# FSMesh
## *Flexibly Securing Mashups by User Defined DOM Environment*

Yi Wang, Zhoujun Li
*State Key Laboratory of Software Development Environment, Beihang University, 37 xueyuan road, Beijing, China*

Tao Guo, Zhiwei Shi
*China Information Technology Security Evaluation Center, Beijing, China*

Keywords: Mashup, Html5, Sandbox, Web Workers, Web Application.

Abstract: A growing trend of nowadays web sites is to combine active content (applications) from untrusted sources, as in so-called mashups, in order to provide more functionality and expressiveness. Due to the potential risk of leaking sensitive information to these third-party sources, it is urgent to provide a secure “sandbox” for playing the untrusted content and allow developers to apply flexible security policy at the same time. In this paper, we propose and implement a new safe framework to prevent untrusted applications from interfering with each other based on HTML5 technology. By creating a separated fake DOM environment in the background, developers can load untrusted content into the “sandbox” and apply their custom security policy in real window or server side when receiving script generated messages from it. The advantage is that it is very flexible as the security policy is also written in JavaScript and requires minimum learning efforts for web developers. The drawback is that it is based on element “web workers” and method “postMessage” introduced in HTML5 and can’t be run in older browsers without these supports.

## 1 INTRODUCTION

A mashup is a web page that integrates content and executes JavaScript from different sources. By combining multiple separated services into a new application, a mashup generates valuable product. Facebook applications, gadgets on the iGoogle homepage, and Google Maps embedded in hybrid applications are all well known and successful mashup examples in real world. In most cases, web pages that display advertisements from ad networks are also mashups, since they also employ JavaScript for animations and interactivity. The evolution within web 2.0 has made mashup an inevitable and important part of web application but flawed with security problems.

*This work has been partially supported by the National Natural Science Foundation of China (No.60973105, No.61170189), the Research Fund for the Doctoral Program of Higher Education No. 20111102130003 and the Fund of the State Key Laboratory of Software Development Environment under Grant No.SKLSDE-2011ZX-03.

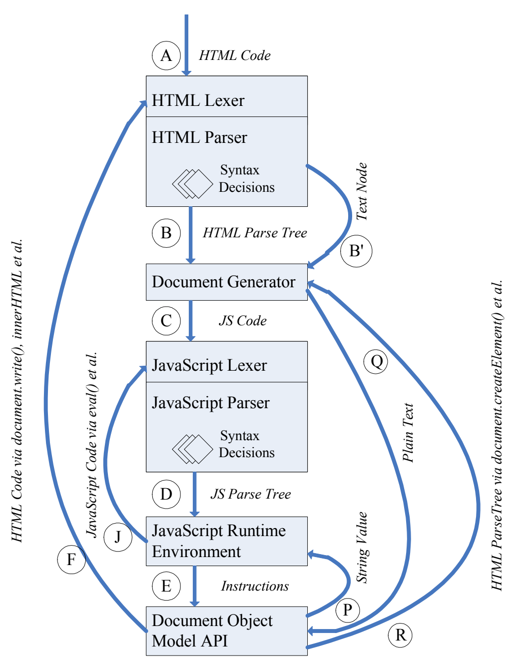



Figure 1: Functional diagram of browsers’ HTML interpretation process.



Wang Y., Li Z., Guo T. and Shi Z..
FSMesh - Flexibly Securing Mashups by User Defined DOM Environment.
DOI: 10.5220/0003899000960102
In *Proceedings of the 8th International Conference on Web Information Systems and Technologies* (WEBIST-2012), pages 96-102
ISBN: 978-989-8565-08-2
Copyright © 2012 SCITEPRESS (Science and Technology Publications, Lda.)

If inserted into the web page directly, these third-party code fragments will run with the same privileges as trusted, first-party code served from the originating site. Hence, the trusted site is susceptible to attacks by maliciously crafted pieces of code. Malicious code may deface web sites to lay down functionality and hijack user sessions or leak cookies to cause disasters on user privacy. When simply embedding code in frames, security threats still arise as techniques such as CSRF (cross-site request forge), clickjacking and frame busting may easily circumvent weak border between the frame and the main window. Moreover, the none or all security insurance provided by SOP(same origin policy) of frames will not enable web developers enough flexibility in controlling the security level of certain mashups.

