Universidad Nacional del Sur

Proyecto Final de Carrera
Ingeniería en Computación

Seguridad en redes LAN: utilizando docker para mejorar la infraestructura.

Salvador Catalfamo

Bahía Blanca – Argentina
2020

Universidad Nacional del Sur

Proyecto Final de Carrera
Ingeniería en Computación

Seguridad en redes LAN: utilizando docker para mejorar la infraestructura.

Salvador Catalfamo

Bahía Blanca – Argentina
2020

Resumen

A lo largo de la carrera, hemos visto como las organizaciones abordan los temas de seguridad en sus sistemas informáticos. Mayormente, se concentran en los equipos que están expuestos a la red pública, dejando de lado los que se encuentran aislados de la misma. Erróneamente, muchas veces se piensa que es suficiente, sin embargo, puede traer graves inconvenientes. Es por eso que realizaremos un estudio teórico/práctico sobre las consecuencias de la navegación en redes internas sin ningún tipo de cifrado de datos ni certificaciones.

Palabras Clave:

Seguridad e Infraestructura

Docker

Linux

Kali

Máquinas Virtuales

Índice general

Capítulo 1

Introducción

1.1. Objetivos

- item 1
- item 2

```
 Este es un bien ambiente para
poner codigo
```

En [5] se [1] ve...

casa

casa

`casa`

casa

casa

1.2. Plan de tesis y principales contribuciones

1.3. Trabajos previos relacionados

[1] Esta es una nota al pie

Figura 1.1: Esta es la figura del escudo de la uns

Capítulo 2

(Ajustar) ¿Qué circula por una red interna?

2.1. Introducción

2.2. Proocolos asociados a la web

2.2.1. ¿Que es el protocolo HTTP?

The Hypertext Transfer Protocol (HTTP) is a stateless application-level request/response protocol that uses extensible semantics and self-descriptive message payloads for flexible interaction with network-based hypertext information systems. HTTP is a generic interface protocol for information systems. It is designed to hide the details of how a service is implemented by presenting a uniform interface to clients that is independent of the types of resources provided. Likewise, servers do not need to be aware of each client's purpose: an HTTP request can be considered in isolation rather than being associated with a specific type of client or a predetermined sequence of application steps. The result is a protocol that can be used effectively in many different contexts and for which implementations can evolve independently over time.

One consequence of this flexibility is that the protocol cannot be defined in terms of what occurs behind the interface. Instead, we are limited to defining the syntax of communication, the intent of received communication, and the expected behavior

of recipients. If the communication is considered in isolation, then successful actions ought to be reflected in corresponding changes to the observable interface provided by servers. However, since multiple clients might act in parallel and perhaps at cross-purposes, we cannot require that such changes be observable beyond the scope of a single response.

2.2.2. Arquitectura

HTTP was created for the World Wide Web (WWW) architecture and has evolved over time to support the scalability needs of a worldwide hypertext system. Much of that architecture is reflected in the terminology and syntax productions used to define HTTP.

2.2.2.1. Client/Server Messaging

HTTP is a stateless request/response protocol that operates by exchanging messages across a reliable transport- or session-layer çonnection". An HTTP çlientïs a program that establishes a connection to a server for the purpose of sending one or more HTTP requests. An HTTP "serverïs a program that accepts connections in order to service HTTP requests by sending HTTP responses. Most HTTP communication consists of a retrieval request (GET) for a representation of some resource identified by a URI. In the simplest case, this might be accomplished via a single bidirectional connection (===) between the user agent (UA) and the origin server (O).

request -¿UserAgent ========== Origin server ¡response

A client sends an HTTP request to a server in the form of a request message, beginning with a request-line that includes a method, URI, and protocol version, followed by header fields containing request modifiers, client information, and representation metadata, an empty line to indicate the end of the header section, and finally a message body containing the payload body (if any, Section 3.3). A server responds to a client's request by sending one or more HTTP response messages, each beginning with a status line that includes the protocol version, a success or error code, and textual reason phrase (Section 3.1.2), possibly followed by header

fields containing server information, resource metadata, and representation metadata (Section 3.2), an empty line to indicate the end of the header section, and finally a message body containing the payload body (if any, Section 3.3).

