

Hacking Browser's DOM - Exploiting Ajax and RIA

Abstract

Web 2.0 applications today make extensive use of dynamic Document Object Model (DOM) manipulations for rendering JSON or XML streams in browsers. These DOM calls along with XMLHttpRequest (XHR) objects are part of the client side logic, either written in JavaScript or in other client side technologies of the likes of Flash or Silverlight. In this scenario, DOM driven Cross Site Scripting (XSS) is a sleeping giant in many application codes and this can be exploited by an attacker to gain access to the end user's browser/desktop. This in turn can become the root cause for another set of interesting vulnerabilities such as Cross Widget Sniffing, RSS Feed reader Exploitation, XHR Response Stealing, Mashup Hacking, Malicious Code Injection, Worm Spreading et cetera. These vulnerabilities need an innovative way of scanning the application and the corresponding standardized methodology needs to be tweaked to suit these. We have seen DOM driven XSS exploited in various different popular portals to spread worms or virus. This is a significant threat that is rising on the horizon but could be mitigated by validating un-trusted content that can potentially poison AJAX or Flash routines. DOM driven XSS, Cross Domain Bypass and Cross Site Request Forgery (CSRF) can together form a deadly cocktail to exploit Web 2.0 applications all over the Internet. In view of all this, this paper is designed to cover the following important issues and concepts:

- Web 2.0 Architecture and DOM manipulation points
- JavaScript exploits by leveraging DOM
- Cross Domain Bypass and Hacks
- DOM hacking for controlling Widgets and Mashups
- Exploiting Ajax routines to gain feed readers
- Scanning and detecting DOM driven XSS in Web 2.0
- Tools for scanning the DOM calls
- Mitigation strategies for a better security posture

The author is going to present this paper along with demos at the Blackhat USA 2010.

Author:

Shreeraj Shah (Founder and Director, Blueinfy Solutions)

Credit for Tools development - Nrupin Sukhadia (Developer, Blueinfy Solutions) and Rishita Anubhai (Web Security Researcher, Blueinfy Solutions)

Understanding the Web 2.0 Architecture and Stream Structures

Web 2.0 applications run in a widely distributed environment. These applications have an integrated boundary structure but are complex in nature with many dependencies on fellow applications. This allows an attacker to easily discover and exploit architectural holes on the application layer. The structure, as shown in *figure 1*, draws attention to how the application runs with a very thin layer of its own components.