There are various ways to secure mashups, from static code transformation to dynamic running protection. But all these methods will ultimately undergo the browser's native HTML interpretation process as depicted in Figure 1(Mike Ter Louw and V. N. Venkatakrishnan, 2009) and we argue that if such process can be reproduced in a user customed way, many problems will be solved without making modification to the browser or untrusted code.

In this paper, we propose an emulated DOM environment "sandbox" framework called FSMesh within which the untrusted code will be parsed and safely executed in the web workers(Ian Hickson, 2011) similar to browser's interpretation process. But its critical window APIs have been implemented with different operation semantics so as to provide web developers the mechanism to communicate with real window and flexibly setup the security policy for certain external source. As our tool is totally implemented in JavaScript and can be easily imported as a library into the original web page. So this paper makes the following contributions:

- New secure mashup's framework
- Flexible mechanism for web developer to setup custom security policy
- Implement a prototype library compatible with html5 technology.

The rest of this paper is structured as follows. In Section II and III, we respectively describe design and implementation of the framework. In Section IV, we present related work on securing mashups or advertisements. In Section V, we report empirical results and analysis. Section VI concludes.

# 2 FSMesh DESIGN

Suppose that a bulletin board web developer wishes to allow users to post contents with html tags and especially enable the execution of script within the blocks, he has to settle three challenges concerning functionality and security. First, posted contents with scripts may contain malicious code, thus security issue must be settled by restricting the accessibility of third-party script to the native window object but allowing the trusted script to access the untrusted DOM elements. Second, the normal user interaction with the third-party elements should be preserved as well as safely handling the event generated by the scripts from them since it is the primary requirement for rich internet application. Finally, flexible and fine-grained security policy must be able to be applied easily by web developers and requires little modification to the current version of web application and browsers.

## 2.1 Architecture

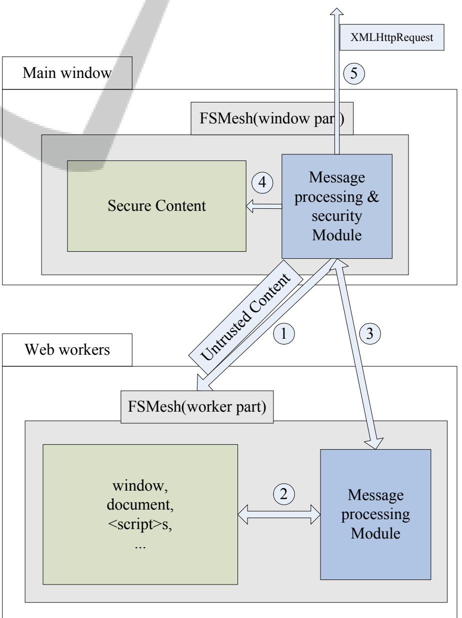



Figure 2: The architecture of FSMesh framework.

The architecture of our FSMesh framework and its inner interaction is depicted in Figure 2. Our approach is to initially relegate the untrusted html snippet into a newly created web workers with FSMesh library imported. The snippet is then parsed and executed under the isolated environment without any interference from the real window. We then detect effects generated by the script that would normally be observable by the user, as the scripts

have not been disabled by our approach. These effects are emulated, subject to policy-based constraints, outside the isolated environment for the user to observe and interact with by message passing mechanism. User actions obtained in the real window are also forwarded to the isolated environment to allow for a response by the script. Thus we enable a controlled interaction between the user and the isolated environment, while blocking several user-defined malicious behaviors.

Web applications that display third-party content on client browsers are exposed to a wide variety of threats. Although our framework can alleviate content security threat, many other security threats posed by embedding untrusted code are out of the scope of our target. We do not address browser vulnerabilities attacks launched through plug-ins, vulnerabilities in image rendering and so on. We also do not protect attacks from opaque content (e.g., Flash), since many possible attack vectors from these binary formats require special treatment.