2.2.2.2. Ejemplo

The following example illustrates a typical message exchange for a GET request (Section 4.3.1 of [RFC7231]) on the URI "http://www.example.com/hello.txt":

Client request:

GET /hello.txt HTTP/1.1 User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3 Host: www.example.com Accept-Language: en, mi

Server response:

HTTP/1.1 200 OK Date: Mon, 27 Jul 2009 12:28:53 GMT Server: Apache Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT ETag: "34aa387-d-1568eb00.ᴬccept-Ranges: bytes Content-Length: 51 Vary: Accept-Encoding Content-Type: text/plain

Hello World! My payload includes a trailing CRLF.

HTTP is defined as a stateless protocol, meaning that each request message can be understood in isolation. Many implementations depend on HTTP's stateless design in order to reuse proxied connections or dynamically load balance requests across multiple servers. Hence, a server MUST NOT assume that two requests on the same connection are from the same user agent unless the connection is secured and specific to that agent. Some non-standard HTTP extensions (e.g., [RFC4559]) have been known to violate this requirement, resulting in security and interoperability problems.

2.2.3. Formato del mensaje (Mejorar intro)

All HTTP/1.1 messages consist of a start-line followed by a sequence of octets in a format similar to the Internet Message Format [RFC5322]: zero or more header fields (collectively referred to as the "headers.ºr the "header section"), an empty line indicating the end of the header section, and an optional message body. The normal procedure for parsing an HTTP message is to read the start-line into a structure, read each header field into a hash table by field name until the empty line, and then use the parsed data to determine if a message body is expected. If a message body

has been indicated, then it is read as a stream until an amount of octets equal to the message body length is read or the connection is closed.

2.2.3.1. Start Line

An HTTP message can be either a request from client to server or a response from server to client. Syntactically, the two types of message differ only in the start-line, which is either a request-line (for requests) or a status-line (for responses), and in the algorithm for determining the length of the message body . In theory, a client could receive requests and a server could receive responses, distinguishing them by their different start-line formats, but, in practice, servers are implemented to only expect a request (a response is interpreted as an unknown or invalid request method) and clients are implemented to only expect a response.

2.2.3.1.1. Request Line A request-line begins with a method token, followed by a single space (SP), the request-target, another single space (SP), the protocol version, and ends with CRLF.

(podria ir un grafico)

The method token indicates the request method to be performed on the target resource. The request method is case-sensitive.

The request-target identifies the target resource upon which to apply the request

2.2.3.1.2. Status Line The first line of a response message is the status-line, consisting of the protocol version, a space (SP), the status code, another space, a possibly empty textual phrase describing the status code, and ending with CRLF.

(podria ir un grafico)

The status-code element is a 3-digit integer code describing the result of the server's attempt to understand and satisfy the client's corresponding request. The rest of the response message is to be interpreted in light of the semantics defined for that status code.

2.2.3.2. Header Fields

Each header field consists of a case-insensitive field name followed by a colon (":"), optional leading whitespace, the field value, and optional trailing whitespace.

(podria ir un grafico o con un formato mejor) header-field = field-name ":.ºWS field-value OWS

The field-name token labels the corresponding field-value as having the semantics defined by that header field.

2.2.3.2.1. Field Extensibility Header fields are fully extensible: there is no limit on the introduction of new field names, each presumably defining new semantics, nor on the number of header fields used in a given message. Existing fields are defined in each part of this specification and in many other specifications outside this document set.

New header fields can be defined such that, when they are understood by a recipient, they might override or enhance the interpretation of previously defined header fields, define preconditions on request evaluation, or refine the meaning of responses.

2.2.3.2.2. Field Order The order in which header fields with differing field names are received is not significant. However, it is good practice to send header fields that contain control data first, such as Host on requests and Date on responses, so that implementations can decide when not to handle a message as early as possible.

2.2.3.2.3. Whitespace This specification uses three rules to denote the use of linear whitespace: OWS (optional whitespace), RWS (required whitespace), and BWS ("bad"whitespace).

The OWS rule is used where zero or more linear whitespace octets might appear. The RWS rule is used when at least one linear whitespace octet is required to separate field tokens.

The BWS rule is used where the grammar allows optional whitespace only for historical reasons.

2.2.3.2.4. Field Parsing Messages are parsed using a generic algorithm, independent of the individual header field names. The contents within a given field value are not parsed until a later stage of message interpretation (usually after the message's entire header section has been processed). Consequently, this specification does not use ABNF rules to define each "Field-Name: Field Value"pair, as was done in previous editions. Instead, this specification uses ABNF rules that are named according to each registered field name, wherein the rule defines the valid grammar for that field's corresponding field values (i.e., after the field-value has been extracted from the header section by a generic field parser).