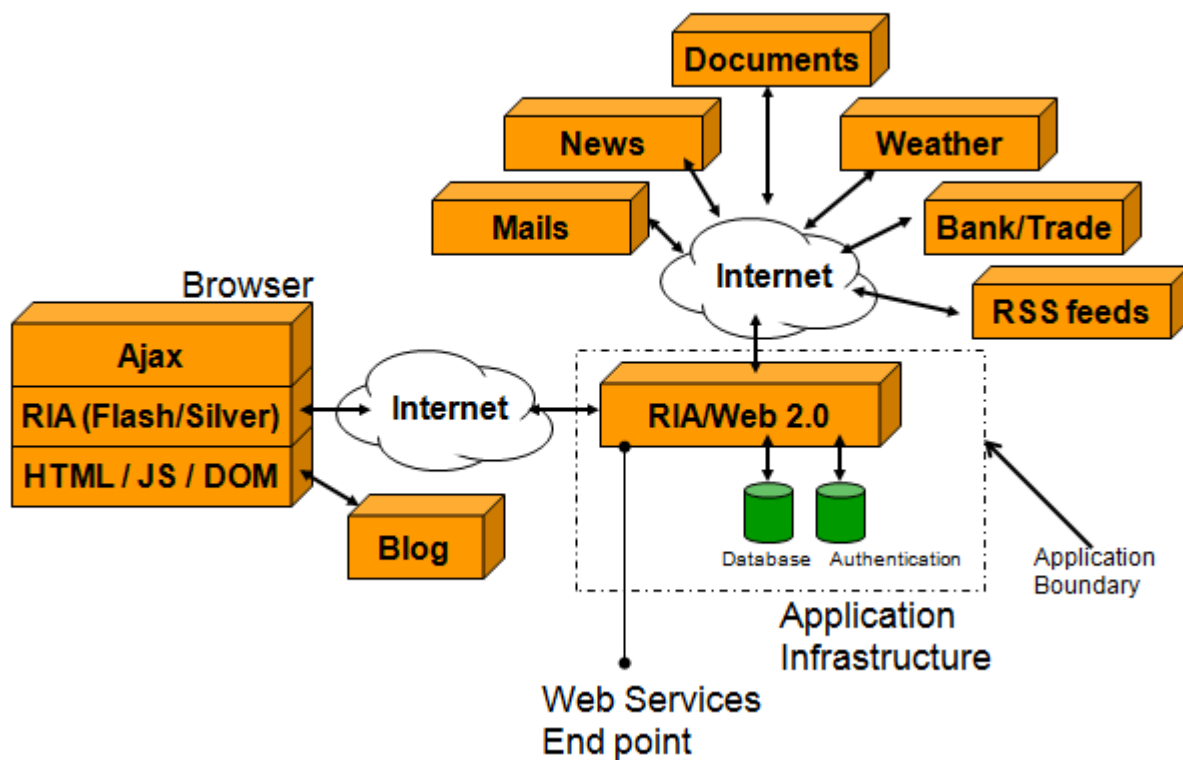


Figure 1 – Web 2.0/RIA Architecture

As shown by the architecture above, Web 2.0 applications are more like “applications of applications” i.e. they are networked with other applications and sources of information. In this case, the target application may allow data to come from other sources of information such as mails, news, documents, weather, banks etc. The application itself may have its own little database and authentication module but other than that most of the data tends to come from outside the application boundary. At the same time, on the browser stack, critical components like Ajax, Flash and Silverlight get leveraged. Interestingly in such a scenario, it is possible to set up a callback mechanism on a cross domain, directly from the browser where the application is not involved in streaming data, but running on its own domain (DOM context)

The above architecture consumes various data streams not being restricted to any one stream in particular like the name-value pair (<http://yahoo.com/index.php?id=5&location=home>). Both, the

application and the browser can send information across in various structures like JSON, XML, JS-Object, JS-Array etc. (as shown in *figure 2*)

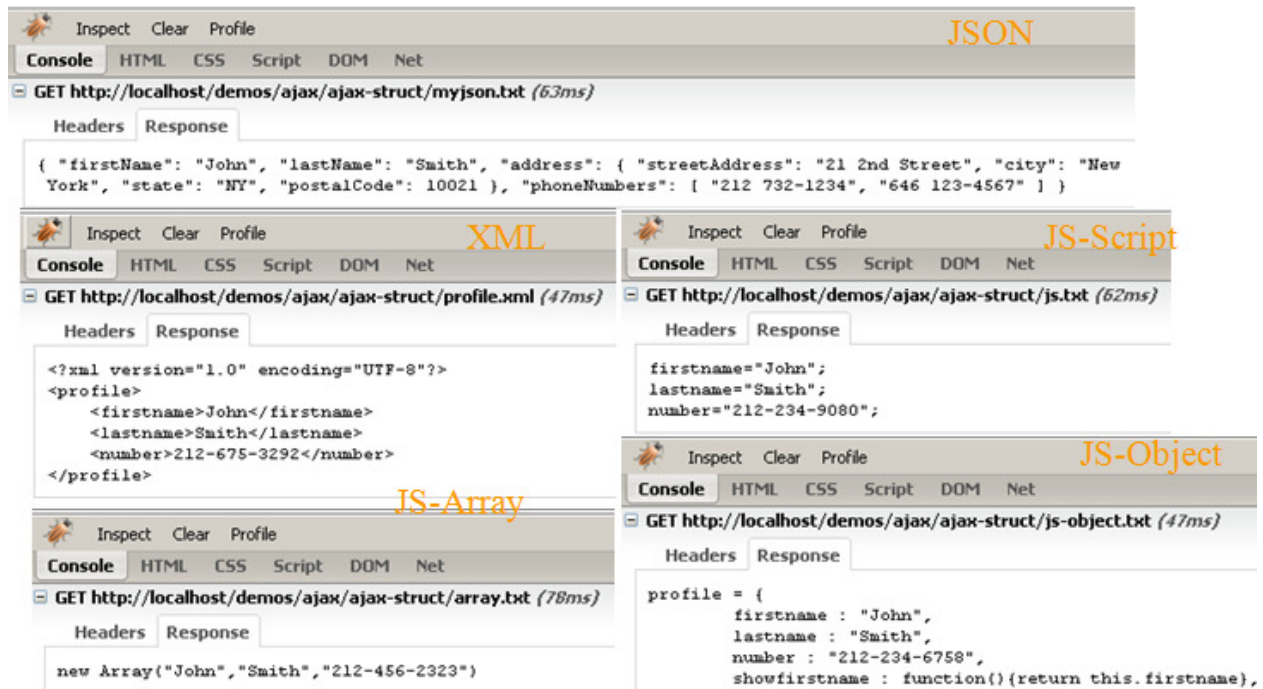


Figure 2 – Different structures for information exchange

Moreover, the protocols involved in transferring some of these structures could be different although majority of the transfers are over HTTP(S). Some of these protocols are like SOAP, XML-RPC or REST.

Hence the complex architecture, porous boundary conditions, involvement of different stream structures, confusing protocols and the vulnerable implementation of client side components can allow attackers to exploit such applications comparatively with more ease.

As the application security dynamics continue to change over the years, on parallel fronts we see some newer and lethal client side attacks surfacing as well. *Figure 3* shows the changing OWASP Top 10 Scenario.

