Web Application Penetration Testing

Introduction

Module 2

This module contains the required background information you will need before you begin your web application security testing.

Note: covering how web applications work is beyond the scope of this course.

2.0 Module Map

2.1 HTTP/S Protocol Basics

2.2 Encoding

2.3 Same Origin

2.4 Cookies

2.5 Session

2.6 Web Application Proxies

HTTP Protocol Basics

Hypertext Transfer Protocol (HTTP) is the basic protocol used for web browsing and, these days, for most other communication on the web.

It is the client-server protocol used to transfer web pages and web application data. The client is usually a web browser that starts a connection to a server web such as **MS IIS** or **Apache HTTP Server**.

During an HTTP communication, the client and the server exchange messages. The client sends **requests** to the server and gets back **responses**. The format of an HTTP message is:

```
HEADERS\r\n
\r\n

MESSAGE BODY\r\n
```

\r (Carriage Return): moves the cursors to the beginning of the line
\n (Line Feed): moves the cursor down to the next line
\r\n: is the same of hitting *enter* on your keyboard

2.1.1. HTTP Request

Let's examine an HTTP request in detail. The following is the content of the request that we send when we open our web browser and navigate to *www.google.com*.


```
GET / HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:36.0) Gecko/20100101
Firefox/36.0
Accept: text/html,application/xhtml+xml
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

2.1.1. HTTP Request


```
GET / HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:36.0) Gecko/20100101
Firefox/36.0
Accept: text/html,application/xhtml+xml
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

An HTTP request to *www.google.com* is initiated. What you see here are the headers (called *HTTP Request Headers*) for this request. Note that a connection to www.google.com on port 80 is initiated before sending HTTP commands to the webserver.

2.1.1. HTTP Request

```
GET / HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64;
rv:36.0) Gecko/20100101 Firefox/36.0
Accept: text/html,application/xhtml+xml
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

REQUEST METHOD

This is your request type (also known as an HTTP **Verb**).
GET is the default request type when you type a URL into the location bar of your web browser and hit Enter.

Other **Verbs** are POST, PUT, DELETE, OPTIONS, TRACE...

2.1.1. HTTP Request

```
GET / HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64;
rv:36.0) Gecko/20100101 Firefox/36.0
Accept: text/html,application/xhtml+xml
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

PATH

This is the file you are requesting. The home page of a website is always "/". Other pages can be requested, of course, such as: /downloads/index.php. Your request always refers to the root folder to specify the requested file (hence the leading "/").

2.1.1. HTTP Request


```
GET / HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64;
rv:36.0) Gecko/20100101 Firefox/36.0
Accept: text/html,application/xhtml+xml
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

PROTOCOL

This is the **HTTP protocol** version that your browser wants to talk with. This basically informs the web server about which version of HTTP you would like to use in any further communication.

2.1.1. HTTP Request

```
GET / HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64;
rv:36.0) Gecko/20100101 Firefox/36.0
Accept: text/html,application/xhtml+xml
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

HOST HEADER

This is the beginning of HTTP Request Headers. HTTP Headers have the following structure: *Header-name:Header-Value*.

The **Host** header allows a web server to host multiple websites at a single IP address. Our browser is specifying in the Host header which website you are interested in.

2.1.1. HTTP Request

```
GET / HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64;
rv:36.0) Gecko/20100101 Firefox/36.0
Accept: text/html,application/xhtml+xml
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

HOST VALUE

After each request header, you will find its corresponding **value**. In this case you want to reach the **Host** *www.google.com*.

Note: Host value + Path combine to create the full URL you are requesting: the home page of www.google.com/

2.1.1. HTTP Request

```
GET / HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64;
rv:36.0) Gecko/20100101 Firefox/36.0
Accept: text/html,application/xhtml+xml
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

USER-AGENT

The **User-Agent** reveals your browser version, operating system and language to the remote web server.

All web browsers have their own user-agent identification string. This is how most web sites recognize the type of browser in use.

2.1.1. HTTP Request

```
GET / HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64;
rv:36.0) Gecko/20100101 Firefox/36.0
Accept: text/html,application/xhtml+xml
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

ACCEPT

The **Accept** header is used by your browser to specify which document type is expected to be returned as a result of this request.

2.1.1. HTTP Request

ACCEPT-ENCODING

```
GET / HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64;
rv:36.0) Gecko/20100101 Firefox/36.0
Accept: text/html,application/xhtml+xml
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

The **Accept-Encoding** is similar to Accept, but it restricts the content codings that are acceptable in the response. *Content codings* are primarily used to allow a document to be compressed or transformed without losing the identity of its media type and without loss of information.

2.1.1. HTTP Request

```
GET / HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64;
rv:36.0) Gecko/20100101 Firefox/36.0
Accept: text/html,application/xhtml+xml
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

CONNECTION

With HTTP 1.1 you can keep your connection to the remote web server open for an unspecified amount of time using the value "k*eep-alive*". This indicates that all requests to the web server will continue to be sent through this connection without initiating a new connection every time (as in HTTP 1.0).

2.1.2. HTTP Response

Now that we know how the request is composed, let us inspect the web server response.

In response to the HTTP Request, the web server will respond with the requested resource, preceded by a bunch of new *Headers*.

These new headers from the server will be used by your web browser to interpret the content contained in the Response content.

2.1.2. HTTP Response


```
HTTP/1.1 200 OK
Date: Fri, 13 Mar 2015 11:26:05 GMT
Cache-Control: private, max-age=0
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
Server: gws
Content-Length: 258

<PAGE CONTENT>
```

The above is an example of a web server response. Note that we cut out the page content since it is not relevant for our study at this time. Let us inspect some of these headers in greater detail.

2.1.2. HTTP Response

STATUS-LINE

```
HTTP/1.1 200 OK
Date: Fri, 13 Mar 2015 11:26:05 GMT
Cache-Control: private, max-age=0
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
Server: gws
Content-Length: 258

<PAGE CONTENT>
```

The first line of a Response message is the **Status-Line**, consisting of the **protocol version** (HTTP 1.1) followed by a numeric **status code** (200) and its relative **textual meaning** (OK).

2.1.2. HTTP Response

STATUS-LINE

```
HTTP/1.1 200 OK
Date: Fri, 13 Mar 2015 11:26:05 GMT
Cache-Control: private, max-age=0
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
Server: gws
Content-Length: 258

<PAGE CONTENT>
```

The more common status codes are:

- **200 OK**, the resource is found.
- **301 Moved Permanently**, the requested resource has been assigned a new permanent URI.
- **302 Found**, the resource is temporarily under another URI.

2.1.2. HTTP Response

STATUS-LINE

```
HTTP/1.1 200 OK
Date: Fri, 13 Mar 2015 11:26:05 GMT
Cache-Control: private, max-age=0
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
Server: gws
Content-Length: 258

<PAGE CONTENT>
```

- **403 Forbidden**, the client does not have enough privileges and the server refuses to fulfill the request.
- **404 Not Found**, the server cannot find a resource matching the request.
- **500 Internal Server Error**, the server does not support the functionality required to fulfill the request.

```
HTTP/1.1 200 OK
Date: Fri, 13 Mar 2015 11:26:05 GMT
Cache-Control: private, max-age=0
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
Server: gws
Content-Length: 258

<PAGE CONTENT>
```

DATE

Date represents the date and time at which the message was originated.

CACHE HEADER

```
HTTP/1.1 200 OK
Date: Fri, 13 Mar 2015 11:26:05 GMT
Cache-Control: private, max-age=0
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
Server: gws
Content-Length: 258

<PAGE CONTENT>
```

The **Cache headers** allow the Browser and the Server to agree about caching rules. Cached contents save bandwidth because, in short, they prevent your browser from re-requesting contents that have not changed when the same resource is to be used.

CONTENT TYPE

```
HTTP/1.1 200 OK
Date: Fri, 13 Mar 2015 11:26:05 GMT
Cache-Control: private, max-age=0
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
Server: gws
Content-Length: 258

<PAGE CONTENT>
```

Content-Type lets the client know how to interpret the body of the message.

2.1.2. HTTP Response

CONTENT-ENCODING

```
HTTP/1.1 200 OK
Date: Fri, 13 Mar 2015 11:26:05 GMT
Cache-Control: private, max-age=0
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
Server: gws
Content-Length: 258

<PAGE CONTENT>
```

Content-Encoding extends Content-Type.

In this case the message body is compressed with gzip.

2.1.2. HTTP Response

SERVER HEADER

```
HTTP/1.1 200 OK
Date: Fri, 13 Mar 2015 11:26:05 GMT
Cache-Control: private, max-age=0
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
Server: gws
Content-Length: 258

<PAGE CONTENT>
```

The Server header displays the Web Server **banner**. Apache and IIS are common web servers. Google uses a custom webserver banner: gws (that stands for Google Web Server).

CONTENT LENGTH

```
HTTP/1.1 200 OK
Date: Fri, 13 Mar 2015 11:26:05 GMT
Cache-Control: private, max-age=0
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
Server: gws
Content-Length: 258

<PAGE CONTENT>
```

Content-Length indicates the length, in bytes, of the message body.

2.1.2. HTTP Response

```
HTTP/1.1 200 OK
Date: Fri, 13 Mar 2015 11:26:05 GMT
Cache-Control: private, max-age=0
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
Server: gws
Content-Length: 258

<PAGE CONTENT>
```

CONTENT

This is the actual **content** of the requested resource. The content can be an HTML page, a document, or even a binary file. The type of the content is, of course, contained in the `Content-type` header.

2.1.3. HTTP Header Field Definitions

For now this information about requests and responses are enough. We will inspect them more in depth later on.

If you want to dig deeper in syntax and semantics of all standard `HTTP/1.1` header fields, please check the following RFC 2616.

It lists and explains all the header fields in detail.

http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.1

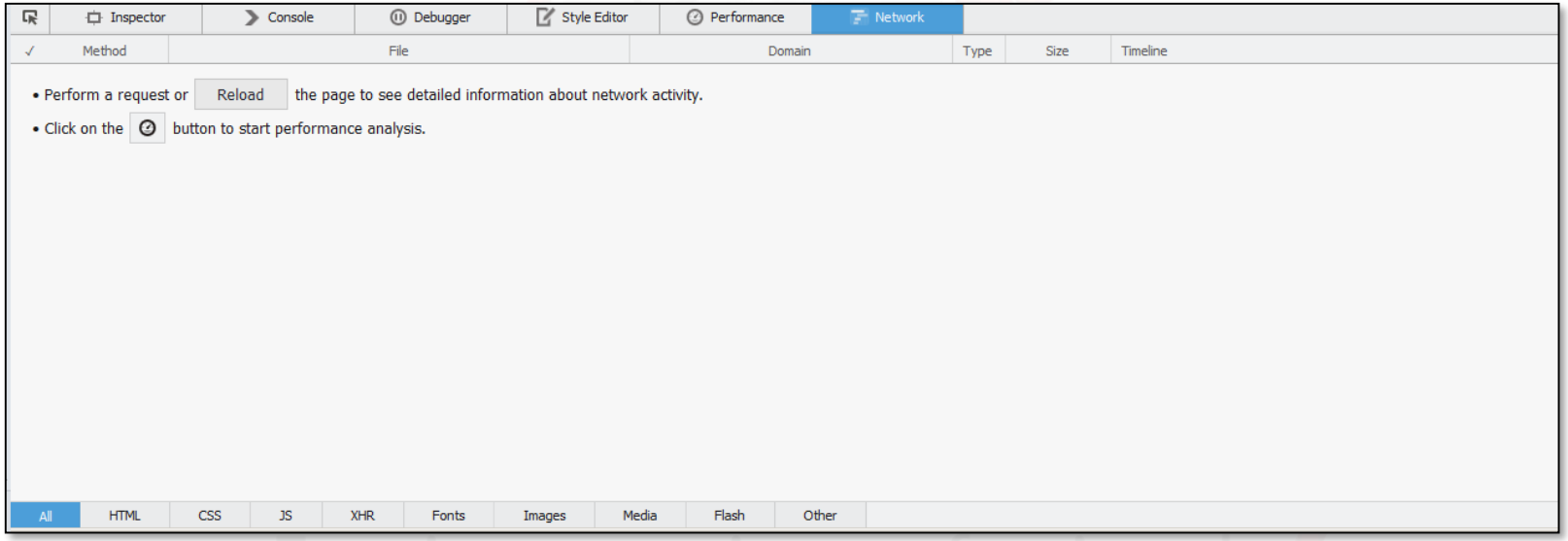
2.1.3. HTTP Header Field Definitions

The best way to understand something is to play around with it a bit.

Firefox (as well as other web browsers) already have some features that allow us to inspect HTTP Headers on the fly.

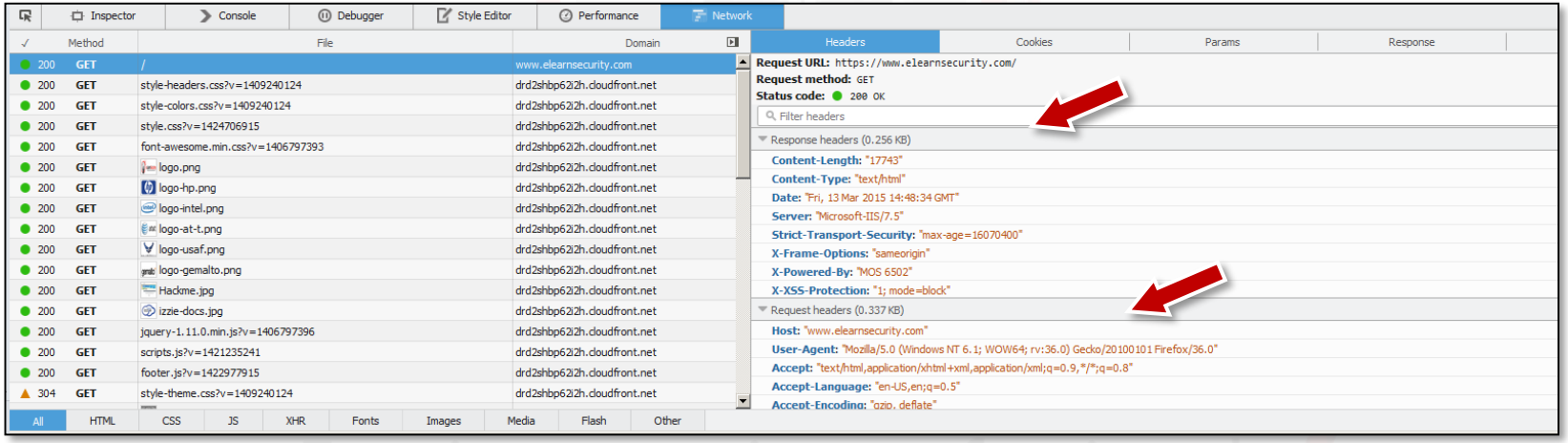
Once Firefox starts, open the options menu and select *Developer -> Network*.

You should see something like the below snapshot. Now we just need to browse any web page we want and all the requests and responses will be listed in this pane.



2.1.3. HTTP Header Field Definitions

By selecting any request or response from the list, a new panel appears on the right and we will be able to inspect information such as headers, cookies, parameters and so on.



2.1.4. HTTPS

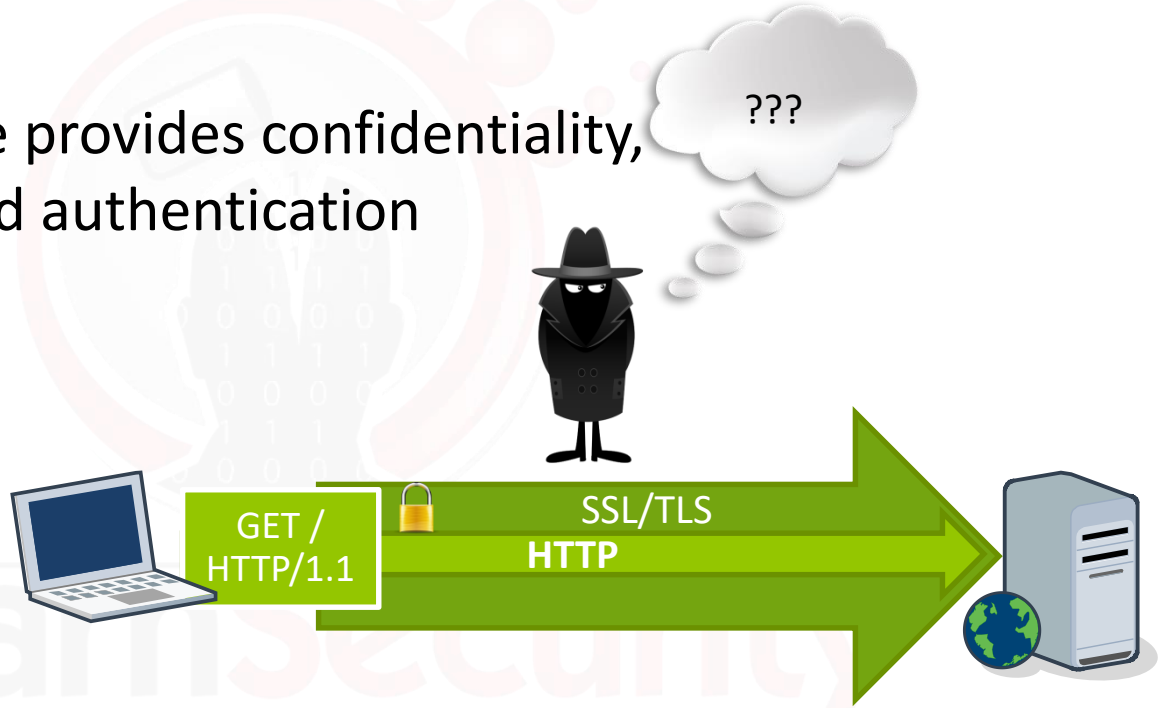
Now that you know how HTTP works, let us see how to **protect it**!

HTTP content, as in every clear-text protocol, can be easily intercepted or mangled by an attacker on the way to its destination. Moreover, HTTP does not provide strong authentication between the two communicating parties.

In the following slides you will see how to **protect HTTP** by means of an **encryption layer**.

HTTP Secure (HTTPS) or HTTP over SSL/TLS is a method to run HTTP, which is a clear-text protocol, over SSL/TLS, a cryptographic protocol.

This layering technique provides confidentiality, integrity protection and authentication to the HTTP protocol.



In other words, when using HTTPS:

- An attacker on the path cannot sniff the application layer communication.
- An attacker on the path cannot alter the application layer data.
- The client can tell the real identity of the server and, sometimes, vice-versa.

HTTPS **does not protect against web application flaws**! All the attacks against an application happen regardless of SSL/TLS.

The extra encryption layer just protects data exchanged between the client and the server. It does not protect from an attack against the application itself.

Attacks such as XSS and SQL injections will still work.

Understanding how HTTP and web applications work is fundamental to mount both stealthy and effective attacks!

ENCODING

2.2.1. Introduction

Information encoding is a critical component of information technology. Its main function is to represent the low-level mapping of the information being handled.

The encoding process, often invisible to end users, occurs each time an application needs to elaborate on any data. Web applications are not excluded from this. Like many other applications, they continuously process thousands of pieces of information even for simple requests.

2.2.1. Introduction

Understanding the encoding schemes used by a web application can give you a big advantage during the detection and exploitation of a vulnerability.

In this part of the course, we will introduce you to the basics of web application encoding. You will find that they are pretty much the same as any other application encoding.

2.2.2. Charset

Internet users, via their web browsers, request billions of pages every day. All of the content of these pages are displayed according to a charset. But what is a character set?

As the word suggests, it contains a set of characters: they represent the set of all symbols that the end user can display in their browser window. In technical terminology, a charset consists of pairs of symbols and code points.

http://www.iana.org/go/rfc2978

2.2.2. Charset

The **symbol** is what the user reads, as he sees it on the screen. The **code point** is a numeric index, used to distinguish, unambiguously, the symbol within the charset. A symbol can be shown only if it exists in the charset.

Examples of charsets are: ASCII, Unicode, Latin-1 and so on.

ASCII

The ASCII (American Standard Code for Information Interchange) charset contains a small set of symbols. Originally it was 128 only, but now it is usually defined with its extended version for a total of 255. It is old and it was designed to support only US symbols.

For example, ASCII cannot be used to display Chinese symbols, among many others. The ASCII charset doesn't contain symbols like © † ∑ α β «.

ASCII

Here are some examples:

CODE	HEX	SYMBOL
65	41	A
66	42	B
67	43	C
68	44	D
…	…	…

You can find the complete list here.

http://www.ascii-code.com/

UNICODE

Unicode (Universal Character Set) is the character encoding standard created to enable people around the world to use computers in any language. It supports all the world's writing systems. Here you can find the whole Unicode charset.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0410	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
0420	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
0430	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п

http://unicode-table.com/en/#0032

2.2.3. CharSet vs. Charset Encoding

Character encoding (or simply encoding) is the representation, in bytes, of the symbols of a charset: a mapping of the symbols to a series of ordered bytes so that your data can get to you.

A symbol can be represented by using one or more bytes.

http://www.w3.org/International/articles/definitions-characters/

2.2.3.1. Unicode Encoding

Unicode has three main types of implementation of character encoding: UTF-8, UTF-16 and UTF-32, where **UTF** stands for *Unicode Transformation Format*.

The numbers 8,16 and 32 are the amount of bits used to represent code points.

For example, the same symbol will be represented as follow:

SYMBOL	UNICODE	UTF-8	UTF-16	UTF-32
!	U+0021	21	00 21	00 00 00 21
W	U+0057	57	00 57	00 00 00 57
◎	U+2B80	E2 AE 80	2B 80	00 00 2B 80
⌗	U+2317	E2 8C 97	23 17	00 00 23 17

2.2.3.2. HTML Encoding

Even in HTML, it is important to consider the information integrity of the URL's, and ensure that user agents (browsers & co.) display data correctly.

There are two main issues to address: inform the user agent on which character encoding is going to be used in the document, and preserve the real meaning of some characters that have special significance

2.2.3.2. HTML Encoding

According to the HTTP 1.1 RFC, documents transmitted via HTTP can send a charset parameter in the header to specify the character encoding of the document sent: this is the HTTP header `Content-Type`.

HTML4

```
<meta http-equiv="Content-Type" content="text/html;charset=ISO-8859-1">
```

HTML5

```
<meta charset="UTF-8">
```

https://tools.ietf.org/html/rfc2616

If not defined, the RFC defines as default charset the *ISO-8859-1*: *"8-bit single-byte coded graphic character sets" aka Latin 1*. Setting an incorrect charset or simply omitting it can bring on some really unexpected behavior. If you intentionally set an incorrect charset, your browser may not display some symbols correctly.

These encoding schemas that we have talked about so far can be applied to all applications.

HTML Entities

In the HTML language, there are many characters (symbols) with special meanings.

For example, the symbol < describes the start of an HTML tag, and obviously that tag will not be shown to the end user; the symbol > indicates the end, and so on.

HTML Entities

If you want to show these symbols in your web document and you want to avoid the symbols being interpreted by your browser as HTML language elements, then you need to use the related entities.

HTML Entities

An **HTML entity** is simply a string (starting with & or &# and ending with ;) that corresponds with a symbol.

When the browser encounters an entity in an HTML page it will show the symbol to the user and will not ever interpret the symbol as an HTML language element.

Let us see some examples.

HTML Entities

As the standard states, character references must start with a **U+0026 AMPERSAND** character (**&**) and following this there are multiple ways to represent character references.

Character Reference	Rule	Encoded Character
Named entity	**&** + named character references + **;**	**<**
Numeric Decimal	**&** + **#** + **D** + **;** **D = a decimal number**	**<**
Numeric Hexadecimal	**&** + **#x** + **H** + **;** **H** = an hexadecimal number (*case-insensitive)*	**<** **<**

http://www.w3.org/html/wg/drafts/html/master/syntax.html#named-character-references

Although the primary purpose of HTML entities is not really to be a security feature however, its use can limit most client side attacks (IE: XSS).

We will discuss this more in the following chapters.

2.2.3.3. URL Encoding (Percent encoding)

As stated in the **RFC 3986**, URLs sent over the Internet must contain characters in the range of the US-ASCII code character set. If unsafe characters are present in a URL, encoding them is required.

This encoding is important because it limits the characters to be used in a URL to a subset of specific characters:

1. Unreserved Chars: [a-zA-z] [0-9] [- . _ ~]
2. Reserved Chars (they have a specific purpose)
   - : / ? # [ ] @ ! $ & " ( ) * + , ; = %

http://tools.ietf.org/html/rfc3986#section-2.1

2.2.3.3. URL Encoding (Percent encoding)

Other characters are encoded by the use of a percent char (%) plus two hexadecimal digits. Reserved chars must be encoded when they have no special role inside the URL. What follows is a list of common encoded characters:

Character	Purpose in URI	Encoding
#	Separate anchors	%23
?	Separate query string	%3F
&	Separate query elements	%26
+	Indicates a space	%2B

2.2.3.3. URL Encoding (Percent encoding)

When you visit a site, URL-encoding is performed automatically by your browser. This happens automatically behind the scenes in your browser while you surf.

Here is a complete URL-encoding reference.

http://www.w3schools.com/tags/ref_urlencode.asp

2.2.3.3. URL Encoding (Percent encoding)

Although it appears to be a security feature, URL-encoding is not. It is only a method used to send data across the Internet but, it can lower (or enlarge) the attack surface (in some cases).

Generally, web browsers (and other client-side components) automatically perform URL-encoding and, if a server-side script engine is present, it will automatically perform URL-decoding.

2.2.3.3. URL Encoding (Percent encoding)

Let us show you some important examples about how web browsers URL-encode simple user requests.

Browser	index.html?arg=test	index.html?arg= test with spaces	index.html?arg= hello world
	arg=test	arg=%20test%20with%20spaces	arg=%3Ch1%3Ehello%20world%3C/h1%3E
	arg=test	arg= test with spaces	arg= hello world
	arg=test	arg=%20test%20with%20spaces	arg=%3Ch1%3Ehello%20world%3C/h1%3E
	arg=test	arg=%20test%20with%20spaces	arg=%3Ch1%3Ehello%20world%3C/h1%3E

2.2.3.4. Base64

Base64 is a binary-to-text encoding schema used to convert binary files and send them over Internet. For example, the e-mail protocol makes massive use of this encoding to attach files to messages.

The HTML language permits the inclusion of some resources by using this encoding. For example, an image can be included in a page by inserting its binary content that has been converted to base64.

The alphabet of the Base64 encoding scheme is composed of digits [0-9] and Latin letters, both upper and lower case [a-zA-Z], for a total of 62 values. To complete the character set to 64 there are the plus (+) and slash (/) characters.

Different implementations however, may use other values for the latest two characters and the one used for padding (=).

2.2.3.4. Base64

The following code will show an image in a web document. The server will send this image without the need to read it from another source like the file system.