No whitespace is allowed between the header field-name and colon. In the past, differences in the handling of such whitespace have led to security vulnerabilities in request routing and response handling.

2.2.3.2.5. Field Limits HTTP does not place a predefined limit on the length of each header field or on the length of the header section as a whole. Various ad hoc limitations on individual header field length are found in practice, often depending on the specific field semantics.

2.2.3.2.6. Field Value Components (no entiendo, tal vez se puede sacar) Most HTTP header field values are defined using common syntax components (token, quoted-string, and comment) separated by whitespace or specific delimiting characters. Delimiters are chosen from the set of US-ASCII visual characters not allowed in a token

token = 1*tchar

/ DIGIT / ALPHA ; any VCHAR, except delimiters

A string of text is parsed as a single value if it is quoted using double-quote marks.

quoted-string = DQUOTE *(qdtext / quoted-pair) DQUOTE qdtext = HTAB / SP /obs-text =

Comments can be included in some HTTP header fields by surrounding the comment text with parentheses. Comments are only allowed in fields containing çomment.ªs part of their field value definition.

comment = "("*(ctext / quoted-pair / comment) ")çtext = HTAB / SP /

The backslash octet (”¨) can be used as a single-octet quoting mechanism within quoted-string and comment constructs. Recipients that process the value of a quoted-string MUST handle a quoted-pair as if it were replaced by the octet following the backslash.

quoted-pair = ”¨(HTAB / SP / VCHAR / obs-text)

A sender SHOULD NOT generate a quoted-pair in a quoted-string except where necessary to quote DQUOTE and backslash octets occurring within that string. A sender SHOULD NOT generate a quoted-pair in a comment except where necessary to quote parentheses [”(.ªnd ”)”] and backslash octets occurring within that comment.

2.2.3.3. Message Body

The message body (if any) of an HTTP message is used to carry the payload body of that request or response. The message body is identical to the payload body unless a transfer coding has been applied.

The rules for when a message body is allowed in a message differ for requests and responses.

The presence of a message body in a request is signaled by a Content-Length or Transfer-Encoding header field. Request message framing is independent of method semantics, even if the method does not define any use for a message body.

The presence of a message body in a response depends on both the request method to which it is responding and the response status code. Responses to the HEAD request method never include a message body because the associated response header fields , if present, indicate only what their values would have been if the request method had been GET (Section 4.3.1 of [RFC7231]). 2xx (Successful) responses to a CONNECT request method (Section 4.3.6 of [RFC7231]) switch to tunnel mode instead of having a message body. All 1xx (Informational), 204 (No Content), and 304 (Not Modified) responses do not include a message body. All other responses do include a message body, although the body might be of zero length.

2.2.3.3.1. Transfer-Encoding The Transfer-Encoding header field lists the transfer coding names corresponding to the sequence of transfer codings that have

been (or will be) applied to the payload body in order to form the message body. Transfer codings are defined in Section 4.

Transfer-Encoding is analogous to the Content-Transfer-Encoding field of MIME, which was designed to enable safe transport of binary data over a 7-bit transport service . However, safe transport has a different focus for an 8bit-clean transfer protocol. In HTTP's case, Transfer-Encoding is primarily intended to accurately delimit a dynamically generated payload and to distinguish payload encodings that are only applied for transport efficiency or security from those that are characteristics of the selected resource. .

For example,

Transfer-Encoding: gzip, chunked

indicates that the payload body has been compressed using the gzip coding and then chunked using the chunked coding while forming the message body.

Unlike Content-Encoding , Transfer-Encoding is a property of the message, not of the representation, and any recipient along the request/response chain MAY decode the received transfer coding(s) or apply additional transfer coding(s) to the message body, assuming that corresponding changes are made to the Transfer-Encoding field-value. Additional information about the encoding parameters can be provided by other header fields not defined by this specification.

Transfer-Encoding was added in HTTP/1.1. It is generally assumed that implementations advertising only HTTP/1.0 support will not understand how to process a transfer-encoded payload.

2.2.3.3.2. Content-Length When a message does not have a Transfer-Encoding header field, a Content-Length header field can provide the anticipated size, as a decimal number of octets, for a potential payload body. For messages that do include a payload body, the Content-Length field-value provides the framing information necessary for determining where the body (and message) ends. For messages that do not include a payload body, the Content-Length indicates the size of the selected representation .