New Top Ten 2004	OWASP Top 10 – 2007 (Previous)	OWASP Top 10 – 2010 (New)
A1 Unvalidated Input	A2 – Injection Flaws	A1 – Injection
A2 Broken Access Control	A1 – Cross Site Scripting (XSS)	A2 – Cross Site Scripting (XSS)
A3 Broken Authentication and Session Management	A7 – Broken Authentication and Session Management	A3 – Broken Authentication and Session Management
A4 Cross Site Scripting (XSS) Flaws	A4 – Insecure Direct Object Reference	A4 – Insecure Direct Object References
A5 Buffer Overflows	A5 – Cross Site Request Forgery (CSRF)	A5 – Cross Site Request Forgery (CSRF)
A6 Injection Flaws	<was T10 2004 A10 – Insecure Configuration Management>	A6 – Security Misconfiguration (NEW)
A7 Improper Error Handling	A10 – Failure to Restrict URL Access	A7 – Failure to Restrict URL Access
A8 Insecure Storage	<not in T10 2007>	A8 – Unvalidated Redirects and Forwards (NEW)
A9 Denial of Service	A8 – Insecure Cryptographic Storage	A9 – Insecure Cryptographic Storage
A10 Insecure Configuration Management	A9 – Insecure Communications	A10 – Insufficient Transport Layer Protection
	A3 – Malicious File Execution	<dropped from T10 2010>
	A6 – Information Leakage and Improper Error Handling	<dropped from T10 2010>

Figure 3 – OWASP Top 10 over years and changes

To name a few noteworthy ones in this context, we have client side attacks involved with Cross Site Scripting, Broken authentication and authorization (client side bypass), CSRF, Unvalidated Redirects and Forwards as well as some minor ones in other sections.

Extensive usage of DOM in Web 2.0 applications

The Document Object Model (DOM) of a browser is becoming a critical part of these next generation Web 2.0 applications. DOM is used by developers and third party libraries extensively. One of the most significant features of the Web 2.0 applications is ‘minimal amount of reload and refresh’. As a result, these applications leverage DOM and Web APIs associated with it to update the pre-loaded DOM. Again, DOM is usually loaded only once when the browser loads the HTML page and thereafter during the rest of the communication with the application DOM remains “in-memory”. Hence, all the new information needs to be injected into the pre-loaded DOM. To achieve this objective, dynamic DOM manipulation is needed which is accomplished by the use of calls like eval() and other calls of the likes of eval().

For example,

```
function getProfile()
{
    var http;
    if(window.XMLHttpRequest){
        http = new XMLHttpRequest();
    }else if (window.ActiveXObject){
        http=new ActiveXObject("Msxml2.XMLHTTP");
        if (! http){
            http=new ActiveXObject("Microsoft.XMLHTTP");
        }
    }
}
```

```

    }
  }
  http.open("GET", "./profile/", true);
  http.onreadystatechange = function()
  {
    if (http.readyState == 4) {
      response = http.responseText;
      document.getElementById('main').innerHTML = response;
    }
  }
}
http.send(null);
}

```

In the above function, an XHR call is made from the pre-loaded DOM and then the new content is changed by “innerHTML” call.

There are various ways to inject new content into the DOM. Another way of rewriting the DOM is shown in the following snippet:

```

if (http.readyState == 4) {
  var response = http.responseText;
  var p = eval("(" + response + ")");
  document.open();
  document.write(p.firstName+"<br>");
  document.write(p.lastName+"<br>");
  document.write(p.phoneNumbers[0]);
  document.close();
}

```

In this case, new content is injected using *eval* and *document.write* calls to the DOM. It is easy to see that there are a large set of DOM calls on these lines that can be utilized by client side scripts. Here is a small list of it:

```

document.write(...)
document.writeln(...)
document.body.innerHTML=...
document.forms[0].action=...
document.attachEvent(...)
document.create(...)
document.execCommand(...)
document.body. ...
window.attachEvent(...)
document.location=...
document.location.hostname=...
document.location.replace(...)
document.location.assign(...)

```

```
document.URL=...  
window.navigate(...)  
document.open(...)  
window.open(...)  
window.location.href=...  
eval(...)  
window.execScript(...)  
window.setInterval(...)  
window.setTimeout(...)
```

One can look at the script utilized by Web 2.0 applications and figure out these calls. It is important how these calls are implemented and what information they consume. These calls form an integral part of the attack surface as they end up providing entry points to many attackers.

On similar lines, DOM can be used by RIA written in Flash or Silverlight as well. It is possible to allow a flash file to access HTML tags and processes at runtime. There are various different ways of doing this.

For example,

Setting access to HTML content and DOM from flash

```
_root.htmlText = true;
```

Accessing the parameter and then further passing it on to the function 'getURL'

```
getURL(_root.input);
```

One can use the HTMLLoader to get access of HTML context from Flash and manipulate the DOM subsequently. Similarly with Silverlight, one can access the HTML DOM by the following object:

```
HtmlDocument doc = HtmlPage.Document;
```

Looking at these implementations one can gauge the severity of the issue from the security perspective. If Web 2.0 or RIA calls are not implemented securely, they lead to entry points for DOM injections and remote script executions. There is ample opportunity for misuse in these cases.

DOM based XSS – a sleeping giant is still a giant

DOM based XSS is very common with Web 2.0 applications and it cannot be determined by scanning or sending just a few requests to the application. It calls for a deeper inspection of the source and the other information sources to conclude the existence of this one and some other similar vulnerabilities.

