UNIVERSITÀ DI TRENTO

Riccardo BONAFEDE
Università di Padova

Matteo Golinelli

HTTP Protocol And Web Security Overview

CYBER CHALLENGE.IT

CYBERSECURITY NATIONAL LABORATORY

https://cybersecnatlab.it

License & Disclaimer

License Information

Disclaimer

- We disclaim any warranties or representations as to the accuracy or completeness of this material.
- Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.
- Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.

Goal

- Present the history of the HTTP
- Show the key features of the protocol
- Present the definition of Web Security and give a classification of the attacks
- List useful tools commonly used in Web Security

Prerequisites

- Lecture:
  - *NS_0.1 – Network Fundamentals*

Outline

- HTTP History
- Key Features and Overview of HTTP
- Security and Web Security
- Tooling

Outline

- HTTP History
- Key Features and Overview of HTTP
- Security and Web Security
- Tooling

HTTP History

- HTTP was introduced at the beginning of the '90s
- The first version of the protocol, **HTTP 0.9**, was released under the World Wide Web initiative
  - Extremely simple
  - Released in 1991
- https://www.w3.org/Protocols/HTTP/AsImplemented.html

HTTP History

- HTTP initial goal was to **share documents**
- Every document was (and still is) written in **HTML**
  - The first version of the language, HTML 1.0, is a barebone language whose main goal was to format texts and to connect them through hyperlinks
- The **first** example of a **browser** for this language was called "WorldWideWeb"

HTTP History

World Wide Web

The WorldWideWeb (W3) is a wide-area hypermedia information retrieval initiative aiming to give universal access to a large universe of documents.

Everything there is online about W3 is linked directly or indirectly to this document, including an executive summary of the project, Mailing lists , Policy , November's W3 news , Frequently Asked Questions .

What's out there?
: Pointers to the world's online information, subjects , W3 servers, etc.

Help
: on the browser you are using

Software Products
: A list of W3 project components and their current state. (e.g. Line Mode ,X11 Viola , NeXTStep , Servers , Tools , Mail robot , Library)

Technical
: Details of protocols, formats, program internals etc

Bibliography
: Paper documentation on W3 and references.

People
: A list of some people involved in the project.

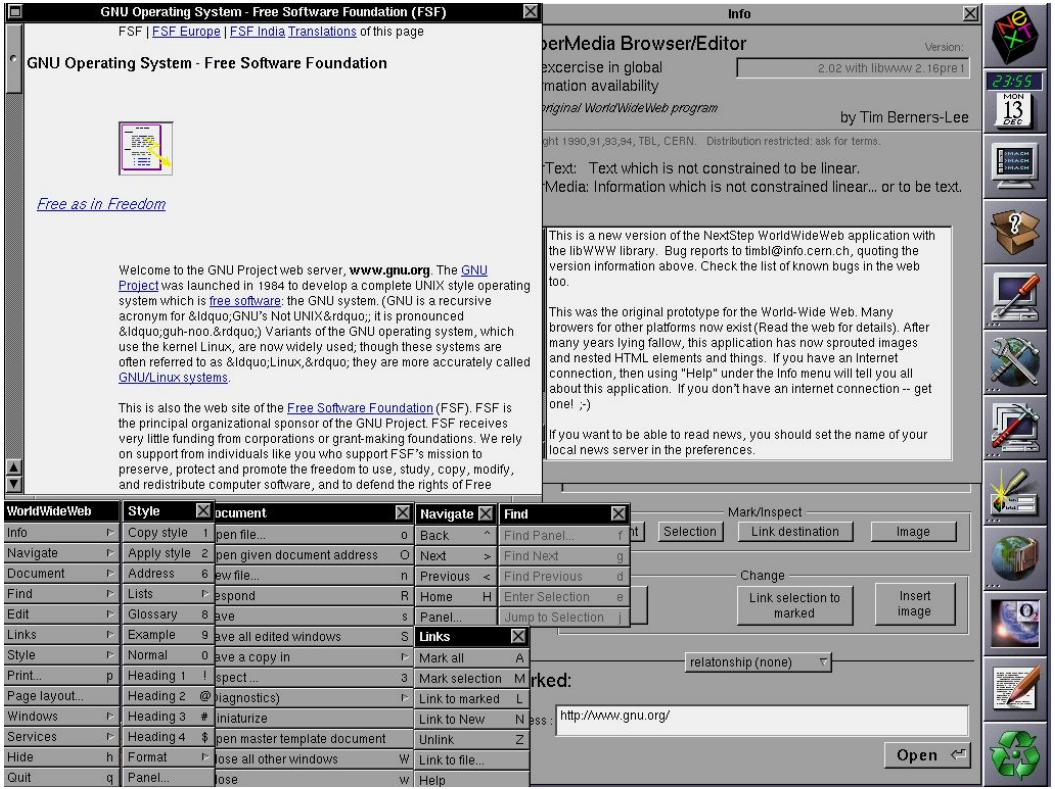
History
: A summary of the history of the project.

How can I help ?
: If you would like to support the web..

Getting code
: Getting the code by anonymous FTP , etc.

HTTP History

- HTTP
- Every
  - The
    lan
    cor
- The f
  calle



ITML

arebone
d to

age was

HTTP - History

- Through the years, **browsers became more and more complex**
- **Mosaic** in mid-1993 brought new features, such as the possibility to embed images into web pages
- Several software houses started to develop their own browser, adding new features to defeat the competition
  - HTML enchantments
  - JavaScript
  - Plugins such as Java/Flash

HTTP

- Through
- **Mosaic** i
  embed i
- Several s
  browser,
  - HTML
  - JavaSc
  - Plugin

complex

possibility to

n

1997.webhistory.org/www.lists/www-talk.1993q1/0182.html

proposed new tag: IMG

Marc Andreessen (*marca@ncsa.uiuc.edu*)
Thu, 25 Feb 93 21:09:02 -0800

- **Messages sorted by:** [date][thread][subject][author]
- **Next message:** Tony Johnson: "Re: proposed new tag: IMG"
- **Previous message:** Bill Janssen: "Re: xmosaic experience"
- **Next in thread:** Tony Johnson: "Re: proposed new tag: IMG"

I'd like to propose a new, optional HTML tag:

IMG

Required argument is SRC="url".

This names a bitmap or pixmap file for the browser to attempt to pull over the network and interpret as an image, to be embedded in the text at the point of the tag's occurrence.

An example is:

<IMG SRC="file://foobar.com/foo/bar/blargh.xbm">

(There is no closing tag; this is just a standalone tag.)

HTTP - History

- This race (called the "**browsers war**") led to a vast diversity of standards
- Each browser implemented its own (often undocumented) heuristics to maintain compatibility with other browsers
  - Often ignoring all the security implications

HTTP - Present

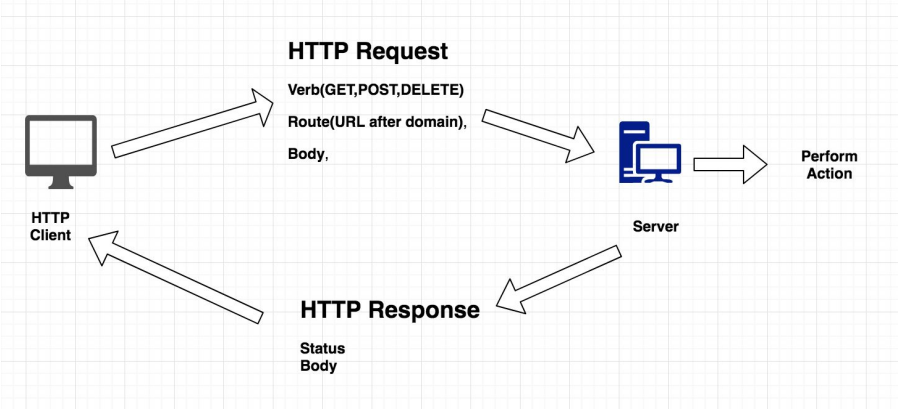
- In an effort to mitigate this anarchy, in **1994** the **W3C consortium** was created
  - The goal was to set mandatory web standards for vendors
- Eventually, vendors started to follow these standards, and by now the vast majority of the problems introduced by the "browsers war" are solved

Outline

- HTTP History
- Key Features and Overview of HTTP
- Security and Web Security
- Tooling

HTTP Overview

- Defined in **RFC2616**[1]
- High level protocol (Application level in the ISO/OSI stack)
- Mainly on TCP
  - Ports 80/443
  - HTTPS: HTTP over TLS
- Human readable
- Client/Server architecture
- Stateless



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

HTTP Overview

- HTTP is about **resources**
- A resource is an asset that a client requests to access, and it can be
  - A HTML file
  - An image
  - A JavaScript file, a CSS file
  - ...

HTTP Overview

- Resources are uniquely represented with **URLs**, acronym of
  - Uniform Resource Locator
- URLs are defined in RFCs **1738**[1] (URLs) and **3986**[2] (URIs)

https://en.wikipedia.org/wiki/Request_for_Comments

1: https://tools.ietf.org/html/rfc1738

2: https://tools.ietf.org/html/rfc3986

HTTP Overview - URLs

- An URL may be the following:

```
http://unitn.it:80/view.php?lang=it
```

HTTP Overview - URLs

- An URL may be the following:

```
http://unitn.it:80/view.php?lang=it
```

Schema

- The **schema** specifies the protocol used, i.e. http or https

HTTP Overview - URLs

- An URL may be the following:

```
http://unitn.it:80/view.php?lang=it
```

Host Port

- The **host** and the **port** represent the address to which the client should connect and send the request

HTTP Overview - URLs

- An URL may be the following:

```
http://unitn.it:80/view.php?lang=it
```

Path

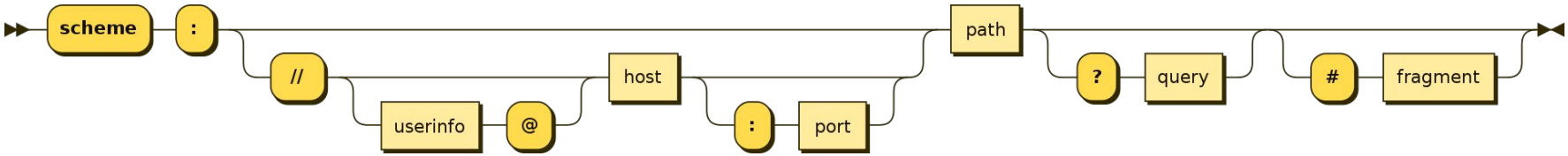
Query parameters

- The **path** is the resource requested to the server
- The **query parameters** are an extension to the URLs

HTTP Overview - URLs

- An URL may be the following:

```
http://unitn.it:80/view.php?lang=it#footer
```



HTTP Overview - URLs

- Optionally an URL can also contain an username and a password. These are used if the server requires an HTTP Basic authentication. An example of such url is the following one:

```
https://admin:password@unitn.it/
```

HTTP Overview - URLs

- The **path** has its own syntax in http, but since its interpretation is reserved to the backend, this syntax may vary from web app to web app
- Usually, it is in the form of

```
/<directory>/<file>?<query>#fragment
```

HTTP Overview - URLs

- The <directory>/<file> part is used to represent the "physical" location of the resource on the filesystem of the server
- Nowadays, it is more a "abstract" location

```
https://twitter.com/rickastley/status/1757825883866325290
```

HTTP Overview - URLs

- The <query> is optional and it can be used to send information to the backend
- It is a dictionary with a **key-value representation**, generally in the form of

```
varname1=value1&varname2=value2
```

HTTP Overview - URLs

- The fragment is a piece of information that is reserved for **clients**
- Clients never send it to the server
- If the fragment appears in a request, the server <u>will ignore it</u>

HTTP Overview - URLencoding

- Looking at the URL syntax one may notice that some characters have a **special meaning** in a URL
  - The character "#" is used for segments, and the server will always ignore every character after it
  - The character “&” is used as a variable separator in the URL-Path
  - ...

HTTP Overview - URLencoding

- What if we want to send the text «hello &#» in a GET variable?

```
http://unitn.it/?var=hello &# world
```

- Because the fragment is reserved for clients, the server will **ignore** the word «world»

HTTP Overview - URLencoding

- This problem is solved using a **particular encoding**, that converts every character in a "not harmful" representation
- This encoding is called “**URL encoding**" or “**Percent Encoding**”

HTTP Overview - URLencoding

- This encoding is very simple
  - Take the hex value of a character you want to encode, and then prepend a "%" symbol

$$\# \longrightarrow \%23$$

- Every reserved character in a URL must be urlencoded
- Every non-printable character must be urlencoded
- Spaces can be represented either with %20, or with the plus sign (+)

HTTP Overview - URLencoding

- So the following not valid URL

```
http://unitn.it/?var=hello &# world
```

- Is rewrote as

```
http://unitn.it/?var=hello+%26%23+world
```

HTTP Overview

- **Requests** and **Responses** are HTTP messages composed of three different parts
  - The Request/Response line
  - The Header Fields
  - A Body (optional)
- Every line in the Request/Response is terminated by the "CR;LF" sequence: `\r\n` or `0x0d0a` in binary
- An empty line separates the last header field from the body

HTTP Overview - Requests


```
GET / HTTP/1.1
Host: www.google.com
User-Agent: curl/7.64.0
Accept: */*


*Body, empty this time*
```

Request Line

Header Fields

Body Field

HTTP Overview - Requests

- The Request line is composed of
  - A **method**
    - "we want to **<METHOD>** the resource"
  - The **resource** for which we are doing the request
  - And, finally, the **protocol version**

```
GET / HTTP/1.1
```

HTTP Overview - Methods

- Tell the server "what we are doing" with the resource
- Standard methods
  - GET
  - POST
  - OPTIONS
  - HEAD

HTTP Overview - Body

- **Generic data** sent to the server
- Its type (or encoding) is defined by the **`Content-Type`** header
- It can be **encoded in different ways:**
  - `application/x-www-form-urlencoded`
  - `text/plain`
  - …
- It can also have a **custom encoding:**
  - `application/json`
  - foo/bar
  - …

HTTP Overview - Headers

- Headers are used to send **additional data** to the server
- Serialized in the form **`name: value`**
- Some are mandatory:
  - **`Host`**
  - **`Content-Encoding/Content-Length`** if there is a body

HTTP Overview - Responses

- A Response is very similar to a Request
- It differs only for the **first line**, which is called "status-line"
- This line is mandatory, and tells the client the type of the response and the version of the protocol used to make the response

HTTP Overview - Responses


```
HTTP/1.1 200 Ok
Host: 127.0.0.1:5000
Date: Thu, 12 Mar 2020 10:31:38 GMT
Connection: close
X-Powered-By: PHP/7.4.3

<b>Hello World!</b>
```

Status-Line

Header Fields

Body field

HTTP Overview – Status Line

- The status-line composed by the **version of the protocol**, an **integer number**, and a **string**
- The number is called **status code.** Status codes are divided into five categories:
  - 1**: Informational Response
  - 2**: Success
  - 3**: Location change
  - 4**: Client Error
  - 5**: Server Error

HTTP Overview – Status code

- Some common status codes are
  - 200: The request was successful
  - 400: The request was malformed
  - 404: The requested resource could not be found
  - 500: The server had a critical error, and could not complete the request

HTTP Overview - Cookies

- In order to make HTTP stateful, **cookies** were introduced
- Cookies are **text information** that a web client receives and stores from a server, and sends back within every request to the host
- They are used mainly for
  - Session management
  - Personalization
  - Tracking

HTTP Overview - Cookies

- HTTP servers can set cookies with the response header field **Set-Cookie**
- Cookies can also be set client-side via JavaScript
- Cookies are composed by a name, a value, and some meta-information
  - The origin (e.g., the server which sends the cookie)
  - The expire date
  - Some security policies

HTTP Overview - Cookies

Response Headers

Access-Control-Allow-Credentials: true

Access-Control-Allow-Headers: X-Requested-With, Content-Type, X-Codingpedia

Access-Control-Allow-Methods: GET, POST, DELETE, PUT

Access-Control-Allow-Origin: *

Content-Length: 65

Content-Type: application/json

Date: Tue, 23 May 2017 08:44:06 GMT

Server: GlassFish Server Open Source Edition 4.1

Set-Cookie: token=y9cHZIrjGqSlipT;Version=1;Comment=;Domain=;Path=/;Max-Age=3600;Expires=Tue, 23 May 2017 09:44:06 GMT

X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server Open Source Edition 4.1 Java/Oracle Corporation/1.8)

HTTP Overview - Cookies

- Browsers will send back cookies to the server in accordance with its scope
- The scope is the "origin" in which each cookie was created
  - If a cookie named "foo" is set by "www.google.com", it cannot be sent to "www.microsoft.com", but only to "www.google.com"

HTTP Overview - Cookies

- In addition to the origin, other security policies can be set
  - Secure and HTTPOnly
  - SameSite
    - None
    - Strict
    - Lax <-- The default on Chrome

Outline

- HTTP History
- Key Features and Overview of HTTP
- Security and Web Security
- Tooling

Security

- As the name suggests, security is about the protection of "something"
- When dealing with computers, we normally identify this "something" with **information**
- Security wants to ensure three main properties of information
  - Confidentiality
  - Integrity
  - Availability

Security

- **Integrity**
  - Maintaining the accuracy and completeness of data
- **Confidentiality**
  - Data must be accessible only to whom is authorized to
- **Availability**
  - Data must be accessible when needed

Security

- A vulnerability is a **weakness in a system** that permits an attacker to violate one or more of the three previous properties
- Every vulnerability must have an **impact**
  - How this vulnerability of this system violates one or more of the three principles?

Security

- The best way to find how a vulnerability impacts a system is to assume the point of view of an attacker
- This is done by testing the application in an offensive manner
- This activity is called **penetration testing**

Web Security

- Web security applies to vulnerabilities that affect web applications
- Typically, web applications are the most exposed assets to an attacker
- And HTTP is really *fragile*

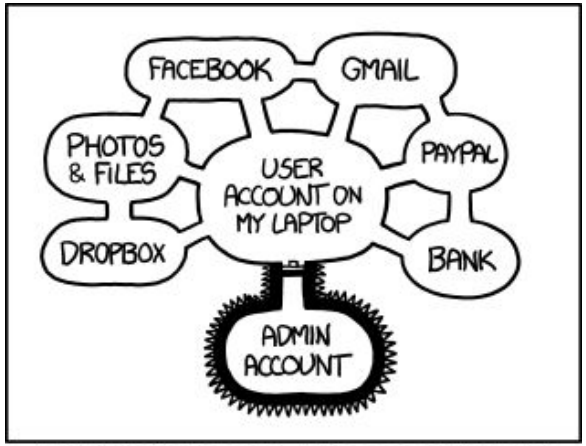
Web Security

- HTTP was created with the intent to serve **static documents**
- The protocol is **simple by design**
  - It is a **stateless** protocol, since there was no need to keep track of the current client
  - **Documents were simple**, there was no need for animations
  - The **security was not a big concern**, at the beginning there was not much to protect on the web

Web Security

- But now we have
  - **Dynamically generated pages**, i.e., pages generated on-the-fly by some code
  - Exceptionally **complex pages**: HTML, CSS, JavaScript, WebAsm, ~~plugins~~... and a lot more
  - A lot of **secrets to protect**
- Such complexity leads to a **huge attack surface**

The web is a mess...



IF SOMEONE STEALS MY LAPTOP WHILE I'M LOGGED IN, THEY CAN READ MY EMAIL, TAKE MY MONEY, AND IMPERSONATE ME TO MY FRIENDS,

BUT AT LEAST THEY CAN'T INSTALL DRIVERS WITHOUT MY PERMISSION.

Web Security

- Web security is about the security of web assets, i.e., everything that runs over HTTP
  - **Server-Side Security**: The impact affects the remote server
  - **Client-Side Security**: The impact affects the client
    - Note: This does not mean that it is a vulnerability of the browser!
  - Rule of thumb: *If you need to send a link to the victim, then it is probably a client-side vulnerability*

Web Security

- In order to find vulnerabilities, there are two main methodologies
  - **Blackbox**
    - We **do not know anything** about the system we are attacking
  - **Whitebox**
    - We **know everything** about the system, we have the source code, we can debug it, ...

Web Security – BlackBox

- **Enumeration**: the **more information** we have about the system **the better**
  - Look at the functions an application implements
  - Try to input random things.
    - If you have to insert a number, try inserting letters, and look at what happens, put a negative number, …
- **Try and error**:
  - we do not know anything about the system, we **need to try attacks in order to discover problems**

Web Security – WhiteBox

- When testing on a WhiteBox environment there is much more a tester can do:
  - **Static Analysis** of the code
  - **Dynamic analysis** of the code: executing the code

Web Security – WhiteBox

- In a WhiteBox environment a tester can discover deeper issues than within a BlackBox environment
- In this way, the tester has an advantage over an attacker, because it can discover more flaws in less time and with less skills
- Because of this, WhiteBox testing is **more effective** than BlackBox testing

Outline

- HTTP History
- Key Features and Overview of HTTP
- Security and Web Security
- Tooling

Web Security – Some useful tools

- Browser
- `curl` or `wget`
  - Command line utilities to make http requests
- Python `requests`
  - A python library to perform HTTP requests
- **Burp suite**, Zap proxy, Caido
  - Live edit raw HTTP requests and responses
- Test server (php dev & python http.server)
- Ngrok
  - creates a public http/tcp tunnel to your machine

On-the-fly HTTP server

- Python
  - Serves every file inside the directory it was launched from
  - `python -m http.server 8080`
- PHP
  - Serves every file inside the directory it was launched from and executes .php scripts
  - `php –S 127.0.0.1:8080`

Launch it from a test directory! You don't want to leak your .ssh directory

ngrok

- What if you need a public server?
  - VPS
  - Ngrok: https://ngrok.com/

ngrok

- Ngrok allows one to create tunnels
- You can run it with the command
  - `$ ngrok http 8080`
    - Create a http tunnel and redirect every request to the local port 8080
  - `$ ngrok tcp 8080`
    - Create a tcp tunnel and redirect every connection to the local port 8080

Demo

- Create an empty folder
- Create an HTML file with some content
- `python -m http.server 8080`
- `ngrok http 8080`

☕ ?