2.2.3.3.3. Message Body Length (no entiendo, ver diferencia con content-length) The length of a message body is determined by one of the following (in order

of precedence):

1. Any response to a HEAD request and any response with a 1xx (Informational), 204 (No Content), or 304 (Not Modified) status code is always terminated by the first empty line after the header fields, regardless of the header fields present in the message, and thus cannot contain a message body.

2. Any 2xx (Successful) response to a CONNECT request implies that the connection will become a tunnel immediately after the empty line that concludes the header fields. A client MUST ignore any Content-Length or Transfer-Encoding header fields received in such a message.

3. If a Transfer-Encoding header field is present and the chunked transfer coding (Section 4.1) is the final encoding, the message body length is determined by reading and decoding the chunked data until the transfer coding indicates the data is complete.

If a Transfer-Encoding header field is present in a response and the chunked transfer coding is not the final encoding, the message body length is determined by reading the connection until it is closed by the server. If a Transfer-Encoding header field is present in a request and the chunked transfer coding is not the final encoding, the message body length cannot be determined reliably; the server MUST respond with the 400 (Bad Request) status code and then close the connection.

If a message is received with both a Transfer-Encoding and a Content-Length header field, the Transfer-Encoding overrides the Content-Length. Such a message might indicate an attempt to perform request smuggling (Section 9.5) or response splitting (Section 9.4) and ought to be handled as an error. A sender MUST remove the received Content-Length field prior to forwarding such a message downstream.

4. If a message is received without Transfer-Encoding and with either multiple Content-Length header fields having differing field-values or a single Content-Length header field having an invalid value, then the message framing is invalid and the recipient MUST treat it as an unrecoverable error. If this is a request message, the server MUST respond with a 400 (Bad Request) status code and then close the connection. If this is a response message received by a proxy, the proxy MUST close the connection to the server, discard the received response, and send a 502 (Bad Gateway) response to the client. If this is a response message received by a user agent, the user agent MUST close the connection to the server and discard the

received response.

5. If a valid Content-Length header field is present without Transfer-Encoding, its decimal value defines the expected message body length in octets. If the sender closes the connection or the recipient times out before the indicated number of octets are received, the recipient MUST consider the message to be incomplete and close the connection.

6. If this is a request message and none of the above are true, then the message body length is zero (no message body is present).

7. Otherwise, this is a response message without a declared message body length, so the message body length is determined by the number of octets received prior to the server closing the connection.

Since there is no way to distinguish a successfully completed, close-delimited message from a partially received message interrupted by network failure, a server SHOULD generate encoding or length-delimited messages whenever possible. The close-delimiting feature exists primarily for backwards compatibility with HTTP/1.0.

A server MAY reject a request that contains a message body but not a Content-Length by responding with 411 (Length Required).

Unless a transfer coding other than chunked has been applied, a client that sends a request containing a message body SHOULD use a valid Content-Length header field if the message body length is known in advance, rather than the chunked transfer coding, since some existing services respond to chunked with a 411 (Length Required) status code even though they understand the chunked transfer coding. This is typically because such services are implemented via a gateway that requires a content-length in advance of being called and the server is unable or unwilling to buffer the entire request before processing.

A user agent that sends a request containing a message body MUST send a valid Content-Length header field if it does not know the server will handle HTTP/1.1 (or later) requests; such knowledge can be in the form of specific user configuration or by remembering the version of a prior received response.

If the final response to the last request on a connection has been completely received and there remains additional data to read, a user agent MAY discard the remaining data or attempt to determine if that data belongs as part of the prior response body, which might be the case if the prior message's Content-Length value

is incorrect. A client MUST NOT process, cache, or forward such extra data as a separate response, since such behavior would be vulnerable to cache poisoning.

2.2.4. Security Considerations

(Agregar una intro)

2.2.4.1. Establishing Authority

HTTP relies on the notion of an authoritative response: a response that has been determined by (or at the direction of) the authority identified within the target URI to be the most appropriate response for that request given the state of the target resource at the time of response message origination. Providing a response from a non-authoritative source, such as a shared cache, is often useful to improve performance and availability, but only to the extent that the source can be trusted or the distrusted response can be safely used.