For example, here is a scenario:

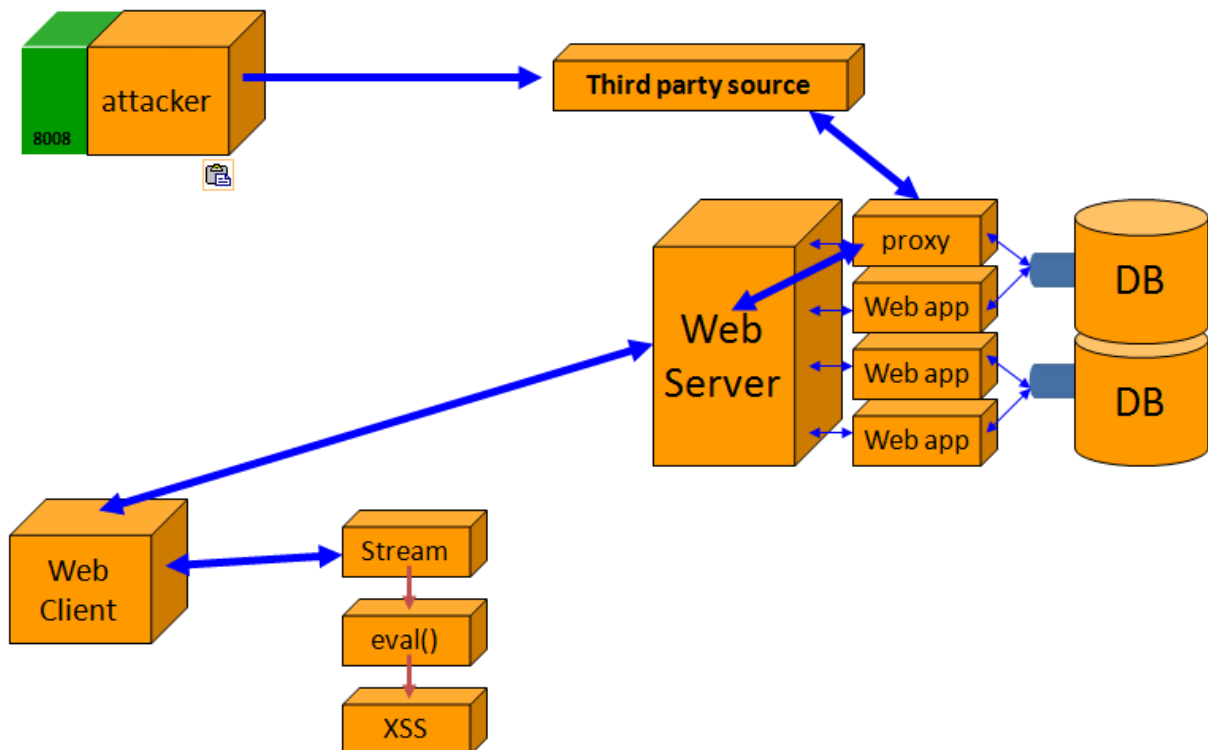


Figure 4 – DOM based XSS

In this scenario the client is fetching a third party stream coming in, say JSON, and ‘eval-ing’ the call without validating the content. An attacker could post the malicious code on a third party site, say a blog or news website. This site would convert the stream into JSON and whenever someone was to ask for it, the stream would be provided via an API. So now, when the target application asks for it, the code tunnels to the victim’s browser and the routine ‘eval’s it. As soon as an ‘eval’ call is executed, it allows malicious code to run in the browser’s current context and causes an XSS!

This one presented above is just one scenario and there are many other cases of a similar nature where DOM manipulating routines can be vulnerable to XSS. Here is another simple scenario.

In the function below, a client side routine takes a parameter coming from the URL and processes it.

```
function querySt(ji) {
    hu = window.location.search.substr(1);
    gy = hu.split("&");
    for (i=0;i<gy.length;i++) {
        ft = gy[i].split("=");
        if (ft[0] == ji) {
```

```
    return ft[1].toString();  
}
```

Now, this is called with the parameter 'pid' and the result passed to the 'eval' call subsequently.

```
var koko = querySt("pid");  
if(koko != null)  
{  
    eval('getProduct('+ koko.toString()+')');  
}
```

Now what if someone has framed the URL used in the above process as shown below:

[http://192.168.81.50/catalog.aspx?pid=3\);alert\('hi'\)//](http://192.168.81.50/catalog.aspx?pid=3);alert('hi')//)

This will execute the alert statement and cause XSS. It is completely possible to pass any script to the browser in this manner and the browser will execute it ignorant of the danger this loophole poses. This is hence a clear case of passing raw script to the function. We will look at variants of such a case in subsequent sections.

Cross Domain bypass and Security Issues

Browser security model and XHR calls will not allow bypassing SOP (Same Origin Policy). Hence, it is not possible to make an XHR call to xyz.com, by being on abc.com. Applications are designed in such a way that they need to make these type of calls and bypass at some points. Therefore, developers either put a proxy on the application or, if possible, leverage callback mechanism.