```
<img_src="
DwAPAAACIISPeQHsrZ5ModrLlN48CXF8m2iQ3YmmKqVlRtW4MLwWACH+H09wdGltaXplZCBi
eSBVbGVhZCBTbWFydFNhdmVyIQAAOw=="alt="Base64 encoded image"
width="150"height="150"/>
```

In this chapter, we have discussed the major encoding schemas, but that is not all of them.

In fact, remember that any web designer or developer could easily create their own encoding schema.

Same Origin

One of the most important and critical points of web application security is same origin policy.

This policy prevents a script or a document from getting or setting properties of another document that comes from a different origin.

CSS stylesheets, images and scripts are loaded by the browser without consulting the policy.

Same Origin Policy (SOP) is consulted when cross-site HTTP requests are initiated from within client side scripts (IE: JavaScript), or when an Ajax request is run.

2.3.1. Origin definition

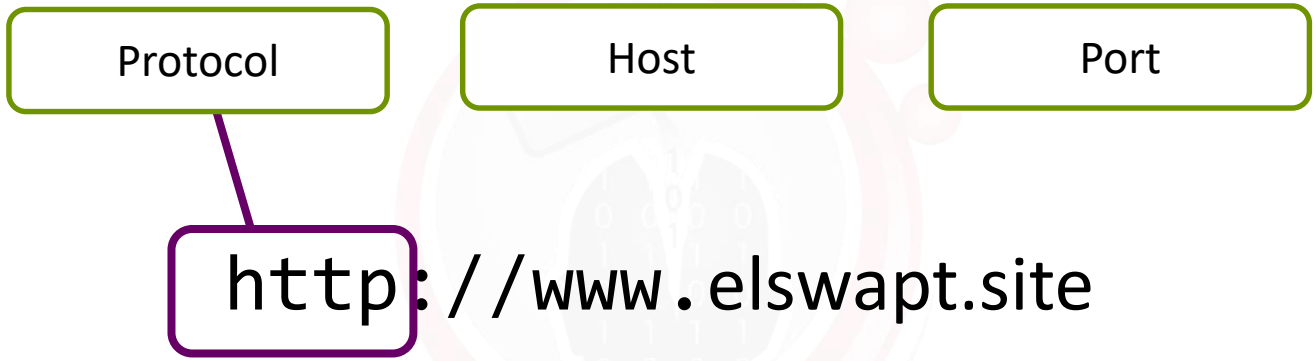
The origin is defined by the following triplet:

Protocol	Host	Port

Take a look at this example:

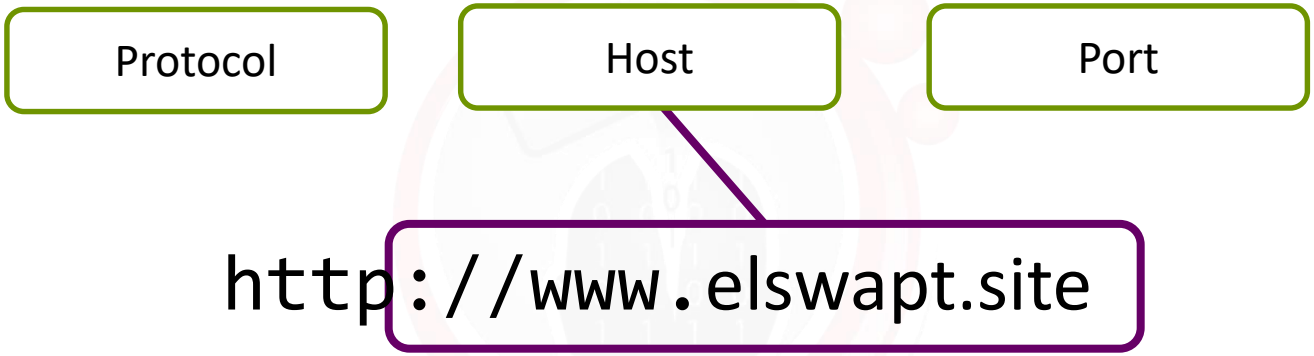
http://www.elswapt.site

2.3.1. Origin definition



The origin of `http://www.elswapt.site` is different from `https://www.elswapt.site` origin because the protocol is different.

2.3.1. Origin definition



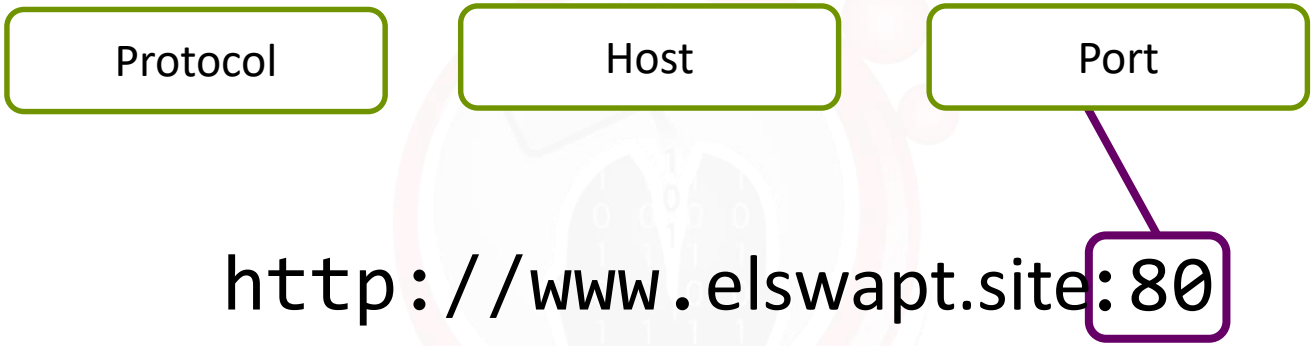
The origin of `http://www.elswapt.site` is different from `http://admin.elswapt.site` origin because the host is different.

http://www.elswapt.site

The hierarchy of domains descends from the right to the left. In the above example:

- `site` is the top-level domain (TLD)
- `elswapt` is the second-level domain (SLD)
  - subdomain of `site`
- `www` is the third level domain
  - subdomain of `elswapt`
- and so on

2.3.1. Origin definition



When not specified, the default port is always 80.

2.3.1. Origin definition

Let us see some SOP examples applied to the following address: http://els.wapt.site/index.php:

URL	SOP	Reason
http://els.wapt.site/admin/index.php	✓	Same protocol, host and port
https://els.wapt.site/index.php	✗	Different protocol
http://els.wapt.site/index.php:8080	✗	Different port
http://www.els.wapt.site/index.php	✗	Different host

Content from `about:blank`, `javascript:` and `data:` inherits the origin.

2.3.1. Origin definition

IMPORTANT

It is important to know that Internet Explorer works a bit differently from other browsers. It has two exceptions:

- **Port**: it does not consider the port as a Same Origin component.
- **Trust Zone**: the Same Origin is not applied to domains that are in highly trusted zone (i.e. corporate domains)

2.3.2. What does SOP protect from?

Why should the browser control where a script can or cannot have access ?

Let us explain it using a simple example.

Suppose you are logged in to your bank site and suppose your friend invites you to visit his new website. Also, suppose your friend is a malicious friend.

2.3.2. What does SOP protect from?

As a general rule, the **S**ame **O**rigin **P**olicy (referred to as SOP) prevents JavaScript, running on a given origin, from interacting with a document from a different origin.

The primary purpose of SOP is to isolate requests coming from different origins.

2.3.2. What does SOP protect from?

What would it happen if SOP did not exist?

Your evil friend could build a crafted page, instigate you to visit it, and once visited by you, access (some) personal information from your bank account.

As you can see without SOP you could not surf the Internet.

The main rule of SOP is:

*"A **document** can access (through JavaScript) the properties of another document only if they have the same origin."*

More precisely, the browser always performs the request successfully but it returns the response to the user only if the SOP is respected.

2.3.3. How SOP Works

With the term *document*, we are referring to an HTML page, an iframe included in the main page, or a response to an Ajax request.

As stated above: images, style information (*.css) and JavaScript files (*.js) are excluded from the previous statement; they are always accessible regardless their origin, and the browser loads them without consulting SOP.

Now, let us look at two examples:

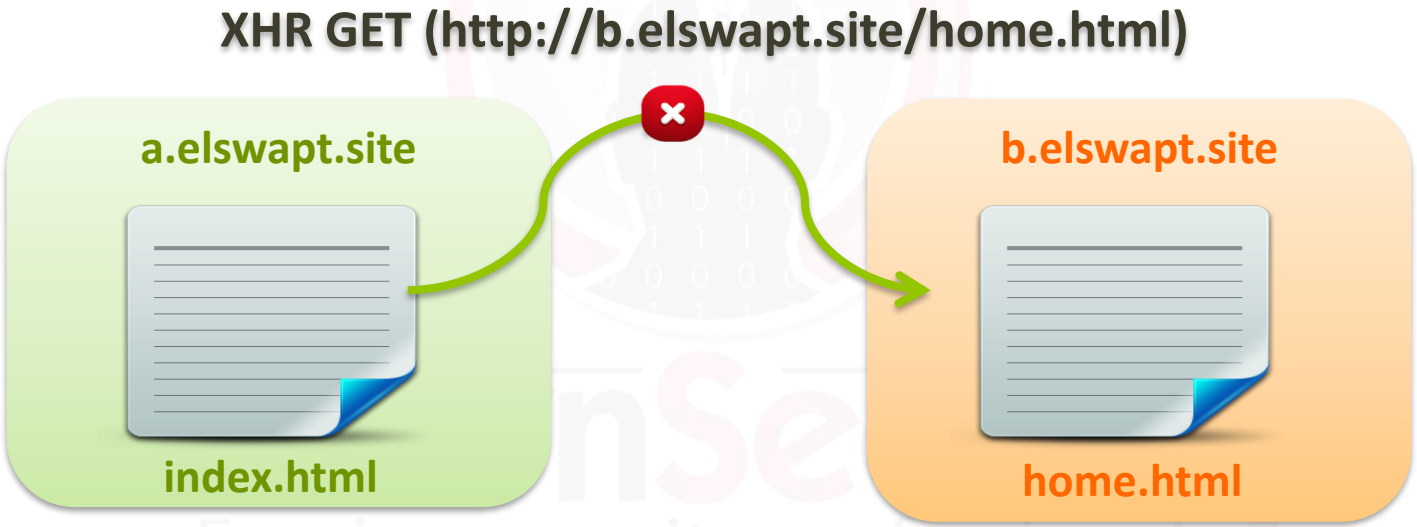
Example 1

Example 2

Example 1

Let us suppose that `index.html` on domain `a.elswapt.site` (referred to as *origin1*: `http://a.elswapt.site`) wants to access, via an Ajax request (*xhr*), the `home.html` page on domain `b.elswapt.site` (referred to as *origin2*: `http://b.elswapt.site`).

The document `index.html` on domain `a.elswapt.site` cannot access, via an Ajax request (*xhr*) the `home.html` page on domain `b.elswpat.site`



Example 2

Let us see another example. We have two documents: the main document http//www.elswapt.site/index.html and the iframe document http//www.elswapt.site/iframe.html.

index.html

```
<html>
…
  <body>
    <iframe src="http//www.elswapt.site/iframe.html">
    </iframe>
  </body>
</html>
```

2.3.3.2. Example2

The two document objects have the same origin, each document can access the other via JavaScript. So, within the main document the following JavaScript instruction would be successful:

```
window.frames[0].body= "Hello world";
```

Similarly, within the iframe document, the following JavaScript instruction would be successful:

```
window.parent.body= "Hello world";
```

2.3.3.2. Example2

If we point the iframe to `http://www.mybank.bank` the previous JavaScript code would fail because the two windows do not have the same origin.

Keep the previous example in mind. SOP often defines the boundaries of many client-side attacks.

There are several exceptions to SOP restrictions:

- `window.location`
- `document.domain`
- `Cross window messaging`
- `Cross origin resource sharing (CORS)`

2.3.4.1. Window.location

A document can always write the `location` property of another document.

"The `window.location` object can be used to get the current page address (URL) and to redirect the browser to a new page."

2.3.4.1. Window.location

Consider two documents on your browser with some existing relationship (i.e.: the first document includes the second via an iframe, or the second document has been opened by the first one with a `window.open` call).

2.3.4.1. Window.location

Each document can write the location property of the other, but cannot read it, except the case where the two documents have the same origin.

2.3.4.1. Window.location

This means that the location property can be always changed, notwithstanding the same origin policy that determines whether a new document can be loaded.

2.3.4.1.1. Window.location - Example

Suppose you have the following document http://www.elswapt.site/index.html:

```
<html>
…
  <body>
    <iframe src="http//www.elearnsecurity.com/index.html">
    </iframe>
  </body>
</html>
```

2.3.4.1.1. Window.location - Example

Within the `index.html` document, the following JavaScript instruction is run successfully:

```
window.frames[0].location=http://www.google.com;
```

2.3.4.1.2. Window.location - Security Issues

A document can always update the `location` property of another document, if they have some relationship.

Typical relationships are: a document is embedded within another via an iframe element, one document is opened by the other via the `window.open` DOM API.

Let us look at some typical examples.

2.3.4.1.2. Window.location - Security Issues

A document **X** included by another document **Y** via an iframe, can always change the location of **Y**.

Code on Y

```
<html>
  <body>
    <iframe src='X'></iframe>
  </body>
</html>
```

Code on X

```
<html>
  <head>
    <script type="text/javascript">
       window.parent.location="http://www.google.com";
    </script>
  </head>
</html>
```

2.3.4.1.2. Window.location - Security Issues

A document **X** opened by document **Y** through the `window.open` DOM API can always change the location of **Y**.

Code on Y

```
<html>
  <body>
    <button name='Click' onclick="window.open(X);">
       Click
    </button>
  </body>
</html>
```

Code on X