Unfortunately, establishing authority can be difficult. For example, phishing is an attack on the user's perception of authority, where that perception can be misled by presenting similar branding in hypertext, possibly aided by userinfo obfuscating the authority component (see Section 2.7.1). User agents can reduce the impact of phishing attacks by enabling users to easily inspect a target URI prior to making an action, by prominently distinguishing (or rejecting) userinfo when present, and by not sending stored credentials and cookies when the referring document is from an unknown or untrusted source.

When a registered name is used in the authority component, the "httpÜRI scheme (Section 2.7.1) relies on the user's local name resolution service to determine where it can find authoritative responses. This means that any attack on a user's network host table, cached names, or name resolution libraries becomes an avenue for attack on establishing authority. Likewise, the user's choice of server for Domain Name Service (DNS), and the hierarchy of servers from which it obtains resolution results, could impact the authenticity of address mappings; DNS Security Extensions are one way to improve authenticity.

Furthermore, after an IP address is obtained, establishing authority for an "httpÜRI is vulnerable to attacks on Internet Protocol routing.

2.2.4.2. Risks of Intermediaries

By their very nature, HTTP intermediaries are men-in-the-middle and, thus, represent an opportunity for man-in-the-middle attacks. Compromise of the systems on which the intermediaries run can result in serious security and privacy problems. Intermediaries might have access to security-related information, personal information about individual users and organizations, and proprietary information belonging to users and content providers. A compromised intermediary, or an intermediary implemented or configured without regard to security and privacy considerations, might be used in the commission of a wide range of potential attacks.

Intermediaries that contain a shared cache are especially vulnerable to cache poisoning attacks. Implementers need to consider the privacy and security implications of their design and coding decisions, and of the configuration options they provide to operators (especially the default configuration).

Users need to be aware that intermediaries are no more trustworthy than the people who run them; HTTP itself cannot solve this problem.

2.2.4.3. Attacks via Protocol Element Length

Because HTTP uses mostly textual, character-delimited fields, parsers are often vulnerable to attacks based on sending very long (or very slow) streams of data, particularly where an implementation is expecting a protocol element with no predefined length.

2.2.4.4. Response Splitting

Response splitting (a.k.a, CRLF injection) is a common technique, used in various attacks on Web usage, that exploits the line-based nature of HTTP message framing and the ordered association of requests to responses on persistent connections [Klein]. This technique can be particularly damaging when the requests pass through a shared cache.

Response splitting exploits a vulnerability in servers (usually within an application server) where an attacker can send encoded data within some parameter of the request that is later decoded and echoed within any of the response header fields of the response. If the decoded data is crafted to look like the response has ended

and a subsequent response has begun, the response has been split and the content within the apparent second response is controlled by the attacker. The attacker can then make any other request on the same persistent connection and trick the recipients (including intermediaries) into believing that the second half of the split is an authoritative answer to the second request.

For example, a parameter within the request-target might be read by an application server and reused within a redirect, resulting in the same parameter being echoed in the Location header field of the response. If the parameter is decoded by the application and not properly encoded when placed in the response field, the attacker can send encoded CRLF octets and other content that will make the application's single response look like two or more responses.

2.2.4.5. Request Smuggling

Request smuggling ([Linhart]) is a technique that exploits differences in protocol parsing among various recipients to hide additional requests (which might otherwise be blocked or disabled by policy) within an apparently harmless request. Like response splitting, request smuggling can lead to a variety of attacks on HTTP usage.

2.2.4.6. Message Integrity

HTTP does not define a specific mechanism for ensuring message integrity, instead relying on the error-detection ability of underlying transport protocols and the use of length or chunk-delimited framing to detect completeness. Additional integrity mechanisms, such as hash functions or digital signatures applied to the content, can be selectively added to messages via extensible metadata header fields. Historically, the lack of a single integrity mechanism has been justified by the informal nature of most HTTP communication. However, the prevalence of HTTP as an information access mechanism has resulted in its increasing use within environments where verification of message integrity is crucial.

User agents are encouraged to implement configurable means for detecting and reporting failures of message integrity such that those means can be enabled within environments for which integrity is necessary. For example, a browser being used to view medical history or drug interaction information needs to indicate to the

user when such information is detected by the protocol to be incomplete, expired, or corrupted during transfer. Such mechanisms might be selectively enabled via user agent extensions or the presence of message integrity metadata in a response. At a minimum, user agents ought to provide some indication that allows a user to distinguish between a complete and incomplete response message (Section 3.4) when such verification is desired.