For example, as shown in *figure 4*, the client side may use a proxy component as follows:

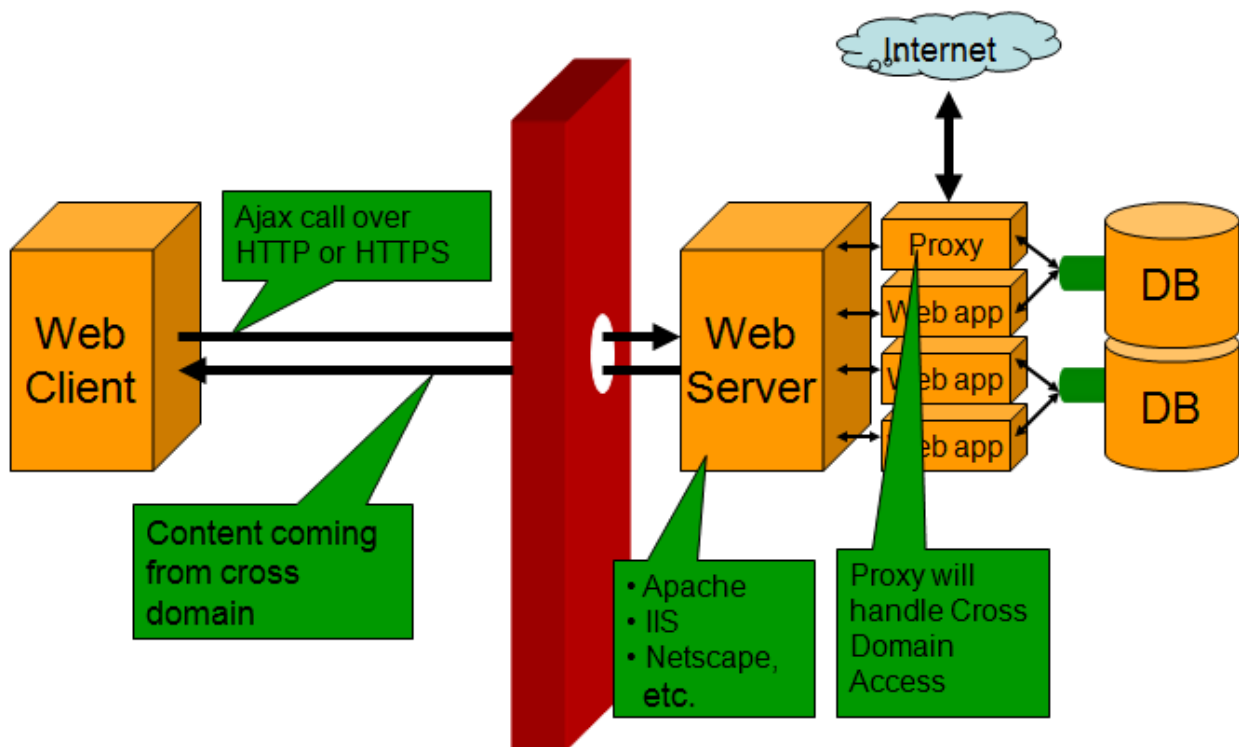


Figure 5 – Cross Domain Call from Proxy

Hence, when the client makes a request for cross domain content it goes via the proxy and gets the stream as shown in *figure 6* (JSON)



Figure 6 – Stream coming via proxy.aspx

Another way to bypass the cross domain call is by directly pushing content to the JavaScript function by wrapping the right callback function name.

