Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Information Security Group
Prof. Dr. David Basin

Design and Implementation of a Rewriting Forward Proxy

March 22, 2005

David Fuchs
fuchsd@student.ethz.ch

Supervision:
Michael Näf

Abstract

The objective of this semester thesis is the design and implementation of a forward HTTP proxy for the ETHZs Information Security Laboratory. While providing basic internet access, the proxy protects external sites from malicious activities from within the lab's network, like SQL injections or HTTP header based attacks.

The proxy solution presented in this document filters HTTP traffic more strictly than conventional web proxies enhanced with filtering rules do. The basic rule is that no unvalidated client-provided information is forwarded to external web resources. Access is only allowed to "known" URLs: Requests are forwarded only if they are contained in a pre-loaded table of allowed web resources or were identified on previously requested html pages. Passing data via the HTTP GET or POST methods is only permitted for selected web resources, and the passed values are validated in a strict manner.

Keywords: forward web proxy, rewriting web proxy, internet security, information security

Contents

Chapter 1

Introduction

1.1 Purpose

The Information Security Group at ETHZ offers an Information Security Laboratory course.[1] The course covers various aspects of operating system and application security and takes place in a separate laboratory environment. The laboratory network is connected to the regular ETHZ network infrastructure via a proxy server and a firewall, and network traffic from and to the Lab is reduced to the necessary minimum.

The Lab consists of 11 Linux workstations and one Linux infrastructure server providing common services. To get a rough estimate about how much WWW traffic the clients generate, the number of issued requests was measured during several course units. The number of requests per hour turned out to be fairly low, peaking at some 1500 requests per hour.

Basic access to internet resources is essential for course students, e.g. for downloading some tool or browsing a manual. At the same time, it is also necessary to protect external sites from (possibly unintended) malicious activity since many of the tools and techniques covered by the course are potentially dangerous. Specially crafted HTTP requests can be used to exploit security weaknesses in remote web servers. An exploit might inject undesired content into a web server's back-end database or completely crash a web server. Many such security weaknesses are known, and many scripts and programs exploiting them are available. The tools utilized by the course students can only be used used against the lab's own infrastructure. However, due to careless or malicious operation of a tool, a remote site might be affected by activities within the Lab.

The presented solution provides basic internet access for the Lab's clients while protecting external sites from malicious activity. The main design goal was the protection of external resources from *unintended* attacks. The proxy works in many ways like a regular forward web proxy. Web

[1] `http://www.infsec.ethz.ch/lab/appliedlab`

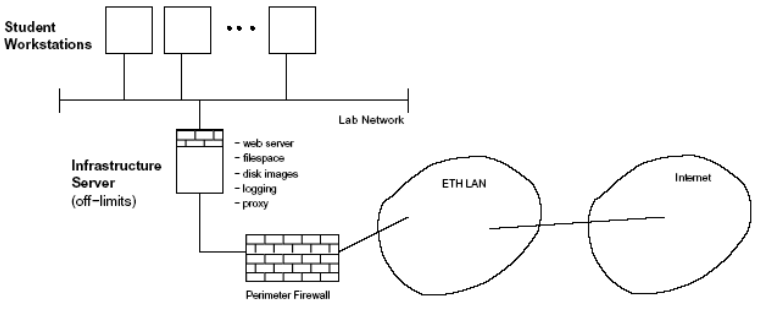

Figure 1.1: Information Security Laboratory

clients configure the proxy's address and port and are subsequently able to browse the WWW via the proxy. HTTP requests are delivered to the proxy, which in turn issues a new request to fetch the requested web page from the actual (external) web server. After receiving the response, the proxy delivers the page to the client.

In addition to this functionality, the proxy does not forward any unvalidated client-provided information to external sites. This is achieved by parsing all web pages retrieved by clients and identifying the hyperlinks contained therein. Incoming requests from clients are forwarded only if the requested URL is found in a pre-loaded table of allowed URLs or has been identified on a previously requested HTML page. Attempts of injecting or altering variable values via the requested URL are therefore blocked by the proxy. For example, a client that requested a HTML page containing the link `http://example.com/somescript.php` will subsequently be able to retrieve this link; an attempt to retrieve `http://example.com/somescript.php?user=admin&loggedin=1` (a possible privilege escalation) will fail.

The pre-loaded table of allowed web resources contains the URLs of commonly used portals like Google[2], ODP[3], SecurityFocus[4] , etc. to which access is always allowed. This table is defined by a Lab administrator.

Besides checking the requested URL, the proxy also replaces any client-provided information in other HTTP header fields by its own, administrator-configured defaults. As an exception, certain HTTP header fields that are necessary for the request are forwarded after validation. Requests using HTTP methods other than GET are generally not allowed; the exception to this being the client data handlers presented in the following paragraphs.

The HTTP protocol allows clients to send data to a web server as part of a request, using the GET or POST method. This mechanism is often used to submit information typed into a form to a web resource processing that information (e.g. a username and a password). Many web resources exhibit security weaknesses when confronted with unexpected client-provided data like unknown encodings or very long strings. Following the basic principle that no client-provided information is forwarded to external web resources, requests containing client-provided data are generally blocked at the proxy.

However, some sites (such as Google) are pretty much useless unless a client is able to pass some data (such as a search query) to the associated web server. The proxy thus provides client data handlers that allow clients to pass client-provided data to selected web resources. Before being forwarded to an external server, all data contained in the request is validated against a predefined set of rules. To this end, an administrator can decide on a set of parameter names and legal ranges for the corresponding values on a per-URL-basis. Whenever a client issues a request containing client-provided data, it will be blocked unless a matching rule allowing the given data is found.

1.2 Terminology

1.2.1 Terms

The following terms are used throughout this document.

- Lab, Security Lab.
  The Information Security Laboratory of the Information Security Group at ETH Zurich.

- Web resource.
  Any service reachable over an internet connection. Usually, this will be a web server. An external web resource is any web resource that is not part of the Security Lab.

[2] if you really don't know Google, check http://www.justfuckinggoogleit.com/ .
[3] http://dmoz.org/
[4] http://www.securityfocus.com/

- Client, user agent.
  A program that sends requests to a web resource or to the proxy for forwarding to the actual resource. Usually the user's browser.

- Client-provided data.
  Data sent by a client as part of a request. This includes the query part of the URL and the request body, but not information contained in header fields. Client-provided data is usually user-provided, e.g. as the input to a HTML form.

- Client-provided information.
  Any information contained in a request. This includes client-provided data, the requested URL, and all header fields sent to a web resource.

- Client data handler.
  Part of the proxy application that identifies client-provided data in a request and decides whether these data will be forwarded to an external web resource. The decision is based on a set of rules defined by a Lab administrator.

1.2.2 Notation

When a mechanism is described in generic grammar, the notation stated below is used.

`<.>`	Indicates a value.
`(.)`	Elements in parantheses are treated as a single element.
`(.|.)`	The pipe indicates a selection.
`[.]`	Elements enclosed by square brackets are optional.
`(.)*`	Elements followed by "*" must be present 0 or more times.
`(.)+`	Elements followed by "+" must be present 1 or more times.
`SP`	White space.
`CRLF`	Line break.
`"string"`	Double-quoted strings or literals stand for themselves.

Chapter 2

Design and Technical Specification

2.1 Cryptographic Primitives

The method chosen to implement the desired functionality is based on the notion of a *Message Authentication Code* (MAC). A MAC algorithm accepts as input a secret key and an input string, and computes a fixed-length representative of the input string. The correctness of a MAC appended to a string is verified using the same secret key.[1] With knowledge of the secret key, alterations of the string can be detected by checking the MAC's correctness. MACs are therefore often called checksum, although many checksum algorithms are not MACs since they do not ensure integrity and authenticity in a cryptographic sense. Without knowledge of the secret key, computation of the MAC is computationally infeasible.[2] A MAC therefore ensures:

- **Message integrity:** Alterations of the string are detected as the MAC is no longer correct.
- **Message authenticity:** Only someone knowing the secret key can compute a correct MAC for a given input string. Authenticity of the MAC is conditional to the authenticity of the secret key.

Note that MACs are only used to ensure a string's integrity and authenticity; the string itself remains unchanged, i.e. MACs do not ensure confidentiality.

There exist different families of MAC algorithms: based on *one time pads*, *block ciphers*, *stream ciphers*, and *hash functions*. Hash functions are well-known cryptographic primitives and are a special case of *one-way functions*. A cryptographic hash function is a function

$$h : A \to B \text{ with } \|B\| \ll \|A\|$$

and the following characteristics:

- The computation of $h(a)$ is efficient, for any $a \in A$.
- **Preimage resistance:** h is a one-way function, i.e. given any $b \in B$, it is computationally infeasible to compute an $a \in A$ such that $h(a) = b$.
- **Second preimage resistance:** for a given input a, it is computationally infeasible to find an input $\hat{a}$ such that $h(a) = h(\hat{a})$.
- **Collision resistance:** it is computationally infeasible to find two different values a and $\hat{a}$ such that $h(a) = h(\hat{a})$. Second preimage resistance is implied by collision resistance.

[1] Note the difference to *Digital Signatures*, which are based on public-key cryptography and can be verified *without* knowledge of the secret key.

[2] Unconditionally secure MACs based on encryption with a one-time pad have also been proposed. The ciphertext of the message authenticates itself, as nobody else has access to the one-time pad. However, there has to be some redundancy in the message. An unconditionally secure MAC can also be obtained by use of a one-time secret key.

Informally, a hash function computes a unique and distinct value (*"fingerprint", "digest"*) for any input. Hash functions used in cryptography usually take an input of arbitrary length and return a fixed-length output. Popular hash functions include MD5[RFC1321] and SHA[SHS]. Common hash functions iteratively process *blocks* of input data in order to hash large inputs.

Many MAC algorithms rely on cryptographic hash functions. They are often called HMACs or *keyed hash functions*, since the hash is computed with the original string plus the secret key as the input for the hash function. A secure HMAC algorithm $HMAC_k(\cdot)$ with secret key k can be constructed from a cryptographic hash function $h(\cdot)$ with block size b as

$$HMAC_k(a) := h(\overline{k} \oplus opad \,|\, h(\overline{k} \oplus ipad \,|\, a)) \;,$$

where $\overline{k}$ is the secret key k padded with 0s so that $\overline{k}$'s length equals b, *opad* and *ipad* are two fixed b-bit constants, $\oplus$ is the bit-wise Exclusive Or operator, and $|$ stands for concatenation. The construction of a HMAC from a hash function as described above is defined in [RFC2104]. A HMAC could also be obtained from a hash function by simply using a concatenation of the secret key and the input string as the hash function's input. The reason why HMAC was defined as above in [RFC2104] is that this design offers significant analytical advantages, so that stronger security results can be proven. A detailed description and analysis of the HMAC function can be found in [BCK96].

2.2 Proxy Architecture

2.2.1 HMAC Tickets

When fetching a HTML document for a client from an external web server, the proxy first parses the received document. HTTP URLs contained therein are identified and tagged with a HMAC computed over the URL and the proxy's secret key. The HMAC is the *ticket* granting access to the proxy when this URL is later requested by a client. Relative URLs are first resolved against the document's base URL, as does a user agent when requesting such a relative URL. They are then rewritten as absolute URLs tagged with a ticket. Replacing relative by absolute URLs is not strictly necessary but advisable, since the resolution against the base URL may be done in different ways. For example, the relative URL `../images/example.gif` within the document `http://example.com/example/example.html` may be resolved as `http://example.com/example/../images/example.gif` or as `http://example.com/images/example.gif`. Both resolutions comply with RFC standards, however if a client resolves a relative URL in a different manner than the proxy, the URL's ticket will no longer be valid. This ambiguity is avoided if all identified URLs are rewritten in their absolute form.

After rewriting the URLs contained in the document, the document is forwarded to the client that requested it. Upon receiving a client's request with a ticket tagged to the URL, the proxy can use the ticket to verify that it has indeed seen this exact URL before.

2.2.2 Rule List

Access to some web resources may always be granted or denied, regardless of the existence of a ticket. The proxy's rule list is the administrator's tool for specifying processing rules for the proxy on a per-URL-basis. Each rule consists of the following items:

- URL(s) the rule will be applied to.
- A target (allow or deny).
- A rule name and description for informational purposes only.
- An optional set of parameters. This is the input for the client data handlers to check whether client-provided data included in the request may be forwarded. (See section 2.2.3.)

If a request containing no ticket is received, the proxy performs a look-up in the set of rules. If a rule denying the request is found, it will be blocked. If a rule allowing the requested URL is found, the request is forwarded. All other requests are blocked by the proxy.

The rules in the rule list have preference over the tickets; i.e. a request that matches a rule with target "deny" will be blocked even if it is tagged with a valid ticket. Denying rules have preference over allowing rules; i.e. a request matching both a denying and an allowing rule will be blocked.

2.2.3 Validating Client-Provided Data: Data Handlers

The HTTP protocol [RFC2616] allows clients to pass client-provided data along with a HTTP request. Data is passed either via the GET method as part of the URL (the URL's *query part*) or via the POST method as part of the request body. The format and actual meaning of the data are not defined by [RFC2616]; it depends on the resource serving the request. Apart from the constraint that binary data and strings containing reserved characters such as "/" must use a suitable encoding if passed via GET, arbitrary values can be passed.

When used without proper validation by the web resource, such client-provided data can often be used as a means to inject wrong variable values into scripts or programs or exploit security vulnerabilities of applications running at the external resource. This can, for example, be used to gain unauthorized access to web pages, inject false values into databases, destroy data or run malicious scripts. Client data sent to external resources is thus minimized by the proxy. For some resources, however, the possiblity of sending client-provided data is an important requirement. Search engines like Google, for instance, are useless without the possibility of submitting a query string. For those resources the proxy forwards client-provided data after validating it. Client-provided data values thus have to be identified, parsed, and matched against a list of allowed values by the proxy.

To describe the structure of client-provided data, its decomposition into parameters has been chosen. A rule allowing client-provided data to be sent to a given URL therefore has one or more parameters associated with it. A parameter consists of

- The parameter name.
- The legal range for the according value.
- A HTTP method (GET or POST).
- An optional "required" attribute if the parameter is to be present. Requests without this parameter will not match the respective rule. By default, parameters are not required.

The HTTP method must be specified because, although unlikely, a parameter that is legal if passed via GET might still have undesired effects if passed via POST, and vice versa. Parameters can be marked as "required" to protect external applications that rely on the existence of a certain parameter without any prior checks, and may fail if the parameter is not present. However, discovering which parameters are required for which application is probably very difficult for an administrator in practice, so this might be a rather theoretical[3] benefit.

The validation of the client-provided data is performed by a client data handler. A client data handler is generated for the parameter set of each rule in the rule list. Requests containing parameters not in the parameter set will not match the rule.

No client-provided data[4] is sent to an external web resource unless a matching allowing rule for the given request is found. Otherwise, upon receiving a POST request or a GET request with a query part in the URL and no ticket, the proxy refuses to serve that request. To check whether a request matches a rule, the client data handler checks whether only allowed parameters are passed

[3] In theory, there is no difference between theory and practice, so it still *is* a benefit.

[4] The query part of a URL with a valid ticket is not regarded as "client-provided", since it was included in the ticket computation and thus must have been present on an external site.

and whether all parameter values are in the legal range specified for them. Only then will the request be forwarded.

The decomposition of client-provided data into parameters is particularly useful if data is encoded using the *application/x-www-urlencoded* format, a very common encoding method. The x-www-urlencoded content type is defined by W3C [HTML40] as the default HTML enctype when submitting form data and has the following format:

```
<parameter name>"="<parameter value> ("&"<parameter name>"="<parameter value>)*
```

White spaces are replaced by "+". Reserved characters such as line brakes and slashes are encoded as "%HH", a percent sign and two hexadecimal digits representing the ASCII code of the character.

Some web resources use non-standard encoding schemes such as

```
<value 1>";"<value 2>";" ...  .
```

For schemes like the above, the decomposition of the data into parametes is not very convenient. Nevertheless, this or similar schemes can still be modeled by regarding the whole data string as a single parameter with an empty name.

The decomposition approach is not adequate for forms using the *multipart/form-data* enctype. The form-data encoding uses a MIME encoding as defined in RFC2045. W3C's HTML standard requires the use of the form-data enctype for submitting forms that contain files, non-ASCII characters and other binary data. Form-data encoded data can only be submitted with the POST method. The MIME encoding (which is somewhat more complicated than the x-www-urlencoded enctype) cannot be modeled with the chosen decomposition approach. Therefore form-data based data submissions are currently not supported by the proxy, i.e. requests containing file uploads or similar client-provided data are always blocked.

2.2.4 Action Upon Requests

Summarizing the above, the proxy's actions upon a request are defined as follows. The conditions are evaluated top to bottom.

1. If a matching denying rule is found, the request is blocked.
2. If a matching allowing rule is found, the request is forwarded. The rule's client data handler ensures that client-provided data matches the parameter set defined for the rule. If present, a ticket attached to the URL will be removed.
3. If the request contains POST data, the request is blocked.
4. If the request has attached a valid ticket, it is forwarded. The ticket is removed from the URL. The request may contain GET data, which is included in the computation of the ticket since it is part of the URL.
5. The request is blocked.

2.2.5 Other Client-Provided Information

Apart from the requested URL, the HTTP method to use, the HTTP version and possibly POST or GET client-provided data, a HTTP request may contain additional information in the form of header fields. The generic format of a header field is

```
<Header-Field-Name> ":" SP <Header-Field-Value> CRLF
```

i.e. each field comprises a separate line in the header. Header fields are usually set by the user agent. As with client-provided data (e.g., values from a HTML form), header fields can cause damage to external web resources. For example, a field value not allowed by the HTTP standard might cause a buffer overflow in a web server.

Some header fields contain information that will only be used by the remote web resource for logging purposes, statistical analysis, tailoring of the response towards a certain user agent or the like. Such information can be stripped off the request without violating the HTTP protocol. Other header fields, however, cannot simply be stripped off. The "Host" header field, for instance, is a required field in version 1.1. At large, the proxy can take one of the following actions for each header field of a client's request:

- Forward the field as-is without any checks. This should generally be avoided since it violates the basic rule that no client-provided information is forwarded to external resources.
  In some cases (Pragma header, for example), the HTTP standard requires proxies to forward fields without any modifications. A trade-off between RFC compliance and the principle of not forwarding unvalidated data is necessary. I decided to completely adhere to the basic rule and thus violate the RFC standard. Thus, no header field is forwarded as-is without prior validation.
- Forward the field after having checked its correctness. Correctness can be checked semantically (e.g., re-compute the request's content length and compare with the value provided by the client) or syntactically (e.g., is the provided MD5-checksum a valid MD5-value).
- Replace the field value provided by the client by a pre-defined default value. This makes sense for values that are known in advance. For example, in the Lab it is known what user agents are utilized.
- Remove the header field. This is reasonable for fields that are not required by the external resource to fulfil a request. The "From" or "Referer" headers for instance are, if at all, merely used for logging or statistical analysis.

The header fields defined for a request as specified by [RFC2616], along with the action taken by the proxy for each field, are listed in the following table.

Header Name	*Description*	*Allowed Values*	*Proxy Action*
Accept	The Accept request-header field can be used to specify certain media types which are acceptable for the response.	(<media-range> [<accept-params>])*	Remove. Server will send any media type
Accept-Charset	The Accept-Charset request-header field can be used to indicate what character sets are acceptable for the response.	((<charset> | "*") [";q=" <qvalue>])+	Replace by default
Accept-Encoding	The Accept-Encoding request-header field is similar to Accept, but restricts the content-codings (e.g. gzip, idenity, ...) that are acceptable in the response.	(<codings> [";q=" <qvalue>])+	Replace by default
Accept-Language	The Accept-Language request-header field is similar to Accept, but restricts the set of natural languages that are preferred as a response to the request.	(<language-range> [";q=" <qvalue>])+	Remove. Server will send any language
Authorization	A user agent that wishes to authenticate itself with a server does so by including an Authorization request-header field with the request.	RFC 2617	Remove
Expect	The Expect request-header field is used to indicate that particular server behaviors are required by the client.	("100-continue" | <future extension>)	allow only "100-continue"

Header Name	*Description*	*Allowed Values*	*Proxy Action*
From	The From request-header field, if given, SHOULD contain an Internet e-mail address for the human user who controls the requesting user agent.	e-mail address (RFC 822)	Remove
Host	The Host request-header field specifies the Internet host and (optionally) port number of the resource being requested. MUST be present in HTTP /1.1.	<FQDN> [":" < port>]	Syntax check
If-Match	Tells the server to send a file iff it matches the specified entity tag. The entity tag is a distinct value a server assigns to a file. The purpose of this feature is to allow efficient updates of cached information. It is also used, on updating requests, to prevent inadvertent modification of the wrong version of a resource.	("*" | <entity-tag>+)	Remove
If-Modified-Since	Used with a method to make a request conditional: if the requested variant has not been modified since the time specified in this field, an entity will not be returned from the server; instead, a 304 (not modified) response will be returned without any message-body.	<HTTP -date>	Remove. Server will send current version
If-None-Match	Similar to If-Match.	("*" | <entity-tag>+)	Remove
If-Range	Tells the server to send only the range part of the document, if the document is unchanged, or the whole document otherwise. Used in conjunction with the Range header.	(<entity-tag> | <HTTP -date>)	Remove
If-Unmodified-Since	Similar to If-Modified-Since.	<HTTP -date>	Remove
Max-Forwards	Only TRACE and OPTIONS methods.	<digit(s)>	Not applicable
Proxy-Authorization	Similar to Authorization, but for proxy host. It applies only for the immediate connection.	RFC 2617	Remove
Range	Retrieves only the part of an entity specified by the range.	<byte-unit> SP <first-byte-pos> "-" <last byte pos> "/" (<total length> | "*")	Remove. Implementation of range is optional
Referer	The Referer request-header field allows the client to specify, for the server's benefit, the address (URI) of the resource from which the Request-URI was obtained (the "referrer", although the header field is misspelled.)	(<absoluteURI> | <relativeURI>)	Remove. Referer is optional
TE	The TE request-header field indicates what extension transfer-codings it is willing to accept in the response and whether or not it is willing to accept trailer fields in a chunked transfer-coding. This is a hop-by-hop header and applies only to the immediate connection.	("trailers" | (<transfer-extension> [<accept-params>]))*	Not applicable. (hop-by-hop)
User-Agent	The User-Agent request-header field contains information about the user agent originating the request. User agents SHOULD include this field with requests.	<user-agent-string>	Replace by default
Cookie	Not an official HTTP header. It is used very often to send safed cookie information back to the server.	<name> "=" <value> (";" <name> "=" <value>)*	Use cookie ticket. (see notes below)

Header Name	*Description*	*Allowed Values*	*Proxy Action*
Allow	The Allow entity header field MAY be provided with a PUT request to recommend the methods to be supported by the new or modified resource. A proxy MUST NOT modify the Allow header field.	<HTTP -method>	Not applicable (PUT only)
Content-Length	Length of request body in bytes.	<digit(s)>	Syntax check
Content-md5	The Content-MD5 entity header field MAY be generated by an origin server or client to function as an integrity check of the entity-body. Only origin servers or clients MAY generate the Content-MD5 header field; proxies and gateways MUST NOT generate it, as this would defeat its value as an end-to-end integrity check.	<base64 of 128 bit MD5 digest as per RFC 1864>	Syntax check
Connection	The Connection general-header field allows the sender to specify options that are desired for that particular connection and MUST NOT be communicated by proxies over further connections.	("close" | "keep-alive")	Not applicable (not forwarded)
Date	The Date general-header field represents the date and time at which the message was originated.	RFC 1123	Remove (optional for requests)
Pragma	The Pragma general-header field is used to include implementation- specific directives that might apply to any recipient along the request/response chain.	("no-cache" | <extension-pragma>)	only "no-cache" Pragma allowed (violates RFC2616)
Transfer-Encoding	Pragma directives MUST be passed through by a proxy or gateway application.	("chunked" | <transfer-extension>)	only "chunked"-Encoding allowed (violates RFC 2616)
Upgrade	The Upgrade general-header allows the client to specify what additional communication protocols it supports and would like to use.	<product-name>	Remove. The proxy cannot handle other protocols than HTTP
Via	The Via general-header field MUST be used by gateways and proxies to indicate the intermediate protocols and recipients between the user agent and the server on requests.	(<received-protocol> <received-by> [<comment>])+	Remove. (violates RFC2616)
Warning	The Warning general-header field is used to carry additional information about the status or transformation of a message which might not be reflected in the message.	<warning in natural language>	Remove

Notes regarding the Cookie header field. Cookies are not part of the official HTTP standard (i.e. [RFC2616]). The Cookie mechanism is specified in [CSPEC]. Nonetheless cookies are used by many web servers and understood by almost every user agent. Cookies are set by the web server using the "Set-Cookie: ..." header. In later requests, a user agent sends the cookie along with its request using the "Cookie: ..." header. A cookie consists of name/value pairs very similar to those used in the x-www-urlencoded scheme. The "Set-Cookie" header can also be used to append additional name/value pairs to an existing cookie. A sample HTTP dialogue (only showing the cookie headers) between a client C and a server S is depicted below.

```
S > C: Set-Cookie:  CUSTOMER=WILE_E_COYOTE; path=/;  expires=Wednesday,
   09-Nov-99 23:12:40 GMT
C > S: Cookie:  CUSTOMER=WILE_E_COYOTE
S > C: Set-Cookie:  PART_NUMBER=ROCKET_LAUNCHER_0001; path=/
```

```
C > S: Cookie:  CUSTOMER=WILE_E_COYOTE; PART_NUMBER=ROCKET_LAUNCHER_0001
```

The same considerations as for client provided data apply to cookies. Therefore, cookies are generally removed by the proxy. A generic mechanism to verify a cookie's integrity (i.e. making sure that each name/value pair of the cookie has indeed been set by the respective server at some prior time) would require the proxy to maintain an internal data structure of all cookie headers received from external resources.

However, if the cookie does not change over time, things become much easier: the same ticket mechanism as for HTTP URLs can be used for cookies. Whenever the proxy detects a "Set-Cookie" header in a response, the HMAC ticket is appended to the cookie. The proxy forwards cookies in client requests iff they have a valid ticket attached. The assumption that the cookie does not change over time is true in many cases, as cookies often contain only a fixed session identifier. In that case the actual data is stored on the external server and accessed via the session identifier. If a cookie does change (i.e. the server adds information to the existing cookie) it will no longer be forwarded by the proxy, as the ticket appended to it does not match the whole cookie.

2.3 Rewriting URLs

2.3.1 URL Format

The generic format of an absolute HTTP URL is defined in [RFC1738] and its updates as follows:

```
"http://" <host>[":"<port>] "/" [<path>]["?"<searchstring>]["#"<reference>]
```

- Host: The fully qualified domain name of a network host, or its IP address as a set of four decimal digit groups separated by ".".
- Port: the port number to connect to. If not specified, the default port 80 is used.
- Path: details of how the resource can be accessed. If neither <path> nor <searchstring> are present, the preceding "/" may also be omitted.
- Searchstring: query string passed to the web server. An arbitrary string except that reserved and unsafe characters such as "?" or "/" must be encoded.
- Reference: The reference (also known as anchor) is not technically part of the HTTP URL scheme. Its meaning depends on the resource interpreting it. Usually the reference part is interpreted by the user agent as a link to a location within the HTML document. References are not sent as part of a request by the user agent.

The URL scheme chosen to fit the proxy's needs (i.e. to contain an additional MAC as the URLs ticket for the proxy) is as follows. The scheme complies with RFC standards, i.e. the rewritten URL is still valid. This is important because some user-agents will refuse to request an invalid link.

```
"http://" <host>[":"<port>] "/" [<path>]["?"<searchstring>] "{"<ticket>"}"
   ["#"<reference>]
```

For example, `http://www.example.com/` becomes
`http://www.example.com/{bf0ce8a4d7614eef377c2ea7e117273dc73185cd}`,
and `http://www.example.com/foo?bar=42` becomes
`http://www.example.com/foo?bar=42{809f78530eccf85603e266404a65583eb2743758}`.

Input for the MAC computation is the entire URL including the scheme part (i.e. "http:") and any optional parts like queries, prepended by the proxy's secret key. As mentioned before, relative URLs are first rewritten to their absolute form. Since the reference part is considered as a link within the document and will not be sent to the proxy, it needs not be tagged with a ticket.

The characters "{" and "}" must be encoded by their respective hexadecimal encodings (i.e. as "%7B" and "%7D", respectively) since they are considered unsafe in URLs. Any two different non-empty strings could be used, as long as reserved and unsafe characters are properly encoded and the strings will never be part of the HMAC. Surrounding the ticket by such markers is just a convenient method to make identification and removal of the ticket easier. Since the HMAC has fixed length and is always appended at the end of the URL, the markers could be abandoned. By using markers it must not be accounted for the fact that different hash functions produce HMACs of different length. In other words, by using markers identification and removal of the ticket become independent of the hash function in use.

2.3.2 HTML Elements

To be able to tag links within a document with the according ticket, the proxy must first identify the links. More precisely, the following HTML elements must be identified and rewritten:

- Anchor elements (A).
  The anchor element only appears in the BODY of a HTML document. When the A element's HREF attribute is set, the element defines a source anchor for a link that may be activated by the user to retrieve a web resource.
  Example: `<A href="http://example.com/foobar.html">`
- Images and scripts (IMG, SCRIPT).
  HTML documents can contain images and scripts. If these are not within the document itself, an external location can be specified with the SRC attribute.
  Example: `<SCRIPT type="text/javascript" src="http://example.com/foobar.js">`
- Link elements (LINK).
  The LINK element only appears in the HEAD of a HTML document. It defines a relationship between the current document and another resource. A common usage is the inclusion of external style sheets or to link to a previous, next, or index page. The linked resource is given by the HREF attribute.
  Example: `<LINK rel="stylesheet" type="text/css" href="foobar.css">`
- Base elements (BASE).
  Relative URIs within a HTML document are resolved according to a base URI. The BASE element allows authors to specify a document's base URI explicitly. When present, the BASE element must appear in the HEAD section of an HTML document, before any element that refers to an external source. The base URI is given by the HREF attribute.
  Example: `<BASE href="http://example.com/foo/bar/">`

2.3.3 Parsing Documents - HTML Parsers vs. Regular Expressions

A possibility to identify the respective elements is to build an entire parse tree of the HTML document, as a web browser does. As the graphical representation of the document depends on the parse tree, its generation is necessary for the browser. The proxy, on the other hand, does not need to know much about the structure of the document. Successful identification of links suffices for the proxy's operation; the proxy does not care about the meaning of a link or its position within the document. In other words, the proxy needs syntactic information only, whereas a browser also needs semantic information.

Therefore, building a parse tree of every document fetched from a web resource would introduce a considerable amount of unnecessary overhead, slowing down response times. Worse still, a large amount of all HTML pages are syntactically incorrect: opening tags miss their corresponding closing tags, elements are improperly nested, and so forth. A HTML parser thus must tolerate a wide range of faults, i.e. try to produce a valid parse tree even if its input is flawed.

Quite a few HTML parsers exist, and most of them can deal with incorrect HTML to a certain degree. Some parsers produce a flat list of the elements in a document rather than a hierarchical

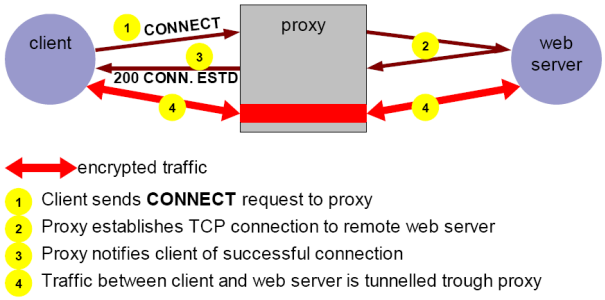

Figure 2.1: HTTPS CONNECT mechanism

parse tree. This eliminates the overhead of generating the parse tree when it is not needed by an application. Examples of such light-weight parsers include the Jericho HTML parser[5], Apache's Xerces-based NekoHTML[6], HTMLparser[7] and many more. (The parsers listed here are open source and free.)

Because the task of identifying the relevant HTML elements within a document is quite simple (only very few HTML elements must be recognized), I decided not to integrate an existing parser into the project. A regular expression package is used instead. Regular expressions provide a convenient way of finding occurences of a given text pattern within an input text. Any further decomposition of the HTML input into elements is not required. Although processing complex regular expressions on large input texts can be quite expensive, the costs for finding the elements described before should be low since the patterns are simple.

2.4 Architecture Deficiencies

2.4.1 Proxying HTTPS

As the HTTP protocol over a TCP/IP connection guarantees neither the authenticity of a web resource nor the confidentiality of the transmitted data, HTTPS was introduced as a more secure means to communicate with web servers. HTTPS is not a new protocol; it merely stands for the conventional HTTP protocol running on top of an additional layer, the *Secure Socket Layer* (SSL). The SSL layer is placed between the transport layer (usually TCP) and the application layer (protocols other than HTTP also make use of SSL). SSL was introduced in 1994 by Netscape and is based on public key cryptography for key exchange. SSL allows mutual authentication based on certificates as well as encryption of data traffic. Keys and certificates are exchanged during the protocol initiation (handshake). The current SSL version v3 and its almost identical successor TLS are considered unbreakable when used with long enough session keys.

As all application-layer data is encrypted, HTTPS connections pose a problem to proxy servers. The client's requests are still sent via the proxy, but the proxy cannot forward the requests since all HTTP headers are encrypted. On this account, an additional HTTP method, CONNECT, was proposed in 1998 [Luo98]. It was standardized in [RFC2817] in 2000. The CONNECT method allows a client to establish a tunnel to a web resource through a proxy. The CONNECT mechanism is defined as follows and illustrated in figure 2.1.

1. The user agents sends a CONNECT request to the proxy: `CONNECT SP <host>":"<port> SP <HTTP-version>`.

2. The proxy establishes a TCP connection to the specified host and port.

[5] `http://jerichohtml.sourceforge.net/`
[6] `http://www.apache.org/~andyc/neko/doc/html/`
[7] `http://htmlparser.sourceforge.net/`

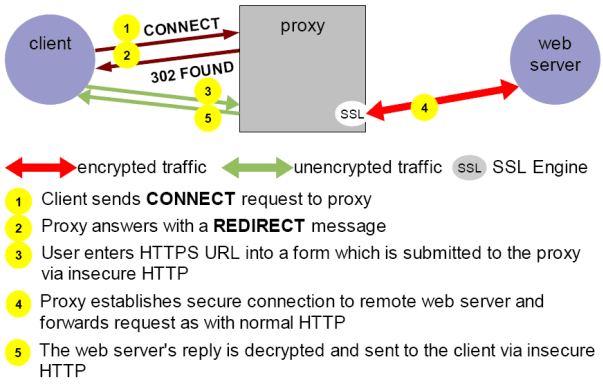

Figure 2.2: Secure connection between proxy and external host only

3. The proxy sends a `200 CONNECTION ESTABLIHED` message to the user agent.

4. The tunnel is established. All data coming from the web resource over this connection is forwarded to the client, and all data coming from the client over this connection is forwarded to the web resource. The application data flow is completely opaque to the proxy, as all data is encrypted.

The CONNECT mechanism as described here is not suited to provide HTTPS support within the requirements for this project, for several reasons:

- The proxy has no control over what data is exchanged with the remote web resource. It cannot even be assumed that the remote resource is a web server. Arbitrary TCP connections can be tunnelled trough the proxy with the CONNECT mechanism. This is clearly a violation of the rule that no client-provided data is forwarded to external resources.

- Incoming HTML documents can no longer be parsed for obvious reasons. Links on those documents cannot be tagged with a ticket, and clients will not be able to retrieve them, although this should be possible according to the specification.

For the proxy to support HTTPS while still meeting the given requirements, it must be able to see transferred data in clear. Sketched below are two different implementation variants that comply with this requirement. Both variants require the proxy to implement the SSL protocol, in contrast to the SSL tunnelling mechanism. Neither variant was implemented for reasons of time.

Secure connection between proxy and external web resource only. [Luo98] discusses an alternative to the CONNECT method: On behalf of the client, the proxy initiates a secure session with the remote resource, and then performs HTTPS transactions on the client's part. The response will be decrypted by the proxy and sent to the client over insecure HTTP . This concept could be implemented by the proxy by the use of HTTP redirections. When receiving a CONNECT from a client, instead of initiating a tunnel, the proxy sends a "HTTP 302 Found" redirect to the client. The client is redirected to a HTML form generated by the proxy, where the user can enter the desired HTTPS URL. The HTTPS URL is then submitted to the proxy, for example as part of the URL: `http://proxy?url=https://...`. A special handler retrieves the HTTPS URL over a secure SSL connection and sends the decrypted data to the client. Figure 2.2 shows a graphical representation of this concept. This approach has several disadvantages:

- The connection between the client and the proxy is normal HTTP, and hence, not secure. Alternatively, a HTTPS connection could be used between client and proxy as well. This would require the implementation of an additional SSL engine on the client side of the proxy.

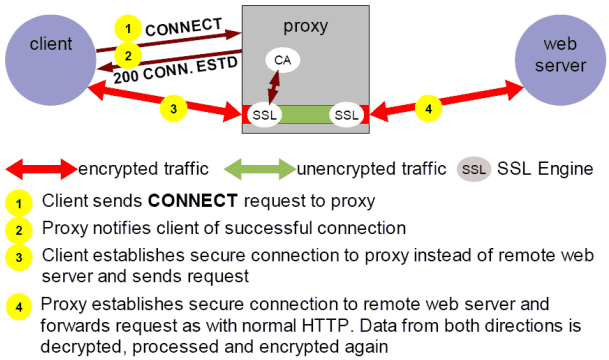

Figure 2.3: Splitting the secure connection at the proxy

- The client will not be able to perform SSL client authentication (authentication based on certificates) to the remote resource, as the proxy will be the authenticated party.
- Since the certificate of the external web resource cannot be checked by the user agent, it should be checked by the proxy. In case the external resource fails to authenticate itself, the user should be warned about this failure. This could be done with a web page generated by the proxy, where the information provided by the resource's certificate is presented and the user has the option to continue or abort his request.
- Although [RFC2817] does not prohibit it, it is questionable whether all user agents will accept a redirect to a CONNECT request.
- The approach is not user friendly, since HTTPS URLs have to be entered twice: First into the browser's location bar as usual, and then a second time into the HTML form provided by the proxy. Many users probably consider this too tedious and therefore abandon the use of secure HTTPS communication.

Splitting the secure connection at the proxy. A more involved solution would allow the proxy to simulate the external end point of the HTTPS tunnel. Instead of opening a TCP connection to the remote web server, the proxy immediately sends a "Connection Established" message to the client upon receiving a CONNECT request. The client will then continue with the SSL handshake, but rather than forwarding the messages involved in the handshake, the proxy itself assumes the role of the external web resource and performs a handshake with the client. The client thinks it is talking to the remote web server but is in fact communicating with the proxy alone. After the handshake is completed, the proxy is able to decrypt all encrypted HTTPS data the client sends. It can then open another secure connection with the actual web resource and forward traffic in both directions. Before being forwarded, traffic is decrypted, processed and encrypted again. The principle is sketched in figure 2.3.

As mentioned above, SSL allows a client to verify the authenticity of a server. To that purpose, the server's SSL certificate is sent to the client during the handshake process. Because the client receives the proxy's certificate in the above scenario, it will detect that the host name of the certificate does not match the name of the host it wants to connect to. As a consequence, most clients will raise a security alert informing the user that the remote resource failed to authenticate itself correctly. The mechanism could, however, be made transparent to the client by augmenting the proxy with its own certification authority (CA). Instead of sending its own certificate, the proxy generates a new certificate on-the-fly during each SSL handshake. The newly generated certificate matches the name of the external resource the client wants to connect to. The proxy signs newly generated certificates with its own CA certificate. The proxy's CA certificate must be contained in the client's list of trusted CA certificates. In that case, the client will accept the

certificate as a valid proof for the web site's authenticity. Advantages and downsides of splitting the secure connection at the proxy are examined below.

- If the proxy is augmented with its own CA, the mechanism is transparent for the client.
- The client will not be able to perform SSL client authentication to the remote resource. SSL client authentication not only requires the client to send its certificate but also to sign a digest of all previous handshake messages to prove knowledge of the associated secret key. While the client certificate can simply be forwarded by the proxy, the same is not true for the signed digest. Since at least one message (the server certificate) is not the same in the two handshakes (client-proxy and proxy-external resource), the digest computed by the client will differ from the one computed by the external resource. The proxy can of course compute the correct digest, but will not be able to sign it with the secret key associated with the client certificate.
- Since the certificate of the external web resource cannot be checked by the user agent, it should be checked by the proxy. In case the external resource fails to authenticate itself, the user should be warned about this failure. This could be done with a web page generated by the proxy, where the information provided by the resource's certificate is presented and the user has the option to continue or abort his request.
- Cryptographic operations are expensive. The generation of a certificate for every HTTPS request will cost a lot of processing power. This might even constitute a security risk since it allows denial of service attacks by simply opening many HTTPS connection at once. At least, generated certificates should be cached for later re-use.
- Although indicated to the user agent, the usual HTTPS end-to-end semantics is no longer provided. Technically, the approach is a man-in-the-middle attack against the SSL protocol, albeit by a trusted host. Though the proxy can in principle be trusted, no secure end-to-end connection exists. Security relevant programming flaws of the proxy software, for example, could lead to exposure of sensitive user data. Users should be made aware of the fact that no real HTTPS connections are used and thus not the same level of security is provided.

2.4.2 Proxying FTP

The File Transfer Protocol uses a URL scheme very similar to the HTTP scheme:

```
"ftp://" [<user>[":"<password>]"@"] <host>[":"<port>] "/" [<path>]
   [";type="("a"|"i"|"d")].
```

At first glance, it seems tempting to use the proxy for FTP in addition to the basic HTTP functionality. FTP URLs on a HTML document could be rewritten and tagged with a ticket using the same method as for HTTP URLs.

To implement FTP support, the proxy must be capable of executing the FTP protocol and translate it into HTTP. Let's consider an FTP link that is contained within a HTML document and clicked by a user. The user agent sends a normal HTTP request containing the FTP URL to the proxy. The proxy then executes an FTP transaction with the specified web resource, generates an HTML page containing the resource's reply, and sends the HTML page back to the user agent via HTTP. This is how many proxies (Squid, for example) in fact handle FTP transactions.

Note, however, that this is not real FTP as the traffic between client and proxy is HTTP-based. Most FTP clients (in contrast to web browsers) are therefore unable to communicate with FTP servers via an HTTP proxy. A real FTP proxy which enforces that only known FTP locations be accessed will be much more complex to implement, because unlike HTTP, the FTP protocol is stateful. In this case, the proxy needs to know to what directories each existing FTP connection has currently access to, based on the analysis of prior directory listings. To this end, the proxy needs an internal data structure. Analyzing FTP directory listings and unblocking listed items for

the respective connection is then the equivalent of parsing HTML pages and tagging URLs with a ticket.

Considering these drawbacks, I decided not to implement FTP support. I rather suggest the use of a secure FTP proxy, also enabling real FTP clients with the possibility to connect to external resources. If FTP access to external resources is not a necessity, the FTP protocol can be blocked entirely.

2.4.3 Missing Links

An URL will only be accessible if it was previously identified by the proxy. The proxy may fail to identify URLs for several reasons:

- The HTML syntax is invalid. A common example are missing double quotes around the HREF attribute.
- The URL is not present in the HTML source but rather dynamically generated by the web browser by executing a script.

While invalid HTML can be handled up to a certain degree by using a fault-tolerant parser or regular expressions that account for common mistakes, a remedy for the second type of failure seems almost impossible. The proxy would need to completely parse and render incoming pages the same way a web browser does to also identify dynamically generated links. This solution is impractical and may bear security risks when improperly implemented. I appraise the impact of unidentified script-generated URLs as very low, however, as only a small fraction of URLs are hidden inside scripts.

2.4.4 Site Access

Direct access to web sites not listed in the rule list is no longer possible, and any submission of data input by a user is blocked unless a specific client data handler exists. This might prove inconvenient or even annoying for users, but is not a deficiency of the proxy architecture but a direct consequence of the basic rule, “no client-provided data is forwarded to external web resources”.

Due to the generic design of the rule list, URLs can be unblocked easily at any time, and data handlers can quickly be set up to allow form submission for sites where data submission is desired.

Chapter 3

Implementation

3.1 Existing Software

3.1.1 Requirements

The goal of this semester thesis is not the implementation of a whole new proxy software from scratch. Implementing the proxy from scratch would be too time-consuming and error-prone (especially with regard to security holes). Rather than that, I evaluated several existing proxy products in terms of their extensibility to meet the project goals as described by the specification. During the evaluation I considered the following points (in decreasing order according to the importance I assigned to them):

- Flexibility and modularity.
  The complexity of integrating new functionality into the existing product. For instance, a product allowing the installation of plug-in classes during run-time is preferred over one requiring modifications of the source code and re-compilation.

- Code safety.
  The programming language used can avert certain types of flaws in a program. These flaws, if present, could be used by an attacker to circumvent the proxy's security features or lead to unpredictable or unstable operation of the proxy.
  Automatic *garbage collection* releases the programmer from the need to manually deallocate objects when they are no longer used. Therefore memory leakage in the case of programming mistakes (eventually leading to unstable programs) is prevented.
  A *type-safe* language prevents code execution with improper input arguments, hence reducing the probability of calling "wrong" code in the program's control flow and eliminating the peril of buffer overflows. Buffer overflows are often security-relevant, since they can be used by an attacker to corrupt the execution stack of an application, causing it to execute arbitrary code.
  The *pointer* data type some languages offer is a very powerful tool, offering direct access to any memory location. If not used with caution, however, it can lead to completely indeterministic program behavior. Most higher-level languages therefore do not offer the pointer data type.
  Finally, *access control* facilities enforce security considerations by restricting access to sensitive parts of the program.

- Documentation.
  This includes comments inside the source code, available API documentation, and informal descriptions of the program logic and structure. A poorly documented program is very hard to extend, because any information about how it operates must be gathered by reviewing its source code.

- Stability and performance.
  Due to the low amount of internet traffic in the Lab environment, performance was not the most important issue during evaluation. However, the final product should provide sufficiently stable operation, i.e. it should constantly run without needing restarts and withstand short bursts in the request rate.

- License and source.
  All evaluated products are freely available. Ideally, the extended product should also be published under a license that allows time-unlimited operation and free access to its source code.

- Ease of use.
  The amount of trouble for the Lab administrator to install and configure the product. Not critical since the set-up is a one-time job.

- Portability.
  The Lab's infrastructure server runs a Red Hat Linux. Portability to other architectures is not required, but might prove useful during development and testing.

3.1.2 Evaluated Products

Early during the evaluation process, after having looked at the source of the two very popular proxies *Squid* and *Apache mod_proxy,* I realized that understanding and altering the C-based code would be a very tedious and time-consuming task. Moreover, the low-level C language does not offer any of the advantages described in the "Code Safety" section to the programmer.

Java, on the other hand, is type-safe and relieves the programmer of error-prone tasks like pointer arithmetics and garbage collection. Due to Java's object-oriented programming paradigm, code is easier to write and review. And since many libraries providing string functions, cryptographic algorithms and much more are by default included in the JSDK development environment, code development can be done much more rapidly than with C.

The aforementioned considerations convinced me to seek for a Java-based solution. Finally, the following products were evaluated. All assessed items were rated from poor (−−) to excellent (++).

- **Squid.**
  The web proxy for Linux and Unix systems. Squid is open-source and licensed by the UC San Diego under GNU GPL. Squid offers high performance and uses DNS- and web-data caches. It supports HTTP-, HTTPS, FTP-, and gopher-URLs. Due to its great popularity and mature development status, Squid is considered very reliable and secure.

Flexibility and Modularity	−−	Modification and recompilation of C source files necessary
Code Safety	−−	Potentially error-prone C source
Documentation	+	A somewhat incomplete programmer's guide is available
Stability and Performance	++	Very fast and stable
License and Source	++	GNU GPL
Ease of Use	+	Text-based config files
Portability	−−	Unix/Linux only

- **Apache with mod_proxy.**
  The mod_proxy module adds proxy functionality to the widely popular Apache web server. The module is very configurable; it can be set up as a forward or reverse proxy and allows for sophisticated access control. Like the Apache web server, mod_proxy is distributed under

the open-source Apache License.

Flexibility and Modularity	−−	Modification and recompilation of C source files necessary
Code Safety	−−	Potentially error-prone C source
Documentation	+ −	Configuration and source well documented, no programmer's guide
Stability and Performance	++	Very fast and stable
License and Source	++	Apache License
Ease of Use	+	Text-based config files
Portability	−−	Unix/Linux only

- **IBM WBI ("webbie").**
  The Java-based WBI architecture is a programmable proxy and web server. It is based on the idea of a web intermediary, a program that can be positioned along the information stream and customizes data as it flows along the stream.

Flexibility and Modularity	++	Customization of data through plug-in Java classes
Code Safety	++	Java offers garbage collection, type-safety, etc.
Documentation	++	Javadoc API and many documents and examples online
Stability and Performance	−	Somewhat slower than C-based programs; can be crashed by issuing many queries at a time
License and Source	−−	Currently licensed under IBM's alphaWorks license. IBM retains the right to terminate the license at any time, thus a special license agreement with IBM would be necessary. The source is not available. IBM is considering to publish the source under GNU GPL.
Ease of Use	++	GUI-based administration
Portability	++	Java VM based

- **PAW (Pro-Active Web Filter).**
  PAW is an open-source HTTP filtering proxy licensed under GNU GPL. It is based on SUN's Brazil framework, a completely Java-based HTTP application framework with a small footprint. PAW only supports the HTTP protocol.

Flexibility and Modularity	++	Customization of data through plug-in Java classes
Code Safety	++	Java offers garbage collection, type-safety, etc.
Documentation	++	Javadoc API and some examples online
Stability and Performance	−	Somewhat slower than C-based programs. Bugs might take a longer time to be discovered, since PAW is not a widely used product
License and Source	++	Modified Apache License
Ease of Use	++	GUI-based and command line administration
Portability	++	Java VM based

- **Muffin WWW Filtering System.**
  The Java based Muffin is mainly designed as a web filter removing advertising banners and other undesired content. It supports HTTP and HTTPS.

Flexibility and Modularity	++	Customization of data through plug-in Java classes
Code Safety	++	Java offers garbage collection, type-safety, etc.
Documentation	+ −	A rather incomplete javadoc API is available
Stability and Performance	−	Somewhat slower than C-based programs.
License and Source	++	GNU GPL
Ease of Use	++	GUI-based and command line administration
Portability	++	Java VM based

Preliminary very simple performance analysis (fetching PHP-generated pages of different sizes from a server over and over again) showed that the Java-based proxies all were about 10 percent slower than their C-based counterparts. The analysis used here was probably too simple since no realistic file size distribution was used. Also, only one file at a time was fetched. A more realistic analysis should simulate concurrent requests as they occur in practice. The result, however, indicated that only a moderate performance loss is to be expected with a proxy running on a Java Virtual Machine. Because performance is not critical, I narrowed the evaluation to the three Java based proxies.

Due to the rather incomplete documentation of Muffin and the restrictive license of WBI, I decided to use PAW as the foundation of the proxy.

3.2 Details of the underlying proxy software

3.2.1 Brazil Framework

The foundation of the PAW proxy is Sun's Brazil web application framework [Brazil]. Brazil began as an extremely small footprint HTTP stack, originally designed to provide a URL based interface for smart cards, so that data on the smart cards can be accessed by an ordinary web browser. The project evolved into a toolkit of reusable parts for a wide range of applications, the common characteristic of which is their URL based interface. Brazil is completely written in Java.

The two most important entities of the Brazil framework are the *Server* object and the *Request* object. Each HTTP transaction along with its current state is represented by a Request object. The Server object handles requests by calling *Handler* objects registered with the server. A handler can perform arbitrary tasks like aggregating content from different web resources, generating content, reading content from a file, rewriting content, or filtering content. Handlers thus provide a very generic mechanism to add desired functionality into the Brazil architecture. A handler can be any Java class implementing the `Handler` interface, defined by the methods init and respond.

- `public boolean init(Server server, String prefix)`.
  This method is called when the server initializes the handler object. The prefix string allows the handler to access handler-specific configuration parameters.

- `public boolean respond(Request request)`.
  Allows a handler to respond to a request. If the request was handled (i.e. a reply sent to the client), true is returned. Otherwise false is returned, and the server passes the request to the next handler object, if present.

Filters are a special case of handlers. When content is retrieved from an external resource, arbitrary modifications of the content can be performed by a filter. The `Filter` interface extends the `Handler` interface by two additional methods:

- `public boolean shouldFilter(Request request, MimeHeaders headers)`.
  Based on the request and the reply headers, the filter decides whether it wants to examine and possibly rewrite the content of the reply. In many cases one wants to examine or rewrite HTML pages but not other content like binary data. In that case, the `shouldFilter` method should return true iff the Content Type header value is "text/html".

- `public byte[] filter(Request request, MimeHeaders headers, byte[] content)`.
  If the `shouldFilter` method returned true, the content and all headers of the reply can be examined and modified by the filter object with this method.

Many useful handlers are already included in the Brazil distribution, for example a CGI script handler, access control handlers, session handlers, a proxy handler, and many more.

3.2.2 PAW

PAW [PAW] basically is the assembly of the required Brazil objects to provide proxy functionality. Its main components are the Brazil Server and the Brazil proxy handler which is registered as the default handler with the Server. A good deal of additional handlers and filters is also provided. For instance, filters that remove ad banners or animated GIFs, log traffic, or perform a virus scan are available. Additionally, a GUI allows easy administration. It allows to register new filters or handlers, switch handlers or filters on or off, and edit the configuration files of a handler or filter. The proxy can alternatively be administrated without the GUI, by editing XML configuration files.

3.3 Plugin Design

Two basic tasks have to be carried out by the proxy in addition to the functionality already provided by PAW. First, a decision must be made for every request whether to allow or deny it. Second, each HTML document retrieved from an external web resource must be parsed and rewritten. Therefore, a handler *(ParanoidHandler)* and a filter *(ParanoidFilter)* are implemented. As the names suggest, ParanoidHandler performs validation of outgoing requests, whereas ParanoidFilter rewrites retrieved documents.

As discussed earlier, ParanoidFilter uses a regular expression package to identify URLs. The GNU `regexp` package[1] is used to that purpose. To compute the HMAC ticket of a URL, Java's `javax.crypto.Mac` implementation of [RFC2104] is used. It allows the application of various hash functions. The desired hash function can be defined in the server's config file.

ParanoidHandler manages the rule list. On proxy startup, a rule object is generated for each rule defined in the XML-based rule configuration file, `rules.xml`. Rule objects are then inserted into a map structure, with the URL they apply to as the key. The map is implemented as a hash table, allowing quick lookup even in very large rule sets. An own map is maintained for each of the two possible rule targets allow and deny.

Each rule object contains the rule's name and description and the respective data handler. The rule's parameters are the input for the data handler checking client-provided data. The allowed range of the parameter value is specified with a regular expression. The syntax of the rule configuration file and a sample rule allowing access to the Google search page are shown in figures 3.1 and 3.2.

As shown in the example, a single rule can be applied to several URLs. Likewise, several rules can be specified for a single URL. If several rules are specified for one URL, they will be evaluated in the order in which they are specified in the rule configuration file, and the first matching rule will be applied. However, rules with target deny are always evaluated before rules with target allow.

[1] http://www.cacas.org/java/gnu/regexp/

Figure 3.1: DTD syntax of the rule file

```
<!ELEMENT rules(rule*)>
<!ELEMENT rule (name, description, url+, param*)>
<!ATTLIST rule
          target (allow|deny) #REQUIRED
>
<!ELEMENT name (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT url (#PCDATA)>
<!ELEMENT method (GET|POST)>
<!ELEMENT param (pname,pvalue)>
<!ATTLIST param
          method (get|post) #REQUIRED
          required (required|no)
>
<!ELEMENT pname (#PCDATA)>
<!ELEMENT pvalue (#PCDATA)>
```

Figure 3.2: Sample rule that allows access to Google

```
<rule target="allow">
    <name>google searches</name>
    <description>this rule allows to send search queries to google.</description>

    <!-- apply this rule to all the following urls: -->
    <url>http://www.google.ch/search</url>
    <url>http://www.google.com/search</url>
    <url>http://google.ch/search</url>
    <url>http://google.com/search</url>

    <!--
        all parameters that are allowed. required parameters can have attribute
        required="required" set.
        allowed parameter values are interpreted as regular expressions.
        it must be specified how a parameter is passed (get|post)
    -->

    <param method="get">
        <!-- search language -->
        <pname>hl</pname>
        <pvalue>^(en|de)$</pvalue>
    </param>
    <param method="get" required="required">
        <!-- the query -->
        <pname>q</pname>
        <pvalue>[a-zA-Z0-9\+%]+</pvalue>
    </param>
    <param method="get">
        <!-- search method -->
        <pname>btnG</pname>
        <pvalue>^(Google-Suche|Suche|Google-Search|Search)$</pvalue>
    </param>
    <param method="get">
        <!-- 'pages only in german' and 'only swiss pages' options -->
        <pname>meta</pname>
        <pvalue>^(|lr=lang_de|cr=countryCH)$</pvalue>
    </param>
</rule>
```

Chapter 4

Results

4.1 Security Considerations

The proxy is mainly designed to protect external resources from *unintended* malicious activities from within the lab. A user could, for instance, unintendedly crash a remote web server by entering a wrong URL or IP address into a tool used for demonstration purposes. Because URLs as well as header fields are validated by the proxy before being forwarded, attacks in this vein will successfully be blocked by the proxy. The proxy therefore complies with the given requirement specification. A short discussion of risks that are not addressed by the specification is conducted in sections 4.1.1 and 4.1.2.

4.1.1 Preventing Unintended Malicious Activity

The proxy does not prevent clients from requesting "dangerous URLs" that are present on existing external web pages. This might happen when a vulnerability of some host is detected and made public by someone outside the lab. For example, the link `http://example.com/somescript.php?user=admin&loggedin=1` will not be blocked by the proxy if it was previously identified on an incoming HTML document. So at least indirect damage can be caused by a user requesting such a link if the respective resource is susceptible to privilege escalation. Note, however, that the risk for the external resource is neither caused nor increased by the Lab, and certainly cannot be eliminated by the proxy.

Also, the proxy will of course not protect clients within the Lab from malicious content on web pages retrieved from external resources. The goal of the proxy is to protect the "outside world" from malicious activity within the lab (and not to protect "us" from "evil guys out there", as most filtering proxies do). To prevent user agents from malicious JavaScript scripts, for example, additional content filters would be necessary.

4.1.2 Preventing Intended Malicious Activity

Although the Security Lab's policy forbids attacks on external resources or even on the proxy itself, some possible intended attacks shall be examined here. To circumvent the proxy's security measures, several types of attack are conceivable.

Brute force attack. If an attacker succeeds in cracking the proxy's secret key, he will be able to add valid tickets to arbitrary URLs as a consequence. Since every tagged URL on a rewritten page constitutes an input/output pair of the keyed hash function, many such pairs are known to the attacker. In principle an attacker can generate any number of input/output pairs, although not for arbitrary inputs.

All the same this type of attack does not seem not very promising, as good keyed hash functions are considered secure in the sense that they do not allow efficient computation of the secret key,

even if many input/output pairs are known. To further reduce the success probability of such an attack, the secret key could periodically be altered. This measure seems unnecessary for the above reason and would introduce the problem that valid tickets suddenly become invalid.

Stealing the secret key. The configuration file of the proxy must be kept safe. Clients must not have access to this file, as the secret key is contained therein.

A particular attempt to steal the secret key is to implement a malicious handler or filter plugin for the proxy. As the secret key is placed in the global configuration file of the proxy, it may in principle be accessed by all handlers and filters. A malicious plugin might even try to disable the ParanoidProxy handler and filter. Only handlers and filters from trusted sources should therefore be installed.

Reflecting URLs. A less general principle than getting hold of the secret key is the illegitimate validation of a single URL. To this end, an attacker must be able to reflect the URL from a site to which he is allowed to submit data. If the site reflects data as it is entered, this type of attack is possible. For example, the attacker might enter "`<a href="some malicous URL">`" as the user name in a login form. The web site then replies, "`user <a href="some malicious URL"> is not known, try again`". The proxy identifies the link and tags it with a ticket, and the attack is successful. To prevent this, the allowed values for parameters should be well reasoned. Above example is no longer possible when the allowed values prohibit special characters such as "<".

Unless an attacker successfully disables the ParanoidProxy handler, he will only be able to carry out URL-based attacks. Header fields are validated and cleaned even if the requested URL has a valid ticket.

4.2 Performance Analysis

While performance was not the key issue during implementation, I still wanted to evaluate whether the proxy noticeably would slow down a realistic amount of web traffic. In particular I was worried about whether parsing and rewriting all incoming sites would significantly slow down response times of the proxy with the amounts of traffic to be expected in the Security Lab. Early implementations of the rewriting filter dramatically increased response times for pages containing a large number of URLs during first tests. By improving the code used for finding the URLs and rewriting them I was able to speed up the processing significantly. Especially the use of string buffers instead of operations on pure strings proved to be magnitudes faster.

I consider degradations of the available throughput caused by the proxy to be less of a problem than increased response times, since throughput values of the lab's internet access are typically fairly low. Measurements conducted during a few courses showed that typically some thousand external URLs per hour are fetched from withing the lab.

To evaluate the proxy's performance and estimate the slow-down caused by parsing incoming sites, I used the tool Web Polygraph [Poly]. Web Polygraph is a freely available, open-source (C++) benchmarking tool for proxies and other web intermediaries. It is developed and maintained by the University of California. To simulate the workload, both a web server and clients fetching documents are simulated. Web Polygraph is very scalable and can run on a single machine as well as on several clients and servers in parallel. A main feature of Web Polygraph is its ability of content simulation. While other benchmarking tools just send random bytes, sends Web Polygraph documents with real HTML content to the proxy under test. This is important when testing the performance of filtering or rewriting proxies. The content is extracted from a database file containing a collection of HTML documents.

To describe the workloads to be simulated, Web Polygraph offers the Polygraph Language (PGL). PGL allows flexible and detailed description of a workload. Adjustable parameters include the number of clients, the average number of documents fetched by a client per time unit, average document sizes, the server's behavior, and much more.

4.2.1 Simulating Typical Activity

In the first (and probably most relevant) test run I tried to model the activity of the lab's internet access as realistically as possible. The goal was to evaluate whether the proxy's influence on throughput and response times is significant under normal circumstances. I used the following parameters:

- The average request rate is 0.6 requests per second or a little more than 2000 per hour. The measurements conducted during the courses peaked at 1500 requests per hour.
- The proxy is tested during 20 minutes with a constant load. Web Polygraph's client process issues requests according to an exponential probability distribution; i.e. the time intervals between request are Poisson distributed.
- Since parsing and rewriting are the most time-intensive tasks of the proxy, only the rewriting filter of the proxy (ParanoidFilter) is activated. The handler granting or denying access to a given URL is disabled, so that all URLs generated by the Web Polygraph client process can be retrieved. Other filters as well as the GUI were not running during the test.
- The content database for the simulation of realistic HTML content is the sample database provided on the Web Polygraph web site.
- The following distribution of content is assumed:
  - HTML documents: 30%, average size: 20 kB with exponential distribution.
  - Images: 23%, average size: 15 kB with exponential distribution.
  - Downloads and other content: 47%, average size: 300 kB and 25 kB with standard deviation 300kB and 10kB, respectively. (Web Polygraph defaults.)

  70% of all requested URLs are images, downloads or other non-HTML content. Only 30% of the requested URLs represent HTML content which needs to be parsed and rewritten. This assumption is based on a 2003 survey of the UC Berkeley [LV03].
  Average document sizes of images and HTML documents are based upon estimates in [LG99]. The assumption about the average size of other content is based on Web Polygraph's default values, which seem reasonable.

The PGL server and client simulators both ran on a 1.6 GHz Intel system with SuSe Linux, while the proxy ran on a 1.3 GHz Athlon system running WinXP. All PGL test programs can be found in Appendix A.

Figure 4.1 shows the distribution of the response times. Mean response time was 62 msec, and 90% of all requests were handled within a 210 msec delay.

To assess the results of the test, I ran a second test under identical assumptions, but with all filters deactivated this time. As figure 4.2 shows, the curve of the response time distribution is steeper than with the filters activated. 90% of all requests were handled within 20 msec. However, the mean response time of 23 msec indicates that the mean values are within the same order of magnitude with filters turned on or off, respectively. A 200 msec delay is acceptable when retrieving a web page. CPU load on the machine running the proxy was typically very low, with slight peaks from time to time.

4.2.2 Response Time Increase for Large HTML Documents

The next test case was designed in order to measure the delay introduced by the rewriting filter for large HTML files. The below size and content distribution was used.

100% of received files are HTML documents. The mean document size is 80 kB (a fourfold increase compared with [LG99]'s estimate) with an exponential distribution. All other test parameters remained unchanged.

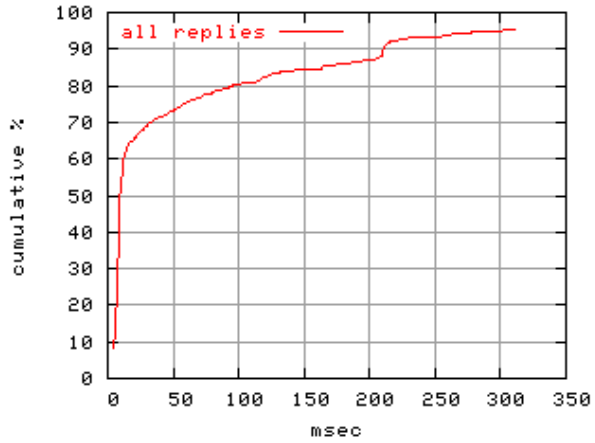

Figure 4.1: Response time distribution. 0.6 req/s, mixed content, filter on.

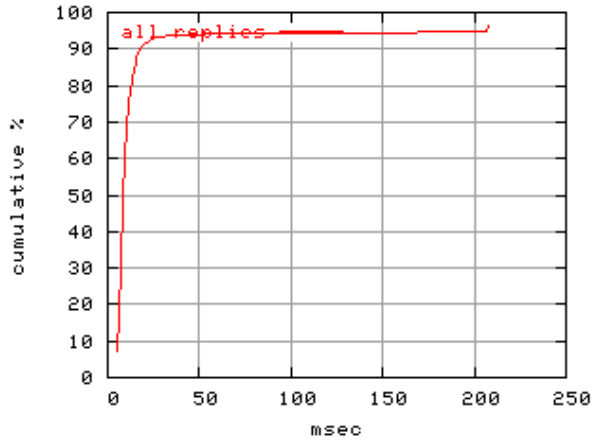

Figure 4.2: Response time distribution. 0.6 req/s, mixed content, filter off.

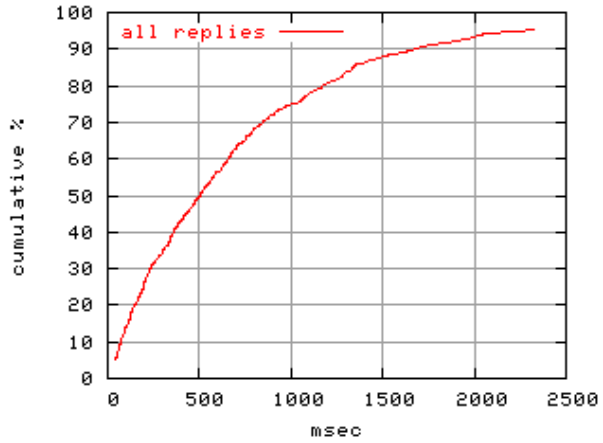

Figure 4.3: Response time distribution. 0.6 req/s, large HTML files, filter on.

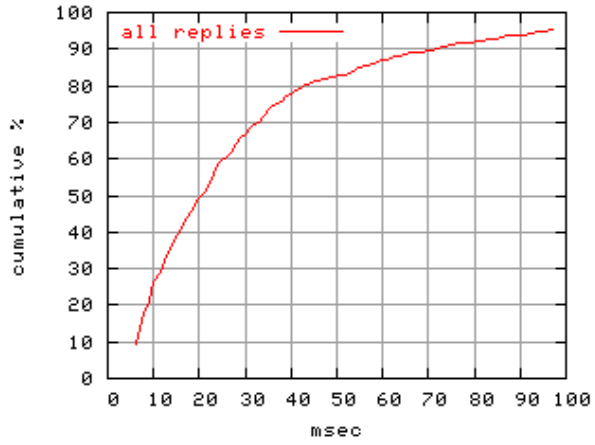

Figure 4.4: Response time distribution. 0.6 req/s, large HTML files, filter off.

Figures 4.3 and 4.4 reveal a dramatic increase in response time introduced by the filter (note the different scaling of the axes). Mean response times were 30 msec and 736 msec with the filter switched off and on, respectively. While response time increases only marginally without the filter, the rewriting of large pages causes considerable delay. Given that only a small fraction of all retrieved documents will be very large HTML files, a delay of still less than a second for those files seems tolerable, though. The processing of such large files can generate high CPU load peaks, temporarily slowing down other applications running on the proxy host.

4.2.3 Best Effort Mode

The last test case uses a best effort test mode. In best effort mode, requests are sent sequentially to the proxy with the highest possible rate. As soon as the reply for a request is received by the client simulator, it issues the next request. In this setting, the throughput rate is the interesting test result. The content distribution used here was the same as in the first test case. This test case is not relevant for the circumstances in the Lab; I used it to estimate the proxy's maximal throughput in larger environments.

Figure 4.5 shows the throughput graph with the rewriting filter off. Troughput is stated in terms of transactions per second (left) and bandwidth (right). Average throughput was 75.4 transactions per second or 14.4 Mbit/s. The corresponding result with the filter switched on is shown in figure 4.6. Throughput was reduced to 20.1 transactions per second or 4.2 MBit/s, or by a factor of about 3.5.

Strangely enough, the mean response time improved in comparison with the first test case, both with the filter switched on or off. The mean response times were 12 msec and 48 msec, respectively. Although the proxy operates at high load (CPU load was constant and very high both with the filter switched on or off, about 80% to 90%), it needs *less* time to serve a request. The probable reason for this outcome is the property of the best effort mode to issue all requests sequentially. Thus the proxy must never handle concurrent requests, resulting in lower response time per request.

On the realism of the best effort mode. Since all requests are issued sequentially, the best effort mode is not a realistic model. In a real situation where the proxy operates at a high load, it will most certainly process many concurrent requests at any time, resulting in high response times. The measured response times are therefore meaningless in practice. On the other hand, the overhead introduced by concurrency should be low compared to the cost of processing a request. To confirm that assumption, I ran an additional test which simulated 10 best effort clients at

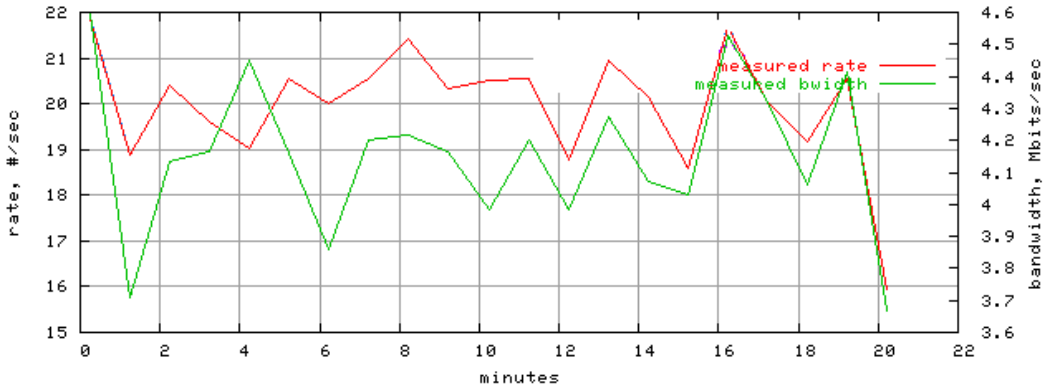

Figure 4.5: Throughput. Best effort mode, filter on.

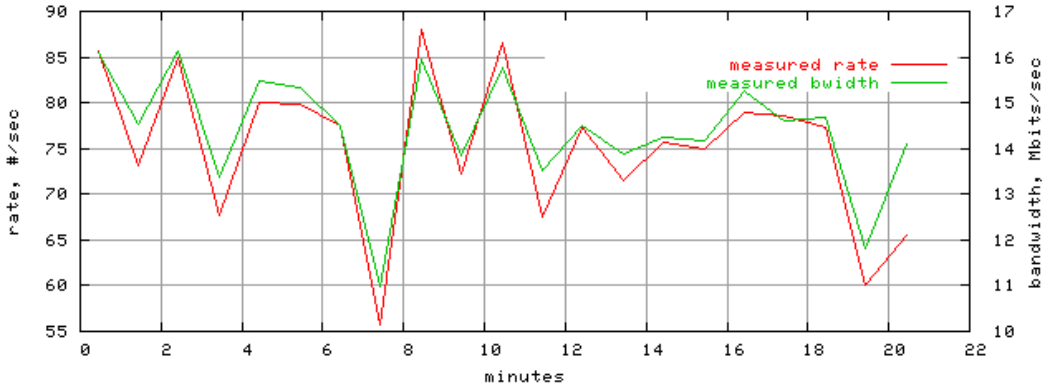

Figure 4.6: Throughput. Best effort mode, filter off.

the same time. While response times got much worse, throughput was not negatively affected by concurrency. The degradation of the throughput caused by the rewriting filter in this test case thus seems a good indication for the value to be expected in a real situation where the proxy operates at high load.

4.2.4 Conclusions

The test results are summarized in table 4.1. For each test run, measured values for the minimal, average, and maximal response time, the 10%-percentile, the 90%-percentile[1], and the throughput rates are listed.

A slow-down originating from the rewriting filter was distinctly measurable in all three test cases. In the first test case that models the Lab's internet traffic, response times were increased by a factor of about 3, but were still in a viable range. For large HTML pages, this factor can go up to 20 or above, as the second test case showed. Finally, throughput is reduced by a factor of 3.5 when the proxy operates at full load.

While these figures are sufficient for the use case defined in the conceptual formulation, they might prove unsatisfactory for the use in larger environments. On the implementation side, the factors probably can be reduced to some degree by code optimizations. A larger decline of these factors could be achieved on the design side by the inclusion of a proxy cache. HTML documents

[1] A value of x msec for the y%-percentile means, "y% of all requests could be served within x msec".

Test case / filter setting	*Response time (msec)*					*Throughput*	
	min	avg	max	10%	90%	xact/s	Mbit/s
Const. rate, mixed/on	3	63	805	5	210	0.6	0.14
Const. rate, mixed/off	3	23	414	5	20	0.6	0.11
Const. rate, HTML/on	6	736	5398	65	1440	0.6	0.49
Const. rate, HTML/off	3	30	279	6	75	0.6	0.39
Best effort, mixed/on	3	49	1078	5	160	20.1	4.17
Best effort, mixed/off	3	13	1631	3	25	75.4	14.39

Table 4.1: Performance tests summary

should first be rewritten and then added to the cache. Using this technique, recently retrieved documents are located in the cache and need not be rewritten when requested again. To increase the cache hit ratio for HTML documents and thus reduce the amount of processing power involved in rewriting HTML pages, an HTML-only-cache can be used. Other file types will then not be cached, thus increasing the proxy's bandwidth usage in turn.

4.3 Future Work

4.3.1 Software Improvements

Future improvements of the software may include:

- Integration of SSL support.
  Implementation of support for HTTPS connections as described in section 2.4.1. Unfortunately, up to JSDK 1.4, Sun's SSL implementation is tied to a particular I/O device (a socket, in fact) making it unsuited for the realization of the second variant (splitting the secure connection), because the proxy's client side socket must be able to receive both encrypted and unencrypted traffic. Starting with version 1.5, Sun offers a more generic SSL implementation where communication with the SSL engine takes place via input- and output-buffers.

- Performance improvement by cache usage.
  A cache as described in section 4.2.4 could improve performance.

- More generic rule list.
  Currently, rules are defined on a per-URL-basis. A more generic rule list could allow wildcards in rule URLs, thus permitting application of the rule to several matching URLs. The possibility to define IP address-based rules instead of URL-based rules could also be added.

- Easier administration.
  Currently, rule administration is done by editing an XML file. If the XML file contains syntax errors, the rule list is not loaded at startup. A helper application could generate the rule file for the administrator based on the administrator's inputs in a GUI.

- Automated test suite.
  A set of automated test cases could simplify the verification of future modifications by checking stability and compliance with the specification.

4.3.2 Reverse Proxy

Because the proxy is tailored to fit the needs of a quite particular environment, its field of application is rather limited. The requirement that external resources must be protected from malicious activity within the internal network is unusual. A far more common requirement is that resources within the internal network must be protected from dangerous external activities. Myke Näf, my

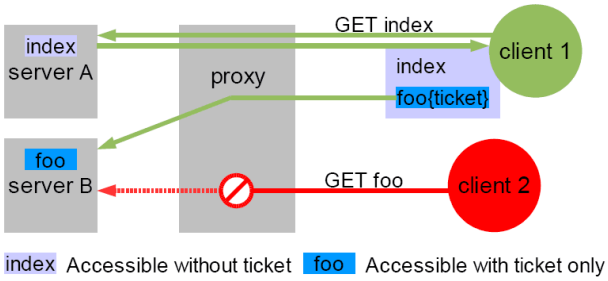

Figure 4.7: Reverse proxy ticket mechanism

supervisor for this semester thesis, pointed out how the proxy might be used to protect internal web servers by acting as a reverse proxy.

A reverse web proxy acts as a gateway to a collection of web servers it is connected to. It provides content located on those servers transparently to clients, i.e. clients will not notice that they are not communicating with a web server directly. In most cases, reverse proxies are used as an "entry point" to servers that are behind a firewall and are not directly accessible. In this setting, all requests that reach a web server originate from the reverse proxy. If the reverse proxy does not forward invalid requests (e.g. requests containing invalid header fields), it serves as an application-layer firewall for the web servers.

Figure 4.7 shows how the ticket mechanism described earlier can be used to provide filtering for a reverse proxy. Requests without a valid ticket attached are generally blocked by the reverse proxy. Only a small set of "entry pages" can be accessed directly without a ticket. On each requested HTML page, URLs pointing to files that are accessible via the reverse proxy are tagged with a ticket. If the proxy is used in reverse mode, the following advantages and downsides emerge.

- Access to files on a web server that should not be accessible (e.g. system files) is blocked, since a URL can only be accessed if it is contained in the rule list or if the request contains a valid ticket.
- Client-provided data is not forwarded to a web server unless a matching allowing rule exists, protecting web servers from SQL injections and similar attacks.
- The set of URLs to which direct access should be possible is known. Also, it is known what parameters may (or must) be passed to which URL. Therefore, the problems discussed in section 2.4.4 can be avoided if the rule list is properly configured. However, this also introduces a considerable amount of configuration overhead, as changes in scripts or HTML documents might require an update of the rule list.
- Proxying HTTPS is no longer a problem, because the reverse proxy rather than the actual web server is the endpoint of the SSL connection.
- Rather than replacing header fields such as "User-Agent" by default values, the reverse proxy can use these fields for logging purposes.
- The reverse proxy must not add tickets to all URLs in a HTML file but only to URLs that are accessible via the reverse proxy. This requires a slight change of implementation.

Chapter 5

Retrospection

The original specification of the semester thesis described a solution where the proxy not only rewrites identified URLs but also maintains an internal data structure which stores identified URLs. URLs are replaced by an encoding which is merely used as an index into the data structure when a client requests such an encoded link. Figure 5.1 shows an example of how the proxy encodes and rewrites URLs in a web page and simultaneously builds the required table.

Early during the design phase I found that maintaining a data structure and rewriting the content of documents is not required to provide the desired functionality. It suffices to store a list of all identified URLs on the proxy. When a client issues a request, the proxy does a lookup in the list and allows the request iff the requested URL is contained in its list. In other words, a URL is both key and data value of a list node, and no special encoding is necessary to provide an extra key.

An early implementation of the proxy therefore integrated such a list of allowed URLs and did not rewrite incoming documents. A balanced binary tree guaranteed lookup of a given URL in time logarithmic to the size of the URL list. As both the handler that decides whether to allow an outgoing request as well as the filter parsing incoming documents need access to the URL list, both handler and filter were implemented in a single Java class. On proxy startup, two instances of that class were generated, one performing request handling, and one parsing incoming documents. The URL list was a static object within this class so that both class instances had access to it.

This approach has a severe disadvantage: The URL list, and with it memory usage, constantly grows. Although a single URL requires not much space and thousands of URLs fit into a few megabytes, the proxy will eventually run out of memory. Some sort of garbage collection removing unused items from the list is therefore required. But deciding which items are unused is impossible for the proxy, because it cannot foresee which URLs are going to be accessed in the future. So a client may want to access an URL that indeed was present on a previously requested page, but the proxy denies the request because it has deleted the relative URL in the meantime. A

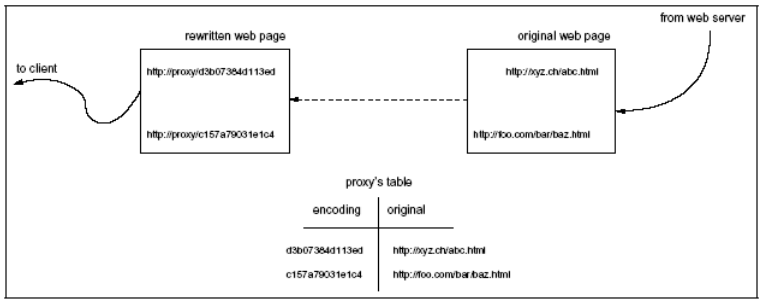

Figure 5.1: Rewriting example.

clever garbage collection algorithm can reduce the impact of this problem. Generational garbage collection, for instance, propagates frequently accessed items into higher generations where fewer collections take place. Old items that are seldomly accessed are deleted first. However, only an oracle could eliminate the problem of wrongly denied requests. In particular bookmarks set by a user will be wrongly denied in many cases, since a bookmarked URL may have been present on a page that was retrieved months ago. The probability that this URL is still present in the proxy's list is thus small. A restart of the proxy even completely destroys the list.

Unfortunately it took me quite a while until I realized that URLs need not be stored at all. It suffices to simply store the information that a given URL is allowed as part of the URL. Of course, the proxy must be able to authenticate this information, as otherwise clients could add it themselves. This is just what MACs do! Once I realized that a MAC can be used to store the fact that the proxy has seen a URL as part of that URL, I abandoned my first approach in favor of the one described in this document. The MAC ticket mechanism elegantly solves all problems described above. Once a URL is tagged with a ticket, it remains valid forever (as long as the secret key of the proxy does not change, which will not happen very often.) Moreover, memory usage of the proxy is far lower.

The downside of the cryptographic approach is that the provided security is only cryptographic. An attacker able to break the hash function (many input/output pairs are known) will be able to compute the secret key and thus generate valid tickets for arbitrary URLs. This downside, however, is only of theoretical nature as no algorithm to efficiently break a "good" hash function is known.

Appendix A

PGL Test Programs

A.1 Mixed Content, 0.6 Requests per Second

```
/*
 * simulating workload for ETHZ infsec lab
 */

/* simulating the servers: */

// content types:

Content cntImage = {
      kind = "image";
      mime = { type = undef(); extensions = [ ".gif", ".jpeg", ".png" ]; };
      size = exp(15KB);
      cachable = 0%;
      checksum = 1%;
};

Content cntHTML = {
      kind = "HTML";
      mime = { type = "text/html"; extensions = [ ".html" : 60%, ".htm" ]; };
      size = exp(20KB);
      cachable = 0%;
      checksum = 1%;
        // use html content from database
      content_db  = "/usr/local/bin/polygraph/workloads/demo.cdb";
};

Content cntDownload = {
      kind = "download";
      mime = { type = undef(); extensions = [ ".exe": 40%, ".zip", ".gz" ]; };
      size = logn(300KB, 300KB);
      cachable = 0%;
      checksum = 0.01%;
};

Content cntOther = {
      kind = "other";
```

```
      size = logn(25KB, 10KB);
      cachable = 0%;
      checksum = 0.1%;
};

Server S = {
kind = "S101";                    //just a label
        //define content distribution and how it is accessed
      contents = [ cntImage: 23%, cntHTML: 30%, cntDownload: 1%, cntOther ];
      direct_access = [ cntImage, cntHTML, cntDownload, cntOther ];
      addresses = ['10.0.0.3:9090' ];  // where to create these server agents
};

/* simulating the clients: */

Robot R = {
      kind = "R101";                  // only a name
      origins = S.addresses;          // where the origin servers are
      addresses = ['10.0.0.3' ];      // where these robot agents will be created
        // realistic request rate
      req_rate = 0.6/sec;
        // assumptions based on requests after using cache
      recurrence = 0;
};

/* phases: */

Phase aPhase = {
      name = "normal_traffic";
      goal.duration = 20min;
      load_factor_beg = 1;
      load_factor_end = 1;
};

schedule( aPhase );

use(S, R);
```

A.2 Mixed Content, Best Effort Mode

The best effort simulation is identical to the simulation in B.1 except for the Robot which is used:

```
Robot R = {
      kind = "R101";                  // only a name
      origins = S.addresses;          // where the origin servers are
      addresses = ['10.0.0.3' ];      // where these robot agents will be created

        // use best effort request rate (default setting)

        // assumptions based on requests after using cache
      recurrence = 0;
};
```

A.3 Large HTML Files, 0.6 Requests per Second

```
/*
 * simulating workload for ETHZ infsec lab
 */

/* simulating the servers: */

// content types:

Content cntHTML = {
      kind = "HTML";
      mime = { type = "text/html"; extensions = [ ".html" : 60%, ".htm" ]; };
        // test behaviour for very large html files only.
      size = exp(80KB);
      cachable = 0%;
      checksum = 1%;
        // use html content from database
      content_db  = "/usr/local/bin/polygraph/workloads/demo.cdb";
};


Server S = {
      kind = "S101";                    //just a label
        //define content distribution and how it is accessed
      contents = [ cntHTML: 100% ];
      direct_access = [ cntHTML ];
      addresses = ['10.0.0.3:9090' ];   //where to create these server agents
};

/* simulating the clients: */

Robot R = {
      kind = "R101";                    // only a name
      origins = S.addresses;            // where the origin servers are
addresses = ['10.0.0.3' ];        // where these robot agents will be created
        // realistic request rate
      req_rate = 0.6/sec;
        // assumptions based on requests after using cache
      recurrence = 0;
};

/* phases: */

Phase aPhase = {
      name = "normal_traffic";
      goal.duration = 20min;
      load_factor_beg = 1;
      load_factor_end = 1;
};

schedule( aPhase );

use(S, R);
```

Bibliography

[RFC1321] R.L. Rivest: *The MD5 Message-Digest Algorithm.* RFC1321, MIT Laboratory for Computer Science, April 1992.

[SHS] National Institute of Standards and Technology (NIST): *FIPS Publication 180-1: Secure Hash Standard (SHS),* April 1995.

[RFC2104] H. Krawczyk, M. Bellare, R. Canetti: *HMAC: Keyed-Hashing for Message Authentication.* RFC2104, February 1997.

[BCK96] M. Bellare, R. Canetti, H. Krawczyk: *Keying Hash Functions for Message Authentication.* Proceedings of Crypto'96, LNCS 1109 (June 1996): pp. 1-15.
Also: `http://www.research.ibm.com/security/keyed-md5.html`.

[RFC2616] R. Fielding et al.: *Hypertext Transfer Protocol – HTTP/1.1.* RFC 2616, The Internet Society, June 1999.

[HTML40] World Wide Web Consortium: *HTML 4.01 Specification.*
`http://www.w3.org/TR/REC-html40/`, December 1999.

[CSPEC] Netscape Corporation: *Persistent Client State HTTP Cookies.*
`http://wp.netscape.com/newsref/std/cookie_spec.html`, 1999.

[Luo98] A. Luotonen: *Tunneling TCP based protocols through Web proxy servers.*
`http://www.web-cache.com/Writings/Internet-Drafts/draft-luotonen-web-proxy-tunneling-01.txt`, Netscape Communications Corporation, August 1998.
Also: A. Luotonen, *Web Proxy Servers.* Prentice Hall 1997, ISBN 0136806120.

[RFC2817] R. Khare, S. Lawrence: *Upgrading to TLS Within HTTP/1.1.* RFC2817, The Internet Society, May 2000.

[RFC1738] T. Berners-Lee et al.: *Uniform Resource Locators (URL).* RFC 1738, December 1994.

[Brazil] Sun Microsystems: *Brazil Web Application Framework.*
`http://experimentalstuff.com/Technologies/Brazil/index.html`.

[PAW] *PAW - Pro-Active Web Filter.*
`http://paw-project.sourceforge.net/index.html`.

[Poly] University of California, A. Rousskov, D. Wessels: *Web Polygraph.*
`http://www.web-polygraph.org/`.

[LV03] P. Lyman, H.R. Varian: *"How Much Information", 2003.*
`http://www.sims.berkeley.edu/research/projects/how-much-info-2003/internet.htm`, UC Berkeley, 2003.

[LG99] S. Lawrence, C.L. Giles: *Accessibility of information on the web.* Nature, 400 (July 8): 107ff, 1999.