2.2.4.7. Message Confidentiality

HTTP relies on underlying transport protocols to provide message confidentiality when that is desired. HTTP has been specifically designed to be independent of the transport protocol, such that it can be used over many different forms of encrypted connection, with the selection of such transports being identified by the choice of URI scheme or within user agent configuration.

2.2.5. Métodos del protocolo HTTP

4.3.1. GET

The GET method requests transfer of a current selected representation for the target resource. GET is the primary mechanism of information retrieval and the focus of almost all performance optimizations. Hence, when people speak of retrieving some identifiable information via HTTP, they are generally referring to making a GET request.

It is tempting to think of resource identifiers as remote file system pathnames and of representations as being a copy of the contents of such files. In fact, that is how many resources are implemented . However, there are no such limitations in practice. The HTTP interface for a resource is just as likely to be implemented as a tree of content objects, a programmatic view on various database records, or a gateway to other information systems. Even when the URI mapping mechanism is tied to a file system, an origin server might be configured to execute the files with the request as input and send the output as the representation rather than transfer the files directly. Regardless, only the origin server needs to know how each of its resource identifiers corresponds to an implementation and how each implementation manages to select and send a current representation of the target resource in a

response to GET.

4.3.2. HEAD

The HEAD method is identical to GET except that the server MUST NOT send a message body in the response (i.e., the response terminates at the end of the header section). The server SHOULD send the same header fields in response to a HEAD request as it would have sent if the request had been a GET, except that the payload header fields MAY be omitted. This method can be used for obtaining metadata about the selected representation without transferring the representation data and is often used for testing hypertext links for validity, accessibility, and recent modification.

4.3.3. POST

The POST method requests that the target resource process the representation enclosed in the request according to the resource's own specific semantics. For example, POST is used for the following functions (among others):

o Providing a block of data, such as the fields entered into an HTML form, to a data-handling process; o Posting a message to a bulletin board, newsgroup, mailing list, blog, or similar group of articles;

o Creating a new resource that has yet to be identified by the origin server; and

o Appending data to a resource's existing representation(s).

4.3.4. PUT

The PUT method requests that the state of the target resource be created or replaced with the state defined by the representation enclosed in the request message payload. A successful PUT of a given representation would suggest that a subsequent GET on that same target resource will result in an equivalent representation being sent in a 200 (OK) response. However, there is no guarantee that such a state change will be observable, since the target resource might be acted upon by other user agents in parallel, or might be subject to dynamic processing by the origin server, before any subsequent GET is received. A successful response only implies that the user agent's intent was achieved at the time of its processing by the origin server.

4.3.5. DELETE

The DELETE method requests that the origin server remove the association between the target resource and its current functionality. In effect, this method is similar to the rm command in UNIX: it expresses a deletion operation on the

URI mapping of the origin server rather than an expectation that the previously associated information be deleted.

Relatively few resources allow the DELETE method – its primary use is for remote authoring environments, where the user has some direction regarding its effect. For example, a resource that was previously created using a PUT request, or identified via the Location header field after a 201 (Created) response to a POST request, might allow a corresponding DELETE request to undo those actions. Similarly, custom user agent implementations that implement an authoring function, such as revision control clients using HTTP for remote operations, might use DELETE based on an assumption that the server's URI space has been crafted to correspond to a version repository.

4.3.6. CONNECT

The CONNECT method requests that the recipient establish a tunnel to the destination origin server identified by the request-target and, if successful, thereafter restrict its behavior to blind forwarding of packets, in both directions, until the tunnel is closed. Tunnels are commonly used to create an end-to-end virtual connection, through one or more proxies, which can then be secured using TLS (Transport Layer Security,).

CONNECT is intended only for use in requests to a proxy. However, most origin servers do not implement CONNECT.

There are significant risks in establishing a tunnel to arbitrary servers, particularly when the destination is a well-known or reserved TCP port that is not intended for Web traffic. For example, a CONNECT to a request-target of .[ex]ample.com:25"would suggest that the proxy connect to the reserved port for SMTP traffic; if allowed, that could trick the proxy into relaying spam email. Proxies that support CONNECT SHOULD restrict its use to a limited set of known ports or a configurable whitelist of safe request targets.