For example,

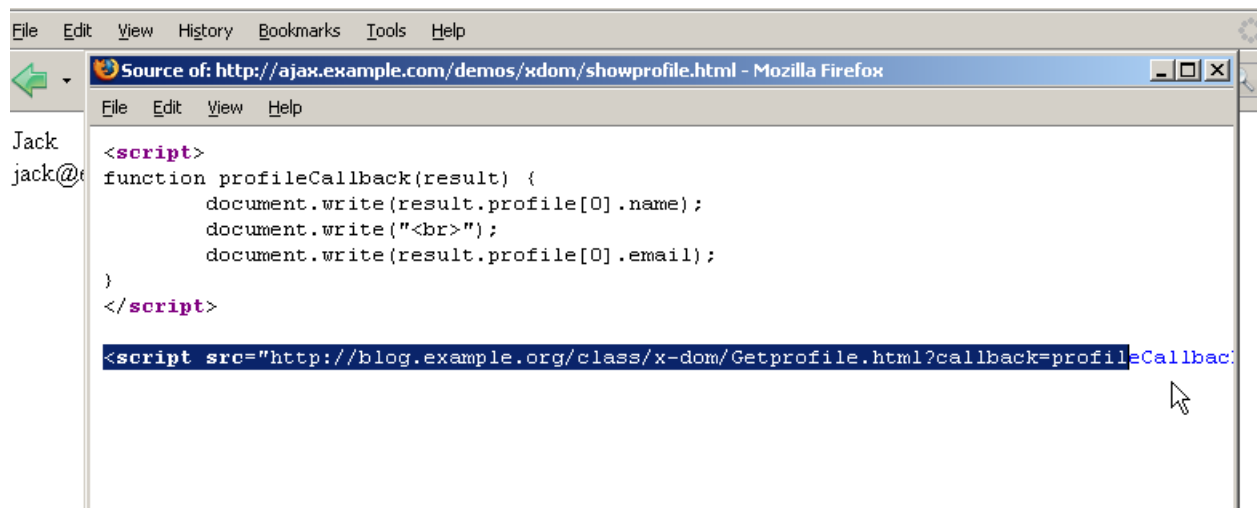


Figure 7 – Callback function for profile

As shown in *figure 7*, we have a `profileCallback` function that gets called as soon as we get the results back. Script tag is where the target URL has been mentioned. Hence, it will make the cross domain call and thereafter the incoming JSON or any other JS stream will automatically kick the function.

In future with the emergence of DOM 2, XHR level 2 and other HTML 5 tags other types of bypass may also be allowed. In this manner, the methods demonstrated above can be leveraged to perform cross domain XSS as well as to establish hidden one way channels as and when required.

Stealing DOM's key variables

Ajax applications use a single DOM throughout the life cycle of the application. At the start of the life cycle of an application, the DOM gets loaded and thereafter the DOM values get injected dynamically by the use of various different streams. Specifically, this dynamic DOM manipulation is caused to occur with the use of 'eval' or 'document.*' calls. It is evident that the use of JavaScript is extensive and consequently such dynamic DOM manipulations allow accessing poorly written JavaScript variables which in turn may involve some key values.

For example, the following is a simple authentication routine that accepts username and password from the browser and sends it to the server using an XHR object.

```

function getLogin()
{
    gb = gb+1;
    var user = document.frmlogin.txtuser.value;
    var pwd = document.frmlogin.txtpwd.value;
    var xmlhttp=false;
    try { xmlhttp = new ActiveXObject("Msxml2.XMLHTTP");
    }
    catch (e)
    { try
        { xmlhttp = new ActiveXObject("Microsoft.XMLHTTP"); }
        catch (E) { xmlhttp = false; }
    }

    if (!xmlhttp && typeof XMLHttpRequest!='undefined')
        { xmlhttp = new XMLHttpRequest(); }

    temp = "login.do?user="+user+"&pwd="+pwd;
    xmlhttp.open("GET",temp,true);

    xmlhttp.onreadystatechange=function()
        { if(xmlhttp.readyState == 4 && xmlhttp.status == 200)
            {
                document.getElementById("main").innerHTML
= xmlhttp.responseText;
            }
        }

    xmlhttp.send(null);
}

```

In the above function, two important lines to be noted are:

```

temp = "login.do?user="+user+"&pwd="+pwd;
xmlhttp.open("GET",temp,true);

```

The loophole here is that the ‘temp’ function has been created just by using ‘temp’ without preceding it with a keyword to limit its scope and lifetime, hence making its scope global. As a result of this, after calling this function ‘temp’ will remain live throughout the application. Hence, an attacker can steal this ‘temp’ variable via XSS and ‘steal’ the username and password

bringing it all down to DOM based XSS. This is a clear way of stealing variables from the current DOM context.

Attacking a Widget's DOM

Web 2.0 applications use the DOM itself for manipulating or injecting new content and modifying the existing DOM. For example, the following is an example of a simple function which reads RSS feed and injects it into the DOM. In this case, there is no validation in place and it is possible to inject a link into the DOM without much effort. The gravity of this possibility as a loophole can be gauged if we think of a case where we may inject a code snippet of the likes of 'javascript:alert(document.cookie)' over here!

```
function processRSS (divname, response) {
    var html = "";
    var doc = response.documentElement;
    var items = doc.getElementsByTagName('item');
    for (var i=0; i < items.length; i++) {
        var title = items[i].getElementsByTagName('title')[0];
        var link = items[i].getElementsByTagName('link')[0];
        html += "<a style='text-decoration:none' class='style2' href='"
            + link.firstChild.data
            + "'>"
            + title.firstChild.data
            + "</a><br><br>";
    }
    var target = document.getElementById(divname);
    target.innerHTML = html;
}
```

This widget would go out and grab the RSS feed thereafter passing it to this routine. It will then lead to a DOM based XSS, after which the script will get executed on the browser. It is possible to attack widgets using this type of a vector.

Cross Widget Access from the DOM

In many applications one can inject a Widget or Gadget. This apparently small HTML code, along with JavaScript, runs on the DOM and allows several operations to be performed. In some cases these Widgets share the same DOM or even a part of the DOM. This allows one Widget to access the important tags and variables of another Widget. In these cases, an attacker can force a malicious widget onto the DOM and monitor other widgets via this one.

