RequestRodeo: Client Side Protection against Session Riding*

Martin Johns and Justus Winter

Security in Distributed Systems (SVS)
University of Hamburg, Dept of Informatics
Vogt-Koelln-Str. 30, D-22527 Hamburg
(johns|4winter)@informatik.uni-hamburg.de

Abstract. The term Session Riding denotes a class of attacks on web applications that exploit implicit authentication processes. There are four distinct methods of implicit authentication found in today's web applications: Cookies, http authentication, IP address based access control and client side SSL authentication. As many web applications fail to protect their users against Session Riding attacks we introduce RequestRodeo, a client side solution to counter this threat. With the exception of client side SSL, RequestRodeo implements protection against the exploitation of implicit authentication mechanisms. This protection is achieved by removing authentication information from suspicious requests.

1 Introduction

Session Riding (also known as "Cross Site Request Forgery") attacks are public at least since 2001 [16]. However this class of web application vulnerabilities is rather obscure compared to attack vectors like Cross Site Scripting or SQL Injection. Session Riding is neither included in the OWASP's top 10 list of web application threats [15] nor in the Web Application Security Consortium's threat classification [3]. As the trend towards web applications continues and an increasing number of local programs and appliances like firewalls rely on web based frontends, the attack surface for Session Riding grows continuously. Session Riding is an attack that targets the user rather than the web application. As long as web applications do not take measures to protect their users against this threat, it is important to investigate possibilities to implement client side mechanisms. In this paper we present RequestRodeo, which is, to the best of our knowledge, the first client-side solution for protection against Session Riding attacks.

* This work was supported by the German Ministry of Economics (BMWi) as part of the project "secologic", www.secologic.org.

2 Technical Background

2.1 Implicit Authentication

With "implicit authentication" we denote processes which cause the web browser to automatically include authentication information in http requests. There are several widely supported methods of implicit authentication:

- **Http authentication:** Http authentication [8] enables the web server to request authentication credentials from the browser in order to restrict access to certain webpages. There are three methods frequently used: Basic, digest and NTLM (a proprietary extension by Microsoft [9]). In all these cases the initial authentication process undergoes the same basic steps (for brevity reasons only a simplified version of the process is given):
  1. The browser sends an http request for a URL for which authentication is required.
  2. The web server answers with the status code "401 Unauthorized" causing the web browser to demand the credentials from the user e.g. by prompting for username and password.
  3. After the user has supplied the demanded information, the web browser repeats the http request for the restricted resource. This request's header contains the user's credentials in encoded form via the `Authorization` field.
  4. The server validates whether the user is authorized. Depending on the outcome, the server either answers with the requested page or again with a 401 status code.

  The browser remembers the credentials for a certain time. If the client requests further restricted resources that lie in the same authentication realm, the browser includes the credentials automatically in the request.
- **Cookies:** Web browser cookie technology [14] provides persistent data storage on the client side. A cookie is a data set consisting at least of the cookie's name, value and domain. It is sent by the web server as part of an http response message using the `Set-Cookie` header field. The cookie's domain value is used to determine in which http requests the cookie is included. Whenever the web browser accesses a webpage that lies in the domain of the cookie (the domain value of the cookie is a valid domain suffix of the page's URL), the cookie is automatically included in the http request using the `Cookie` field. Cookies are often used as authentication tokens by today's web applications. After a successful login procedure the server sends a cookie to the client. Every following http request that contains this cookie is automatically regarded as authenticated.
- **Client side SSL authentication:** The Secure Sockets Layer (SSL) and its successor the Transport Layer Security (TLS) [4] protocols enable cryptographically authenticated communication between the web browser and the web server. To authenticate the communication partners X.509 certificates and a digital signature scheme are used.

All these methods have in common, that after a successful initial authentication, the web browser includes the authentication tokens (either the cookie, the http authentication credentials or the SSL signature) automatically in further requests without user interaction.

IP address based authentication: A special case of implicit authentication is often found in intranets. Instead of actively requesting user authentication, the web application passively uses the request's source IP address as authentication token, only allowing certain IP (or MAC) addresses. Some intranet servers do not employ any authentication at all because they are positioned behind the company's firewall. In this case every web browser that is behind that firewall is authorized automatically.

2.2 Cross Site Request Forgery

"Session Riding" or "Cross Site Request Forgery" (CSRF) [18] is an attack technique that exploits implicit authentication. The attack is executed by causing the victim's web browsers to create hidden http requests to restricted resources. In the case that a successful request for such a resource causes the web application to commit further persistent actions (e.g. like altering database fields, sending email or changing the applications state), these actions are done using the victim's authentication tokens.

Example: a (rather careless) site for online banking provides a HTML form to place credit transfers. This form uses the GET method and has the action URL `http://bank.com/transfer.ext`. The form is only accessible by properly authenticated users, employing one of the techniques described above. If an attacker is able to trick a victim's browser to request the URL `http://bank.com/transfer.ext?amount=10&ano=007`, while the victim's browser maintains an authenticated state for the banking site, the owner of the account with the number 007 might gain €10. To execute the attack the attacker manufactures a harmless appearing webpage. In this webpage the attacker includes HTML or Javascript elements, that cause the victims web browser to request the malicious URL. This can be done for example with a hidden image: `<img src="http://bank.com/transfer.ext?amount=10&ano=007">`. If the attacker successfully lures the victim to visit the malicious website, the attack can succeed.

All access control methods described in Section 2.1 are vulnerable against CSRF as long as no countermeasures against this attack method have been implemented by the web application.

CSRF attacks are not necessarily limited to submitting a single fraudulent request. Workflows that require a series of http requests (i.e. web forms that span over more than one webpage) might be vulnerable as well, as long as certain conditions are fulfilled: The content and identifiers of every step of the workflow's web forms are know prior to the attack and the workflow does not employ a separate mechanism to track the workflow's progress (i.e. a request id) but uses the implicit communicated session identifier. If these conditions are satisfied an attacker can create in most cases a series of hidden iframes that host malicious

web forms. These forms are automatically submitted sequentially via JavaScript using the iframe's `onload`-events, thus simulating a user that is filling the forms in their proper order.

Cross Site Scripting (XSS) [5] and CSRF attacks are frequently confused as they are clearly related. Both attacks are aimed at the user and often require the victim to access a malicious webpage. Also the potential consequences of the two attack vectors can be similar: The attacker is able to submit certain actions to the vulnerable web application using the victim's identity. The causes of the two attack classes are different though. A web application that is vulnerable to XSS fails to properly sanitize user provided data before including this data on a webpage, thus allowing an attacker to include malicious JavaScript in the web application. This JavaScript consequently is executed by the victim's browser and initiates the malicious requests. XSS attacks have more capabilities beyond the creation of http request and are therefore more powerful than CSRF attacks: A rogue JavaScript has almost unlimited power over the webpage it is embedded in and is able to communicate with the attacker. A XSS can e.g. obtain and leak sensitive information.

3 Securing Web Applications against CSRF

This paper mainly focuses on client side protection against CSRF. However it is important to discuss the server side as well in order to understand why CSRF vulnerabilities are still an issue.

3.1 How NOT to do it

There exist misconceptions about possibilities to protect web applications against CSRF attacks:

- **Accepting only http POST requests:** A frequent assumption is, that a web application which only accepts form data from http POST request is protected against CSRF, as the popular attack method of using IMG tags only creates http GET requests. This is not true: Hidden POST requests can be created e.g. by using HTML forms in invisible iframes, which are automatically submitted via JavaScript.
- **Referrer checking:** An http request's referrer [7] indicates the URL of the webpage that contained the HTML link or form that was responsible for the request's creation. The referrer is communicated via an http header field. To protect against CSRF web application check if a request's referrer matches the web applications domain. If this is not the case, the request is usually rejected. Some users prohibit their web browsers to send referrer information because of privacy concerns. For this reason web applications have to accept requests, that do not carry referrer information. Otherwise they would exclude a certain percentage of potential users from their services. It is possible for an attacker to reliably create referrerless requests (see below). For this

reason any web application that accepts requests without referrers cannot rely on referrer checking as protection against CSRF.
In the course of preparing this paper, we conducted an investigation on the different possibilities to create http requests without referrers in a victim's browser. We found three different methods to create hidden request that do not produce referrers. Depending on the web browser the victim uses, one or more of these methods are applicable by the attacker.