## 2.2 Content Confinement using Web Workers

For basic security policy, the developer wants to ensure outside script does not access the native global window object. To enforce the policy, we leverage the new element introduced in html5 "web workers" as the ideal environment to place user defined html parsing module and execute script. Web workers defines an API for running scripts, basically JavaScript, in the background independently of any user interface scripts. This allows for long-running scripts that are not interrupted by scripts that respond to clicks or other user interactions, and allows long tasks to be executed without yielding to keep the page responsive. To enforce the developer's content security policy, scripts are executed in the FSMesh in web workers. The only way to communicate between the main window and web workers is message passing which means any access to code or data in main window is not permitted. Web workers is currently supported by some mainstream browser vender such as Safari, Chrome, Opera and Mozilla Firefox.

## 2.3 Controlled Interaction between Environments

As the web workers is running in the background, it has to invoke the native rendering mechanism of browser by sending secure content to the main window as known of a process called content mirroring. Moreover, user's interaction with the content in main window will be passing forward to the web workers as scripts are all have to be executed in that environment for security insurance with the process called event forwarding. User custom security policy will be defined in main window by developers and applied during the two processes.

### 2.3.1 Content Mirroring

After content is parsed by the html parser into a document tree and onload scripts are executed on the web workers side, it will reflect the nodes of the document into the main window by transforming them into string and sending out via message passing. On the other side, when receiving the html string from web workers, main window will again parse it into document. But this time, no script will be executed as they have been already evaluated in web workers and also not allowed for security issues. The document will be transformed into secure content for them to fill in the block formerly removed.

### 2.3.2 Event Forwarding

To facilitate this interaction, events must be captured on secure content in the main window and forward these events to the web worker for processing. For example, if some element possesses onmousemove attribute and should invoke its script when mouse move over it, we replace the original script with clean "catching event" handler on the mirrored element. The handler only listens for the mouse-move event and forwards it to the shadow page via a message. While the web worker receives such message, it will dispatch the same event to the element and execute the script safely. For event handler defined by "addEventListener" on normal document element, if it is related with keyboard or mouse event then we treat the element in the same way as above. Other kinds of event type or event handler registered with window are discarded.

### 2.3.3 Policy Definition

As mentioned earlier, access of elements will be regulated by the FSMesh security module where policy definition takes place. These policies are specified by defining validation procedures against possible malicious content leading to or communicating with outer space in the untrusted code. These cases are listed in Table 1 and only

minimum trust level is defined (no trust). In this way, web developer can define any kind of ways to filter or validate the content or content will be eliminated without treatment.

# 3 FSMesh IMPLEMENTATION

The implementation of FSMesh framework is mainly consisted of DOM emulation module and message interaction module. For the first module, we port Envjs (server-side DOM environment implemented in JavaScript) (John Resig, 2011) of Rhino version to web worker for DOM emulation. A few changes are needed for original code base including replacing core part concerning platforms and work around internal limitations of web workers. For the message interaction module, as the core of security policy enforcement, we introduce content synchronization on both sides and filtering functions in the main window side.

Table 1: Possible malicious content requiring security policy enforcement.

| Malicious content | Example | Default action |
|---|---|---|
| ‘href’ in <a> element | <a href=”malicious.com”>… | Remove ‘href’ |
| ‘action’ in <form> element | <form action=”malicious.com”>… | Remove ‘action’ |
| ‘src’ in <img> element | <img src=”mal.com/mal.php”>… | Remove ‘src’ |
| http request | <script><br>var xhr=new XMLHttpRequest();<br>xhr.open(method, url);<br>xhr.send(content);<br></script> | No http request allowed |
| location redirection | <script><br>window.location=”malicious.com”;<br></script> | Location not changed |

## 3.1 DOM Emulation

Envjs contains a html5 parser together with definition of html elements that support parsing html string and execute script node inside. The Rhino version contains code that is specific for java environment which is unsuitable for web workers. Unlike server side program which has no restriction for system access, browsers limit web workers’ ability in writing files, setting read-only predefined properties(eg. location, navigator). We work around this problem by initially assigning “window” as a blank object and set its prototype to native web workers’ “this” instead of directly assigning “window” to be “this” . In this way, we can freely extends “window” object or will be forbidden when redefining read-only properties of web workers otherwise. Injected scripts will be evaluated in “this” and newly defined variables in “this” can be accessed transparently by “window” via the property finding mechanism through the prototype chain. This process is depicted in Figure 3.

