Security Vulnerabilities in Modern Web Browser Architecture

Marin Šilić
Faculty of Electrical Engineering and Computing, University of Zagreb
Unska 3, 10000 Zagreb, Croatia
Phone: +385 1 6129 549
E-mail: marin.silic@fer.hr

Abstract - **The Web today has become the most used and popular platform for application development. In the beginnings of the Web, applications provided users just ability to browse and read content. The expansion and adoption of the new web technologies has led to a significant increase in development and, more importantly, usage of the web applications that allow users to create their own content and impact their life (e.g. e-banking, e-commerce, social networks). Web 2.0 applications introduced new possibilities for both users and application developers, but also created new security concerns. Almost every Internet user uses a web browser to access any content on the Internet. Each web application is designed and developed to be executed inside the web browser. Web browser mediates between users and applications. In such architecture, malicious applications could be loaded and executed inside the web browser, making it a vulnerable point in preserving security. Modern web applications demand for a new web browser architecture design that will meet new security requirements arisen with the Web 2.0. In this paper, we study web browser's vulnerabilities, analyze popular web browsers architecture and present how they cope with potential security threats.**

I. INTRODUCTION

In its basis, Web was designed for browsing static web pages and reading content. With the recent technological improvements, the Web has become a platform for application development. The turning point was invention and adoption of AJAX technology which turned from the old concept of static web pages to the new concept of creating interactive web applications. AJAX and similar web application development technologies, often referred to collectively as Web 2.0 technologies, led to the creation of variety of numerous worldwide-oriented web applications. Contemporary web applications like e-banking, e-commerce, social-networking sites, blogs, and video-sharing sites provide users not just the ability to view information and access content, but also the ability to contribute and create their own content on the Web, express their creativity and share knowledge and information with others.

The nature of Web 2.0 applications requires users to provide their identity and private data like user-names, passwords, credit card numbers, mailing addresses, social security numbers, etc. Those applications are designed to be executed inside the web browser, which is a mediator between users and applications. Web browser exploits have bigger impact than ever before, and thus web browser designers have to pay more attention to security than ever before. Knowing the security holes in the web browser, attackers can create malicious web applications in order to compromise other users' security.

Many different web applications can be executed simultaneously within the web browser. Some applications can have significant reflection on user's life, while some can be malicious applications with the only intention to compromise security. Each application in the browser has its own security settings that define application's privileges and rights for the user's local file system. For example, web application should be allowed to access local file system in order to upload a certain file only with user's explicit approval. On the other hand, browsers have their local storage where user's sensitive data like passwords, cookies, bookmarks, browsing history, temporary files, and cache are stored. Modern browsers need to assure that web application can not access that storage, and can only get private data (e.g. cookies) related to that particular application.

In order to protect the user, some browsers enforce strict security policy, which isolates applications inside the browser by their origin and does not allow subresources from other origins. Such a restrictive policy would require architectural restructuring of existing Web. On the other side, users expect browsers to be compatible with the existing Web architecture and render their popular applications. The desirable goal in browser design is to achieve user's protection and still to provide compatibility with existing web applications.

The majority of modern browsers still use the original monolithic architecture design. Monolithic browser architecture has many disadvantages that concern client code execution. Failure caused by one web application crashes down the entire browser instead of just the application that caused it. In terms of better user experience, user should be able to use other opened applications. From the aspect of security, if the browser as a vulnerable monolithic structure gets compromised, attacker could execute his arbitrary code with user's privileges and rights and cause damage on local machine. Modern applications require browser architecture that provides both browser security and compatibility with the existing Web architecture. That can be achieved with modular browser architecture where, in contrast to monolithic one, each application is executed in its own sandbox with restricted privileges.

Section II explains modular browser architecture and compares it with the monolithic one. In section III, we review Google Chrome browser, as an implementation example of the modular browser architecture. We analyze how Chrome responds to major threats on browser security. Section IV describes related browsers based on modular architecture and compares them with Chrome. The paper finishes with conclusions in Section V.

II. MULTI-PROCESS BROWSER

As a result of recent break through in the Web technology utilization contemporary web applications behave more like complex programs that demand resources than simple documents for browsing. Most of current web browser architectures are still monolithic, usually designed for browsing and rendering static web pages. Monolithic architectures do not provide enough isolation between concurrently executed web programs and execution often ends in misbehavior as a lack of security, fault-tolerance, memory management or performance. Early PC operating systems had same program isolation issues. MS-DOS and MacOS allocated single address space and programs interfered with each other, unlike modern operating systems that isolate each program in its own separate process. Thus, modern browser should isolate web programs and modularize their execution assigning each web program to the specific operating system process within the browser.

A. Monolithic browser architecture

Figure 1 shows monolithic web browser architecture most common for current web browsers. In that architecture, all web programs browser components are placed in a single operating system process. Document Object Model (DOM) tree is a web page representation that can be accessed and modified by the script code. HTML Renderer component parses each page code and generates DOM tree. JavaScript Engine is responsible for running script code that manipulates DOM tree.

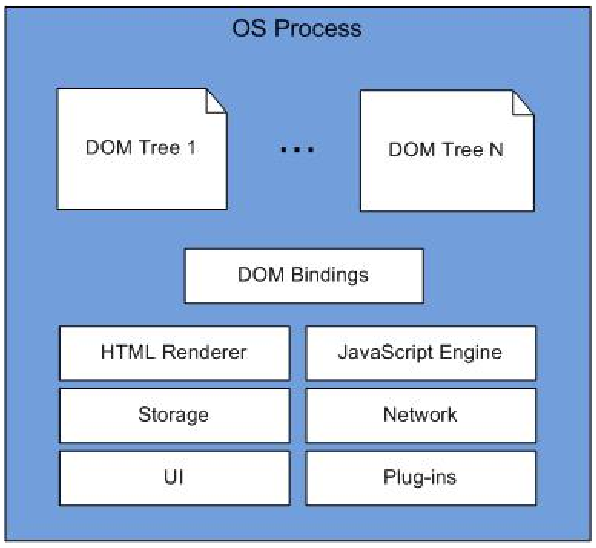

Figure 1. Modular browser architecture

Monolithic architecture has a lot of disadvantages that concern user experience, fault-tolerance, accountability, security, memory management and performance. From user experience and fault-tolerance point of view, any web program or browser component that encounters crash, takes down the complete web browser. Some browsers with monolithic architecture provide reload feature on browser start up after the crash. But still, as a result of the crash user might lose valuable data (e.g. unsaved email drafts, e-banking transactions, purchase orders) stored as a JavaScript state in memory. After the browser restarts, misbehaved application that caused crash might cause the crash again, in this case reload feature is pointless. Considering accountability, monolithic architecture provides resources usage statistics for the entire web browser. However, web program responsible for a poor performance of the entire browser can not be identified in a monolithic architecture. Another disadvantage of monolithic browser architecture is memory management. Browser process in OS is a long life process compared to the life of web programs that are executed in the browser. Some web program running in the browser might require lot of memory allocation and contain memory leaks, which can result in a large and fragmented memory space that is allocated to the browser process. Once the web program like that finishes, the memory still remains large and fragmented. As far as performance is concerned, monolithic architectures can cause resource demanding web programs to compete for CPU on with each other. Also, monolithic architecture can block a browser UI thread because web program's actions, like executing synchronous XMLHttpRequest. Both of this causes user-perceived delays on UI level and lower performance for the entire browser. Security of monolithic architectures entirely rely on the browser components logic to completely isolate different web programs and prevent any information flow between web objects in different web programs. However, bugs omitted in browser design or implementation, leave space for malicious web programs and attackers to install malware, steal files or access private data and compromise user's security.

Despite all its disadvantages, monolithic architecture is preserved in the majority of web browser because it is difficult and challenging to isolate web programs in the browser and still keep browser compatibility. One approach could be to isolate each web page in the browser, but this would break many popular applications like sites that use pop-up windows or embed content in a separate frame from a different location. Another approach could be to isolate web programs by their origins. However, sometimes pages with different origins need to communicate with each other and sometimes pages with the same origins are not related at all.

B. Modular browser architecture

Figure 2 presents web programs isolation model, implemented in Google Chrome web browser, based on open source Chromium project [1]. The key point in modular browser design is to isolate web programs, but provide compatibility with the current Web. Presented model introduces ideal abstractions: *web program* and *web program instance*.

Web program is a set of connected web pages containing all their subresources that provide certain functionality. For example, iGoogle page contains of parent page, script libraries and images, and gadgets sites embedded in their frames. Since browsers allow users to visit multiple instances of the same page, e.g. user can open two iGoogle pages in different tabs, web program instance abstraction is introduced.

Web program instance is defined as a set of pages from a web program that are connected in the browser and allowed to access and manipulate each others content. Web program abstraction is realized using *site*, while web program instance abstraction is realized using *site instance*.

Browsers allow related pages to communicate by enforcing *Same Origin Policy (SOP)* [2]. SOP conducts access control based on the page origin, which includes protocol, full host name, and port of each page. Pages with the same origins are grouped together and allowed to manipulate each others content. Page subresources can be included from some other origins, but their origin is considered same as the origin of the enclosing page. If origins do not match, pages are mainly isolated. Origin does not provide enough distinction among pages to define site because web page can change its origin dynamically. The origin can be changed within a limited range, from sub-domain to more general domain and only up to the *registry controlled domain name* [3], which is the most general part of the host name before suffix (e.g. *.fer.hr* can be changed to *.hr*). Site is defined as a set of web pages with origins within the specific origin range, limited with protocol and registry controlled domain name.

Sometimes pages with a different origin are connected and share communication channel. This is the case when a page opens content in a new window, the opener page keeps reference on a new window and the opened page can access the opener using property *window.opener*. For example, Gmail chat window opens in a new window when the conversation starts. Second case is when a page embeds content from different origin in a separate frame. For example, iGoogle page contains more gadgets pages in separate frames. Top window can access its frames using property *windows.frames*, and each gadget can access parent window using property *window.parent*. Connections between pages that share a communication channel are kept as long as the parent browser window is alive. Even if the user navigates to another page or opens a new page in a new tab or window references among those pages are kept. Chromium isolation model defines another term, *browsing instance* as a set of connected windows or frames that keep a reference to each other.

Site instance, the concrete realization of web program instance abstraction, is defined as a set of connected pages that belong to the same site within the browsing instance. All pages from the same browsing instance can reference each others windows, but only the pages from the same site instance can access each others DOM contexts. On figure 2 there are two browsing instances presented. First browsing instance *B1* contains two site instances: *Sa* and *Sb*. Second browsing instance *B2* contains site instance *Sa*. Site instance *Sa* contains pages *Pa* and *Pb*, while site instance *Sb* contains page *Pc*. Site instance *Sa* from *B1* and *Sa* from *B2* belong to the same site, but do not reference each other, although SOP would allow them to communicate. On the other hand, site instances *Sa* and *Sb* from *B1* belong to different sites, they reference each other, but still are not allowed to manipulate each other DOM according to the SOP.

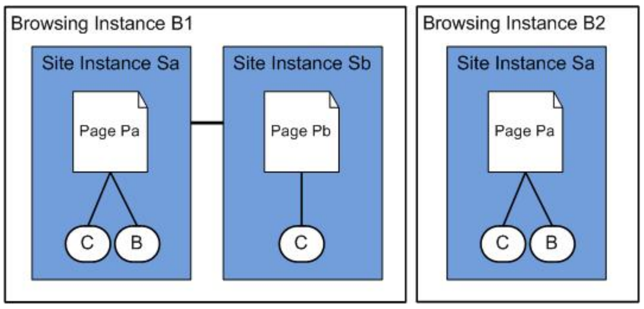

Figure 2. Isolation model in Chromium

One approach to accomplish isolation is to assign each site instance to a one operating system process. In this case, there would be too many processes allocated for web browser execution. Another approach is to assign each browsing instance to a one operating system process. Chromium manages to isolate web programs and modularize web browser execution. Each web program is running in its own operating system process.

In comparison to monolithic architecture, modular architecture is superior considering user experience, fault-tolerance, accountability, security, memory management and performance. Considering user experience and fault-tolerance, each web program that crashes does not effect the execution of other running programs. In modular architecture each program performance can be easily monitored, thus modular architecture is superior in accountability. Memory management in modular architecture is effectively conducted, each program has its own process and allocated memory, once the program finishes memory is released and can be assigned to some other program. The fact that each program has its own process, assures better performance. Modular architecture leaves scheduling issues to the OS and web programs can run in parallel. Security aspects of modular architecture are presented in Section III.

III. SECURITY OF CHROME

The definition of web program isolation model is used as a base for modular architecture implementation. This section presents modular browser architecture implemented in Google Chrome browser. Furthermore, this section analyzes security aspects of Chrome's architecture.

A. Chrome architecture

Architecture of Chrome [4] browser is given in Figure 3. Chrome consists of three different modules: *rendering engine*, *browser kernel* and *plug-ins*. Each of these modules is isolated in its own operating system process. Rendering engine converts HTTP responses into rendered bitmaps, browser kernel interacts with OS, and plug-ins module is responsible for each plug-in execution.

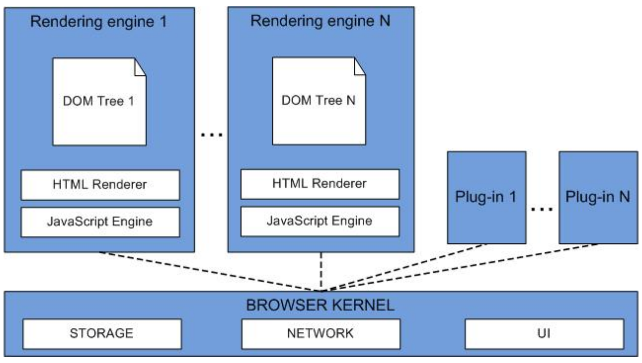

Figure 3. Chrome architecture

Rendering engine runs in a sandbox with restricted privileges and no access to OS. Each isolated web program in the browser is assigned to its own rendering engine. Rendering engine is responsible for parsing web content, creating DOM tree representation in memory, manipulating the DOM tree while executing script instructions. Also, rendering engine enforces SOP policy and manipulates directly with untrusted web content. Historically most of the web browser security vulnerabilities were detected in the parsing and decoding tasks. Thus, rendering engine does most of the parsing like HTML, CSS, XML, JavaScript, regular expressions parsing and image decoding. To interact with the user and OS, rendering engine uses simple and restricted browser kernel APIs.

Browser kernel runs with full user privileges on behalf of the user. It manages each instance of the rendering engine and implements browser kernel APIs. Browser kernel is responsible for storage management, which includes cookies, bookmarks, passwords, because such activity requires file system access. Browser kernel is executing network operations, e.g. downloads the image, but sends it to rendering engine to decode it. Also, browser kernel is interacting with OS, handles user inputs and forwards it to a rendering engine that has a focus. Browser kernel, keeps the information about granted privileges to each rendering engine such as list of files that certain rendering engine may upload.

Plug-ins runs in its own process outside the rendering engine and browser kernel. Web compatibility requires plug-ins to run outside the sandbox, plug-ins may require access to microphone, web cam or local file system. Thus, plug-ins can not be placed inside rendering engine since rendering engine runs in a sandbox. Plug-ins could be placed within the browser kernel, but in this case, crash in plug-ins would take down the entire browser. However, bug omitted in plug-in design or implementation could be exploited to compromise security and arbitrary code with full user's privileges.

B. Security aspects

System compromise threat refers to malicious arbitrary code execution with full privileges on behalf of the user. The majority of browser's vulnerabilities that concern this threat are detected in rendering engine that can be compromised. Compromised rendering engine runs within a Windows sandbox, with Windows restricted security token, unlike browser kernel that runs with Windows user's security token [5]. Although Windows sandbox restricts rendering engine to communicate with OS, there are potential issues that can be exploited in order to compromise the system. Sandbox does not perform security token check if the sandboxed process is accessing the FAT32 file system. Most of existing devices use NTFS file system, but some USB devices use FAT32 formatting. In this scenario, compromised rendering engine could read and write the content on the USB drive. Also, Windows sandbox does not perform security token or requires OS handle when low-level privilege process attempts to open TCP/IP socket. However, these issues rather concern Windows sandbox then Chromium architecture. System can be compromised if the browser kernel gets compromised. Browser kernel can be tricked from a compromised rendering engine. While executing APIs, as a lack of parameters validation, browser kernel can perform unauthorized network or system task. Another way how system could get compromised is to exploit vulnerabilities in vendors' plug-ins that run outside the sandbox by default.

Data theft threat refers to the ability to steal local network or system data. This often happens in case compromised rendering engine requires uploading or downloading a file. In Chrome architecture rendering engine runs in a sandbox and has no direct access to the local file system. When uploading a file, rendering engine uses browser kernel API for file upload. Browser kernel shows the upload file picker window and remembers which file is selected. This action is considered as a explicit user authorization to the associated rendering engine to upload that particular file and that authorization lasts for the lifetime of the associated rendering engine. In the next step, browser kernel uploads the file to the site which instance is running in the associated rendering engine. Also, when downloading files, rendering engine uses browser kernel API to download file. Since the download is initiated by the user, browser kernel is authorized to download the resource from the download URL. Some malicious site may include subresources with URLs that use file scheme. Chrome architecture prevents rendering engine to issue network tasks, like requesting a resource from a specified URL. Rendering engine rather uses browser kernel API to include subresources, then browser kernel analyzes the resource URL and downloads the resource. Most of the rendering engines are not allowed to include subresources from URLs that uses file scheme. However, local files can be viewed in Chrome browser, but in a dedicated rendering engine.

Cross domain compromise Code originating from one *fully qualified domain name (FQDN*) [6] can execute code in the context of, or read data from, another FQDN domain without permission. One such attack is XML eXternal Entity (XXE) attack, in which the attacker's XML document, hosted at http://attacker.com/, includes an external entity from a foreign origin [7]. For example, the malicious XML document might contain an entity from the https://bank.com or file:///etc/passwd. If vulnerable to XXE attacks, the browser will retrieve the content from the foreign origin and incorporate it into the attacker's document, making him able to read the content. Chrome, like many other browsers, uses *libXML* to parse XML documents. However, the architecture of Chrome is designed in such a way that it delegates parsing tasks to a sand-boxed rendering engine. The rendering engine does not prevent the content from retrieving URLs from foreign origins, but passes the requests to the browser kernel. If the external entity URL was a web URL, browser kernel serviced the requests. However, if the external entity URL was from the user's file system (i.e. from the file scheme), then the browser blocked the request, preventing the attacker from reading confidential information such as passwords. Chrome's modular architecture with sand-boxed rendering engines does not completely defend against the XXE vulnerability because the attacker is able to retrieve URLs from foreign web sites. To block such requests, the browser kernel would need to sacrifice compatibility with the Web architecture (e.g. ban cross-site images).

The threats that involve *session hijacking* compromise the session token by stealing or predicting a valid session token to gain unauthorized access to the honest web server. Cross-site scripting (XSS) and cross-domain request forgery (CSRF) have become the two most scaled attacks regarding session hijacking. According to The Open Web Application Security Project (OWASP), those two kind of attacks have been marked as No2 and No5 top security risks for web applications for the year 2010 [8]. In short, XSS exploits the client's trust of the content received from the server (by just sending text-based attack scripts that exploit the interpreter in the browser). This allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites. A CSRF attack tricks (via image or script tags) a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other authentication information, to a vulnerable web application. This allows the attacker to perform any action on a vulnerable web server the victim is authorized to use. Chrome's architecture does not protect an honest web site if the site contains XSS or CSRF vulnerabilities. Chrome expects these sites to repair their vulnerabilities. The only helpful thing Chrome has is support for *HttpOnly* cookies, which can be used as a partial mitigation for XSS [9].

User interface compromise threat refers to the ability to trick the user into making incorrect trust decision, or directly provide confidential data using script UI manipulation. Popular attack that manipulates UI is *clickjacking* [10]. In clickjacking, attacker loads subresource from some other origin and places it to be transparent content in front of the visible content. User thinks he clicks on the objects he sees, but actually he clicks on the transparent content. Also, user interface compromise refers to implementing annoying scripting actions like hogging the CPU or memory, moving dialogs faster then user can respond, opening dialogs in endless loop. From today's perspective, no modern browser provides enough security restrictions to defend itself from scripting disruptions. Introduction of any limitations that are related to window manipulation or disabling pop-ups, provides less functionality and options for site developers and lowers compatibility with Web. However, Google Chrome uses limitations on windows manipulation [11] with scripting languages such as taking full screen, specifying screen dimensions and position, hiding URL bar and status bar.

IV. RELATED ARCHITECTURES OVERVIEW

OP Browser [12] introduces modular architecture that consists of following browser components: UI, web page, storage, network and browser kernel. Each component runs in its own operating system process. Browser kernel runs with full privileges and behaves like the operating system micro-kernel. It coordinates the communication among browser components using message passing mechanism. Compared to Google Chrome, proposed architecture provides strong isolation among web sites and higher level of protection, but makes impossible to implement some popular web features like inter-frame communication, file uploads and downloads. These characteristics make OP browser incompatible with many popular web sites. OP browser enforces restrictive plug-ins security policy. Architecture does not allow plug-ins to run with full privileges in their own process on the whole browser level. Instead, plug-ins run within the web page component with restrictive privileges. Plug-ins are allowed to access the resources that correspond to the origin of the site whence the plug-in object is embedded. Sandboxing plug-ins using restrictive policy provides higher protection, but makes browser less compatible with the current Web.

Tahoma [13] architecture introduces a new concept for web application execution. Each web site is running on its own virtual machine within the protected framework named browser operating system (BOS). BOS manages each virtual machine's network and UI tasks. Each virtual machine manages its own storage, cookies, bookmarks, history and has no access to the user's local file system. There is a strict isolation among different running virtual machines. Tahoma architecture introduces new possibility for web application execution. Since the web application is running on virtual machine, web application developer can deploy application in machine code language. Each web site owner should create manifest file for his web site. That manifest file contains information about the site like the list of URLs site is communicating with or weather the site uses machine code or standard HTML renderer. When first visiting the site, user receives site manifest and needs to approve the site before execution begins. Tahoma architecture is revolutionary and provides high level of protection. However, this architecture is completely incompatible with the current Web and requires current Web restructuring.

Gazelle browser [14] architecture contains browser kernel and rendering engine process, similar like Google Chrome. Google Chrome places resources within the same renderer according to the registry controlled domain name policy, while Gazelle places resources within the same renderer according to the SOP. Gazelle architecture provides stronger isolation that concerns inter-frame scripting. In case the web page embeds content in a separate frame, Gazelle places parent and child frame in to different renderer processes and allows them to communicate using limited browser kernel API. On the other side, Google Chrome places the parent and child frame in the same renderer process, but the communication among them is restricted according to the SOP. Cross scripts and style sheets are placed within the same renderer process both in Chrome and Gazelle. However, because of the cross-scripting and inter-frame communication limitations, Gazelle is not quite compatible with the current Web. For, example, Gazelle does not allow the frame to change its *document.domain* property, which is essential for inter-frame communication before *postMessage* event introduction. Also, Gazelle introduces opaque display policy, which disallows cross-site content to be transparent and overlap the host site. This policy enhances the overall browser security and reduces UI manipulation, but still it is not quite compatible with the current Web. Gazelle browser tends to protect different web sites from each other, while Chrome focuses on host machine's and user's protection.

V. CONCLUSION

In this paper we described the concept of web program isolation in the browser as a response to new security challenges and performance demands introduced with the Web evolution in recent years. We compared the new modular architecture to the monolithic architecture most used in current web browsers and showed that modular architecture is superior.

We reviewed modular architecture implemented in Google Chrome web browser. We analyzed Chrome's behavior concerning the most popular security web browser threats. We showed that modular architecture of Chrome mitigates most serious treats that are related to system compromise and data theft.

However, Chrome's architecture does not provide the full protection. Threats that are related to cross-site attacking, session hijacking and user interface compromise are not mitigated. We reviewed similar architectures implemented in OP, Tahoma and Gazelle web browser. These architectures sacrifice compatibility with the current Web in order to provide higher level of security that Chrome.

REFERENCES

[1] C. Reis and S.D. Gribble, "Isolating web programs in modern browser architectures", Proceedings of the 4th ACM European conference on Computer systems, April 01-03, 2009, Nuremberg, Germany

[2] Jesse Ruderman, "The Same Origin Policy", http://www.mozilla.org/projects/security/components/same-origin.html, 2001.

[3] Mozilla, "Public Suffix List", http://publicsuffix.org/, 2007.

[4] A. Barth, C. Jackson, C. Reis, and Google Chrome Team, "The Security Architecture of the Chromium Browser", Technical report, Stanford University, 2008. http://crypto.stanford.edu/websec/chromium/chromium-security-architecture.pdf.

[5] Microsoft: "Restricted Tokens", February 2010. http://msdn.microsoft.com/en-us/library/aa379316(VS.85).aspx

[6] Indiana University Knowledge Base: "Fully qualified domain name", October 2009. http://kb.iu.edu/data/aiuv.html

[7] G. Steuck: „XXE (Xml eXternal Entity) attack", October 2002. http://www.securiteam.com/securitynews/6D0100A5PU.html

[8] The Open Web Application Security Project: OWASP Top10 - 2010 rc1, "The Ten Most Critical Web Application Security Risks", http://www.owasp.org/images/0/0f/OWASP_T10_-_2010_rc1.pdf

[9] Microsoft: "Mitigating cross-site scripting with HTTP-only cookies", http://msdn.microsoft.com/en-us/library/ms533046.aspx

[10] M. Balduzzi, M. Egele, E. Kirda, D. Balzarotti, C. Kruegel: "A Solution for the Automated Detection of Clickjacking Attacks", ASIACCS'10, Beijing, China, 2010.

[11] M. Zalewski: "Browser Security Handbook", 2009. http://code.google.com/p/browsersec/wiki/Main

[12] C. Grier, S. Tang, S. T. King: "Secure web browsing with the OP web browser", 2008 IEEE Symposium on Security and Privacy

[13] R. S. Cox, J. G. Hansen, S. D. Gribble, H. M. Levy: "A Safety-Oriented Platform for Web Applications", 2006 IEEE Symposium on Security and Privacy

[14] H. J. Wang, C. Griery, A. Moshchukz, S. T. Kingy, P. Choudhury, H. Venter: "The Multi-Principal OS Construction of the Gazelle Web Browser", MSR Technical Report MSR-TR-2009-16