```
<html>
  <head>
    <script type="text/javascript">
       window.opener.location="http://www.google.com";
    </script>
  </head>
</html>
```

2.3.4.2. Document.domain

Another important exception is related to the DOM property called `document.domain`.

This property describes the domain portion of the origin of the current document.

A document with the URL

- `http://subdomain.domain.com/index.html`

has the `document.domain` property set to

- `subdomain.domain.com`

2.3.4.2. Document.domain

This property can be changed. A document can update its own `document.domain` to a higher level in the domain hierarchy, except for the top level (e.g. `.com`).

The second-level domain (e.g. `domain.com`) can be specified but it **cannot** be changed (e.g. from `domain.com` to `whitehouse.gov`).

By changing the `document.domain` property, a document partially changes its own origin.

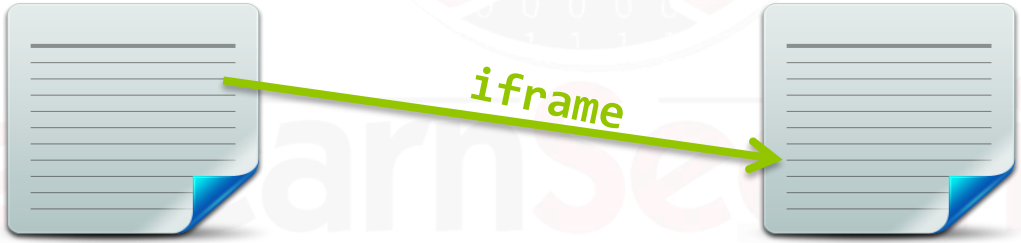
2.3.4.2.1. Document.domain Example

Let us say that a document with the URL

- `http://a.elswapt.site/index.html`

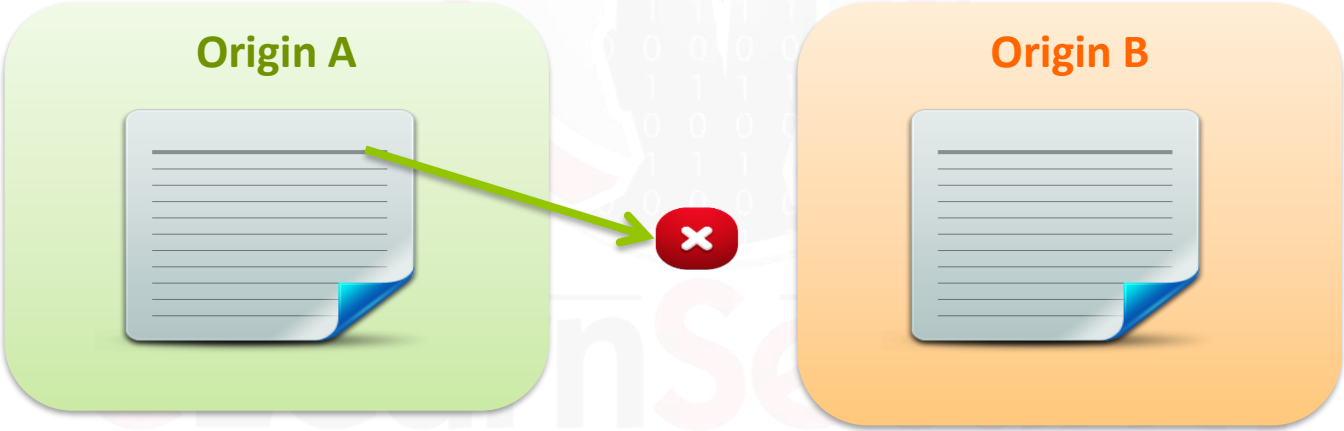
includes, via an iframe, another document belonging to a different origin

- `http://b.elswapt.site/home.html`



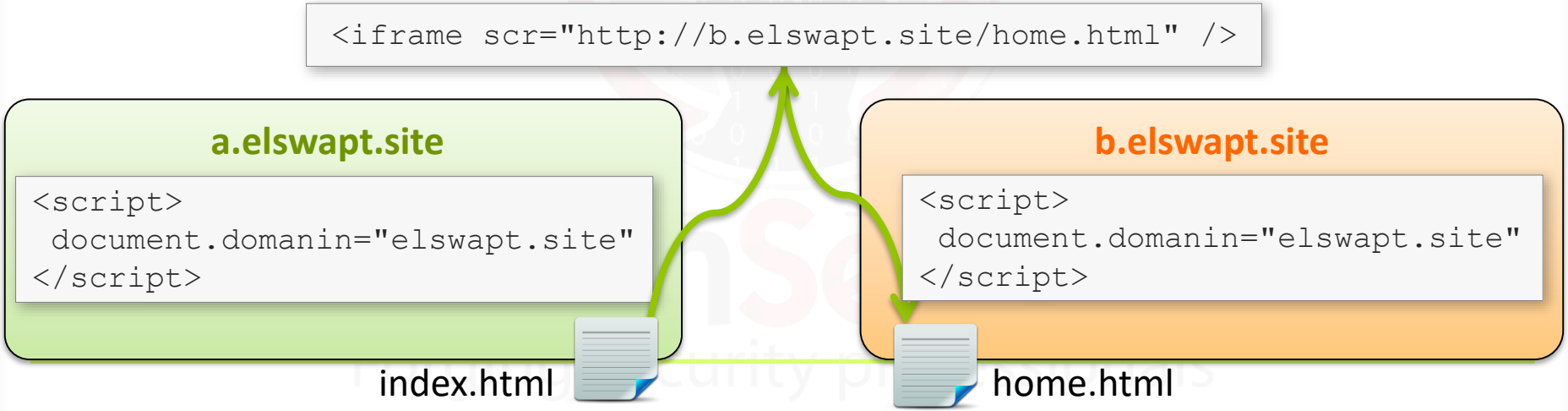
2.3.4.2.1. Document.domain Example

Due to the SOP, the JavaScript code running from the context of the main document cannot access, via JavaScript, the iframe content because the two documents come from different origins.



2.3.4.2.1. Document.domain Example

The SOP is circumvented, though, if the following JavaScript code (`document.domain="elswapt.site"`) is run by **each of the two documents**. In this manner, the two documents can be considered to have the same origin.



2.3.4.3. Cross window messaging

The new HTML5 feature known as *Cross Window Messaging* permits different documents (iframes, popups, current window) to communicate with each other regardless of the same origin policy by using a simple synchronous mechanism.

Do not worry; this mechanism will be dealt with in-depth in the HTML5 module.

2.3.4.4. Cross Origin Resource Sharing (CORS)

Cross origin resource sharing is a set of specs built to allow a browser to access a few resources by bypassing the same origin policy. The CORS architecture uses custom HTTP response headers and relies upon server-side components or server-side scripting languages.

This also will be dealt with in-depth in the HTML5 module

If you have a FULL or ELITE plan you can click on the image on the left to start the video

Cookies

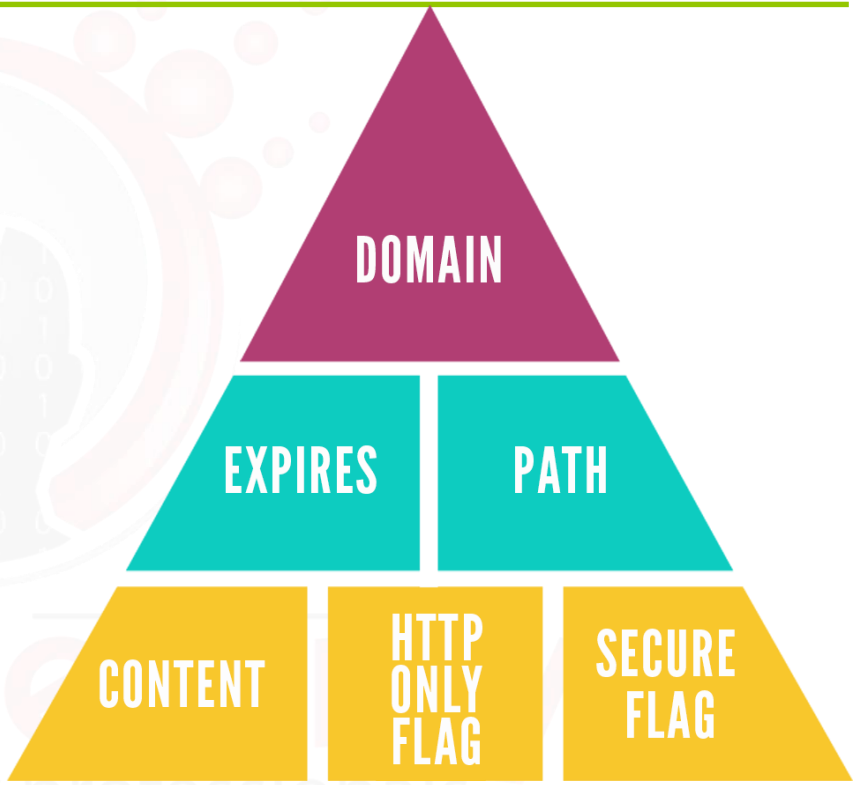
HTTP is a **stateless** protocol. This means that a website cannot retain the state of your visit between different HTTP requests without mechanisms such as sessions or cookies. Each visit without a session or a cookie looks like a new user to a server and a browser.

To overcome this limitation, in 1994 sessions and **cookies** were invented. **Netscape**, a leading company at that time, invented cookies **to make HTTP stateful.**

Cookies are just textual information installed by a website into the "cookie jar" of the web browser. The **cookie jar** is the storage space where a web browser stores the cookies.

They are fragments of text containing variables in the form of `name=value`.

A server can set a cookie via the **Set-Cookie** HTTP header field in a response message. A cookie has a predefined format. It contains the following fields.

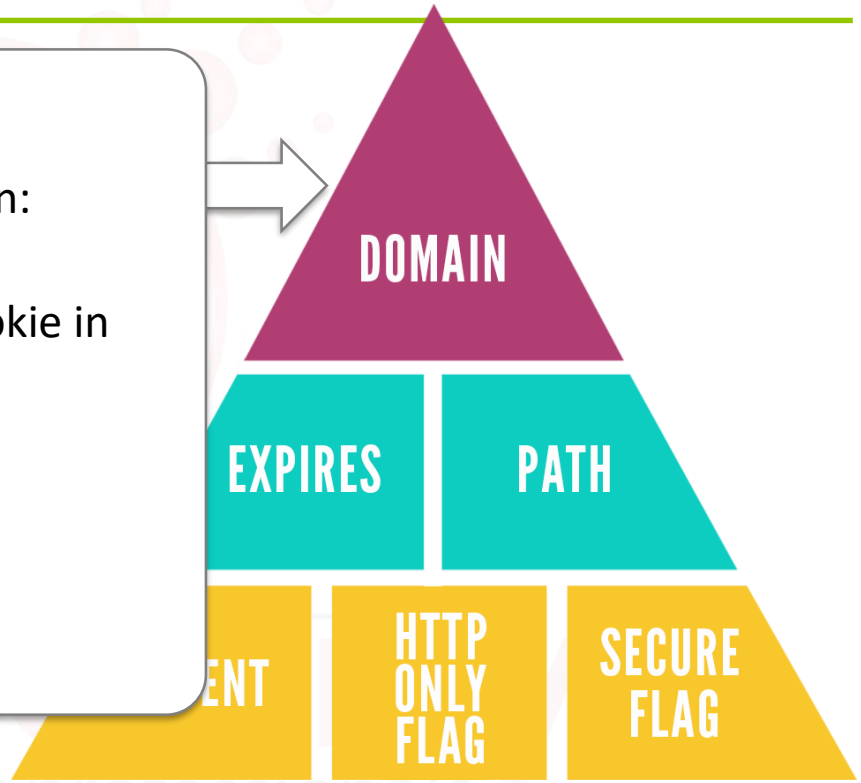


A website only sets a cookie for its domain.

- e.g. google.com sets a cookie for the domain: `google.com` or `.google.com`

This means that the browser will install the cookie in the cookie jar and will send this cookie for any subsequent request to:

- `google.com`
- `www.google.com`
- `maps.google.com`



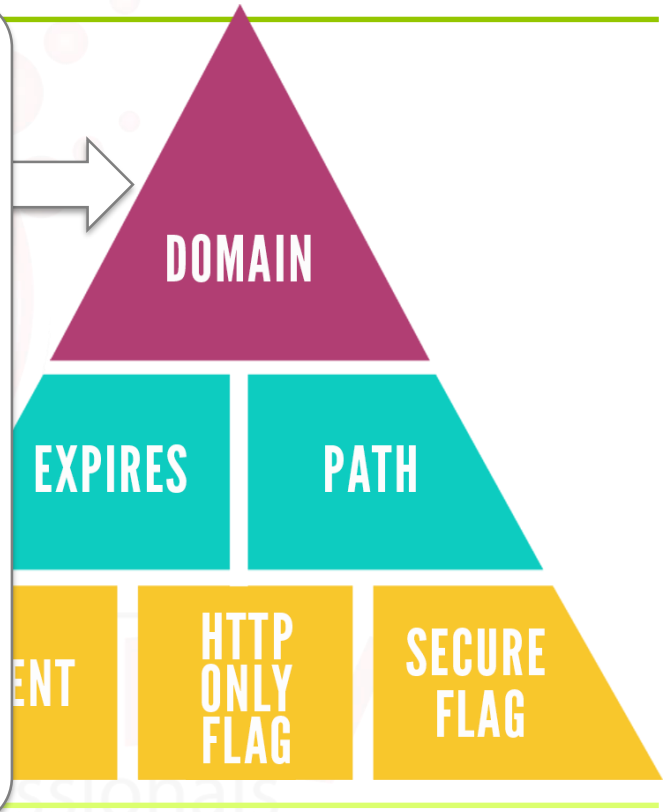
The **scope** of this cookie will be `*.google.com`

Domain A cannot set a cookie for domain B.

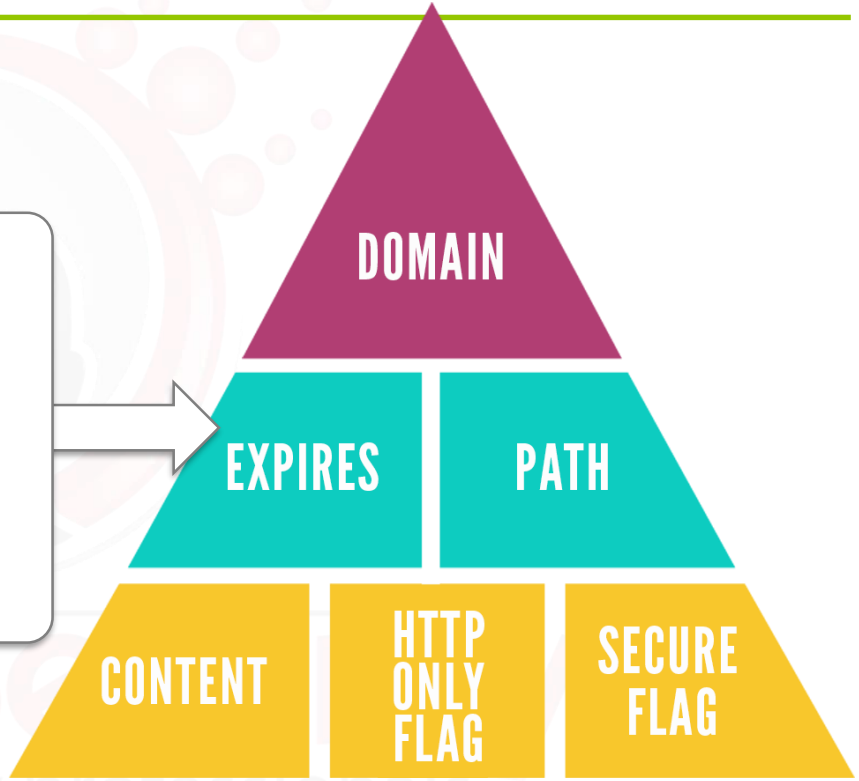
The browser will sends A's cookie in accordance with the above domain scope (to A and all of its subdomains), including the path and the expiration date.

There are two important considerations about the domain field:

- a leading ".", if present, is ignored
- If the **server does not specify** the domain attribute, the browser will automatically set the domain as the server domain and set the cookie **host-only** flag. This means that the cookie will be sent **only to that precise hostname**



Expires gives the cookie a time constraint.
The cookie will only be sent to the server if it is *not* expired.
Session cookies expire when the session exits.



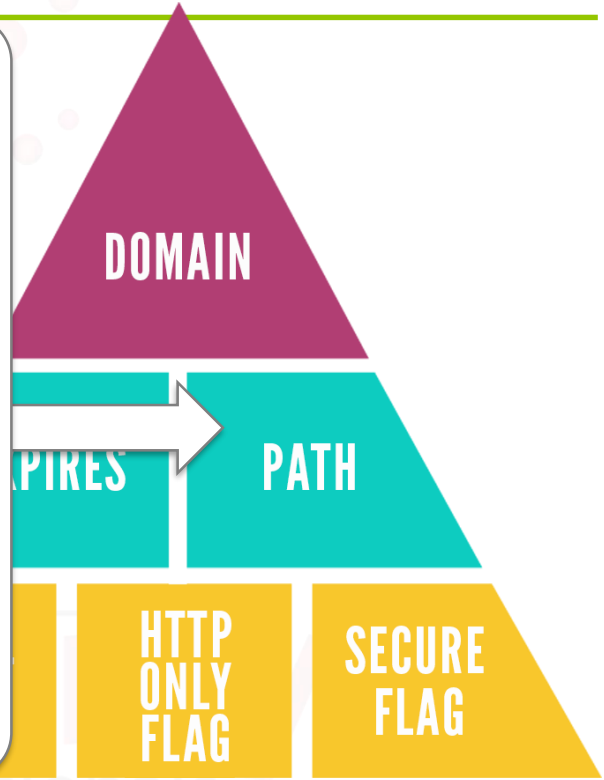
The cookie **Path** field specifies for which requests, within that domain, the browser needs to send the cookie.

For cookies with path =`/downloads`, all subsequent requests to:

- `/downloads`
- `/downloads/foo`
- `/downloads/foo/bar`

will include this cookie.

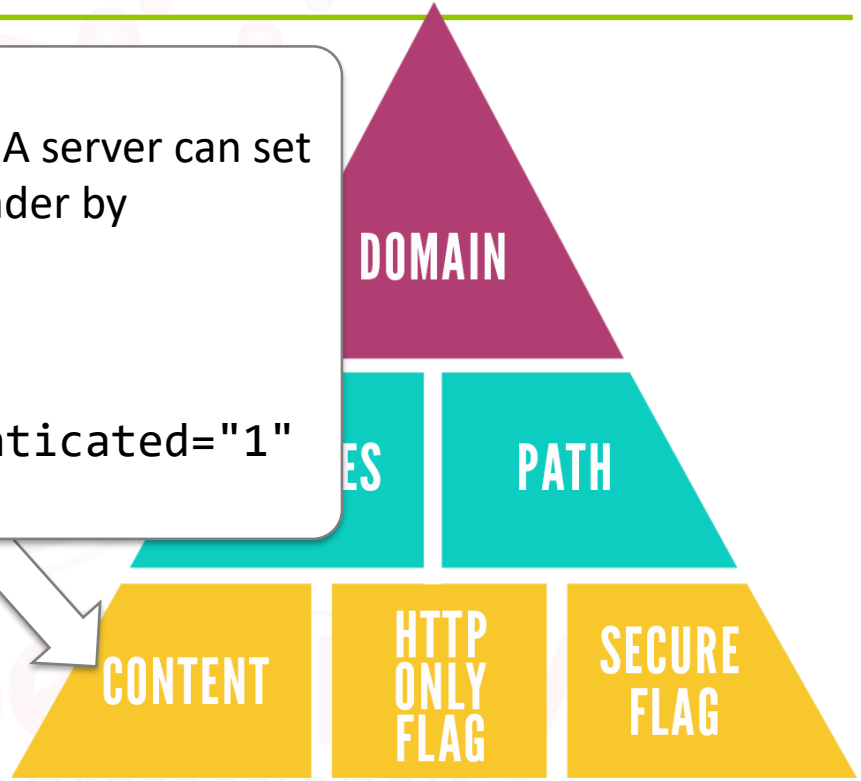
The browser will *not* send this cookie for requests to `/blog` or `/members`



A cookie can carry a number of values at once. A server can set multiple values with a single `Set-Cookie` header by specifying multiple `KEY=Value` pairs.

For example:

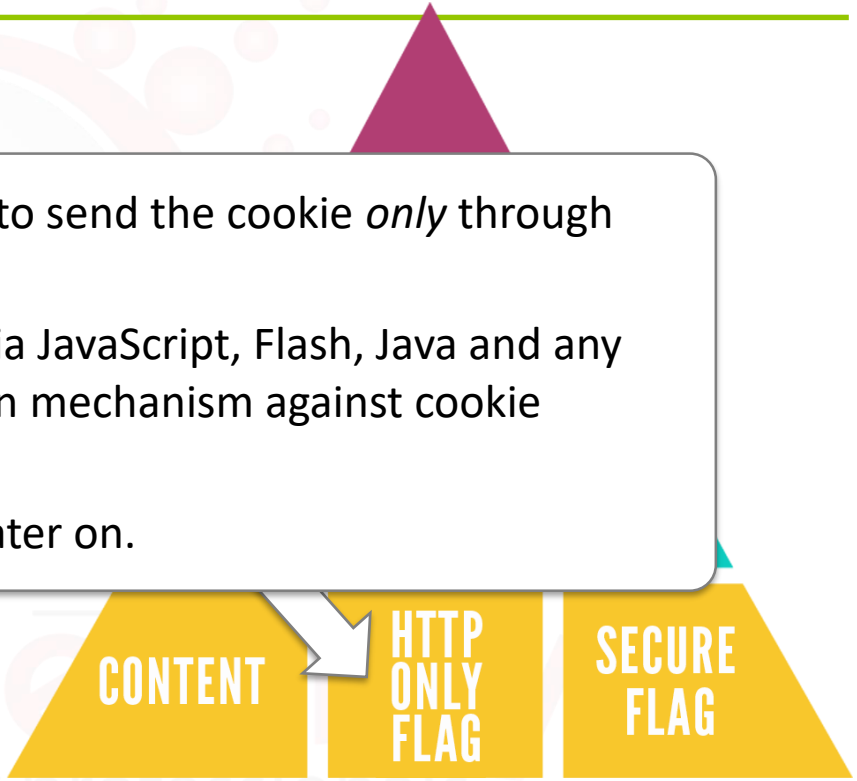
```
Set-Cookie: Username="john"; Authenticated="1"
```



The **HttpOnly flag** is used to force the browser to send the cookie *only* through HTTP.

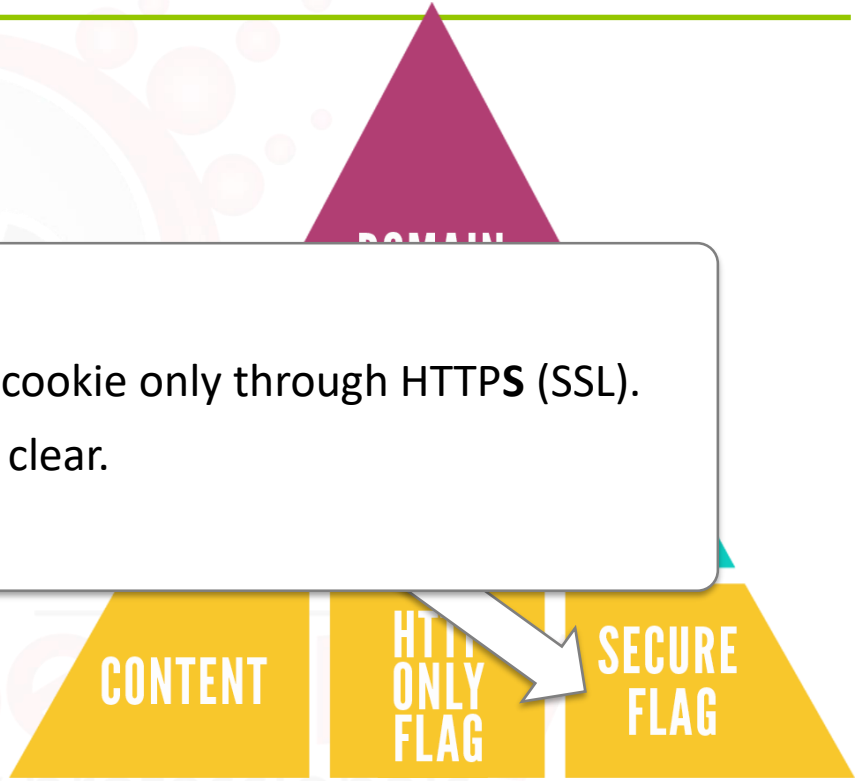
This flag prevents the cookie from being read via JavaScript, Flash, Java and any other non-HTML technology. This is a protection mechanism against cookie stealing via XSS.

You will see how to exploit XSS vulnerabilities later on.



The **Secure flag** forces the browser to send the cookie only through HTTP**S** (SSL).

This prevents the cookie from being sent in the clear.



2.4.1. Cookies Domain

The *domain* attribute represents the domain value for which the cookie is valid. Together with the *path*, *secure* and *expires* attributes, it is useful during the process in determining if a cookie must be submitted along with a new HTTP request.

RFC6265 distinguishes cookies set with a specified domain value from cookies without.

http://tools.ietf.org/html/rfc6265

2.4.1. Cookies Domain

An important difference in the previous RFC is the leading dot at the start of the domain value string. As opposed to the previous RFC, the leading dot (.) has no particular significance; the browser does not care about it.

For example, this means that a browser will handle cookies with these domain values the:

- .elswapt.site
- elswapt.site

2.4.1. Cookies Domain

Let us have a look at the following different cookie domain settings. We will see when and how the cookie will be sent.

- Specified cookie domain
- Unspecified cookie domain
- Internet Explorer exception

2.4.1.1. Specified Cookie Domain

A cookie with a domain value specified will be sent by the browser when one of the following conditions occurs:

1. Cookie domain value = target domain

2. Cookie domain value is different from the target domain **AND** Cookie domain value is a suffix of the target domain.

2.4.1.1. Specified Cookie Domain

Cookie domain value
=
target domain

1

Suppose that the *cookie domain value* is `els.wapt.site` and that the target domain requested by the browser is `els.wapt.site` (it is the same).

If we request the following page:

- `http://els.wapt.site/index.php`

the cookie will be sent.

2.4.1.1. Specified Cookie Domain

2

Cookie domain value is different from the target domain
AND
Cookie domain value is a suffix of the target domain.

Suppose that the cookie domain value is `wapt.site` and that the target domain requested by the browser is `els.wapt.site`. For example we are requesting the page

- `http://els.wapt.site/index.php`

The cookie will be sent because the string `wapt.site` represents a suffix of the string `els.wapt.site`.

2.4.1.1. Specified Cookie Domain

Let us see some examples. A page on the target domain `elswapt.site` sets a cookie with domain value `elswapt.site`.

The browser will send this cookie in HTTP requests matching the following URLs:

- http[s]://elswapt.site/*
- http[s]://www.elswapt.site/*
- http[s]://www.lab.elswapt.site/*
- http[s]://*.elswapt.site/*

2.4.1.1.1. Security Implications

A page on the target domain `*.elswapt.site` can set a cookie with domain value `elswapt.site`.

Example: A page on the target domain `a.b.elswapt.site` sets a cookie with domain value `elswapt.site`.

The browser will send this cookie in requests matching the following URLs: `http[s]://*.elswapt.site/*`

2.4.1.1.1. Security Implications

A page on the target domain `elswapt.site` cannot set a cookie with domain value `*.elswapt.site`.

Example: A page on the target domain `elswapt.site` cannot set a cookie with domain value `a.elswapt.site`.

This means that lower-level subdomains can set cookies for higher domains. Indeed `b.a.elswapt.site` can set a cookie for `a.elswapt.site` or `elswapt.site`.

On the reverse side, higher domains cannot set cookies for lower-level subdomains. Indeed `elswapt.site` cannot set cookies for `anysubdomain.elswapt.site`.

2.4.1.2. Unspecified cookie domain

When a cookie does not contain a domain value, it is assumed that the **host-only-flag** is set to true. A cookie with the host-only-flag value will be sent only to the target domain that set it.

Note that the RFC uses the term host instead of domain.

2.4.1.2. Unspecified cookie domain

If a page on the target domain `elswapt.site` sets a cookie without the domain value, the browser will send this cookie only in HTTP requests that exactly match the following URLs:

`http[s]://elswapt.site/*`.

2.4.1.3. Internet Explorer Exception

Internet Explorer does not distinguish between cookies with a *specified* domain value from ones with *unspecified* values. Cookies with unspecified domain values will be interpreted by the browser as if they had a domain value corresponding to the target domain set in it.

Let us look at an example of this.

2.4.1.3. Internet Explorer Exception

A page on the target domain `elswapt.site` sets a cookie **without** a domain value. IE differs from other browsers, and will consider sending this cookie as if its domain value was set to `elswatp.site` therefore, it will send this cookie in HTTP requests that match the following URLs:

- http[s]://elswapt.site/*
- http[s]://www. elswapt.site/*
- http[s]://www.lab. elswapt.site/*
- http[s]://*. elswapt.site/*

2.4.2. Inspecting the Cookie Protocol

This following slides will depict the process of cookie installation within a web browser.

This example, although pretty simplistic, should finally shed some light on how cookies are handled.

Login

A login page is a great place for a session to begin and also a good point at which a cookie is installed in your browser.


```
POST /login.php
Host: www.google.com

usr=John&Pass=mypass
```

Set-Cookie

The web site will respond with a **Set-Cookie** response header. This header contains the cookie to be installed in the browser and to be included in all subsequent requests to www.google.com


```
HTPP /1.1 200OK
...
Set-Cookie: domain=google.com; path=/; expires=espires=Mon;
16-Apr-2013 19:03:22 GMT; authenticated='1'; httpOnly;
secure;
<PAGE CONTENT>
```

Cookie

For every subsequent request, the browser will consider: Domain scope, Path, Expiration, Flags. If all the above checks pass, a cookie header that contains the cookie will be inserted into the Request header.


```
GET /mail.php
Host: www.google.com
Cookie=authenticated="1";
```

2.4.3. Cookie installation

Let us see some examples wherein the browser accepts cookies sent by the web server and some others where the cookies are rejected.

Examples of Correct Cookie Installation

Examples of Incorrect Cookie Installation

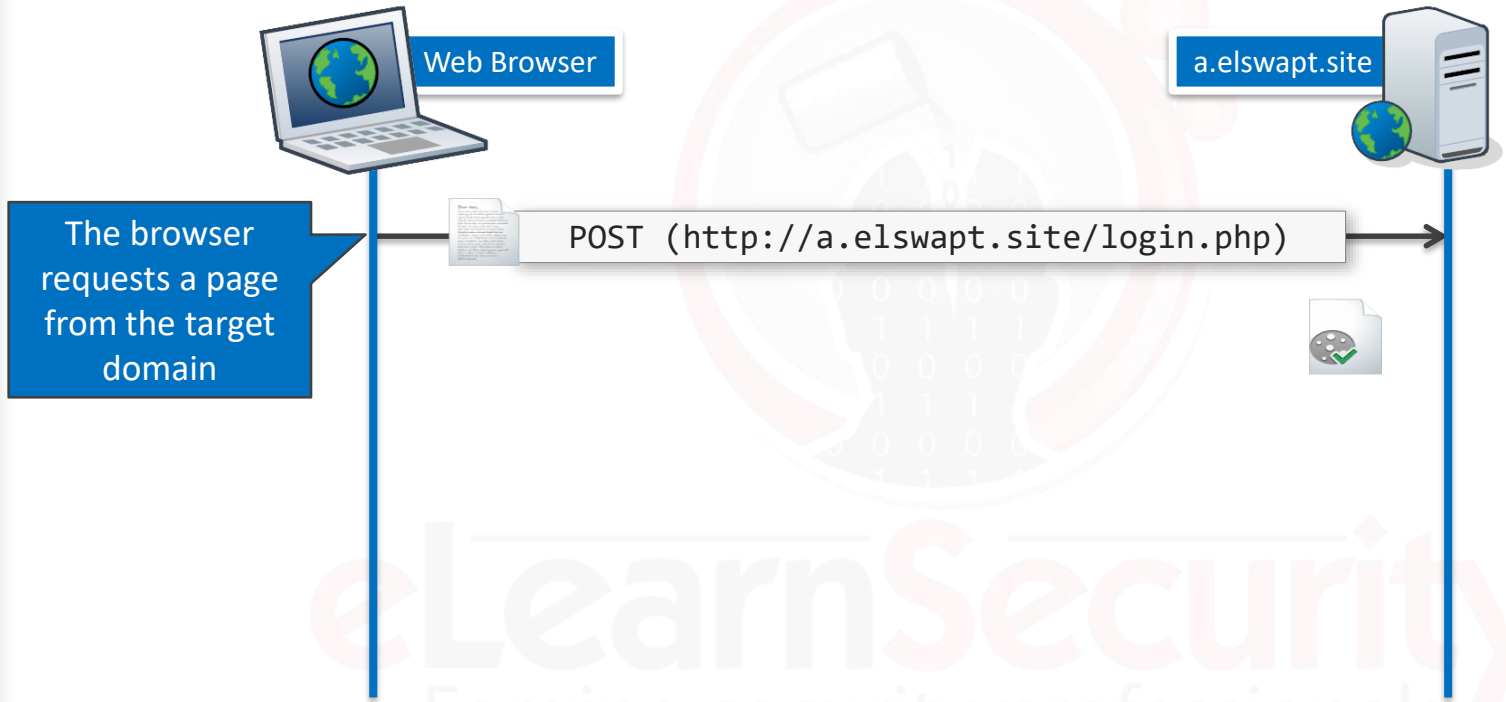
Example #1

The browser requests a page from the target domain `a.elswapt.site` and the web server sends a response, including a cookie without a domain value.

Let us see how this works in the next slide.

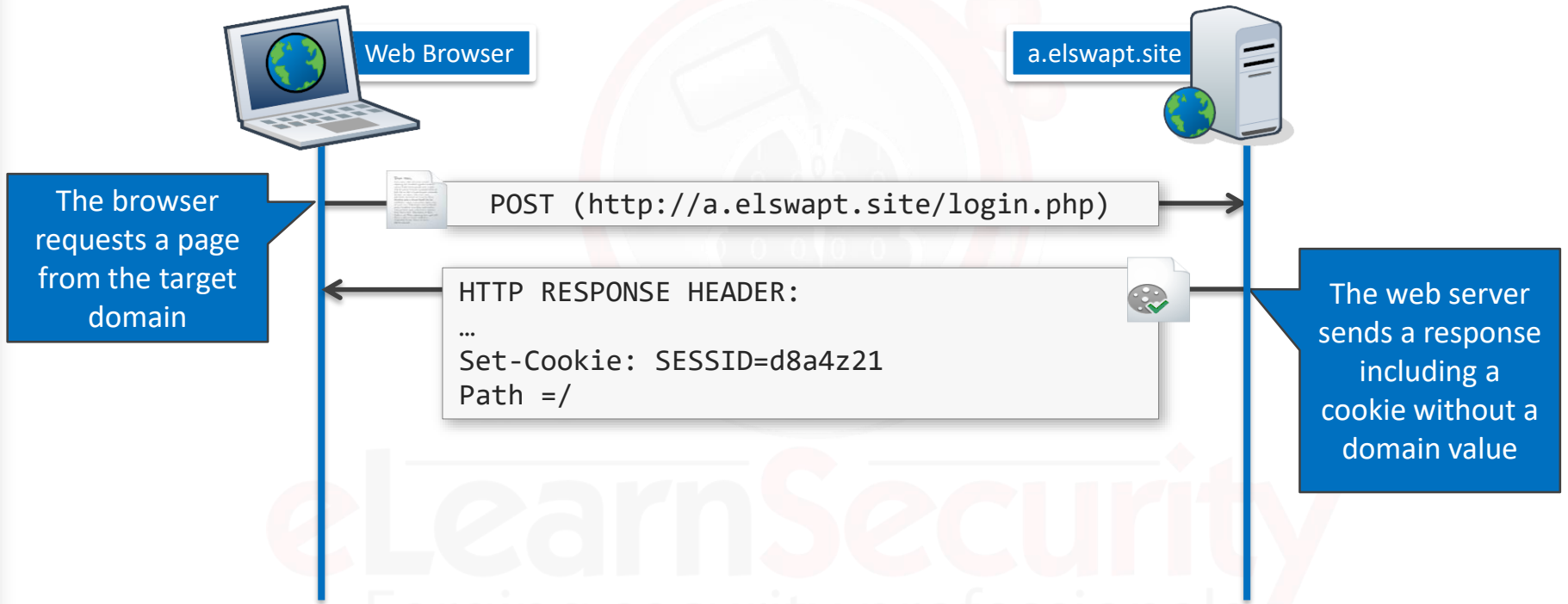
2.4.3.1. Correct Cookie Installation

Example #1



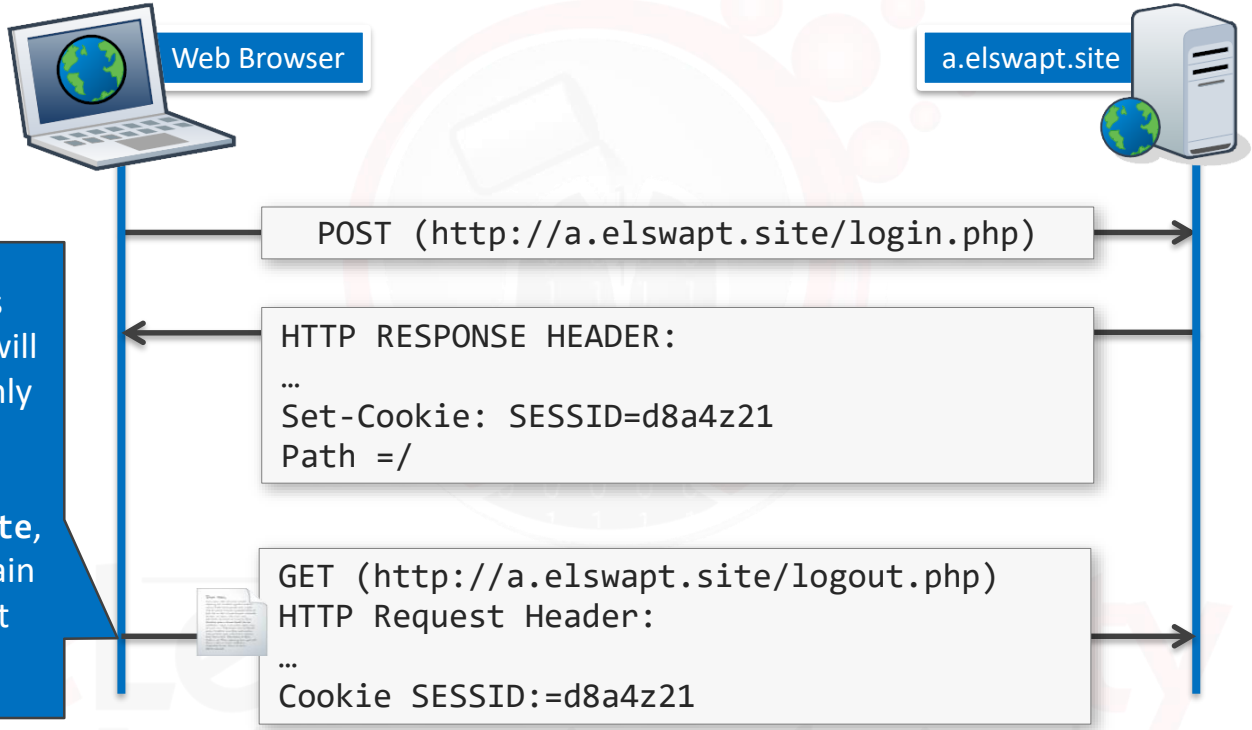
2.4.3.1. Correct Cookie Installation

Example #1



2.4.3.1. Correct Cookie Installation

Example #1



2.4.3.1. Correct Cookie Installation

Example #1

This cookie will be sent in each HTTP request matching the following URLs:

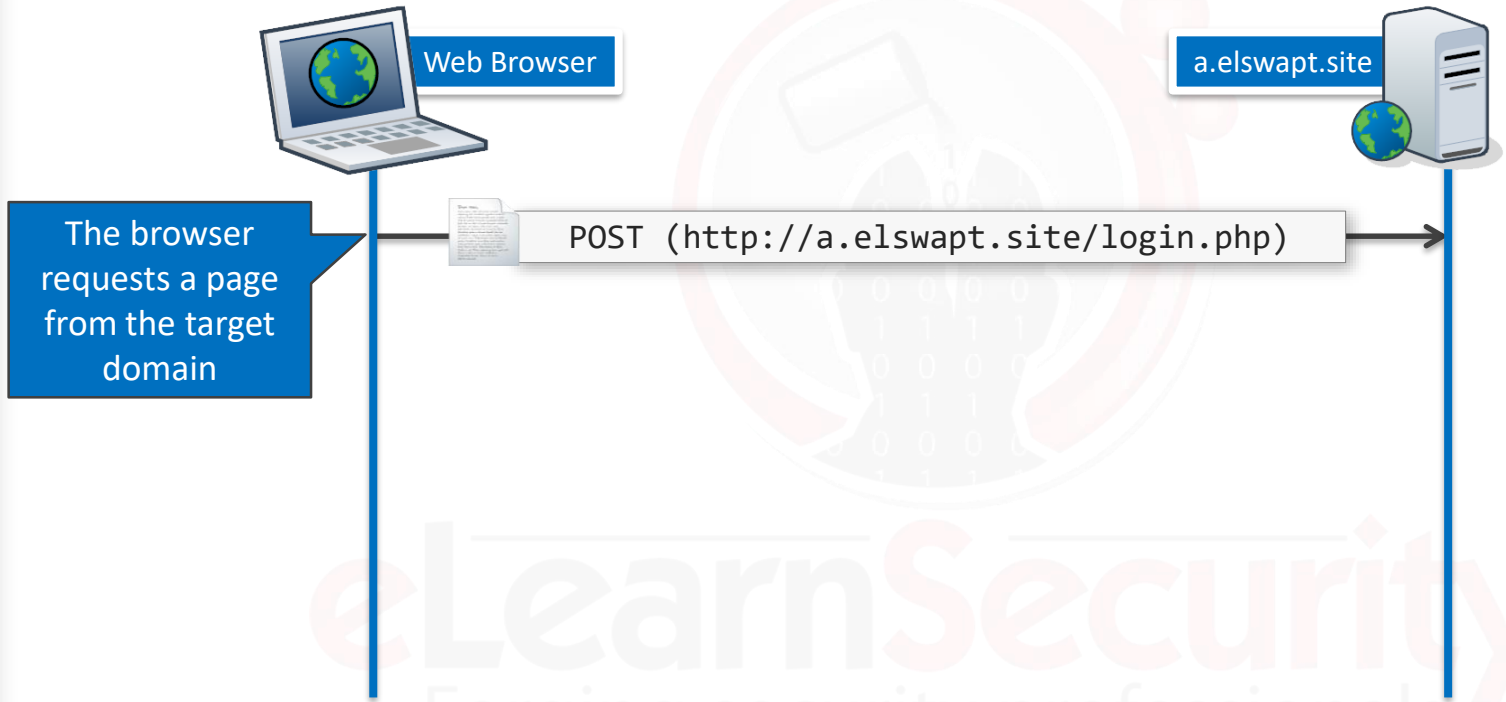
- http://a.elswapt.site/*
- https://a.elswapt.site/*

Example #2

The browser requests a page on the target domain `a.elswapt.site` and the web server sends a response including both a cookie with the domain value `.elswapt.site` and the path value `/`.

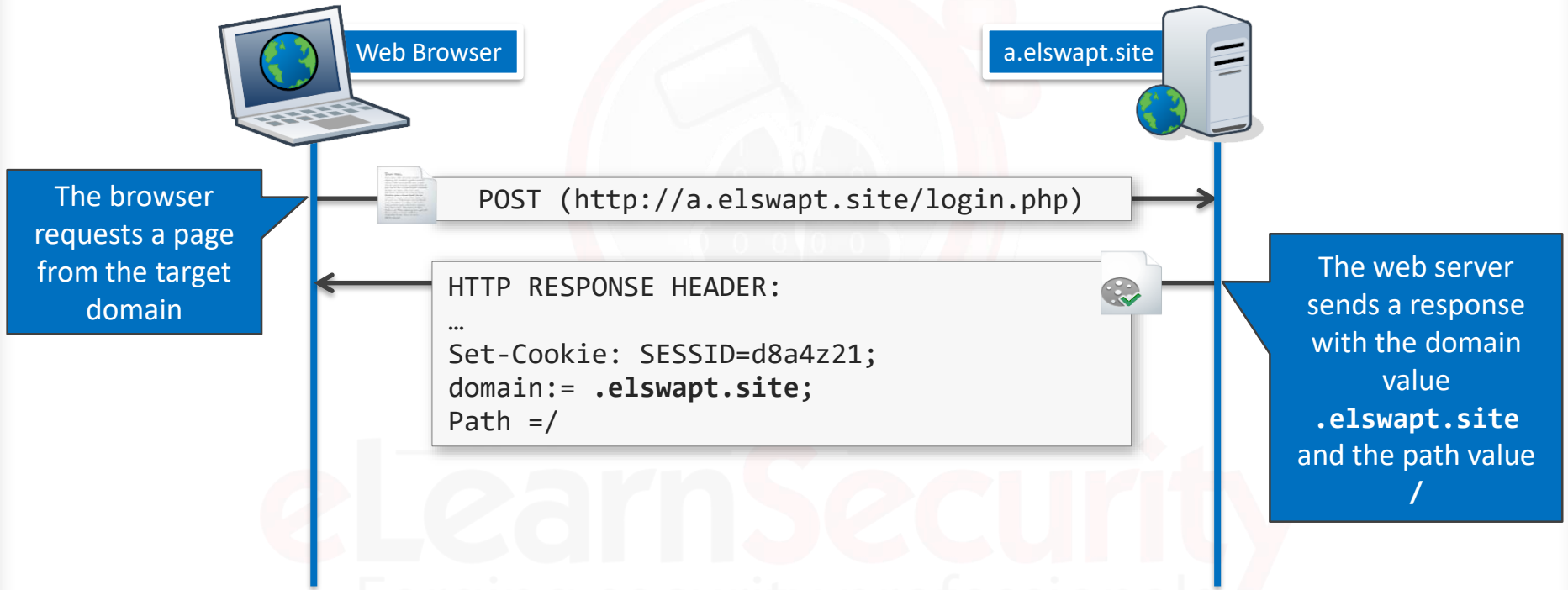
2.4.3.1. Correct Cookie Installation

Example #2



2.4.3.1. Correct Cookie Installation

Example #2



Example #2

The cookie is accepted because the domain value `.elswapt.site` is a suffix of the domain emitting the cookie, `a.elswapt.site`, therefore it will be accepted and sent in each request matching the following URLs:

- http://elswapt.site/*
- https://elswapt.site/*
- http://*.elswapt.site/*
- https://*.elswapt.site/*

Example #2

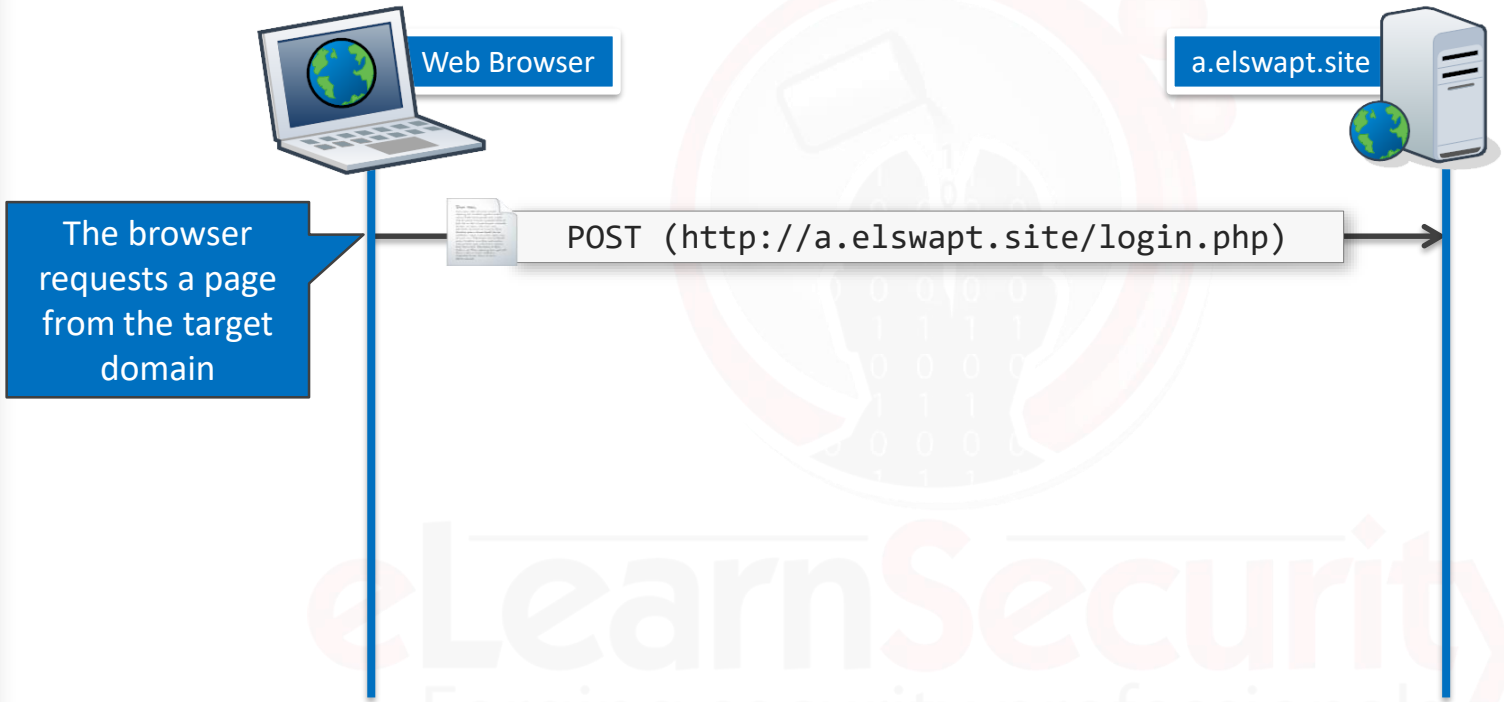
This is what will happen. The cookie previously set is sent to both **a** and **b** subdomains.

Example #3

The browser requests a page from the target domain `a.elswapt.site` and the web server sends a response including both a cookie without a domain value and the path value of `/learning`.

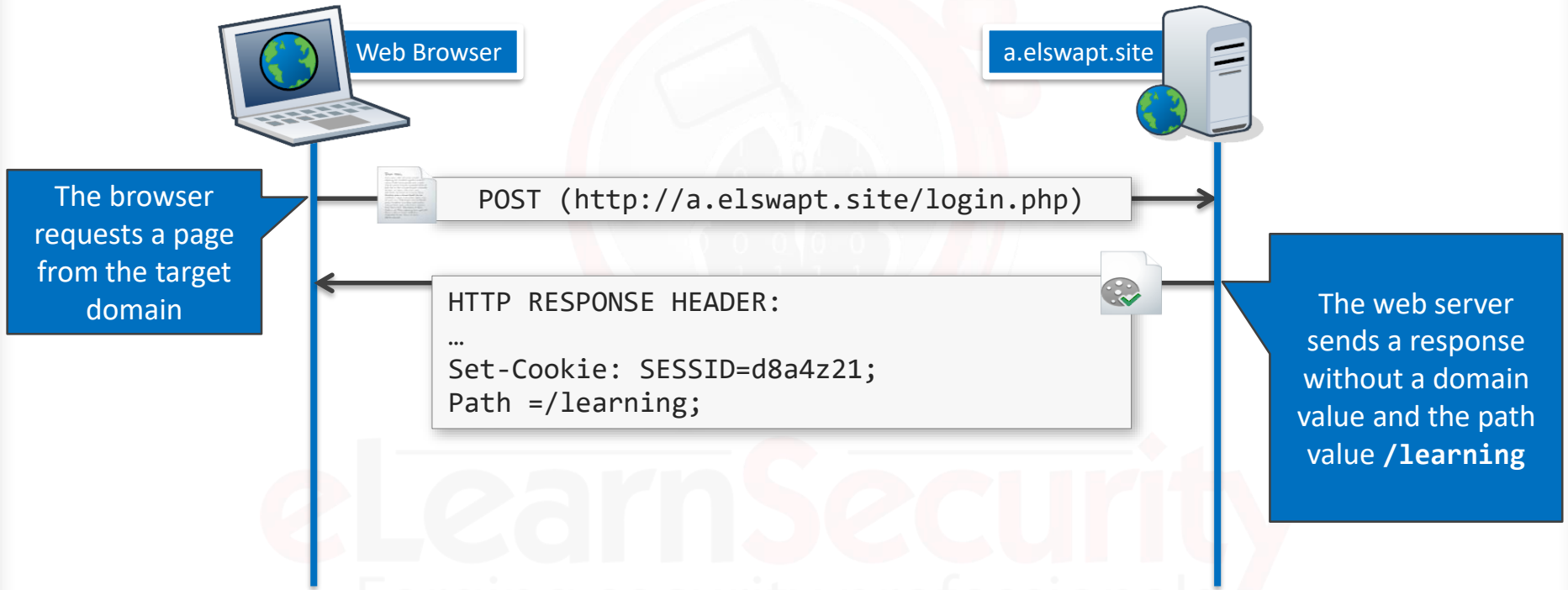
2.4.3.1. Correct Cookie Installation

Example #3



2.4.3.1. Correct Cookie Installation

Example #3



Example #3

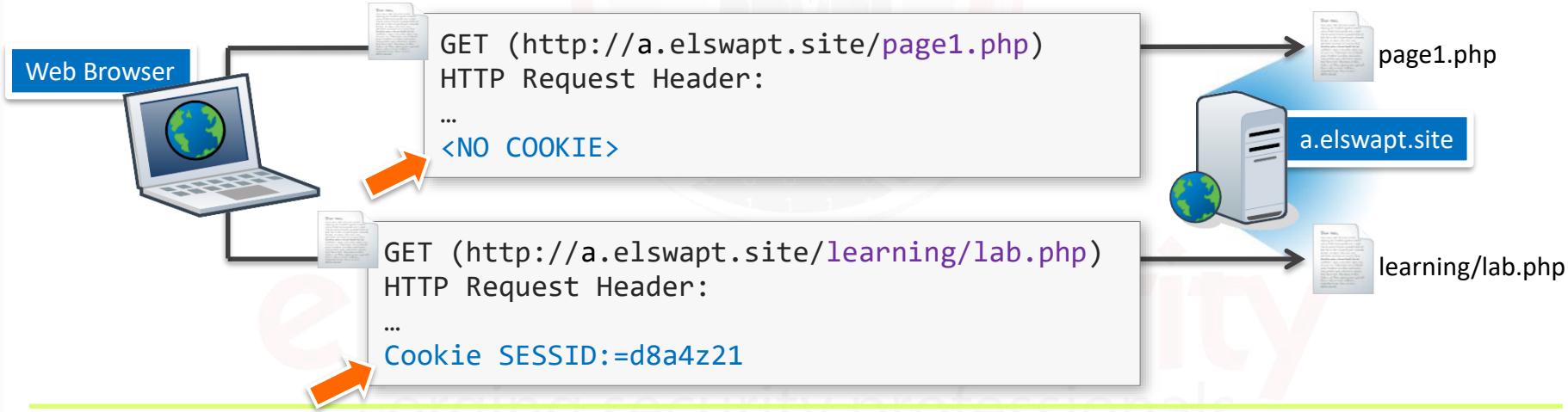
The cookie is accepted and will be available only to the target domain `a.elswapt.site` and path `/learning/*`.

So, this cookie will be sent in each request matching the following URLs:

- http://a.elswapt.site/learning/*
- https://a.elswapt.site/learning/*

Example #3

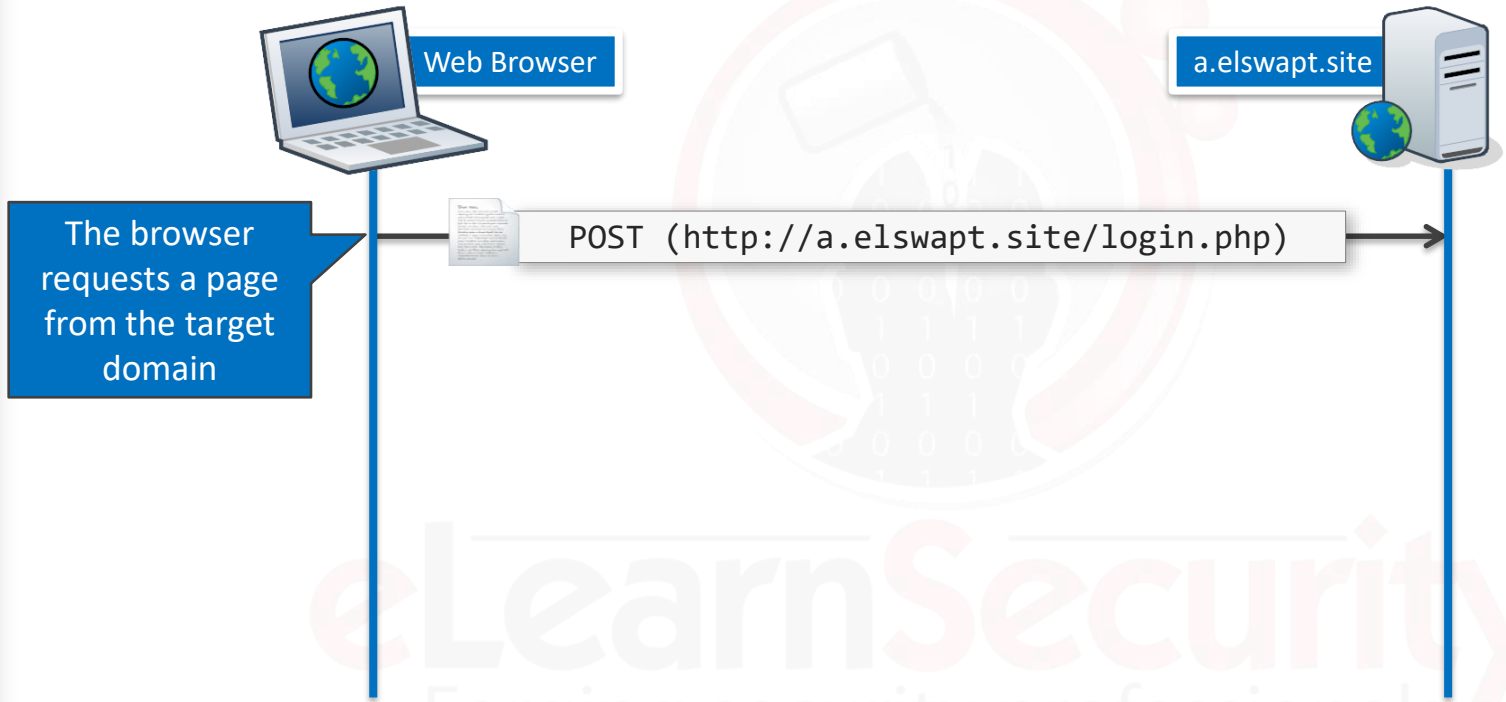
This is what will happen. The cookie will be sent for resources in the `/learning/` path.



Example #4

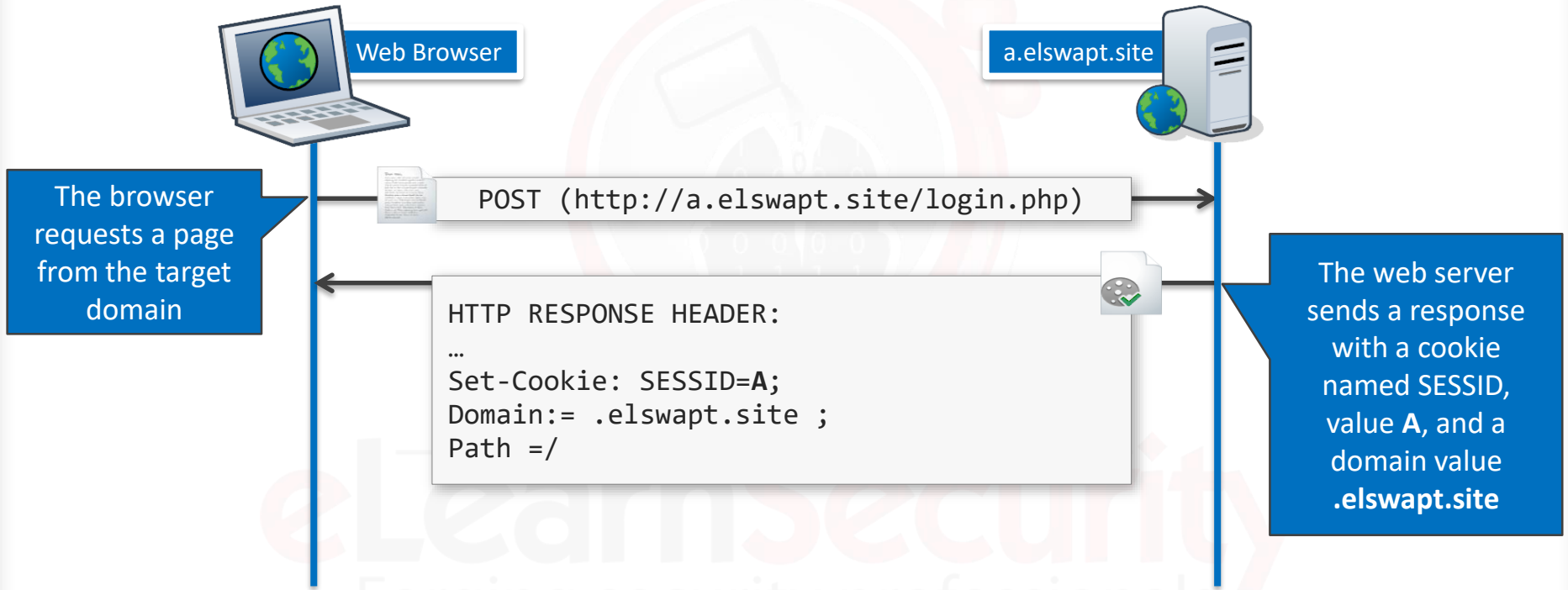
The browser requests a page from the target domain `a.elswapt.site` and the web server sends a response, including a cookie named `SESSID`, value `A`, and a domain value `.elswapt.site`.

Example #4



2.4.3.1. Correct Cookie Installation

Example #4



Example #4

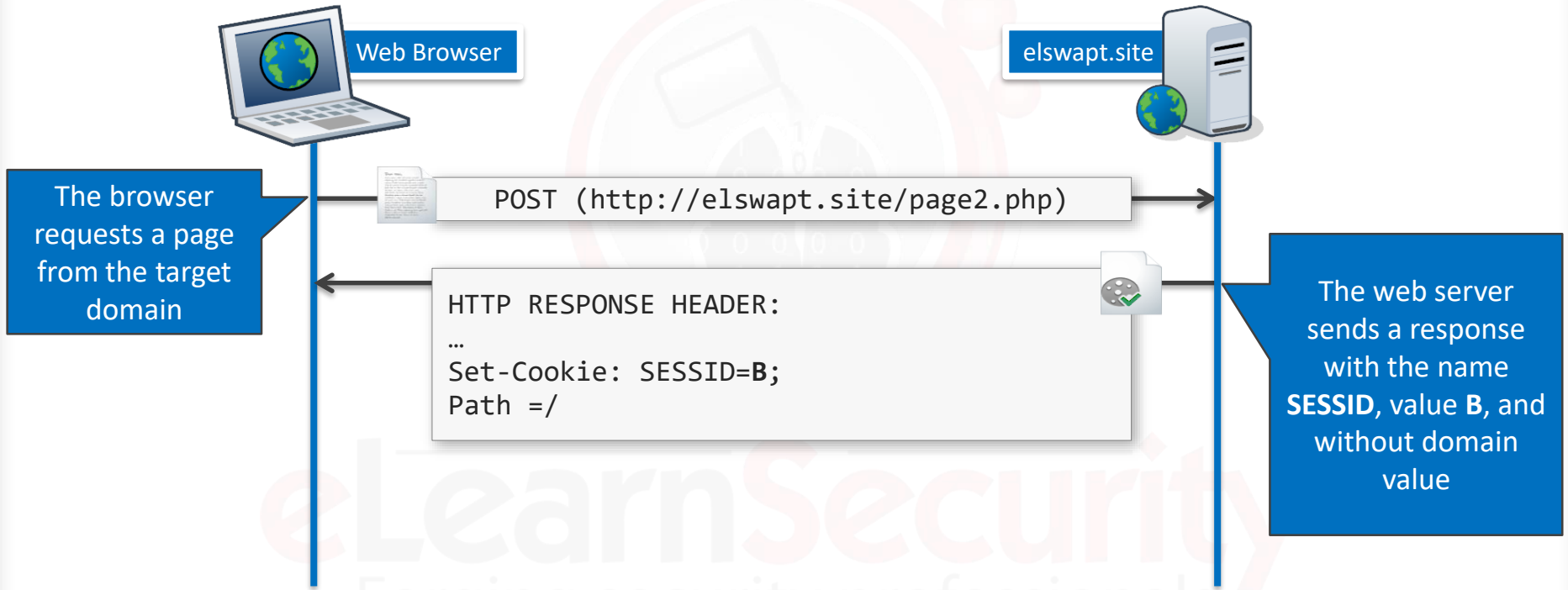
After that, the browser requests a second page from the target domain `elswapt.site` and the web server sends a response including a cookie with the name `SESSID`, value `B`, and without domain value.

2.4.3.1. Correct Cookie Installation

Example #4

2.4.3.1. Correct Cookie Installation

Example #4



Example #4

Both cookies will be accepted and stored by the browser. They will not interfere with one another as they are two different cookies.

2.4.3.2. Incorrect Cookie Installation

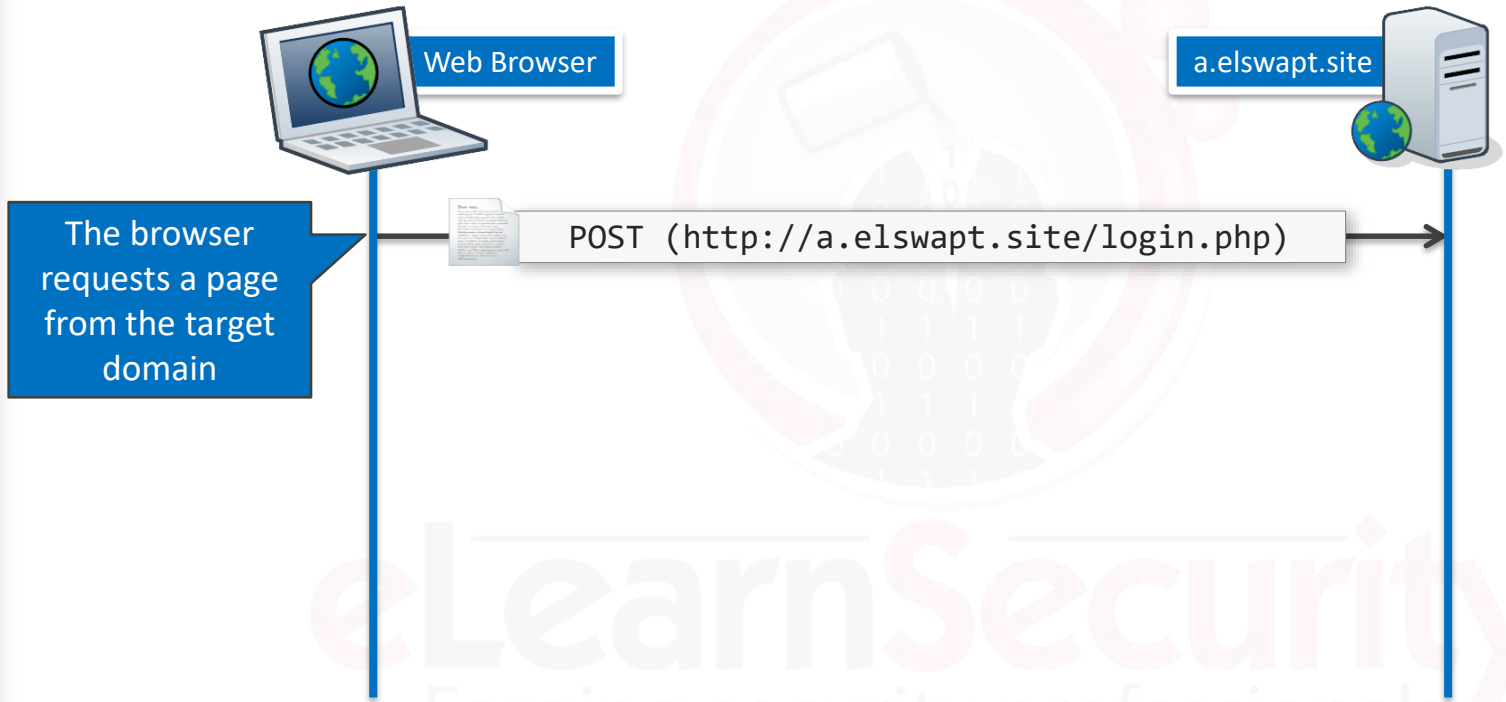
Let us see some examples where the cookie sent by the web server is not accepted by the browser.

Example #1

The browser requests a page from the target domain `a.elswapt.site` and the web server sends a response including a cookie with domain value `b.elswapt.test`.

2.4.3.2. Incorrect Cookie Installation

Example #1

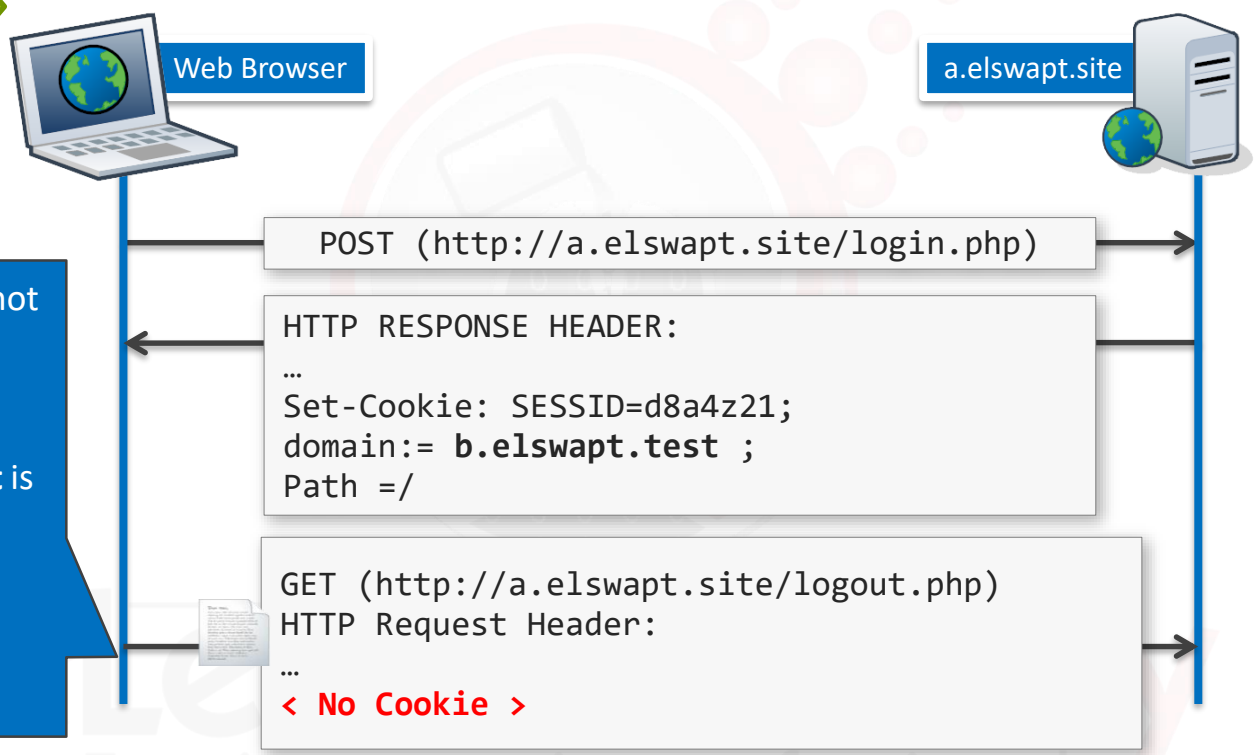


2.4.3.2. Incorrect Cookie Installation

Example #1

2.4.3.2. Incorrect Cookie Installation

Example #1



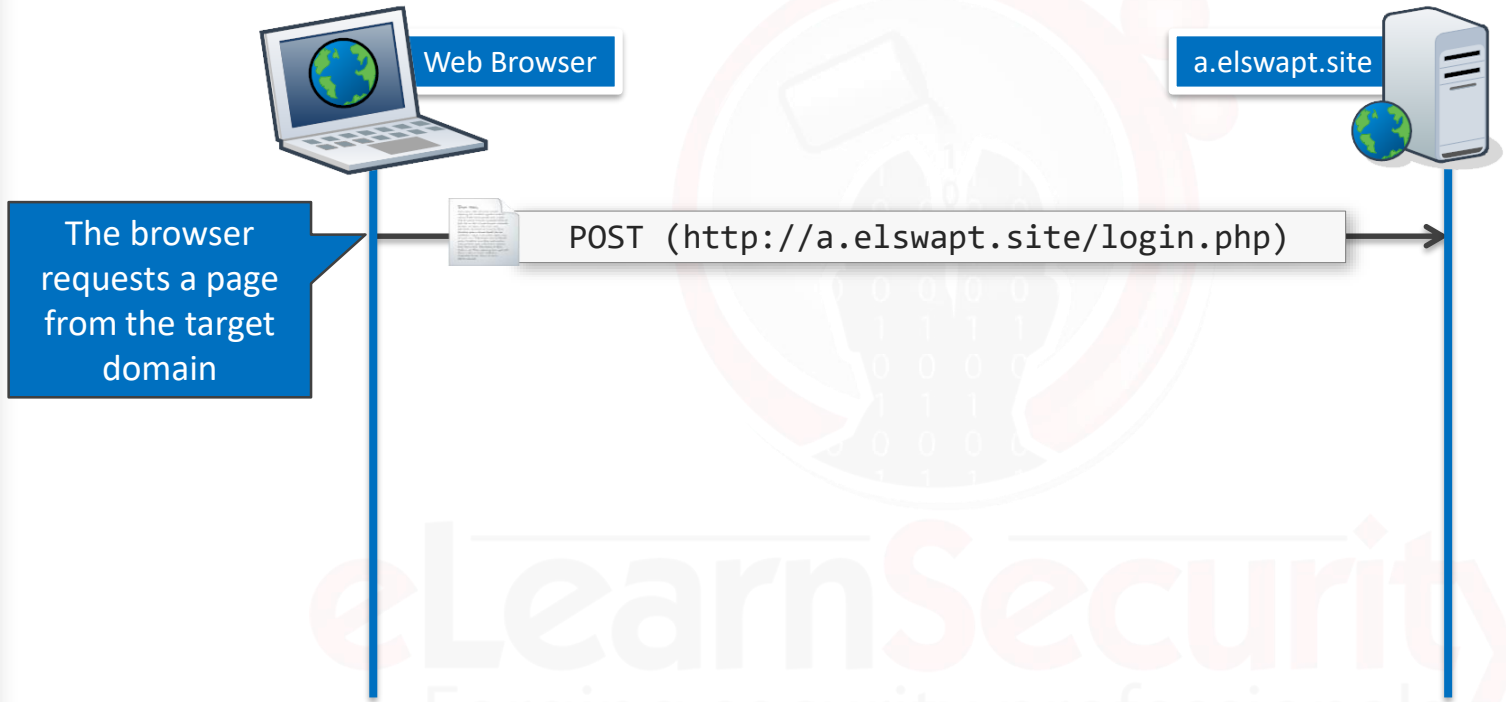
The cookie is not accepted because the domain **b.elswapt.test** is not a suffix of the domain **a.elswapt.site** that sent the cookie.

Example #2

The browser requests a page from the target domain `a.elswapt.site` and the web server sends a response including a cookie with domain value `b.elswapt.site` .

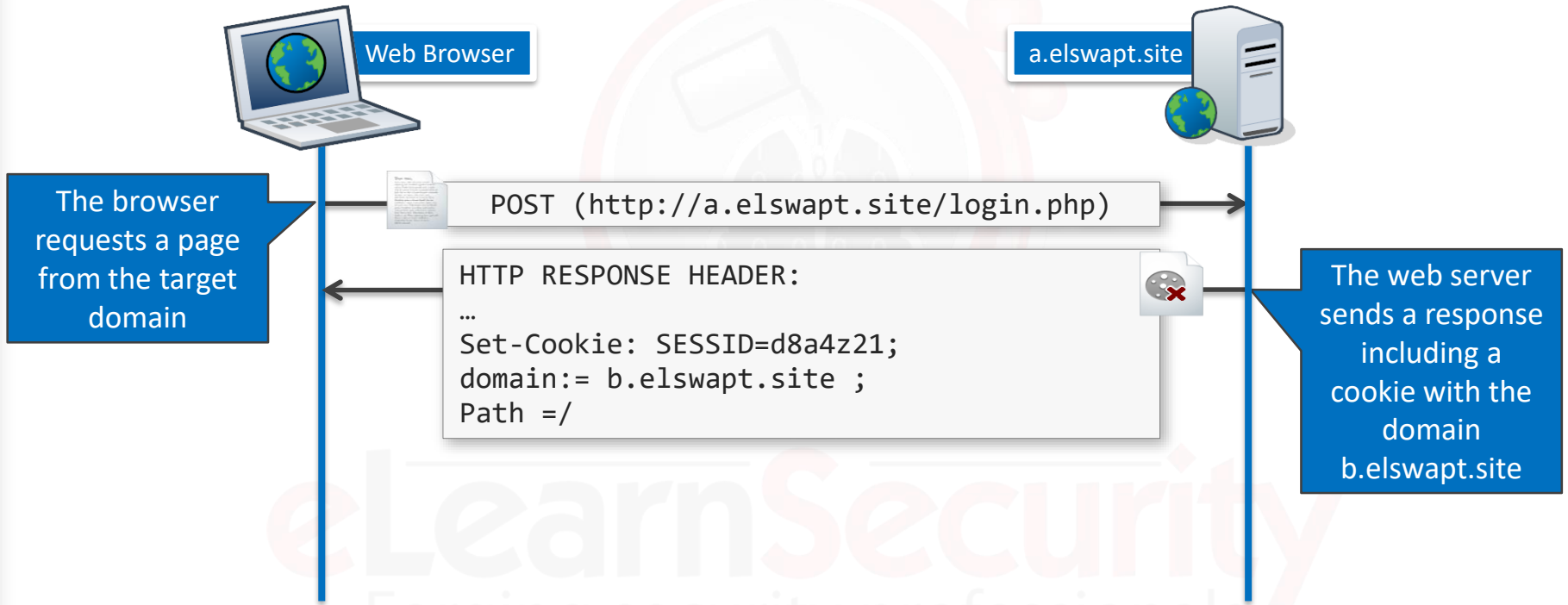
2.4.3.2. Incorrect Cookie Installation

Example #2



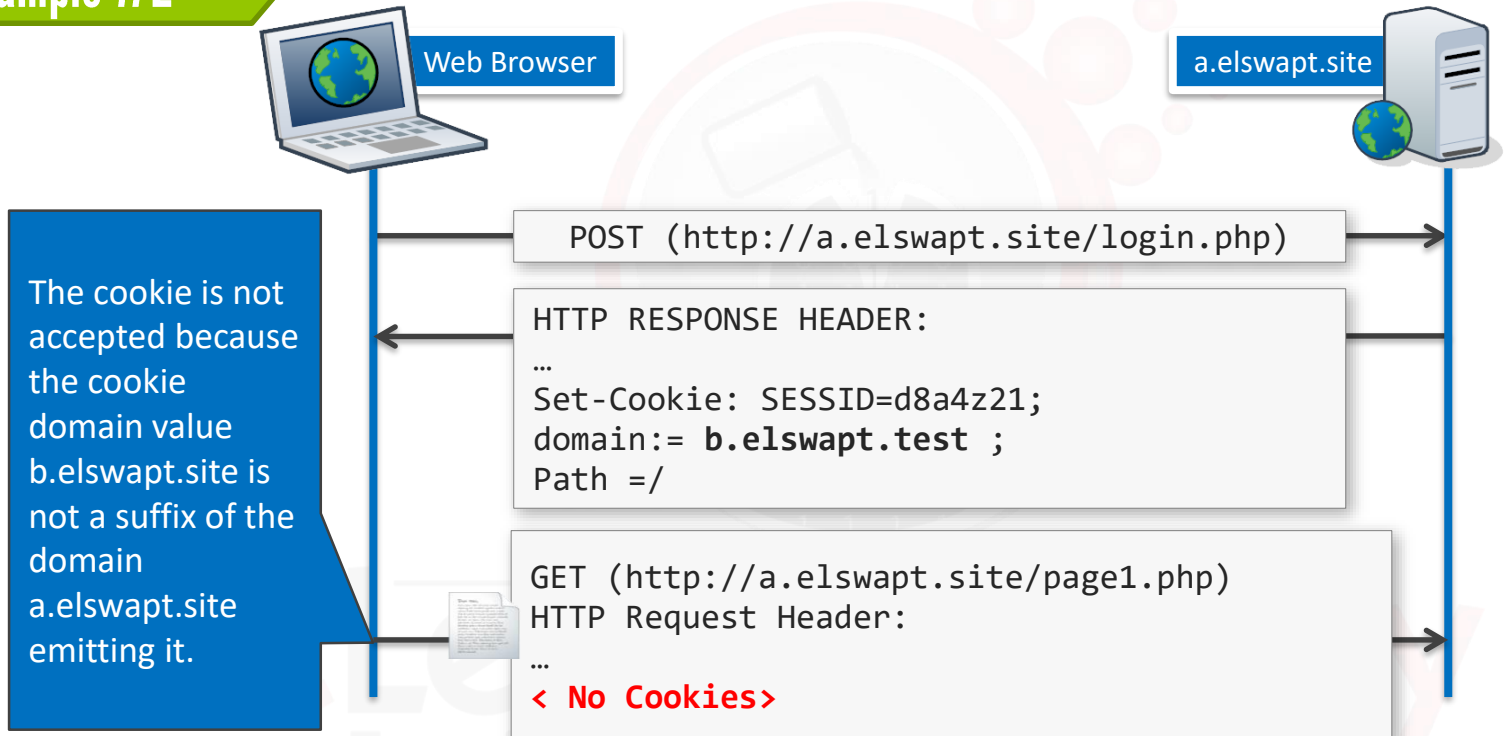
2.4.3.2. Incorrect Cookie Installation

Example #2



2.4.3.2. Incorrect Cookie Installation

Example #2



Session

2.5. Session

Sometimes the web developer prefers to store information on the **server side** instead of the client side.

This happens in order to hide the application logic or just to avoid the back and forth data transmission, which is typical behavior of cookies.

HTTP sessions are a simple mechanism that allows websites to store variables specific for a given visit on the **server side**.

Each user session is identified by a either a session id or token, which the server assigns to the client.

The main difference between cookies and session variables is that cookies are stored on the client whereas session variables are on the server.

Also, session variables expire with the session and sessions usually expire sooner than cookies do.

The session mechanism works through the use of a session token (or session ID).

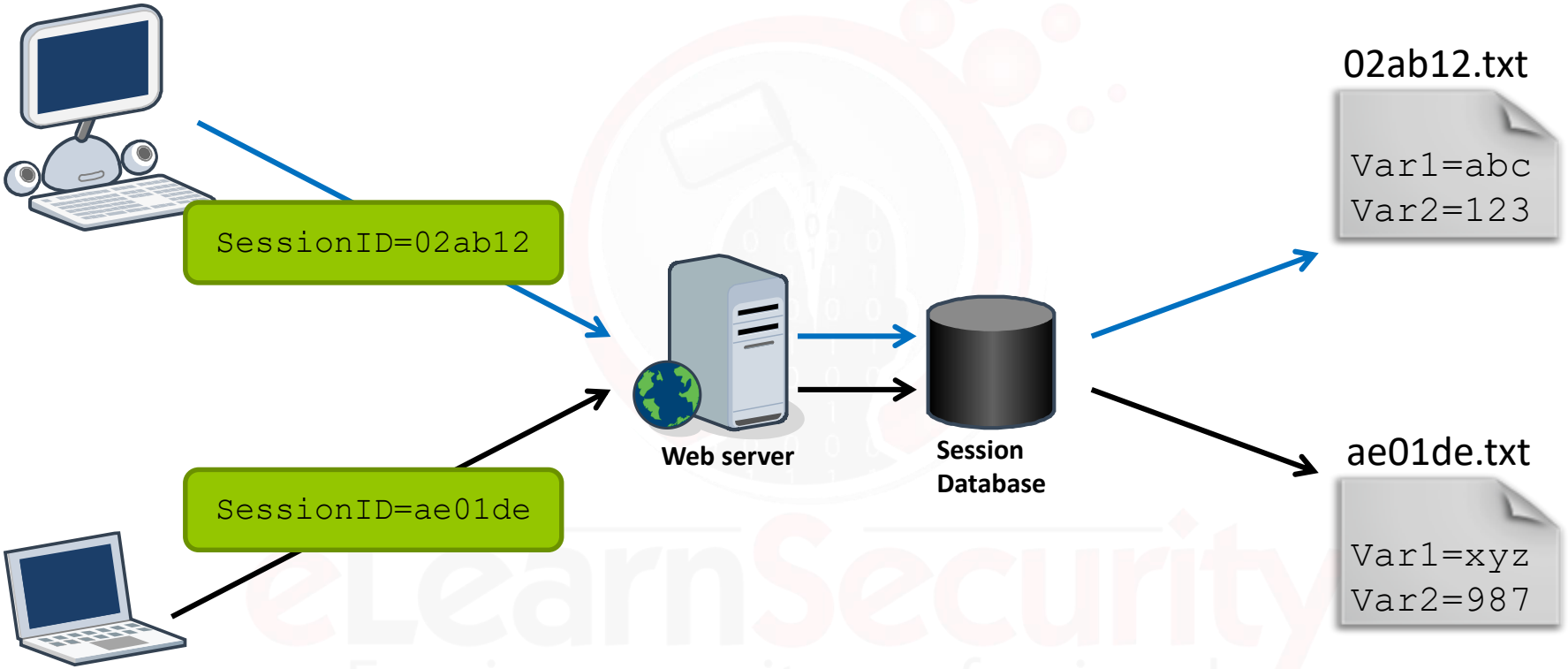
The session ID is assigned to the client by the webserver and the client will present this ID for each subsequent request in order to be recognized.

The session ID provided by the client will let the server retrieve both the state of the client and all of its associated variables.

You can picture the session ID like primary key in a relational database where clients' variables are stored.

Session IDs can be stored within text files, databases or memory on the server.

2.5. Session



Session cookies just contain a single parameter formatted in a key value pair. See the example below:

```
SESSION=0wvCtOBWDH8w
PHPSESSID=l3Kn5Z6Uo4pH
JSESSIONID=W7DPUBgh7kTM
```

Websites running PHP, typically install session cookies by using the "`PHPSESSID`" parameter name, while JSP websites typically use "`JSESSIONID`". Each development language has its own default session parameter name and typically allow for developers to customize its functionality (IE: Changing PHPSESSID to PSESSID).

If needed, servers install session cookies after a browser performs some type of activity, such as:

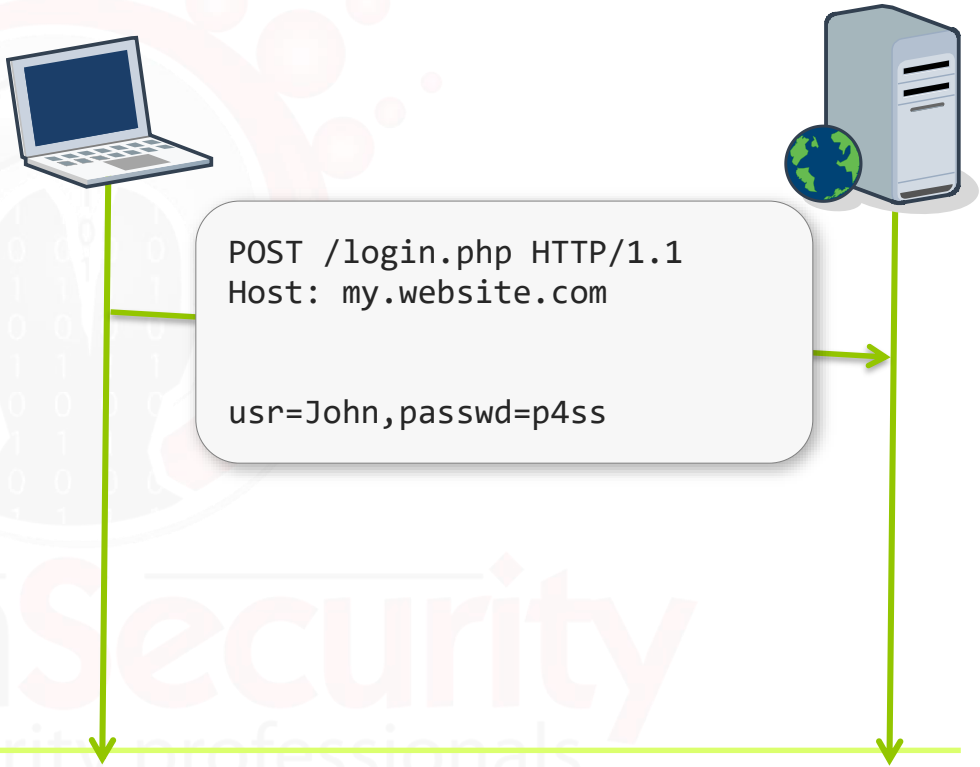
- Opening a specific page.
- Changing settings in the web application.
- Logging in.

Then the browser uses the cookie in subsequent requests.

A session could contain many variables, so sending a small cookie keeps the bandwidth usage low.

In the following example you can see a session cookie in action.

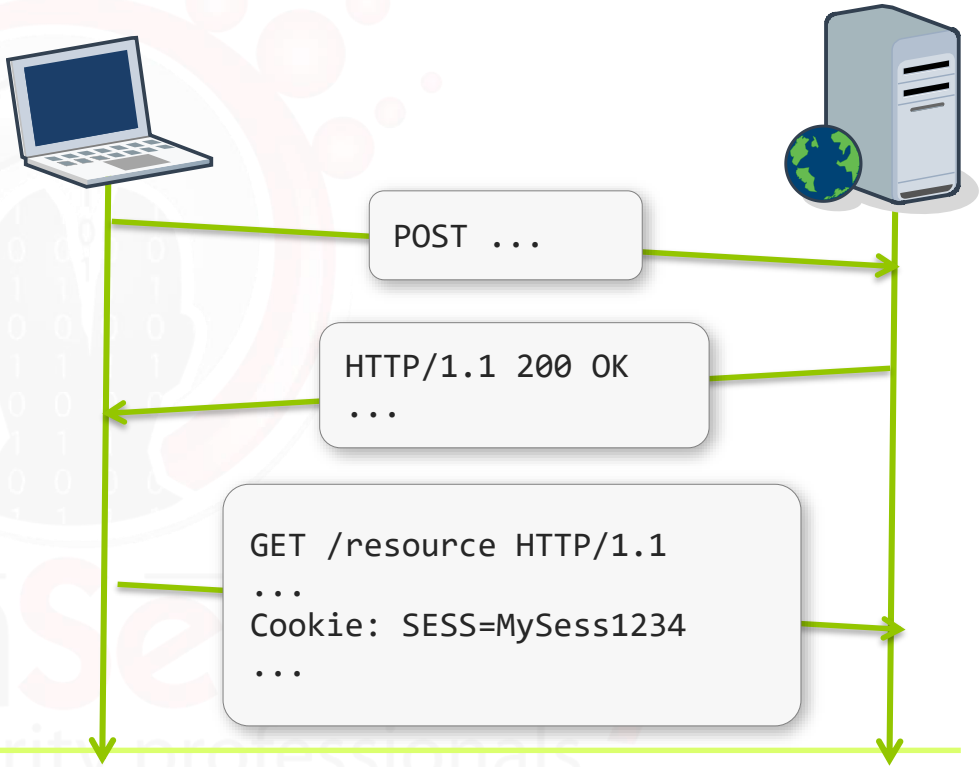
The client uses a login form to POST the user's credentials.



The server sends back a response with a Set-cookie header field.

The cookie contains the **session ID**.

The browser will send back the cookie according to the cookie protocol, thus sending the **session ID**.



Since the web browser has a cookie in its jar, any subsequent request will carry the (session) cookie with it.

As an alternative to session cookies, session ID's can also be sent via the GET method appended to the requesting URL.

```
http://example.site/resource.php?sessid=k27rds7h8w
```

In the following video you will see how the cookie protocol works:

- Installation of cookies
- Manipulation of cookies
- Dissecting a cookie

Moreover you will see how HTTP sessions work and how they use cookies.

If you have a FULL or ELITE plan you can click on the image on the left to start the video

Web Application Proxies

2.6. Web Application Proxies

Most web applications contains a great deal of objects like scripts, images, style sheets, client and server side intelligence.

Having **tools** that help in the **study** and **analysis** of web application behavior is critical.

An **intercepting proxy** is a tool that lets you analyze and modify any request and any response exchanged between an HTTP client and a server.

By intercepting HTTP messages a pentester can study a web application behavior and manually test for vulnerabilities.

2.6. Web Application Proxies

The most used web application proxies are:

- Burp Suite (the intercepting proxy feature).
- ZAP.

Proxies are fundamental while analyzing web applications and will become your best friends for web-app testing.

http://portswigger.net/burp/

https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

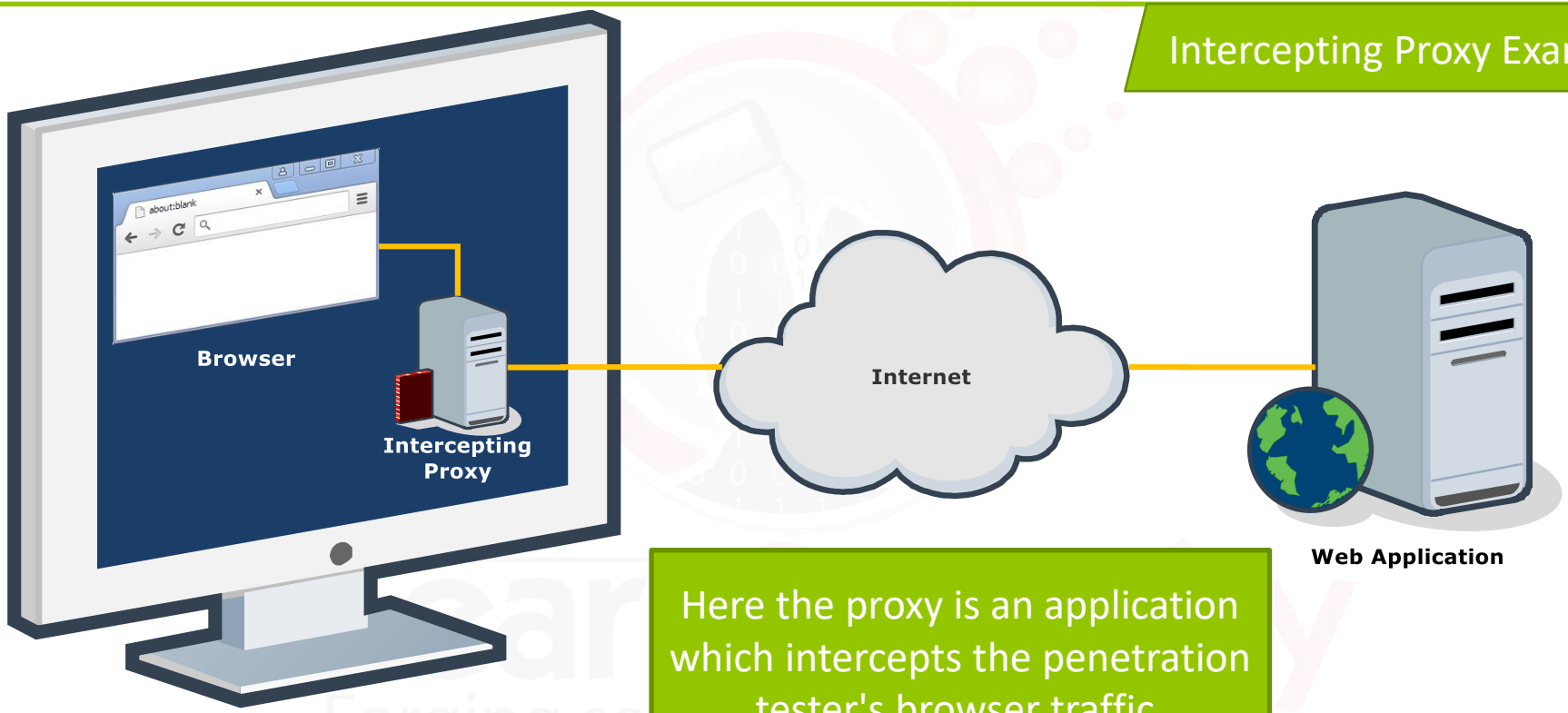
Do not confuse intercepting proxies with common web proxy servers like Squid.

Proxy servers have different purposes: bandwidth optimization, content filtering and more.

The next two images will help make that clearer.

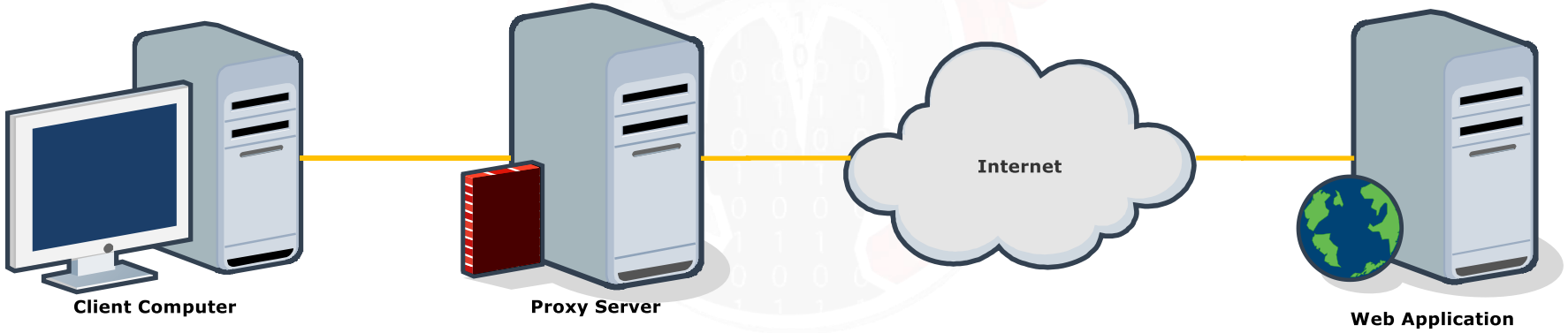
http://www.squid-cache.org/

Intercepting Proxy Example



Here the proxy is an application which intercepts the penetration tester's browser traffic.

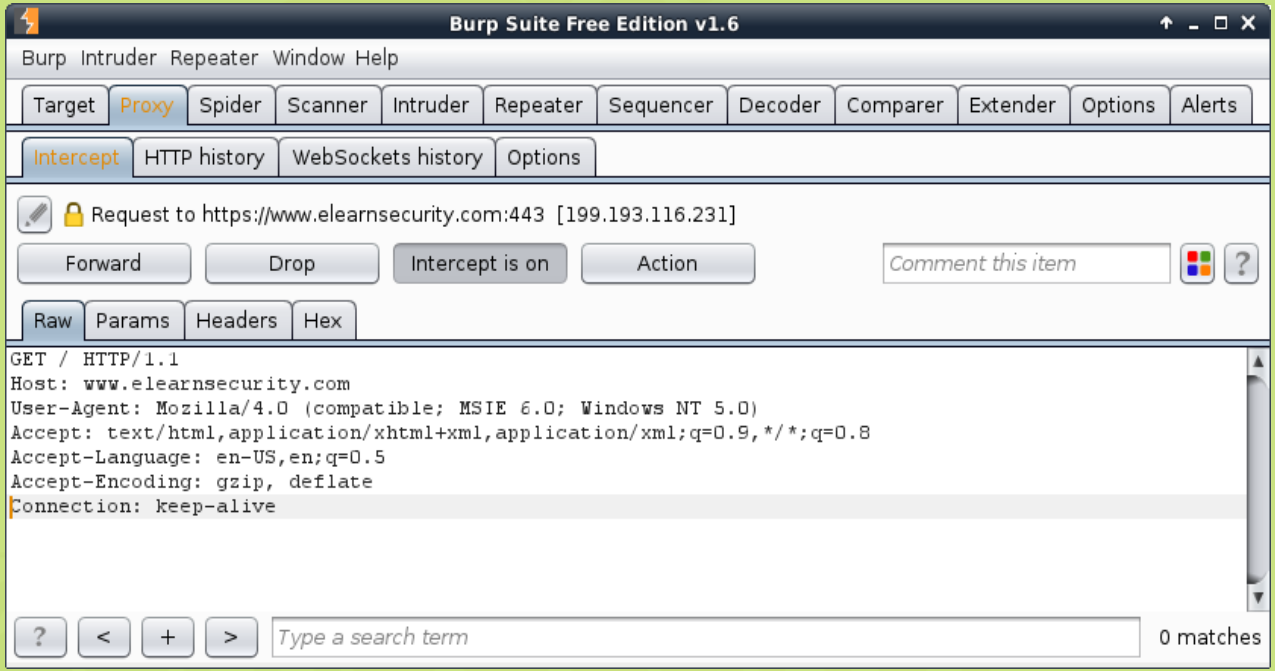
Proxy Server Example



Here the proxy server filters all the traffic coming from the internal network.

Burp suite offers one of the best proxies available.

You can download the Free Edition here.



http://portswigger.net/burp/download.html

Burp suite will let you:

- Intercept requests and responses between your browser and the web server.
- Build requests manually.
- Crawl a website, by automatically visiting every page in a website.
- Fuzz web applications, by sending patterns of valid and invalid inputs to test their behavior.

By using Burp, you can **intercept and modify** requests coming from your browsers **before** they are sent to the remote server.

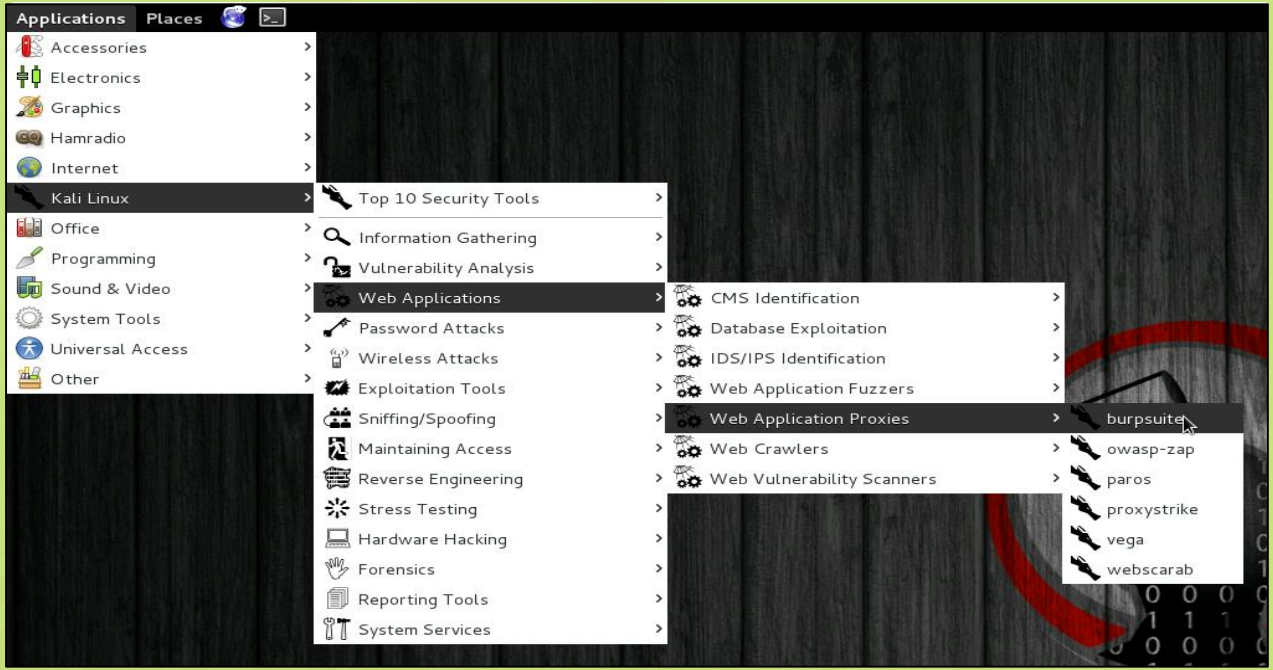
You can modify the **header** and the **body** of a message either **by hand or, automatically**.

In the following slides you will see how to launch, configure and use Burp Suite with your browser.

Attempt to understand all the settings by trying them on your computer!

2.6.1. Burp Suite

Launch Burp Suite: in Kali you will find it under *Kali Linux > Web Applications > Web Application Proxies > burpsuite*.



If you want to run it on another operating system, you can download it from the Portswigger website.

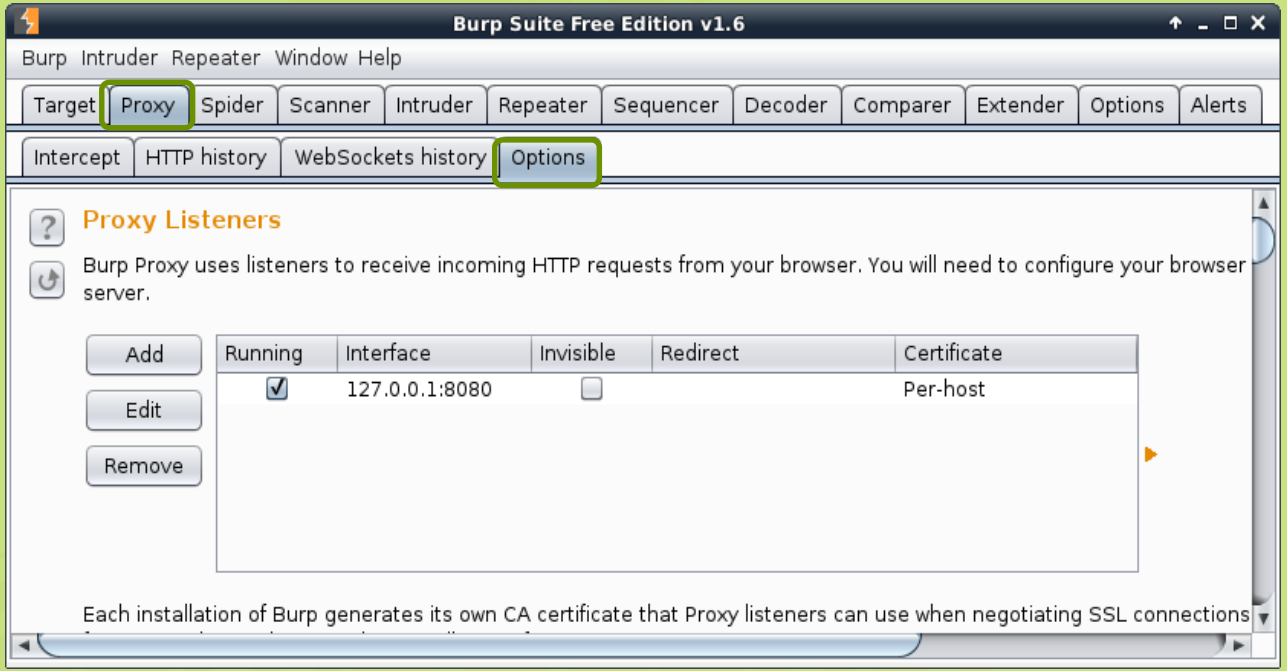
To run Burp you can double click on the jar file you downloaded or run

```
java -jar burpsuite_free_v1.6.jar
```

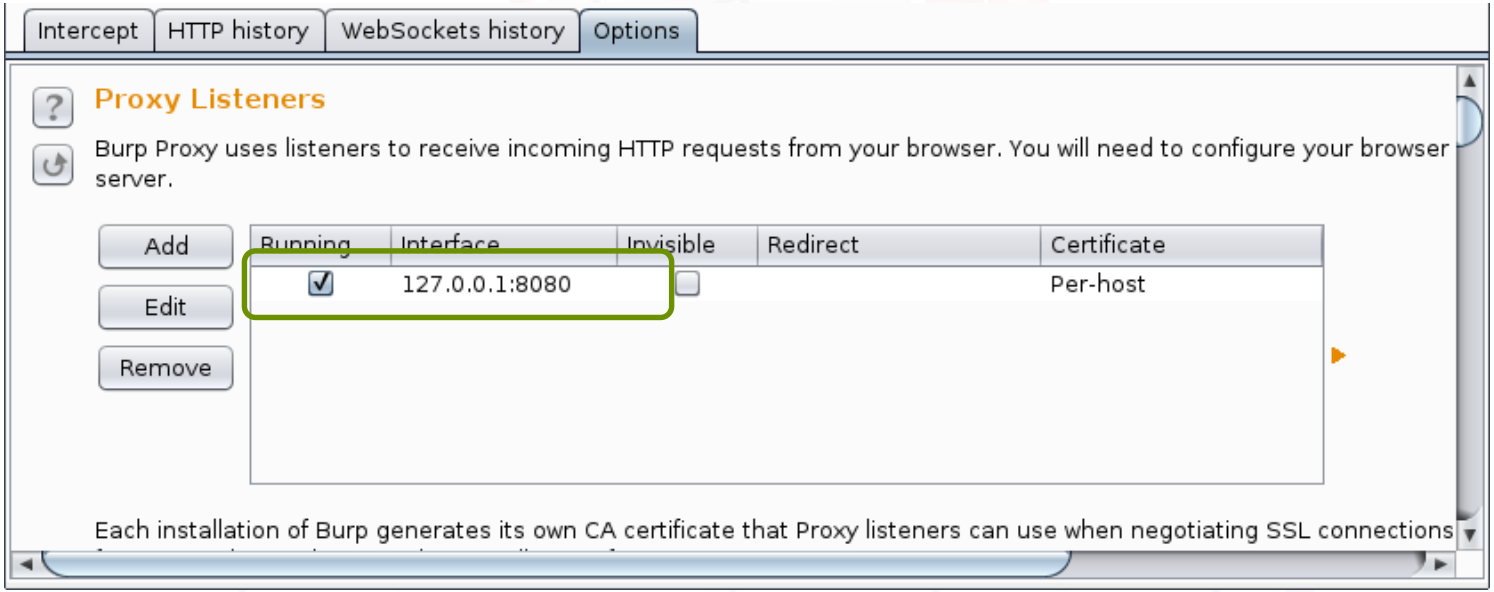
from the console.

http://portswigger.net/Burp/downloadfree.html

Now go to the *Proxy* tab and then to the *Options* sub-tab.



Here you can start and stop the proxy and configure the *host:port* pair on which burp will listen.

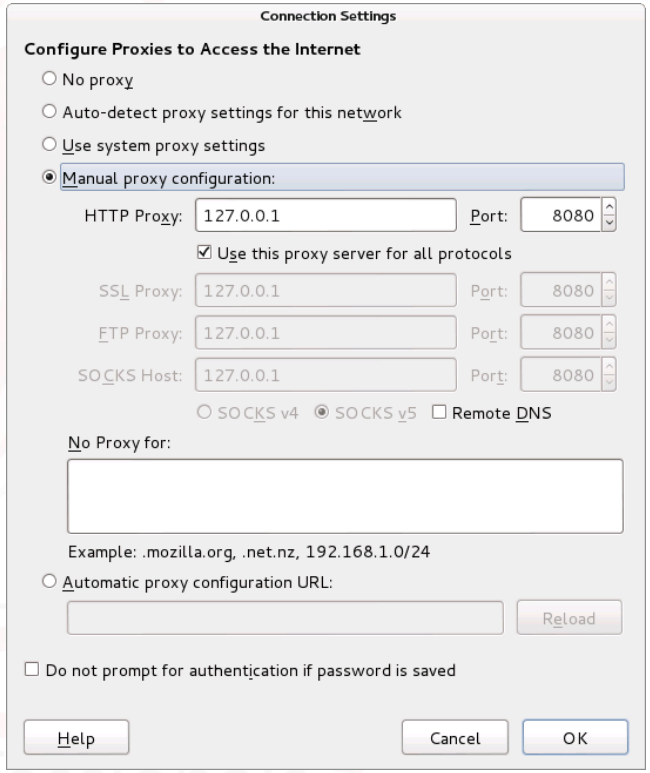


Scrolling down you can find other configuration items to fine tune which messages to intercept, how to automatically change message content and more.

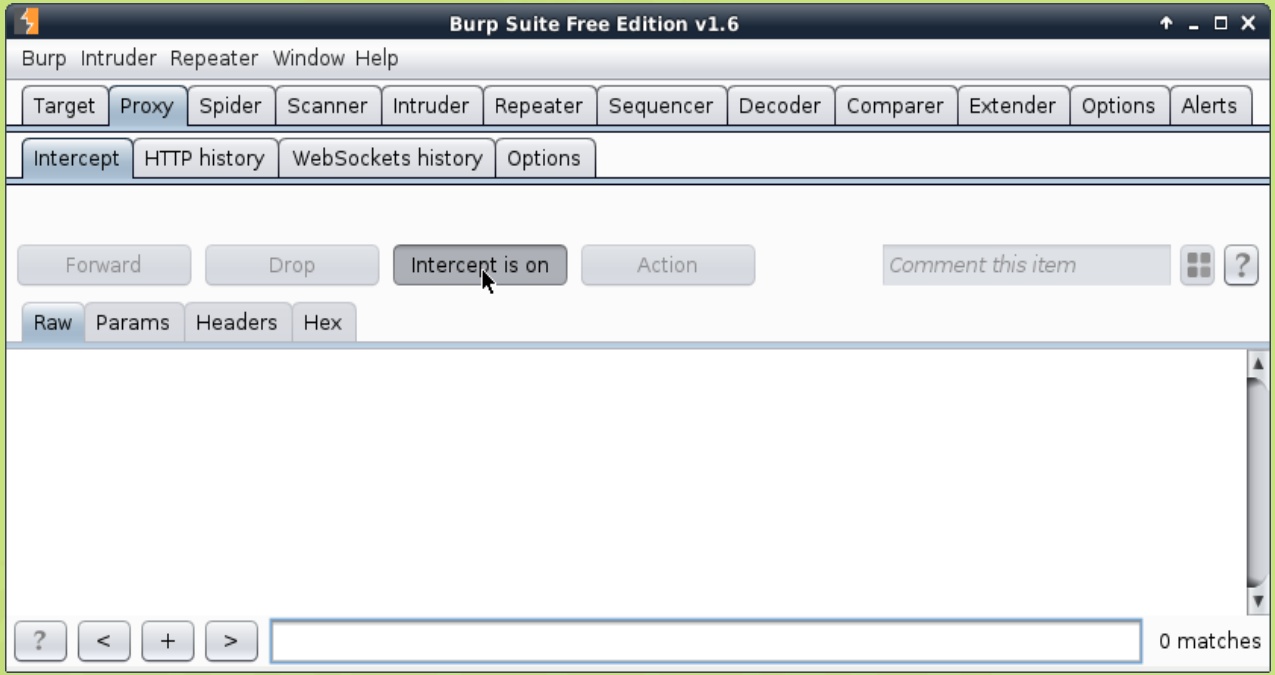
For now, just leave the default options as they are, you will see how to use those features later on.

Once Burp Proxy is configured, you have to configure your browser to use it as the proxy for every protocol.

In Firefox you have to open the *Preferences* window, go to the advanced tab, click on the *Network* sub-tab and finally open the *Connection settings* window.

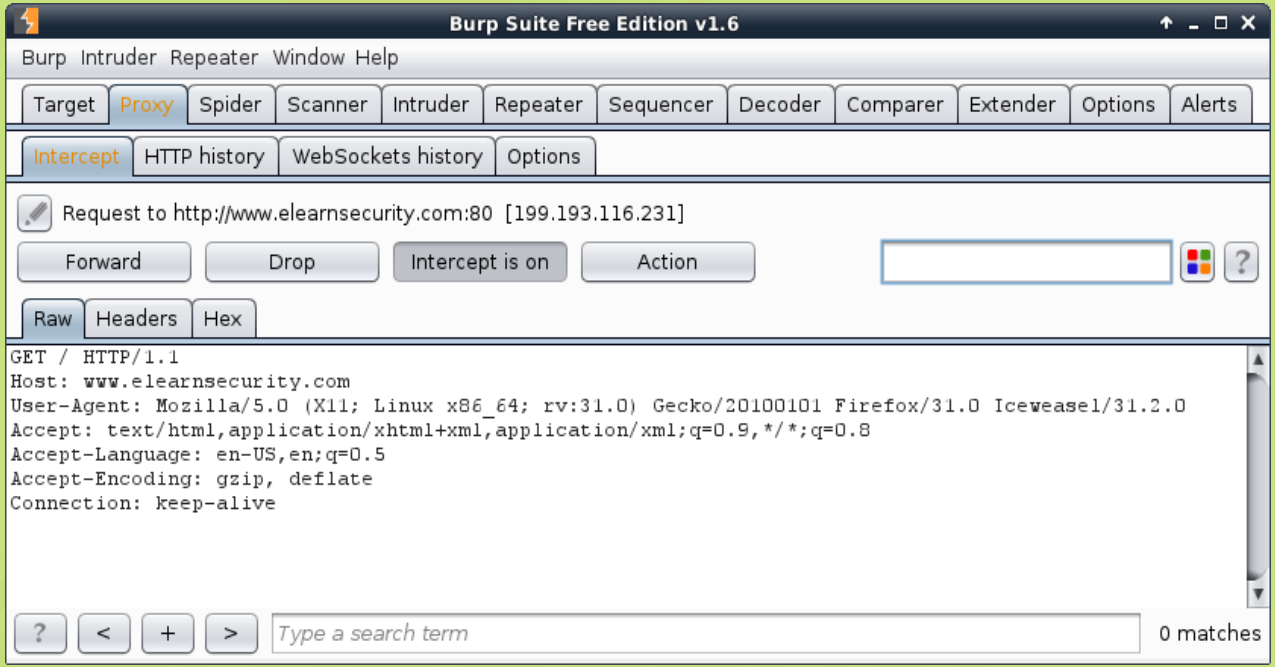


To intercept traffic switch to Burp and go to *Proxy > Intercept* and click on the *Intercept is off* button to enable interception.

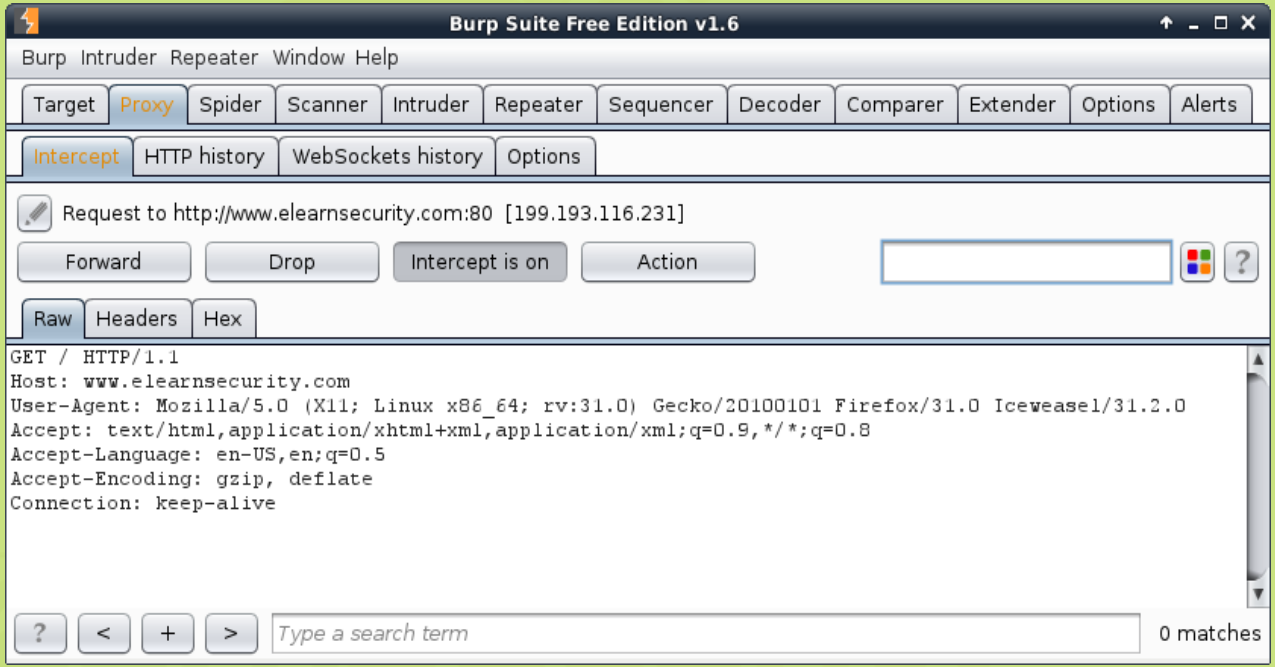


2.6.1. Burp Suite

Now open a website with your browser, Burp will pop up intercepting the request. You can, optionally, modify and then forward it by clicking on *Forward*.

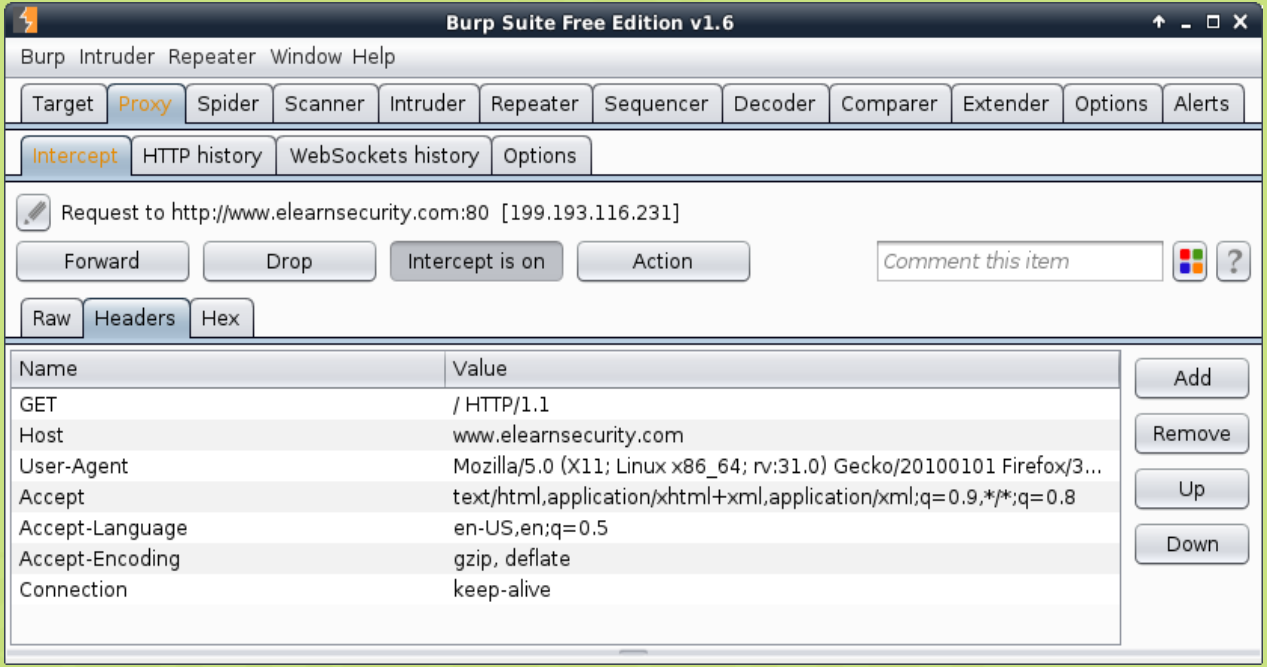


When *Intercept* is on, every browser request stops at Burp Proxy. You can modify the entire request or just its headers.

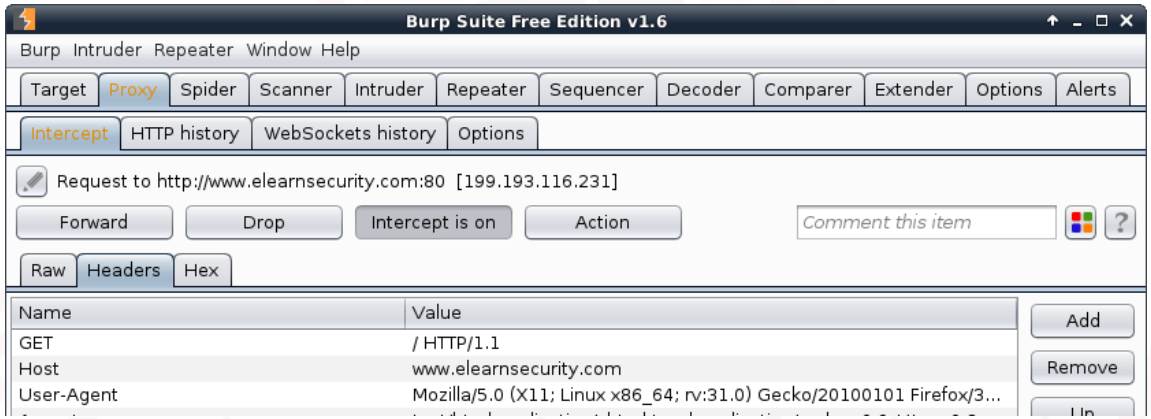


2.6.1. Burp Suite

You can modify the headers both in the R*aw* tab or in the *Headers* tab. Remember to forward the request after editing it!



Raw and **Headers** tab present the very same information with a different format. The *Headers* tab simply presents the headers in a tabular manner.



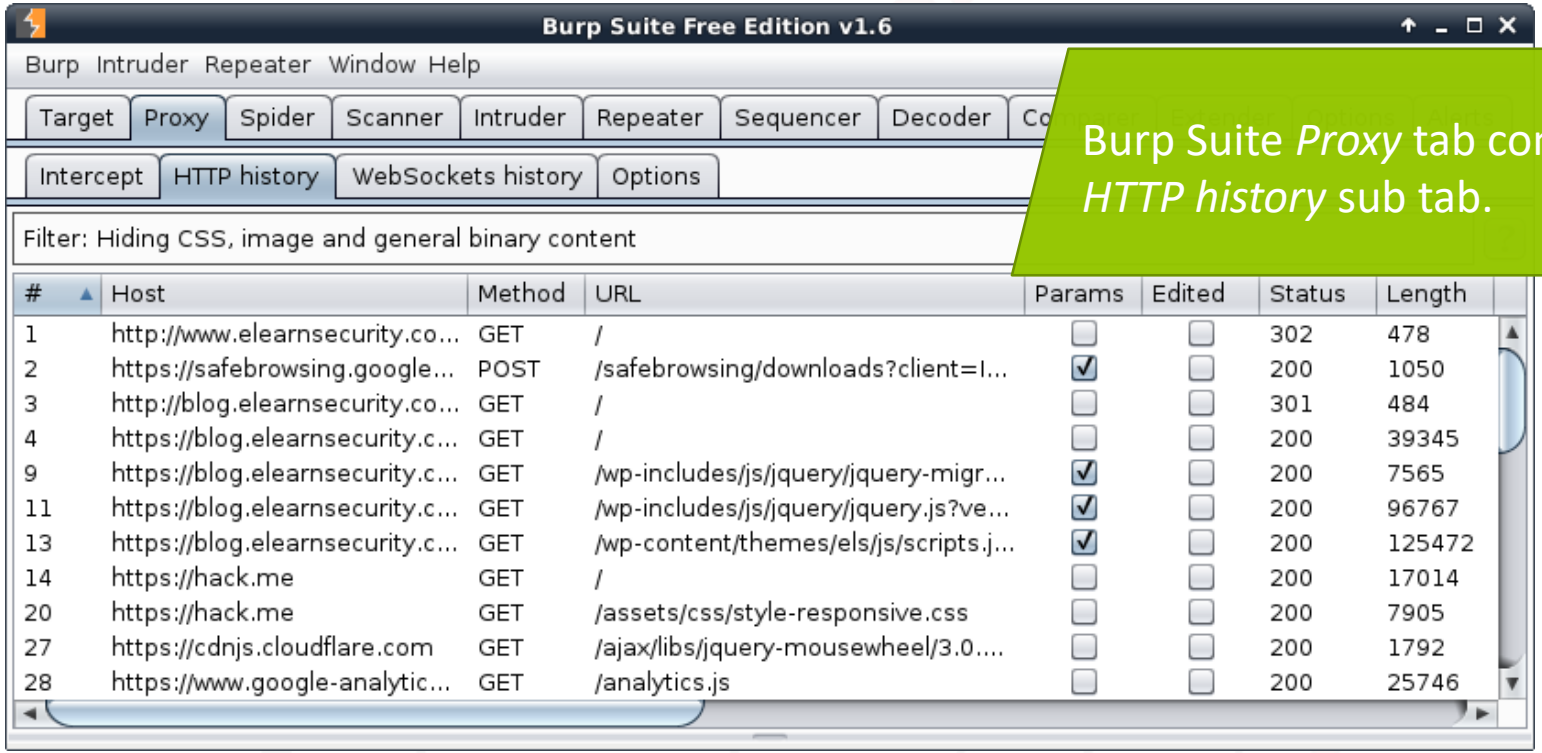
You do not need to manually intercept and forward every request though.

Even if you leave the master interception off, Burp will still collect information on the HTTP traffic coming to and from your browser.

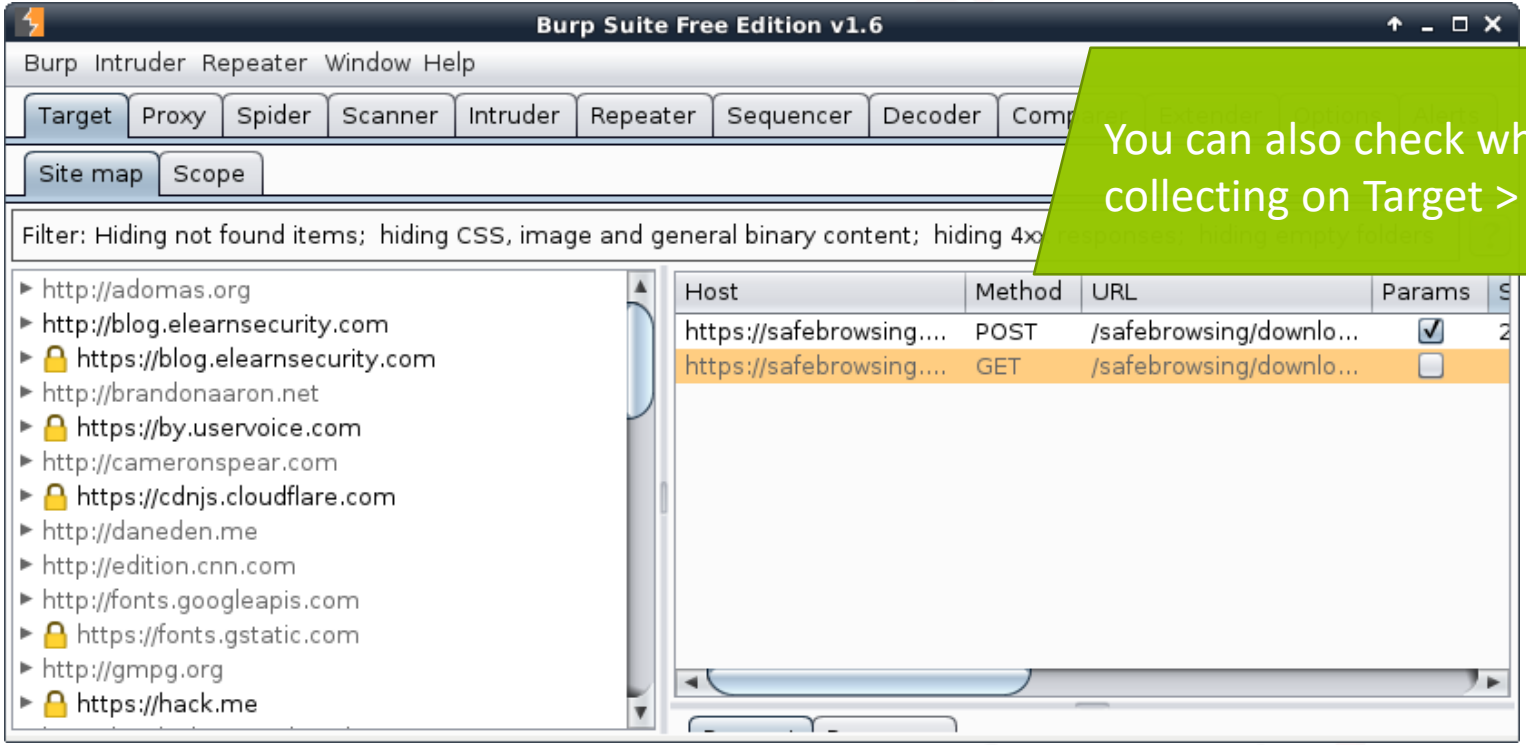
You can check what Burp is collecting in two ways:

- On the *Proxy > History* tab
- In the *Target > Site Map* tab.

2.6.1. Burp Suite



Burp Suite *Proxy* tab contains an *HTTP history* sub tab.



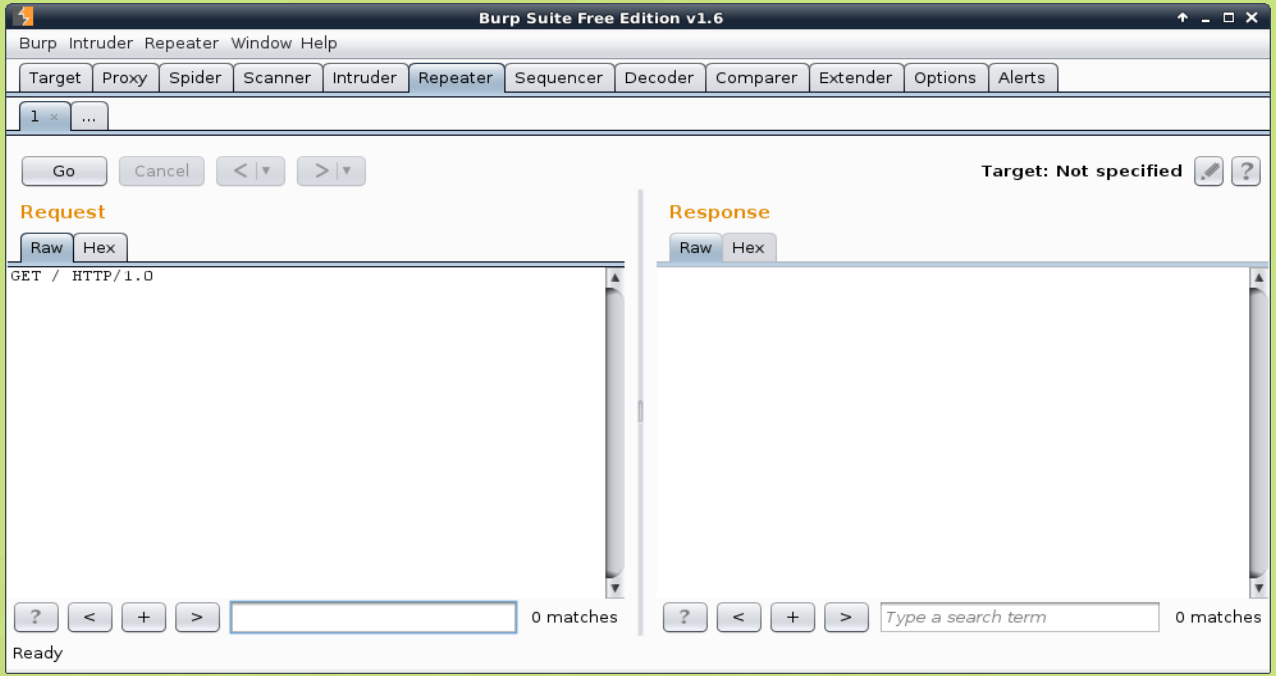
You can also check what Burp is collecting on Target > Site Map

Another feature is **Burp Repeater** which lets you manually build raw HTTP requests.

You can do the same thing by using other tools such as *netcat* or *telnet*, but *Burp* provides you:

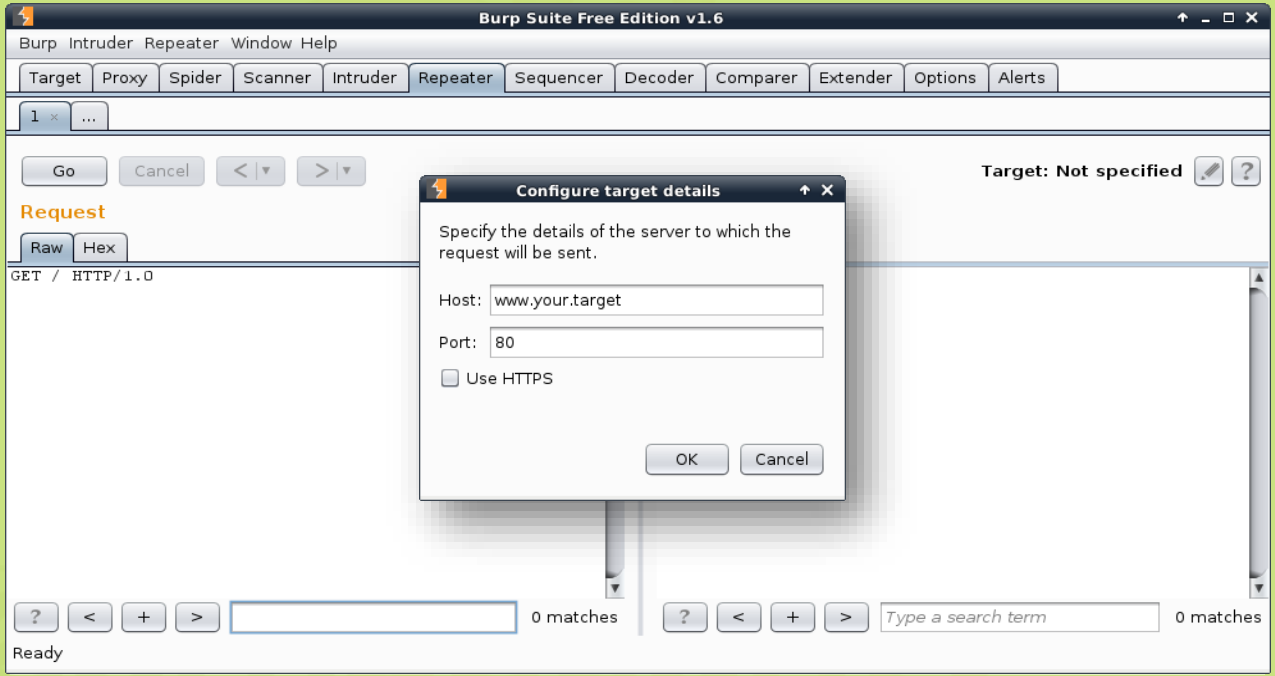
- Syntax highlighting
- Raw and rendered responses
- Integration with other Burp tools

To create a request, first you have to set your target by clicking on the pencil icon in the upper right corner of the tab.



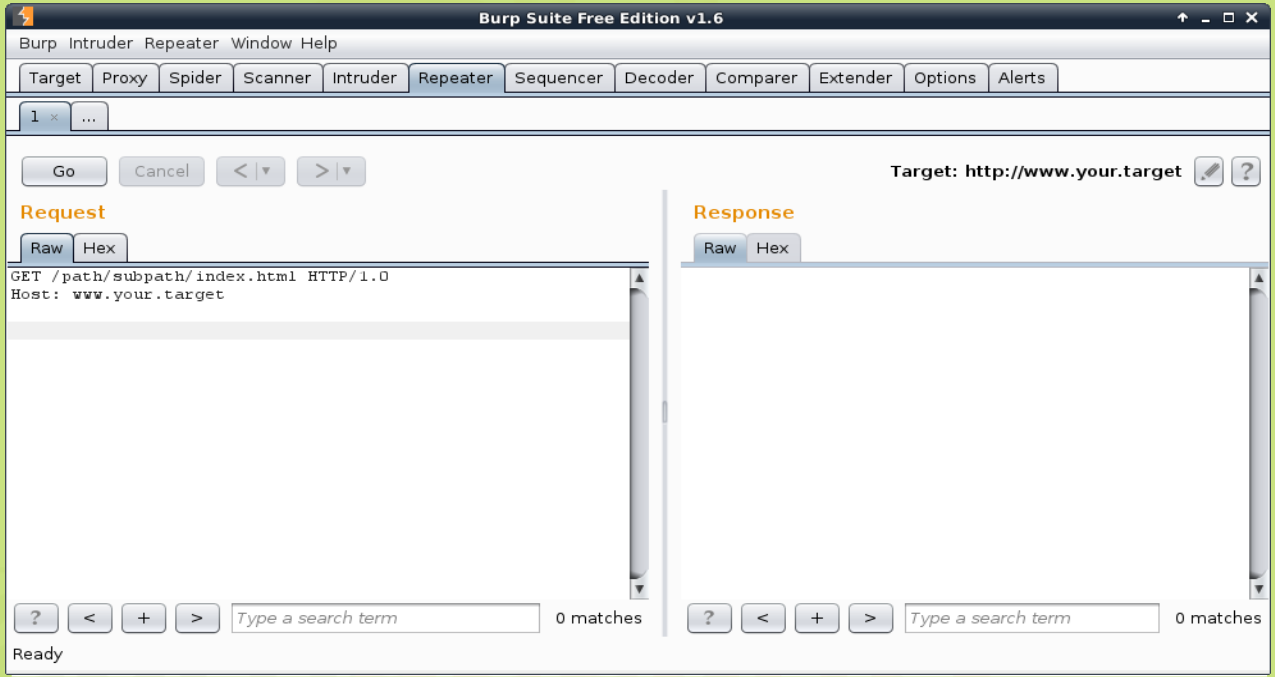
2.6.1. Burp Suite

Then you can set your target host and port.

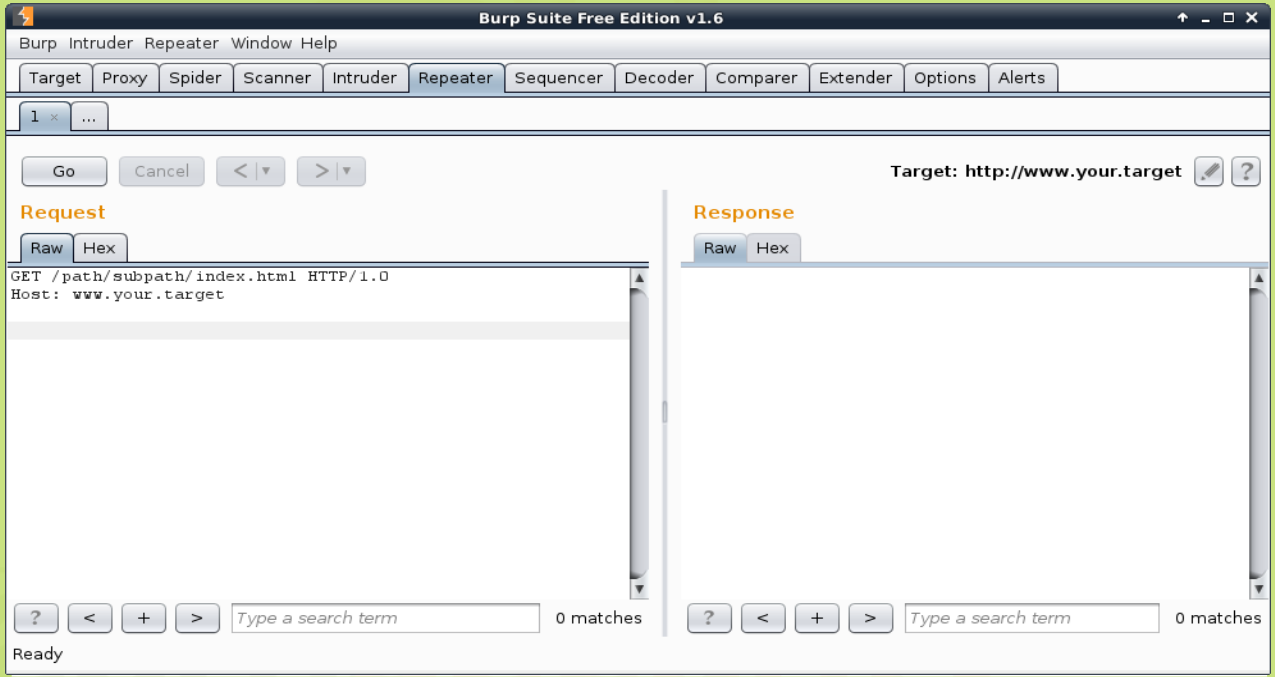


You can define your request by using this text area.

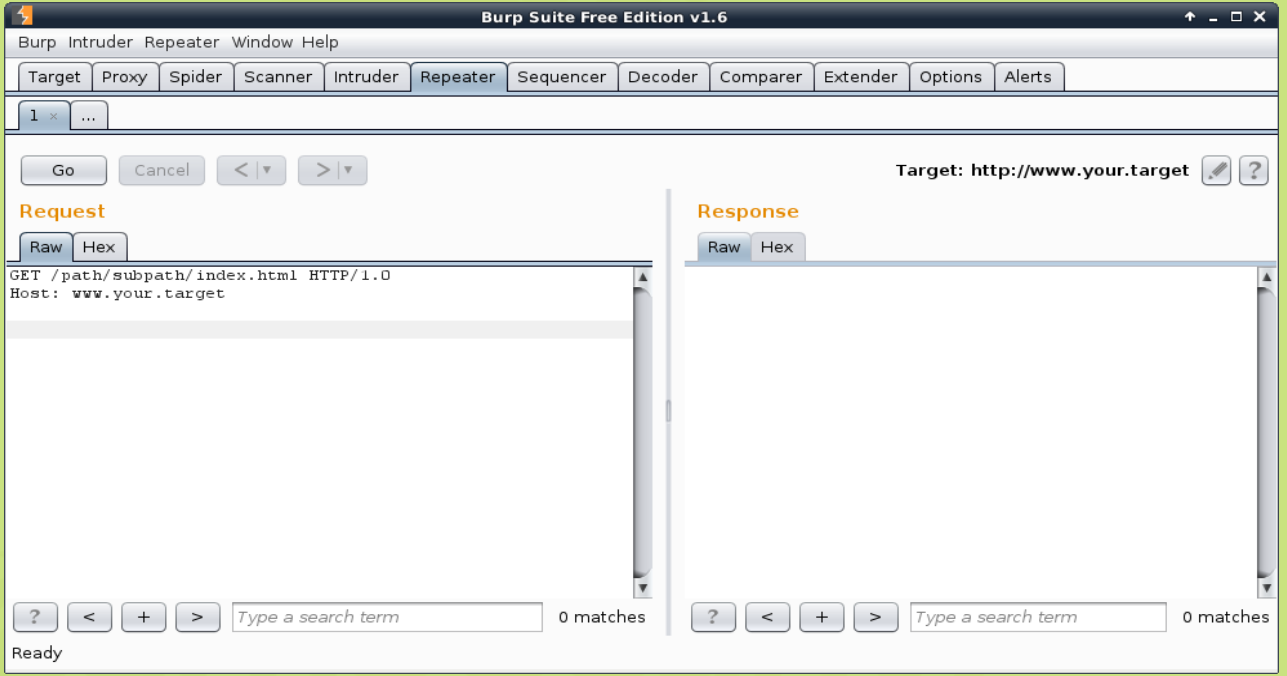
Every request must have at least an **HTTP VERB** (GET, POST, ...)



A **Host** header.

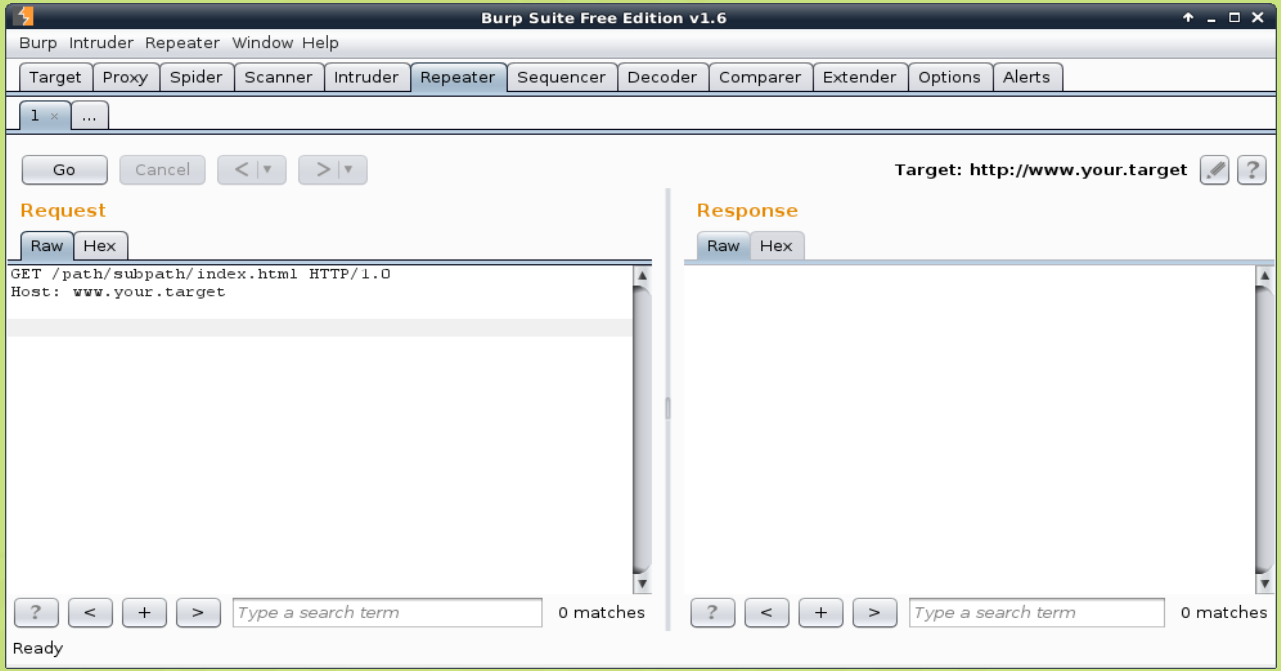


And **two empty lines** after the headers.

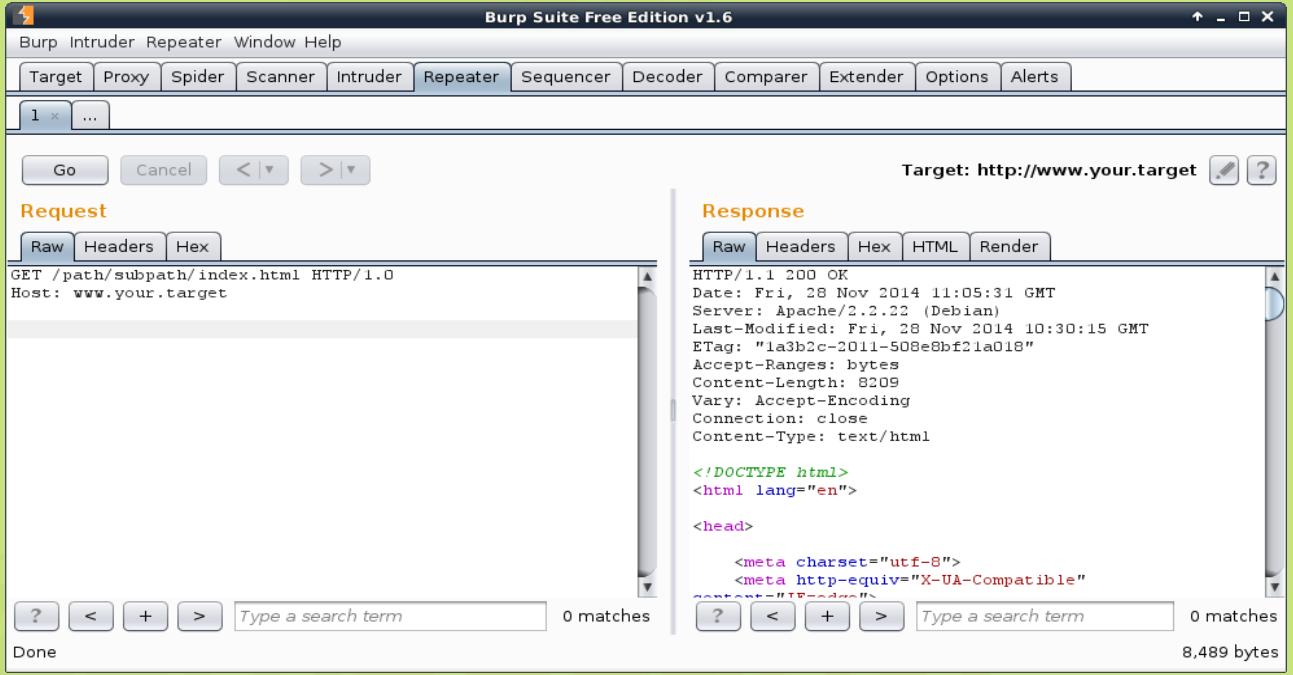


2.6.1. Burp Suite

When your request is complete, you can click the **Go** button to send it to the server.

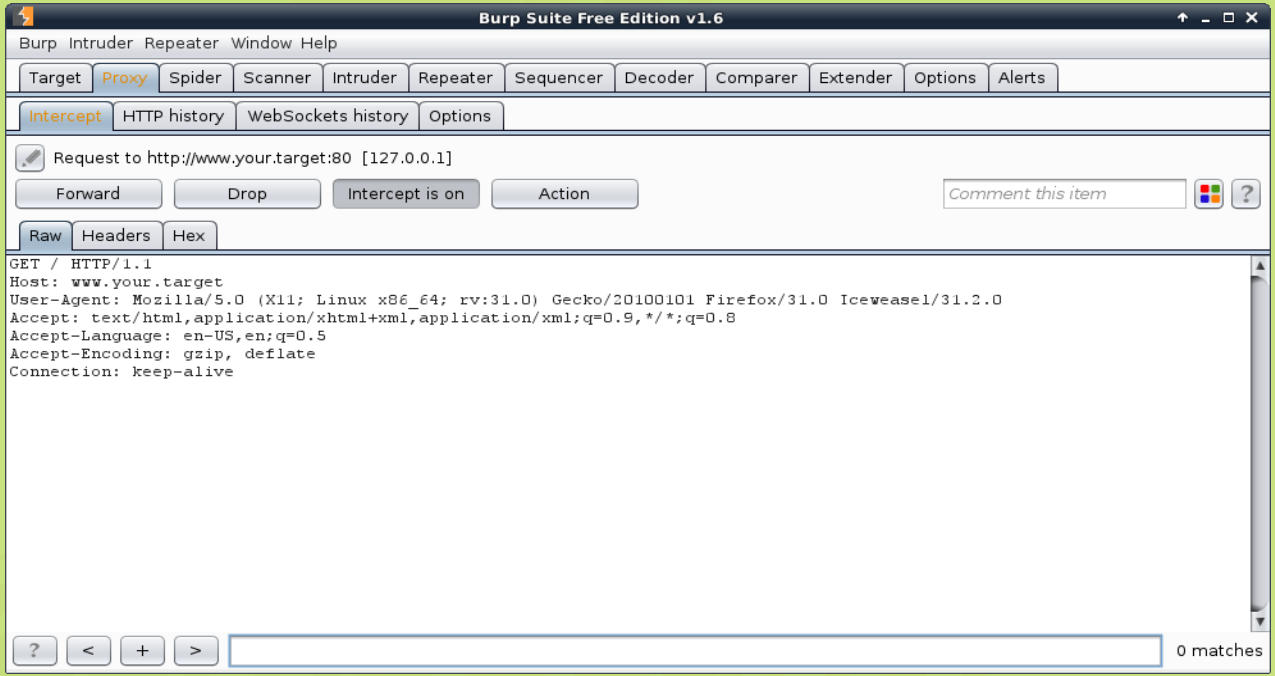


Burp displays the response in the **Response** panel.

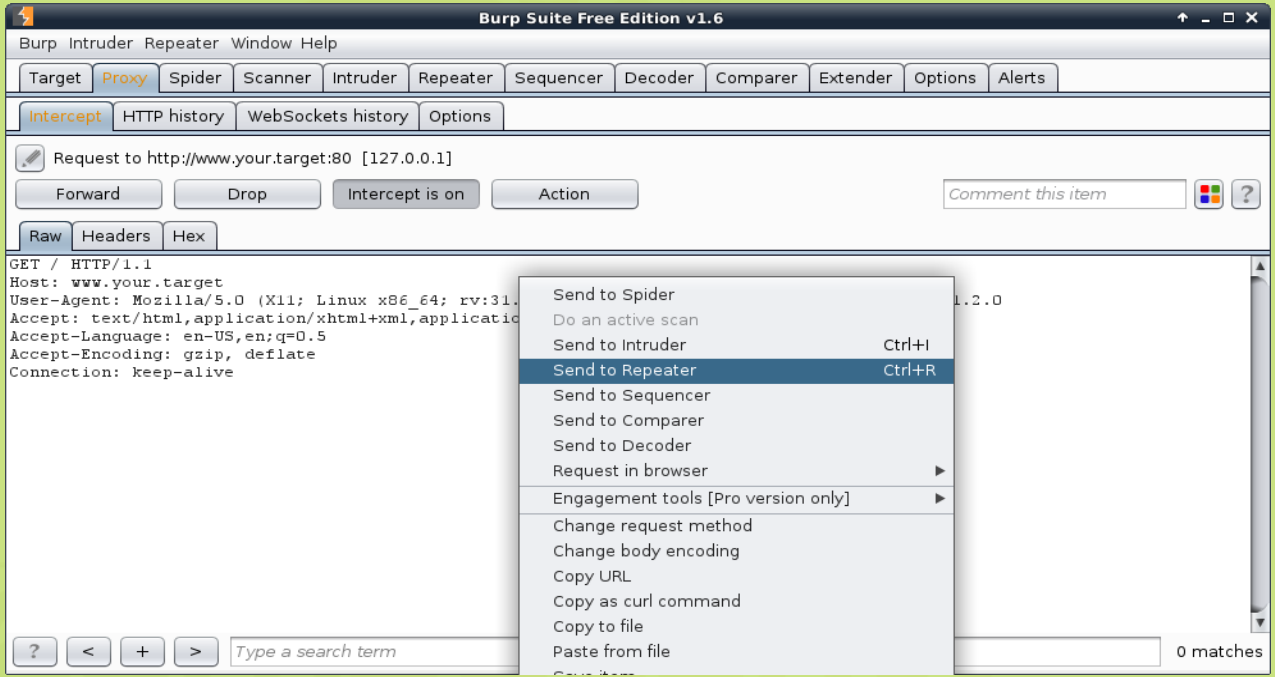


2.6.1. Burp Suite

An easier method to build requests is to intercept a browser request with the proxy and send it to the Repeater function.



You can do that with the **Ctrl+R** shortcut or by right clicking in the request body and selecting **Send to Repeater**.



In the following two videos you will learn how to define the target scope in Burp and OWASP ZAP, how to configure the Proxy to intercept only the traffic in scope.

Moreover you will see how to configure and use other features of both Burp and ZAP, according to the penetration test engagement goals.

If you have a FULL or ELITE plan you can click on the image on the left to start the video

If you have a FULL or ELITE plan you can click on the image on the left to start the video

References

References

URIs, URLs, and URNs: Clarifications and Recommendations 1.0

http://www.w3.org/TR/uri-clarification/

SSL/TLS Strong Encryption: An Introduction

http://httpd.apache.org/docs/2.2/ssl/ssl_intro.html

HTTP State Management Mechanism (cookies)

http://tools.ietf.org/html/rfc6265

ZAP

https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

HTTP Status Code Reference

http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

HTTP Overview, History, Versions and Standards

http://www.tcpipguide.com/free/t_HTTPOverviewHistoryVersionsandStandards.htm

Burp Suite

http://portswigger.net/burp/

Burp Suite

Same Origin

HTTP Cookies and Sessions

OWASP ZAP

Same Origin Policy

Test different Same Origin Policy configurations

Cookies

Test different cookies configuration on different domains