Some core window functions required to utilize native window of the main page is implemented as post message to real window with message type specified. When the main page received such messages, it will carry out corresponding actions if security policy permits. As it is not possible to send request to different origin web site for content by browser’s SOP rule, we work around the function of XMLHttpRequest by making the request to an interface of same web site with the url as an argument. Developer will set up the interface in the server side as a proxy to retrieve from outer web site for content.

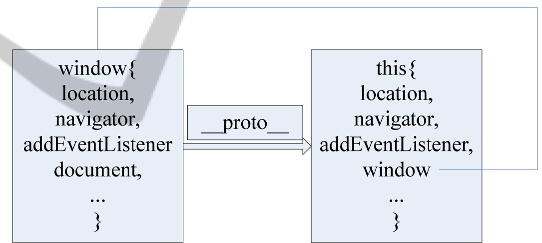



Figure 3: Relationship of “this” and “window” in web workers.

## 3.2 Content Synchronization

To reflect changes in web workers to main page and handle event generated in main page in web workers requires the developer to modify original contents in three ways. The first modification is to remove the post content. Second, we add the trigger scripts for every removed contents to initiate content passing. The third modification to the original page is to implement event handling procedures in order to pass correct event to the web workers. In the same time, corresponding receiving mechanism in the web workers side will be implemented for event to dispatch to the right element.

## 3.3 Security Policy Enforcement

Although scripts inside web worker have no access

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

to object reside in main window, some actions have to be realized in the main window when more authority is required. So it relies on the developer to decide the actions for different security policies. We now describe in detail the individual permissions granted by policies, how policies are specified, and how multiple policies are combined to form a composite policy.

As described earlier in Table 1, we have to deal with the 5 cases where malicious actions can be taken when content are mirrored on the main window. FSMesh framework allows the developer to override 5 API implementations to achieve the goal of ensuring such security requirements to be met, described in Figure 4. The default behavior of these APIs is no operation which is highest in security level but lowest in functionality level. It depends on the specific cases to gain perfect trade-off between the two factors and decisions are made by the developers.

```
setLocation( url ):
Redirect the current location to url.
makeHttpRequest( url, content ):
Make a http request to url with content if 'post'.
linkFiltering( link ):
Filter malicious href in <a> element.
imgFiltering( src ):
Filter malicious src in <img> element.
formActionFiltering( form ):
Filter malicious action in <form> element.
```

Figure 4: Security policy in main window.

# 4 EVALUATION

We evaluated FSMesh by self-made testing mashups with regard to security and overhead performance. For the case of security, the mashups are mostly extracted from cross site scripting website consisted of various kinds of script attacks from third-party code. For the case of overhead performance, we made incremental size of mashups to investigate the time delay and memory cost caused by message passing and html parsing of FSMesh framework.

## 4.1 Security Insurance

We set up Apache web server on our computer with main test page as the index page and choose Firefox 4.0 as the our testing web browser. We then inject various kinds of script blocks into the mashups in the main page aiming to test prevention ability against different objectives. These script codes together with their attack goal are shown in Table 2. Our experiments results support our claims that it provides strong defense against several potential attack vectors in which mashups are often exposed.

It should be noted that although attackers may inject code that can override the core function of the part of web workers for FSMesh and may transfer malicious content as messages to main window. But they won't be able to find a way to change the code of the main window's part which ensures the enforcement of security policy as the only communication method between the two parts is message passing.

## 4.2 Overhead Performance

To measure mashups rendering latencies caused by our FSMesh framework, we placed incremental size of mashups both on original main page and on modified FSMesh page instrumented with time benchmarking code. There were a total of 10 instances of mashups with size from 10 nodes to 1000 nodes. The test pages are rendered in Firefox v4.0.1 on an AMD Athlon X2 4450 (2.30 GHz) PC with 2.0GB RAM. Each modified FSMesh page import our implementation source code (663 kB of JavaScript), which was cached by the web browser and is not optimized. Time and memory cost are measured for the 10 test pages with and without FSMesh framework and result for each case is the average of 10 tests for accuracy.