For example, assume there is a Widget which accepts username and password from the user. Below is another simple Widget that can set up a trap on the former.

```

4  function regEvent4me()
5  {
6      var objs = document.getElementsByName("txtUser");
7      if (objs.length > 0)
8      {
9          var thefield = objs[0];
10         thefield.onblur = GetU;
11     }
12     objs = document.getElementsByName("txtPass");
13     if (objs.length > 0)
14     {
15         var thefield = objs[0];
16         thefield.onblur = GetP;
17     }
18 }

```

Figure 8 – Setting a trap

Hence the malicious widget is listening to a mouse event and as soon as username and password entries are done it can force the GetU and GetP function calls to execute. These functions would go ahead to steal the content and send it across the network where the attacker may be listening in anticipation. It is now evident how it is important to analyze the DOM architecture and usage when it comes to Widget platforms.

Other DOM based attacks with Web 2.0 Applications

We have seen some instances of DOM hacking so far and there exist yet more ways in which the DOM can be hacked and attacked, if JavaScript coding and usage is not done with due anticipation of the attackers. In due light of this, few more examples which we intend to cover during our presentations would be:

- **DOM hacking with Mashups** – Mashup applications use API and Open calls to fetch information from un-trusted sources and bundle them on the target domain application. This content is not verified or validated in many cases. An attacker may well be able to pass on a malicious stream to these Mashup applications and they in turn can pollute the DOM accordingly. Essentially, the stream gets injected in vulnerable calls and allows executing malicious script on the client end. For example, it may be aimed to pollute a profile residing on your site and inject values which get executed on a third party mashup.
- **DOM side logic hacking** – This is another interesting area in Web 2.0 applications. Web 2.0 applications use large amounts of client side logic and some of the business level logic tends to get shifted to the client side. This allows the attacker to dissect calls and understand the core logic behind the process, identify access controls, encryption logic, secret flags etc. All this logic being embedded in the client side DOM and in its stack of variables makes it easy to access and manipulate unauthorized content. For example,

accessing another user's banking information by analyzing client side call and injecting different values in the DOM.

- **CSRF with XML, SOAP, JSON or AMF streams** – It is also possible to inject dummy HTML forms which may force cross domain XML or SOAP requests from the browsers. This can cause Cross Site Request Forgery on poorly designed forms.
- **Cross Technology Access** – Browsers may run two different technology stacks and it allows one technology to access another one. For example, consider Flash and Ajax running on the same DOM. It is possible to access Flash or Silverlight variables from the DOM through JavaScript if the application is designed in a way permitting loopholes. It allows Cross Technology Access and can cause a security breach. An instance of this is the case of retrieving username and password variables from a Flash file via JavaScript through XSS.

DOM scanning and vulnerability detection

Detecting this set of aforementioned vulnerabilities would be difficult with simple blackbox testing i.e. it is not possible to merely try fuzzing and send request A, expecting a specific response B if a vulnerability exists. It needs static code analysis and some debugging tricks while the execution of the script is going on. Here are a few techniques to perform scanning:

- One can perform DOM scanning and look for various calls which are relevant and those that are using XHR. All the streams which are injected in the DOM should be analyzed and their impact needs to be thoroughly looked into. If the stream is seen to use calls like eval or document.write then one can exploit them. JavaScript scanning and static analysis is required to identify these vulnerabilities. (Appendix A – DOMScan)
- Another method is to employ debugging for JavaScript at run-time and analyze the flow of execution amongst all the components under a browser's DOM umbrella. There are powerful debuggers like Firebug or IE8's debugger tool that can be used for this purpose. Using a deobfuscator or a JavaScript tracer – one can deploy a technique to deobfuscate the script and look for all statements which are being executed while various events are being fired. This exposed execution surface of the call then helps in identifying possible loopholes or weaknesses. (Appendix A – DOMTracer)

Conclusion

It is imperative to analyze DOM management of the application with browser context. Poorly written applications can be attacked and hacked via DOM manipulation calls. There are new techniques, methodologies and tools required for this analysis. Just depending on automated tool will not solve the problem. We have already witnessed several DOM based attacks and exploits.

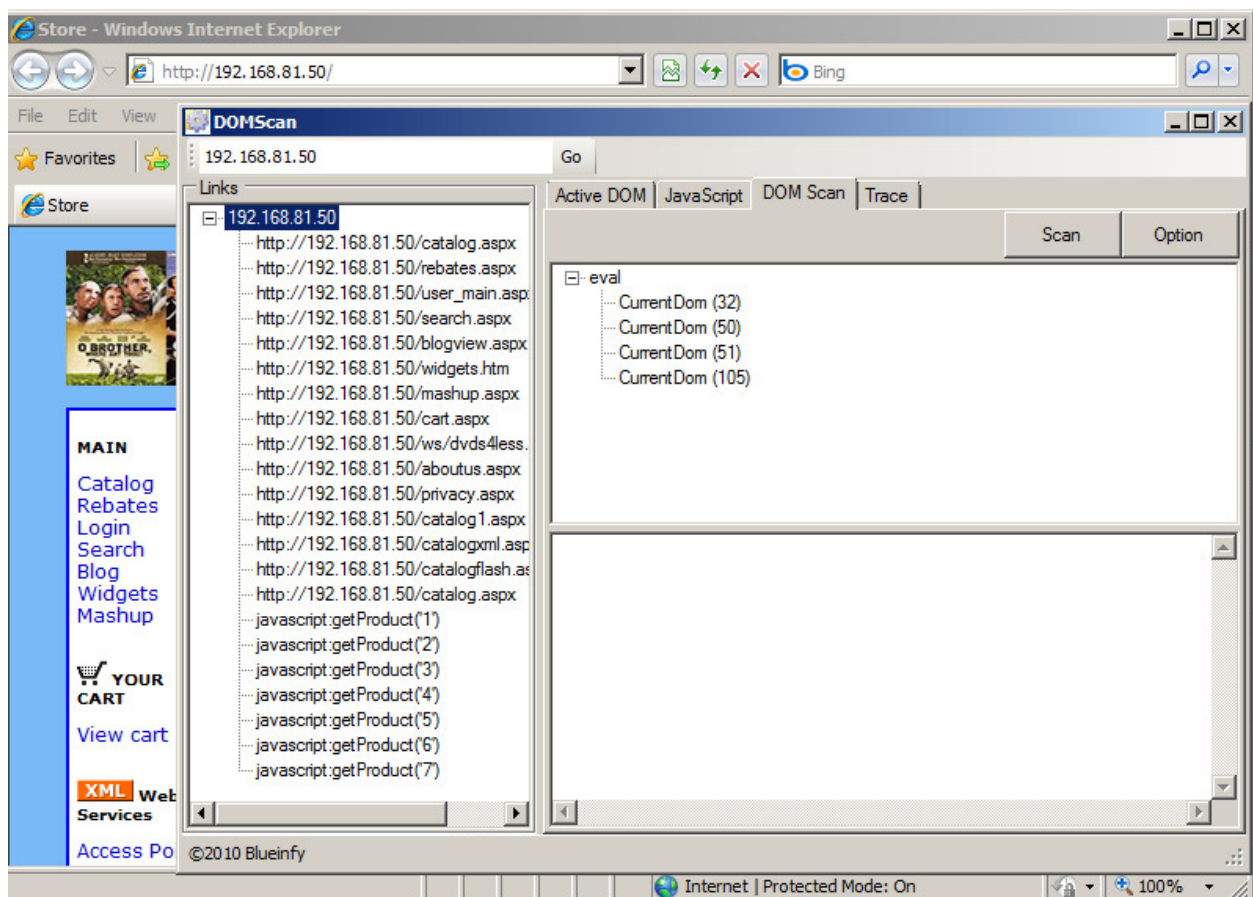
Appendix 1 – Tools and Concepts

DOMScan

DOMScan is utility to drive IE and capture real time DOM from the browser. It gives access to active DOM context along with JavaScripts. One can observe the DOM in detail using this utility. It has predefined rules to scan DOM. One can run the scan on existing DOM and fetch interesting entry points and calls. It allows tracing through JavaScript variables as well. Using this utility one can identify following vulnerabilities.

- DOM based XSS
- DOM based vulnerable calls
- Source of abuse and external content loading methods
- Possible DOM logic and business layer calls
- Same Origin Bypass calls and usage
- Mashup usage inside DOM
- Widget Architecture review using the tool

Here is a simple screenshot for the tool



DOMTracer

The DOM as seen in all the aforementioned cases needs to be analyzed in many aspects. Runtime analysis of the DOM is vital and aids one to look at the calls made during the 'dynamic DOM manipulation' that was discussed above. The DOMTracer is a Firefox Extension for this same purpose.

It has been written using the standard method of writing extensions using the XUL platform and the JavaScript language in majority. Synoptically, there are two layers – the

- Underlying XUL Runner layer and
- The Chrome layer on top of it.

The chrome layer is the layer where majorly there are two kinds of files, namely the '.xul' and the '.js'. The '.xul' files are xml files specifying the outlook of the extension in terms of 'overlays' that are to be layered over the browser. The '.js' files primarily must contain the functions that are called by the xul file and these functions handle events of the extension. The link between this chrome layer and the lower layer is what is popularly known as the XPCOM interface which provides hooks to the underlying services implemented in the lower layer. With this in mind, DOMTracer is an extension which utilizes the 'Debugger Service' via XPCOM:

```
const debug = Cc["@mozilla.org/js/jsd/debugger-service;1"].getService(Ci.jsdIDebuggerService);
```

After the above code, 'debug' has been used for its attributes, properties and methods by DOMTracer. The brief documentation of APIs can be referenced from:

<http://www.oxymoronical.com/experiments/apidocs/interface/jsdIDebuggerService>
where the last component can be replaced for other interfaces as per needed.

These services, via XPCOM, provide means to override existing methods and handle various events such as the creation of a Script, destruction of a script, calls of various functions, et cetera. Moreover, additional facilities to guide the debugger such as switching it on or off and also changing the flags affected by the debugger are also present under this interface.

DOMTracer uses this concept at the core and handles various events which show the calls made at runtime in the extension window. The code could also be altered in some cases to show the scripts as alerts if the demand is: to be able to have the browser paused at each function call and wait for the analyzer's intervention before the next call is traced:

