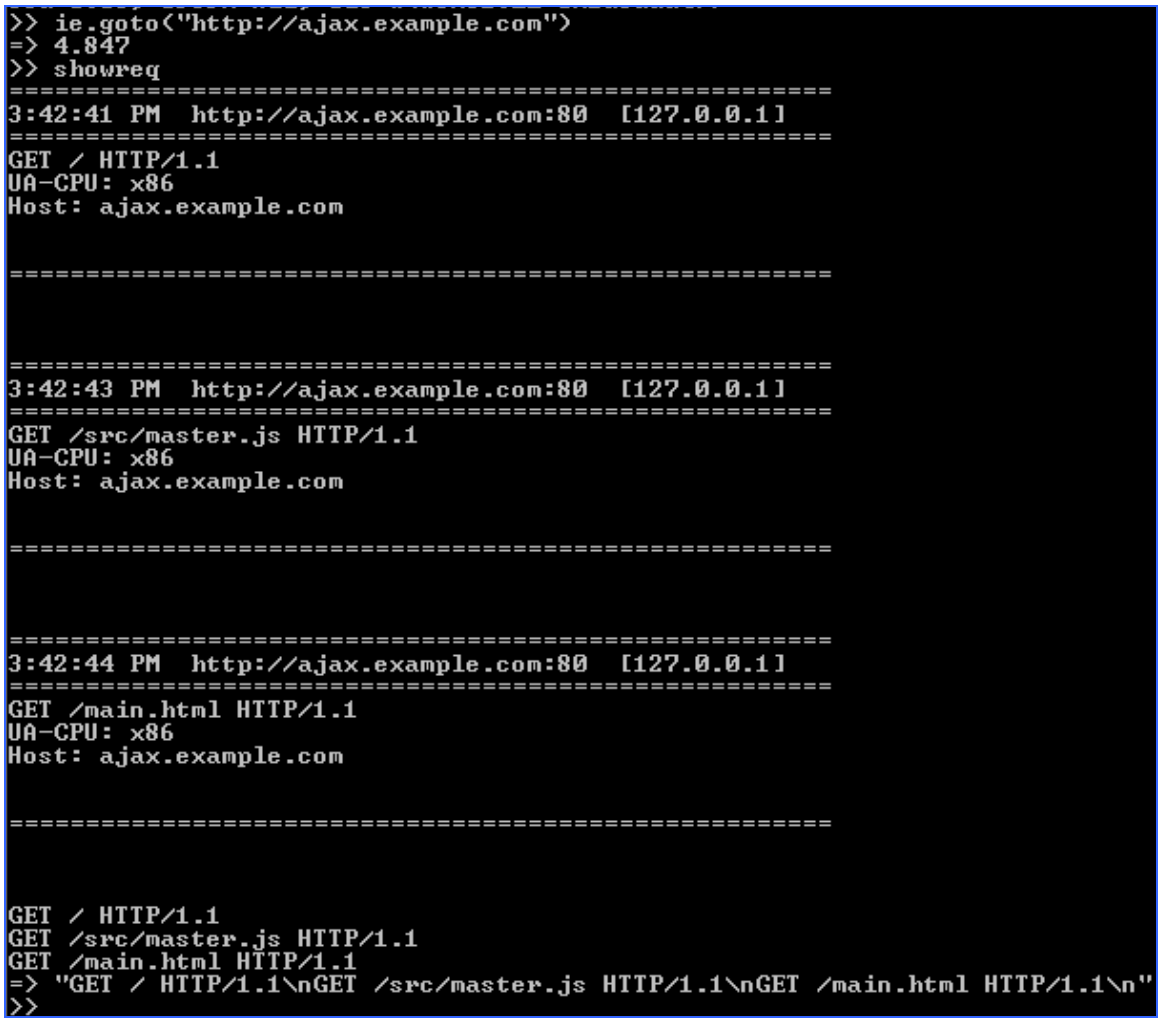

Figure 3.0 – Traffic for page 1

The preceding three requests are sent and the page *main.html* is loaded in the browser. Use the following method to see the links on the page after DOM refreshing –

```
>> ie.show_links
index name        id          href
          text/src
1                             http://ajax.example.com/login.asp
          Login
2                             http://ajax.example.com/news.asp
          News
3                             javascript:loadmyarea()
          Your area
=> nil
>>
```

There are three links. One of the three links calls JavaScript code. A complete HTML around the link can be obtained to harvest *onClick* or *mouseover* events as well.

```
>> ie.links[3].html
=> "<A href=\"javascript:loadmyarea()\">Your area</A>"
>> ie.links[3].href
=> "javascript:loadmyarea()"
```

Now it is possible to generate a “click” event for the link automatically as shown below:

```
>> ie.links[3].click
=> ""
>>
```

The state of the browser is now changed. New traffic is also generated as shown below:

Figure 4.0 – Request from XHR

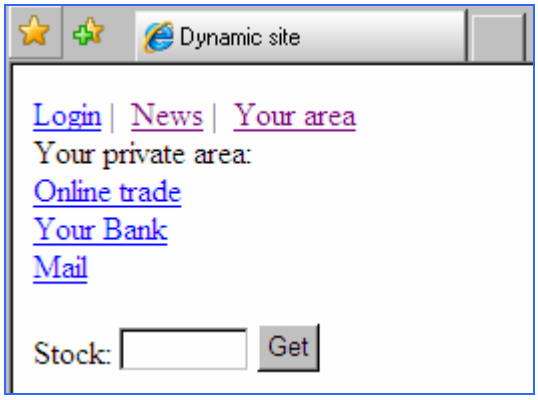

Figure 5.0 – “Your area” is clicked

It is clear that we have a new request with XHR, generated on wire for “myarea.asp”. The page is loaded in the browser.

These are the total number of links on the page now:

```
>> ie.show_links
index name         id          href
          text/src
1                              http://ajax.example.com/login.asp
          Login
2                              http://ajax.example.com/news.asp
          News
3                              javascript:loadmyarea()
          Your area
4                              http://ajax.example.com/trade.asp
          Online trade
5                              http://ajax.example.com/bank.asp
          Your Bank
6                              http://ajax.example.com/mail.asp
          Mail
=> nil
>>
```

Next, analyze buttons and generate events as shown below:

```
>> ie.buttons.length
=> 1
>> ie.buttons[1].html
=> "<INPUT onclick=getPrice(this.form) type=button value=Get name=button>"
>> ie.buttons[1].click
=> ""
>>
```

It is clear that "button" also calls *getPrice* that internally calls *getQuote* which in turn makes an XHR call. Click it to fire the XHR call. Here is how the browser would look like after the "click".

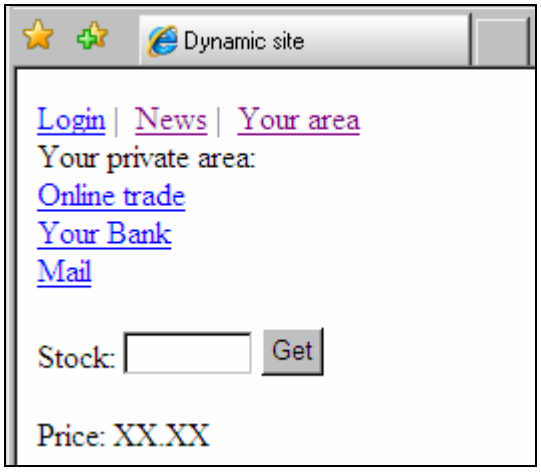

Figure 6.0 – "Button" is clicked and "Price" is obtained

The traffic for this event is shown in Figure 7.0.

Figure 7.0 – Button click created XML call on the wire

The traffic on the wire, as illustrated in Figure 7.0, shows that the event has sent an XML stream to the resource. Access to another resource is now available. It is also possible to do XML fuzzing on this resource.

All links are collected and traffic analyzed along with "firing" events. This page is completely analyzed. The following list of resources resides on the server:

/src/master.js
/login.asp
/news.asp
/trade.asp
/bank.asp
/mail.asp
/myarea.asp
/myquote.asp **[XML call]**

This way crawling can be done using a combination of "protocol-driven" and "event-driven" approaches. This dual approach can help in extracting the maximum information in an automated fashion. Given that everything is done using Ruby scripts, a recursive crawling mechanism can be built to provide a thorough analysis of each and every page. *Ruby with Watir* can make automated crawling easy and possible.

Conclusion

Times are changing for the web application assessment space with the introduction of Web 2.0 technological components like Ajax, Flash and SOA. To keep pace with these changing times, it is imperative to put new methods, tools and techniques into action while still keeping old jewels. The concept discussed in this paper can help in putting forth automated methods for Ajax-driven applications.

References

[1] Past articles
- Vulnerability Scanning Web 2.0 Client-Side Components [http://www.securityfocus.com/infocus/1881]
- Hacking Web 2.0 Applications with Firefox [http://www.securityfocus.com/infocus/1879]

[2] Watir - http://wtr.rubyforge.org/ [Tutorials,Documents and Build]
[3] rbNarcissus - [http://idontsmoke.co.uk/2005/rbnarcissus/]