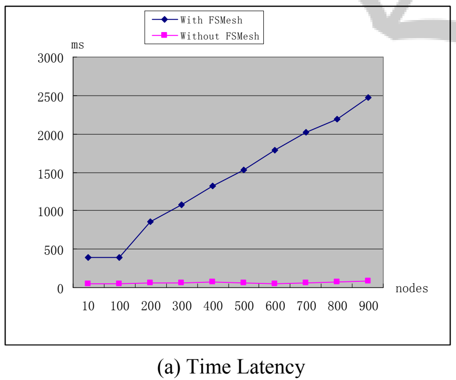
Results of this experiment are shown in Figure 5. Upper line in Figure 5a shows the time required to render mashups in FSMesh with increasing number of nodes, while the lower line indicate the same case for normal rendering. Figure 5b shows the space cost for both situations with upper line representing FSMesh and lower line as normal rendering.

It is clear to see that the time latency is comparably high for our framework and unlike native rendering, it increase dramatically as the number of nodes rise. The reason for this is mainly because html parsing is a time and space consuming task and web workers' implementation in Firefox still has issues in trace jit optimization. To settle this single problem, it may suffice to replace the our prototype javascript implementation with native compiled version installed as a plugin in browser. Another way is to extend the functionality of web worker to support built-in parsing and DOM manipulating. But these approaches will contradict our main idea of non-modification to web browsers, as web browser should only be responsible for browsing the content of web pages. Thus, optimizing the performance of javascript engine will be an

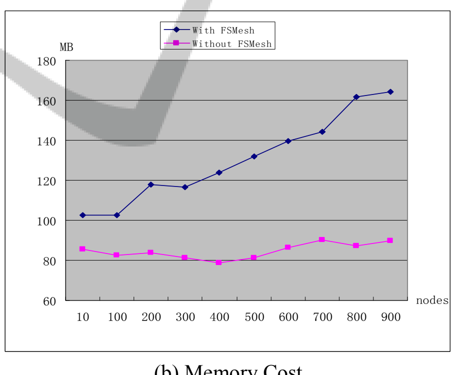
Table 2: FSMesh prevents scripts' attack actions by execute in web workers.

| Attack Objective | Injection codes | No FSMesh | FSMesh |
|---|---|---|---|
| Execute arbitrary code in window | <img src=JavaScript: alert('XSS') /> | execute | execute in WW |
| | <script src=http://xss.org/xss.js></script> | execute | execute in WW |
| | window.location="http://malicious.com/"; | redirect | no redirect |
| | var style = document.defaultView. getComputedStyle (link, null); | get style | get style in WW |
| | <img src=nonexistent.gif onload=alert(1)>"; | execute | execute in WW |
| | window.prompt("enter your password"); | execute | execute in WW |
| | document.write("<script>alert(/XSS/)</script>"); | execute | execute in WW |
| Confidential information leak | document.write("cookie is:"+document.cookie); | output cookie | output nothing |
| | xhr.open("POST",malicious_url); xhr.send(document.cookie); | post cookie to malurl | action filtered |
| Frame busting | top.location="http://malicious.com/"; | redirect | No redirect |
| Content integrity violation | var elm=document.getElementById ("some_id_in_main_page"); | get element | get element in WW |
| | var elems=document.getElementsByTagName("a"); | get elements | get elements in WW |
| Clickjacking | <iframe src="some_url"></iframe> | iframe created | iframe filtered |

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS




(a) Time Latency




(b) Memory Cost

Figure 5: Rendering overhead with increasing number of DOM nodes.

important area for future work in order to make such framework practical and it is actually possible. With the issues in web workers' trace jit optimization settled, as well as cutting off useless functions in our framework, the performance of FSMESH will be comparable to native compiled version.

## 5 RELATED WORKS

PostMessage (Hickson and Hyatt, 2011) and sandbox attribute (Hickson and Hyatt, 2011) are extensions in HTML5 and augments the origin-based separation of iframes. The sandbox attribute imposes a set of restrictions, preventing scripts or browser plugins to run or preventing forms from being submitted.

MashupOS(H. J. Wang et al., 2007) argments html tags to support additional trust levels within a mashup. OMash(S. Crites et al., 2008) represent web pages as objects, which have public interfaces for interaction and encapsulates the internal state of a web page, including associated resources such as cookies.

The ADsafe subset(D. Crockford 2011) , Facebook JavaScript (FBJS) (Facebook 2011) and Caja(M. S. Miller et al., 2011), which are secure JavaSript subsets, are active protection mechanism, which applies a rewriting process to normal JavaScript by rewriting variable and function names

to a unique namespace. Although specific issues with ADSafe and FBJS(S. Maffeis and A. Taly, 2009) in the security of JavaScript subset are discovered, they do not break the fundamental of the language. More importantly, the security of caja-based JavaScript subsets has been able to prove to be capability safe(S. Maffeis et al., 2010).

ConScript enables the specification and enforcement of fine-grained security policies for JavaScript in the browser(B. Livshits and L. Meyerovich, 2009). Self-protecting JavaScript(P. H. Phung, 2009) provides similar security features, but does not require specific support within the browser. Policy enforcement is achieved by wrapping security-sensitive JavaScript operations before normal script execution.

AdJail(M. Ter Louw et al. 2010) offers a technique to mediate access to advertisements, which are embedded as a DOM object and executed in an iframe while interacting with hosting page. Our work is comparable with AdJail in the way of seperating untrusted content and interacting with hosting page, but we make use of the more secure “web workers” element in HTML5 considering the drawback of frames. They utilize the html parsing and script execution by native frame window with hooked DOM manipulating APIs. But we argue that such process is not capable of controlling every phase of html parsing and browser behavior, and also too coarse in the case of applying particular security rules. Instead, by creating a fully implemented fake DOM environment in the secure web workers, we have everything at our hand. Thus, any attacks targeting at frame or navigation will be nullified while may easily compromise their technique.

## 6 CONCLUSIONS

In this paper, we present FSMesh as a solution for the problem of confinement of third-party mashups to prevents attacks on confidentiality and integrity. The new safe framework which is based on HTML5 technology creates a separated fake DOM environment in the background which allows developers to load untrusted content into the “sandbox” and apply their custom security policy in real window. The benefit of FSMesh is both in flexibility security policy enforcement and minimum modification of original content. Our approach offers developers an easy solution for confining untrusted content in main stream browsers without steep learning curve or installing new software. Although the current framework suffers big overhead performance problem, we believe that it can be overcome by optimization. We plan to make the idea possible by first inspecting the source code of web workers for browsers and make extension for it to support the function of FSMesh.

## REFERENCES

I. Hickson and D. Hyatt (2011). Html 5 working draft cross document messaging. http://www.w3.org/TR/html5/comms.html#crossDocumentMessages.

I. Hickson and D. Hyatt (2011). Html 5 working draft - the sandbox attribute. http://www.w3.org/TR/html5/the-iframe-element.html#attr-iframe-sandbox.

H. J. Wang, X. Fan, J. Howell, and C. Jackson (2007). Protection and communication abstractions for web browsers in mashupos. ACM SIGOPS Operating Systems Review, 41(6):16.

S. Crites, F. Hsu, and H. Chen (2008). Omash: Enabling secure web mashups via object abstractions. In Proceedings of the 15th ACM conference on Computer and communications security, pages 99-108. ACM.

D. Crockford (2011). Adsafe. http://www.adsafe.org/.

Facebook (2011). FBJS. http://developers.facebook.com/docs/fbjs/.

M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay (2008). Caja: Safe active content in sanitized javascript. http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf.

S. Maffeis and A. Taly (2009). Language-based isolation of untrusted javascript. In 22nd IEEE Computer Security Foundations Symposium, pages 77-91.

S. Maffeis, J. C. Mitchell, and A. Taly (2010). Object capabilities and isolation of untrusted web applications. In Proceedings of IEEE Security and Privacy'10. IEEE.

B. Livshits and L. Meyerovich (2009). Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. Technical report, Microsoft Research.

P. H. Phung, D. Sands, and A. Chudnov (2009). Lightweight self-protecting javascript. In Proceedings of the 4th International Symposium on Information, Computer, and Communications Security, pages 47-60.

M. Ter Louw, K. T. Ganesh, and V. N. Venkatakrishnan (2010). Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements. In 19th USENIX Security Symposium.

John Resig (2011). Envjs – Bring the browser to the server. http://www.envjs.com/.

Mike Ter Louw and V. N. Venkatakrishnan (2009). Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In IEEE Symposium on Security and Privacy, Oakland, CA, USA.

Ian Hickson (2011). Web Workers.http://dev.w3.org/html5/workers/. July 2011

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS