Binonymizer – A Two-Way Web-Browsing Anonymizer

Tim Wellhausen
Gerrit Imsieke
(Tim.Wellhausen, Gerrit.Imsieke)@GfM-AG.de

12 August 1999

Abstract

This paper presents a method that enables Web users to surf anonymously such that neither the requested content nor the referencing URLs are traceable by someone who has access to the user's local storage, especially the history files. This is achieved by scrambling the "filename part" of the accessed URLs on a per-session basis. The method builds on standard encryption techniques so that packet sniffing is also made impracticable. These features provide the user with privacy that he might desire as an employee in a company that analyzes the employees' surfing behavior. A combination of these techniques with already established anonymizing proxy services[1] is presented. This provides increased privacy both towards the Web service and towards the employer. The latter is true because, together with the encryption and URL scrambling, this obscures the Web server's host name, only exposing the proxy's name.

1 Introduction

World Wide Web users expose information about their interests to many parties:

- Sensitive information like credit card numbers together with the user's name may be extracted of un-encrypted TCP/IP packets.
- Users may be re-identified by Web services through the use of cookies.
- Users' surfing behavior may be tracked by analyzing their history and cache files.

The first problem has been addressed by encryption technology like Netscape's Secure Socket Layer (SSL) which emerged to the IETF standard Transport Layer Security (TLS) [1]. SSL/TLS provides security by encrypting the TCP traffic between a browser and a server. The amount of security depends on the length of the key used. Even with key lengths of 40 bits mass sniffing of all traffic that is caused by Web surfing becomes

[1] e.g., anonymizer.com

impracticable, since the value of the gained information doesn't outweigh the costs of decoding.

Users may disable cookies and refuse to give their real name in order to prevent Web services from filing their surfing behavior. In addition, there are anonymizer servers that retrieve specified URLs for the user, concealing all details about the user from the Web service.

But current anonymization techniques only address one side of the problem: concealing a Web surfer's identity from a remote service. Many organizations make use of automatic traffic filtering and analysis tools. These include general packet sniffing, dedicated E-mail filters, and Web browser cache and history file analyzers[2].

Users can protect themselves by disabling caching and deleting history files. But since in larger organizations user data is often stored on network devices, tools may be conceived that analyze history files in real-time, access the surfed URLs, and retrieve the content. In the most widespread current browsers, there are no means of disabling history file logging. But even if there were, by imposing mandatory user profiles organizations could prevent users from reconfiguring their browsers. Since users usually don't bother to protect their privacy, it is desirable that they are lead to secure surfing by the Web services, i.e., the Web services provide secure surfing facilities and place links to these facilities on their Web pages.

The current paper doesn't try to suggest how browsers could be improved to provide more security, but takes the currently available browsers like Netscape 3.x/4.x and Microsoft 3.x/4.x/5.x as "given fate". Strategies and technologies that provide increased privacy rely on how these browsers handle caching, history logging, and secure connections.

The remainder of this paper is organized as follows: section two explains the requirements for the system that we propose. Section three presents the architecture, section four the implementation of a working prototype. Section five concludes this paper.

2 Requirements

To provide better privacy it is essential that the whole process is completely transparent for users. Therefore, our approach has to avoid the installation of a separate tool for clients. A standard Web browser should be sufficient. The only requirements for browsers are that they are able to open secure connections and accept certificates used by the servers.

The resulting system will be used in a commercial environment, consisting of standard software like Web servers, application servers and databases. The changes that are needed to employ this system in an existing environment have to be small. In our approach, we only add functionality to an existing Web server without any further changes to other server side applications.

Furthermore, the service the system provides has to be fast. It is possible to use this service for many parts of a web site or even the whole site. If this service proves to be the bottleneck for high performance it is very likely that it won't be deployed.

[2] Examples for tools???

With the existence of dedicated tools that filter and analyze the surf behavior of users, it is important that the technology presented in this paper is "sufficiently" secure. The authors don't recommend using this service exclusively in an environment with very high security requirements because of possible vulnerabilities of the underlying encryption technology. But, as mentioned earlier, it has to be impractical to log all activities of users surfing on a web site enhanced with this technology.

3 Architecture

The system consists of two main parts: a Web server that is enhanced by several standard modules and the "scrambler". The Web server is responsible to answer user requests and to send the required documents to the clients. The scrambler resides in an independently running process that communicates with the Web server. The two main functions of the scrambler are scrambling plain URLs and resolving scrambled URLs (SURLs).

The following figure shows the architecture of the system.

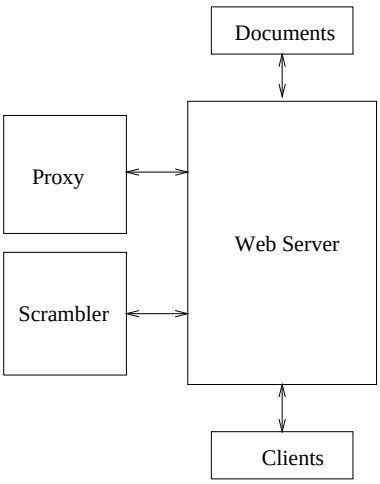

Figure 1: Architecture

For a better understanding how the system works, we demonstrate the life cycle of a scrambled URL:

The first user request consists of a plain, valid URL. The server locates the appropriate document and parses it. Each URL found in this document is sent to the scrambler. Now, two different cases have to be distinguished: either the URL refers to a document that resides locally on the same server or it is an external URL. In the latter case, a proxy is needed to retrieve the requested documents from external servers. This proxy is described in a section of its own. In either case, it doesn't matter which media type is referenced and which HTML tag is used.

After a document is parsed and all URLs are substituted, the Web server sends the document to the user. This connection is secured by the use of a SSL or TLS connection between the Web server and the client. Therefore, the content of the document is protected against packet sniffi ng.

Current Web browsers store URLs of visited web sites in a history list, even URLs of securely received documents. This makes it possible for a tool to retrieve the same document the user has requested. To avoid this attack SURLs are only valid in the context of the same SSL/TLS session between client and server.

If the user requests an SURL the browser sends the SURL back to the Web server. The Web server is able to determine whether the requested URL is scrambled and contacts the scrambler, if necessary. The scrambler receives the SURL and examines it. If it is not valid within the current session it denies access to the requested document. Otherwise, it returns the appropriate plain URL to the Web server. Now, the Web server can retrieve the new document.

Proxy Server

The scrambler is able to process both local URLs, i.e. URLs of documents that reside on the same server, and URLs of documents that reside on other servers. The Web server can easily retrieve a local document and send it to the user. But additional functionality is needed to process external documents.

In the latter case, the Web server has to forward the request to the referenced server, retrieve the requested document, scramble it and send it to the user. For the user, it is not obvious that the requested document doesn't reside on the same server.

This functionality is provided by a proxy server that retrieves requested documents and stores them in a separate cache. This cache accelerates the access to the external resources in case that they are requested again.

4 Implementation

Several existing technologies have been adopted to create a working prototype. We used the open-source Web server Apache, enhanced by several existing modules, and implemented the scrambler as a Java application.

The following publicly available packages have been used:

- Apache Web server – version 1.3.6, including the standard modules mod_rewrite and mod_proxy
- SSL library openSSL – version 0.9.3a
- Apache module SSL – version 2.3.10
- Sun Java Servlet Development Kit – version 2.0
- Apache module JServ – version 1.0
- Apache module JSSI – version 1.1.2

The openSSL library combined with the Apache module SSL provides the functionality for secure connections. Sun's JSDK together with the Apache module JServ provides support for using Java servlets. JSSI is a servlet that includes parsing functionality.

The following figure demonstrates the architecture of our implementation:

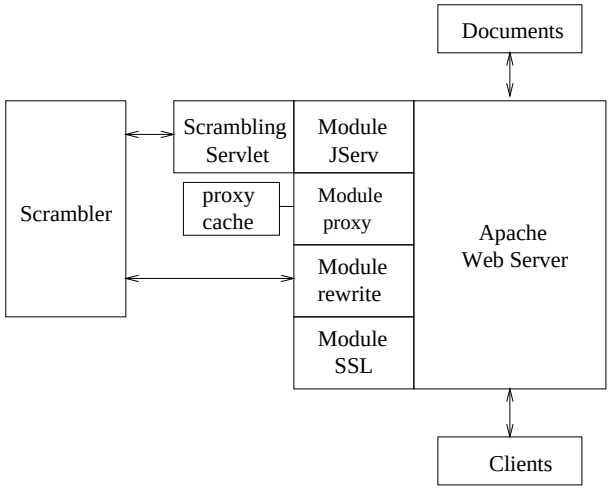

Figure 2: Architecture of the implementation

4.1 The URL scrambler

In our prototype, the URL scrambler is written in Java and starts as a separate process. On the one hand, this may have performance disadvantages. On the other hand, the flexibility of Java made it possible to develop the prototype quickly.

The scrambler communicates with the Apache server over sockets. One communication channel is needed to connect with the document parser. The other channel connects with the module "rewrite" that rewrites requested URLs.

For each user connected to the server the scrambler maintains information about the session. This information consists of the SSL session id and a hash table of URL-SURL pairs, which ensures the fast lookup of requested SURLs. The SSL session id is a string with 64 hexadecimal characters. It is created by the SSL Apache module and can be assumed to be unique.

Each time a URL is sent, the scrambler tries to find an existing session. If no session is found a new session is created. Otherwise, the existing session is reused. The SURL that is created subsequently consists of the session id and a document id that is unique for this session.

Whenever an SURL is received, the scrambler retrieves the session id and checks whether this session id is valid, i.e. whether it is identical to the current SSL session id. In this case, a lookup follows into the appropriate hash table to find the URL

belonging to this SURL. If there is a URL, it is returned. Otherwise, the SURL is returned unchanged, leading to an error message generated by the Web server.

For example, the URL in an HTML document might look like
`/test/test.html`,
whereas the generated SURL the scrambler produces might look like
`https://localhost:8443/secure/83628AF54D[...]8A8234F0000`.

4.2 The servlet module

We used a servlet engine and an existing servlet written for the Apache Web Server to facilitate the communication with the scrambler while parsing a requested HTML document. This servlet parses the document and provides hooks to add special handlers that cope with specified HTML tags/attributes.

We used this functionality to create handlers for all HTML tags that contain references to other resources. These handlers extract the referenced URLs, send them to the scrambler and replace them with the resulting SURLs.

4.3 The URL rewrite module

The standard Apache module "mod_rewrite" is used for translating incoming SURLs into URLs. If installed this module can be configured to translate arbitrary requested URLs into another form. Rules determine which requested URLs the module modifies.

In our case, the module starts a perl script that communicates with the scrambler over a socket. Every time a requested URL has the form "`/secure/...`" this script sends the requested URL to the scrambler and gives the result back to the rewrite module. The rewrite module in turn makes the Web server process the modified URL instead of the requested one.

5 Limitations

Although the system is working well, there are several limitations some of which are caused by the modules we have chosen to adopt. But there are also general restrictions of such a system.

The servlet engine that we use allows only to start servlets for documents with specified extensions. For example, a document called start.html has the extension .html and is recognized. Documents that don't have a common extension won't be recognized and, therefore, won't be scrambled.

Documents that incorporate script code, for example JavaScript which is commonly used, cause more serious problems. Within the script code URLs can be stored that are not easily detectable because they are not embraced by special tags. It might be possible to apply heuristics in such cases, but these heuristics will presumably not always work.

Furthermore, new technologies are emerging rapidly that do not rely on standard HTML. Using Java/XML solutions that render data on the client's side, for example, makes it impossible to use a scrambling service on a Web server. Every time new technologies emerge changes for the scramble might be necessary to keep it working.

6 Conclusion

This paper gave a short overview over existing issues in the area of privacy for WWW users. It demonstrated that it is possible to increase the protection of privacy of users without adding new technologies on the client side. Rather, it demonstrated and explained a system that enhances Web servers. This system is capable of modifying URLs in HTML documents is such a way that it is only possible for the same user to retrieve the according documents.

The authors are aware that this system doesn't provide perfect protection for users. For example, it might still be possible to create tools that manipulate Web browsers directly to get access to the content of a retrieved document.

On the other hand, the proposed system is one more step to protect users from being kept under surveillance. Therefore, the authors would like to encourage web site administrators to use the proposed system to offer their users more security.

References

[1] Specifi cation of Transport Layer Security (TLS) protocol v1.0:
`ftp://ftp.isi.edu/in-notes/rfc2246.txt`