1. Page refresh via meta tag: This method employs the "HTTP-EQUIV = Refresh" meta tag. The tag specifies a URL and a timeout value. If such a tag is found in the "head" section of an HTML document, the browser loads the URL after the given time. Example:
   `<META HTTP-EQUIV=Refresh CONTENT="0; URL=http://path_to_victim">`
   On some web browsers the http GET request, which is generated to retrieve the specified URL, does not include a referrer. It is not possible to create POST request this way.
2. Dynamically filled frame: To generate hidden POST requests, the attacker can use an HTML form with proper default values and submit it automatically with JavaScript. To hide the form's submission the form is created in an invisible frame. As long as the `src` attribute of the frame has not been assigned a value, the referring domain value stays empty. Therefore the form cannot be loaded as part of a predefined webpage. It has to be generated dynamically. The creation of the form elements is done via calls to the frames DOM tree [12].
3. Pop under window: The term "pop under" window denotes the method of opening a second browser window that immediately sends itself to the background. On sufficiently fast computers users often fail to notice the opening of such an unwanted window. Such a window can be used to host an HTML form that is submitted either automatically or by tricking the victim to click something. The form can be generated by calls to the DOM tree or by loading a prefabricated webpage. Depending on the victim's browser one of these methods may not produce a referrer (see below for details).

To examine the effectiveness of the described methods, we tested them with common web browsers. See table 1 for the results of our investigation. The only web browser that was resistant to our attempts was Opera.

Method/Browser	IE 5	IE 6*	IE 7**	FF 1.07	FF 1.5	O 8	S 1.2
META Refresh				X	X		
Dynamic filled frame	X	X	X	X	X		X
Pop up window (regular)	X	X	X				
Pop up window (dynamically filled)				X	X		

IE: Internet Explorer; FF: Firefox; S: Safari; O: Opera; *: IE 6 XPSP 2; **: IE 7 (Beta 2)

Table 1. Generating referrerless requests ("X" denotes a working method)

3.2 How to do it

Using random form tokens: To prevent CSRF attacks, a web application has to make sure that incoming form data originated from a valid HTML form. "Valid" in this context denotes the fact that the submitted HTML form was generated by the actual web application in the first place. It also has to be ensured that the HTML form was generated especially for the submitting client. To enforce these requirements, hidden form elements with random values can be employed. These values are used as one time tokens: The triplet consisting of the form's action URL, the ID of the client (e.g the session ID) and the random form token are stored by the web application. Whenever form data is submitted, the web application checks if this data contains a known form token which was stored for the submitting client. If no such token can be found, the form data has been generated by a foreign form and consequently the request will be denied. See [18] for a similar approach.

Using explicit authentication: There are methods to communicate authentication tokens explicitly: Authentication tokens can be included into the web application's URLs or transported via hidden fields in HTML forms. These techniques are resistant to CSRF attacks.

3.3 Reasons for the Existence of CSRF Vulnerabilities

In today's web applications CSRF problems can be found frequently. There are several reasons for this. Compared to vulnerability classes like Cross Site Scripting (XSS) [5] or SQL Injection, CSRF is rather obscure. While in many cases the consequences of CSRF attacks can be as severe as XSS exploits, web application developers are often unaware or dismissive when it comes to this vulnerability class. Furthermore, most web application frameworks lack a central mechanism for protection against CSRF, opposed to e.g. XSS for which numerous filtering functions are provided. In addition, automatic approaches like "taint checker" [11] for detecting SQL Injection and XSS problems do not exist for CSRF.

4 Client Side Protection against CSRF

We propose a client side solution to enable security conscious users to protect themselves against CSRF attacks. Our solution works as a local proxy on the user's computer.

4.1 Concept

As described in Section 2.2 the fundamental mechanism that is responsible for CSRF attacks to be possible is the automatic inclusion of authentication data in any http request that matches the authentication data's scope. Our solution is to partly disable the automatism that causes the sending of the authentication data.

The proxy identifies http requests which qualify as potential CSRF attacks and strips them from all possible authentication credentials. We chose to implement our solution in form of a proxy instead of integrating it directly into web browser technology because this approach enables CSRF protection for all common web browsers.

Identification of suspicious requests: The proxy resides between the client's web browser and the web application's server. Every http request and response is routed through the proxy. Because of the fact that the browser and the proxy are separate entities, the proxy is unable to identify how an http request was initiated. To decide if an http request is legitimate or suspicious of CSRF, we introduce a classification:

Definition 1 (entitled). *An http request is classified as* **entitled** *only if:*

- *It was initiated because of the interaction with a web page (i.e. clicking on a link, submitting a form or through JavaScript) and*
- *the URLs of the originating page and the requested page satisfy the "same-origin policy" [17]. This means that the protocol, port and domain of the two URLs have to match.*

Only requests that were identified to be entitled *are permitted to carry implicit authentication information.*

To determine if a request can be classified as *entitled*, the proxy intercepts every http response and augments the response's HTML content. Every HTML form, link and other means of initiating http requests is extended with a random URL token. Furthermore the tuple consisting of the token and the response's URL is stored by the proxy for future reference. From now on, this token allows the proxy to identify outgoing http requests with prior http responses. Every request is examined whether it contains a URL token. If such a token can be found, the proxy compares the token value to the values which have been used for augmenting prior http responses. This way the proxy is able to determine the URL of the originating HTML page. By comparing it with the request's URL, the proxy can decide if the criteria defined in definition 1 are met. If this is not the case, all implicit authentication information is removed from the request.

Removal of authentication credentials: As discussed in Section 2.1 there are two different methods of implicit authentication used by today's web applications that include credentials in the http header: Http authentication and cookies. If the proxy encounters an http request, that cannot be classified as *entitled*, the request is examined if its header contains `Cookie` or `Authorization` fields. If such header fields are found, the proxy triggers a reauthentication process. This is done either by removing the `Cookie` header field or by ignoring the `Authorization` field and requesting a reauthentication before passing the request on to the server. Following the triggered reauthentication process, all further requests will be *entitled* as they originated from a page that belongs to the web application (beginning with the webpage that executed the reauthentication).

Prevention of IP address based attacks: To protect resources, that filter access based on the request's IP address, the proxy verifies that the destination URL of every request, which has not been classifies as *entitled*, is reachable from the outside. To achieve this, the proxy has to employ an entity that resides outside the client's intranet (see below for details). If the request's destination URL is not "world reachable", the proxy drops the request and replies with a confirmation dialog, to ensure that the request was intended by the user and not part of an CSRF attack.

Client side SSL authentication: Our solution is not yet able to prevent CSRF attacks that exploit client side SSL authentication.

4.2 Implementation

We implemented a proof of concept of our approach using the Python programming language with the Twisted [6] framework. Free Python interpreters exist for all major operating systems. Thus, using our solution should be possible in most scenarios. We call our implementation "RequestRodeo". In the next paragraphs we discuss special issues that had to be addressed in order to enforce the solution outlined in Section 4.1.

Augmenting the response's HTML content: The process of adding the random tokens to a webpage's URLs is straight forward: The proxy intercepts the server's http response and scans the HTML content for URLs. Every URL receives an additional GET parameter called `__rrt` (for "RequestRodeoToken"). Furthermore JavaScript code that may initiate http requests is altered: The proxy appends a JavaScript function called `addToken()` to the webpage's script code. This function assumes that its parameter is a URL and adds the GET token to this URL. Example: The JavaScript code

```
document.location = someVariable;
```

is transformed to

```
document.location = addToken(someVariable);
```

This alteration of URLs that are processed by JavaScript is done dynamically because such URLs are often assembled on script execution and are therefore hard to identify reliably otherwise.

Removal of header located authentication credentials: The following aspects had to be taken into consideration:

- Cookies: If a `Cookie` header field is found in a suspicious request, it is deleted before the request is passed to the server. To ensure compatibility with common web applications, our solution somewhat relaxes the requirements of Definition 1: The proxy respects a cookie's domain value. A cookie is therefore only discarded if its domain does not match the domain of the referring page. Otherwise e.g. a cookie that was set by `login.example.org` with the domain value ".example.org" would be deleted from requests for `order.example.org`.
- Http authentication: Simply removing the authorization data from every request that has not been classified as *entitled* is not sufficient. The proxy

cannot distinguish between a request that was automatically supplied with an `Authorization` header and a request, that reacts to a 401 status code. As the web browser, after the user has entered his credentials, simply resends the http request that has triggered the 401 response, the resulting request is still not *entitled* because its URL has not changed. Therefore the proxy has to uniquely mark the request's URL before passing it on to the server. This way the proxy can identify single requests reliably. It is therefore able to determine if an `Authorization` header was sent because of a "401 Unauthorized" message or if it was included in the message automatically without user interaction.

Whenever the proxy receives a request, that was not classified as *entitled* and contains an `Authorization` header, the following steps are executed (see figure 1):

1. The proxy sends a "302 temporary moved" response message. As target URL of this response the proxy sets the original request's URL with an added unique token.
2. The client receives the "temporary moved" response and consequently requests the URL that was provided by the response.
3. The URL token enables the proxy to identify the request. The proxy ignores the `Authorization` header and immediately replies with a "401 Unauthorized" message, causing the client browser to prompt the user for username and password. Furthermore the proxy assigns the status *entitled* to the URL/token combination.
4. After receiving the authentication information from the user, the client resends the request with the freshly entered credentials.
5. As the request now has the status *entitled*, the proxy passes it on to the server.

An analog process has to be performed, whenever a not *entitled* request triggers a "401 Unauthorized" response from the server. The details are left out for brevity.

Usage of an outside entity to prevent IP based attacks: As described earlier, in intranet scenarios the possession of a local IP address is often considered to be sufficient authentication. For this reason, the proxy has to make sure that the target of a request, that was not classified as *entitled*, is accessible from any host on the internet. We introduce a *reflection service* that is installed on a host outside of the cooperate intranet. The URL of every questionable request is submitted to this service. The reflection service poses a HEAD request for the URL. If this request succeeds, it is save to assume that no IP based authentication mechanism is in place. The outcome of this test is communicated back to the proxy. If the reflection service was not able to access the URL, the proxy withholds the http request and replies with a confirmation dialog instead. Only if the user explicitly confirms his intend of requesting the protected content, the request is passed on to the server.

For performance reasons, the proxy keeps track of all IP addresses that were checked this way. Every IP address is therefore only checked once, as long as

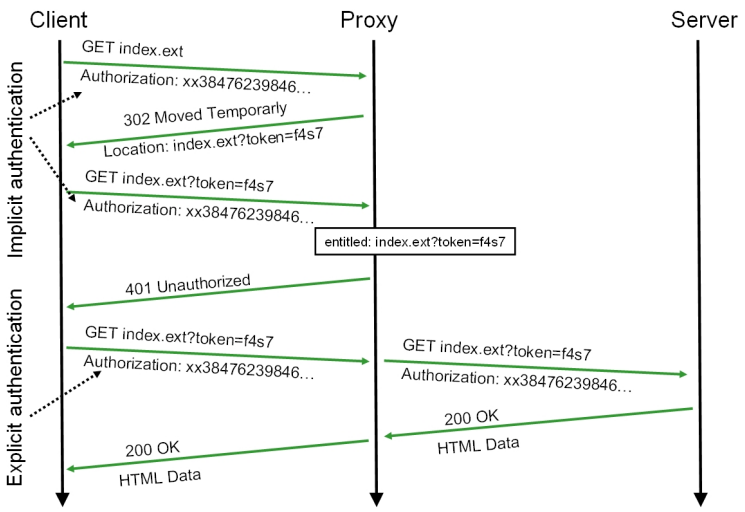

Fig. 1. Intercepting implicit http authentication

the user's IP address does not change. The reflection server will be distributed together with the proxy to enable privacy conscious users to set up their own service.

Notification: Whenever the proxy removes implicit authentication credentials, an unobtrusive notification element is added to the http response's HTML content in order to notify the user about the proxy's action. In our prototype this is done via a small floating sign.

5 Discussion

As described above our solution identifies http requests that pose potential CSRF attacks. For these requests the implicit authentication processes are disabled. With the exception of intercepting requests for intranet resources, the http request themselves are not prevented, only the authentication information is removed. For this reason our solution interferes as little as possible with the usage of web based applications. For example if a web application provides additionally to the restricted resources also pubic content, this public content can be referenced by outside webpages without the interference of the proxy. The requests for these public items may initially contain authentication credentials, which are subsequently removed by the proxy. But this removal does not influence the server's response, as no authentication was required in the first place.

With the single exception of local attacks (see below), the in Section 2.2 described CSRF attacks are prevented reliably, as all http requests originating from an attacker's website or from outside the web browser (e.g. from an email application) are identified as not being *entitled*

5.1 Limitations

Our solution cannot protect from "local" CSRF attacks. With local CSRF attacks we denote attacks that have their origin on the attacked web application. If e.g. an application allows its users to post images to one of the application's webpages, a malicious user may be able to use the image's URL to launch a CSRF attack. Our proxy would consider the image request as *entitled* as the image is referenced by a webpage that belongs to the application.

As mentioned above, our solution is furthermore not able to prevent CSRF attacks on client side SSL authentication. This issue could be solved, if the proxy instead of the browser would handle the client side SSL authentication.

Some webpages use JavaScript to create parts of the page's HTML code locally. As Javascript is a highly dynamic language, our current implementation may fail in some cases to correctly classify all included URLs as *entitled*. For this reason, we are considering the implementation of strict referrer checking as a second line of defense in certain cases.

We designed our solution to interfere as little as possible with a user's browsing. The most notably inconvenience that occurs by using the proxy is the absence of auto login: Some web applications allow the setting of a long lived authentication cookie. As long as such a cookie exists, the user is not required to authenticate. In almost every case, the first request for a web application's resource is not *entitled*, as it is caused either by entering the URL manually, selecting a bookmark or via a web page that does not belong to the application's domain. For this reason the proxy removes the authentication cookie from the request, thus preventing the automatic login process.

5.2 Future Work

As noted above, our solution does not yet protect against attacks on client side SSL authentication. An enhancement of our solution in this direction is therefore desirable.

Another future direction of our approach could be the integration of the protection directly into the web browser. This step would make the process of augmenting the HTML code unnecessary, as the web browser has internal means to decide if a request is *entitled*. Furthermore, such an integration would also enable protection against attacks on client side SSL authentication, as no interception of encrypted communication would be necessary. As noted above, we decided to implement our solution at first in form of a local web proxy to enable a broad usage with every available web browser.

5.3 Related Work

To date, little attention has been paid to CSRF attacks. The work conceptually closest to ours is NOXES [13], a client side proxy for protection against Cross Site Scripting attacks. NOXES prevents the communication of sensitive data (like session identifiers) to third parties by disallowing dynamically generated http

requests. Therefore, a malicious JavaScript is not able to leak information that was gathered on runtime, as only http requests that were statically embedded in the HTML code are allowed to pass the proxy.

Google's "Safe Browsing Toolbar" [10] is an extension for the Firefox web browser. This extension intends to provide client side protection against "phishing" attacks. The protection is done by sending information about every visited webpage to a central entity that compares these information to a database of known phishing attacks. If the submitted information matches one of the stored signatures, the web browser displays a warning.

Furthermore, from a technological point of view, related approaches can be found in the domain of personal web firewalls like WebCleaner [1]. These firewalls are also implemented as a client side web proxy that intercepts all http communication.

6 Conclusion

In this paper we presented RequestRodeo, a client side solution against CSRF attacks. Our solution works as a local http proxy on the user's computer. RequestRodeo identifies http requests that are suspicious to be CSRF attacks. This is done by marking all incoming and outgoing URLs. Only requests for which the origin and the target match, are allowed to carry authentication credentials that were added by automatic mechanisms. From suspicious requests all authentication information is removed, thus preventing the potential attack. Furthermore, special measures have been implemented to protect local resources in the client's intranet. For this reason we introduced the concept of a reflection server, which assures that the requested resources are not protected by external mechanisms like firewalls, before allowing http requests that have a non local origin to access these resources. By implementing the described countermeasures RequestRodeo protects users of web applications reliably against almost all CSRF attack vectors that are currently known.

References

1. Webcleaner - a filtering http proxy. [application], <http://webcleaner.sourceforge.net>, (03/20/06).
2. Jesse Burns. Cross site reference forgery - an introduction to a common web application weakness . Whitepaper, <https://www.isecpartners.com/documents/XSRF_Paper.pdf>, 2005.
3. Web Application Security Consortium. Threat classification. whitepaper, <http://www.webappsec.org/projects/threat/v1/WASC-TC-v1_0.pdf>, 2004.
4. T. Dierks and C. Allen. The tls protocol version 1.0. RFC 2246, <http://www.ietf.org/rfc/rfc2246.txt>, January 1999.
5. David Endler. The evolution of cross-site scripting attacks. Whitepaper, iDefense Inc., 20. May 2002.
6. Abe Fettig. *Twisted Network Programming Essentials*. O'Reilly, first edition edition, October 2005.

7. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. RFC 2616, <http://www.w3.org/Protocols/rfc2616/rfc2616.html>, June 1999.
8. J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. Http authentication: Basic and digest access authentication. RFC 2617, <http://www.ietf.org/rfc/rfc2617.txt>, June 1999.
9. Eric Glass. The ntlm authentication protocol. [online], <http://davenport.sourceforge.net/ntlm.html>, (03/13/06), 2003.
10. Google. Safe browsing for firefox. [application], <http://www.google.com/tools/firefox/safebrowsing/>, (03/20/06), 2006.
11. Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th conference on World Wide Web*, pages 40–52. ACM Press, 2004.
12. Philippe Le Hégaret, Ray Whitmer, and Lauren Wood. Document object model (dom). W3C recommendation, <http://www.w3.org/DOM/>, January 2005.
13. Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: A client-side solution for mitigating cross site scripting attacks, security. In *Security Track of the 21st ACM Symposium on Applied Computing (SAC 2006)*, April 2006.
14. D. Kristol and L. Montulli. Http state management mechanism. RFC 2965, <http://www.ietf.org/rfc/rfc2965.txt>, October 2000.
15. OWASP. Top ten most critical web application security vulnerabilities. whitepaper, <http://www.owasp.org/documentation/topten.html>, January 2004.
16. John Percival. Cross-site request forgeries. [online], <http://www.tux.org/ peterw/csrf.txt> (03/09/06), June 2001.
17. Jesse Ruderman. The same origin policy. [online], <http://www.mozilla.org/projects/security/components/same-origin.html> (01/10/06), August 2001.
18. Thomas Schreiber. Session riding - a widespread vulnerability in today's web applications. Whitepaper, SecureNet GmbH, <http://www.securenet.de/papers/Session_Riding.pdf>, December 2004.