4.3.7. OPTIONS

The OPTIONS method requests information about the communication options available for the target resource, at either the origin server or an intervening intermediary. This method allows a client to determine the options and/or requirements associated with a resource, or the capabilities of a server, without implying a resource action. An OPTIONS request with an asterisk ("*") as the request-target

applies to the server in general rather than to a specific resource. Since a server's communication options typically depend on the resource, the "*request is only useful as a "ping.or "no-op"type of method; it does nothing beyond allowing the client to test the capabilities of the server. For example, this can be used to test a proxy for HTTP/1.1 conformance (or lack thereof).

A server generating a successful response to OPTIONS SHOULD send any header fields that might indicate optional features implemented by the server and applicable to the target resource (e.g., Allow), including potential extensions not defined by this specification. The response payload, if any, might also describe the communication options in a machine or human-readable representation. A standard format for such a representation is not defined by this specification, but might be defined by future extensions to HTTP. A server MUST generate a Content-Length field with a value of "0if no payload body is to be sent in the response.

4.3.8. TRACE

The TRACE method requests a remote, application-level loop-back of the request message. The final recipient of the request SHOULD reflect the message received, excluding some fields described below, back to the client as the message body of a 200 (OK) response with a Content-Type of "message/http". The final recipient is either the origin server or the first server to receive a Max-Forwards value of zero (0) in the request .

TRACE allows the client to see what is being received at the other end of the request chain and use that data for testing or diagnostic information. The value of the Via header field is of particular interest, since it acts as a trace of the request chain. Use of the Max-Forwards header field allows the client to limit the length of the request chain, which is useful for testing a chain of proxies forwarding messages in an infinite loop.

2.2.6. Response Status Codes

The status-code element is a three-digit integer code giving the result of the attempt to understand and satisfy the request.

HTTP status codes are extensible. HTTP clients are not required to understand the meaning of all registered status codes, though such understanding is obviously

desirable. However, a client MUST understand the class of any status code, as indicated by the first digit, and treat an unrecognized status code as being equivalent to the x00 status code of that class, with the exception that a recipient MUST NOT cache a response with an unrecognized status code.

For example, if an unrecognized status code of 471 is received by a client, the client can assume that there was something wrong with its request and treat the response as if it had received a 400 (Bad Request) status code. The response message will usually contain a representation that explains the status.

The first digit of the status-code defines the class of response. The last two digits do not have any categorization role. There are five values for the first digit:

o 1xx (Informational): The request was received, continuing process

o 2xx (Successful): The request was successfully received, understood, and accepted

o 3xx (Redirection): Further action needs to be taken in order to complete the request

o 4xx (Client Error): The request contains bad syntax or cannot be fulfilled o 5xx (Server Error): The server failed to fulfill an apparently valid request

2.2.7. Overview of Status Codes

The status codes listed below are defined in this specification, Section 4 of [RFC7232], Section 4 of [RFC7233], and Section 3 of [RFC7235]. The reason phrases listed here are only recommendations – they can be replaced by local equivalents without affecting the protocol.

Responses with status codes that are defined as cacheable by default (e.g., 200, 203, 204, 206, 300, 301, 404, 405, 410, 414, and 501 in this specification) can be reused by a cache with heuristic expiration unless otherwise indicated by the method definition or explicit cache controls [RFC7234]; all other status codes are not cacheable by default.

Code	**Reason-Phrase**
100	Continue
101	Switching Protocols

200	OK
201	Created
202	Accepted
203	Non-Authoritative Information
204	No Content
205	Reset Content
206	Partial Content
300	Multiple Choices
301	Moved Permanently
302	Found
303	See Other
304	Not Modified
305	Use Proxy
307	Temporary Redirect
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Payload Too Large
414	URI Too Long
415	Unsupported Media Type
416	Range Not Satisfiable

417	Expectation Failed
426	Upgrade Required
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported

2.2.7.1. Informational 1xx

The 1xx (Informational) class of status code indicates an interim response for communicating connection status or request progress prior to completing the requested action and sending a final response. 1xx responses are terminated by the first empty line after the status-line (the empty line signaling the end of the header section). Since HTTP/1.0 did not define any 1xx status codes, a server MUST NOT send a 1xx response to an HTTP/1.0 client.

A client MUST be able to parse one or more 1xx responses received prior to a final response, even if the client does not expect one. A user agent MAY ignore unexpected 1xx responses.

A proxy MUST forward 1xx responses unless the proxy itself requested the generation of the 1xx response. For example, if a proxy adds an .Expect: 100-continue"field when it forwards a request, then it need not forward the corresponding 100 (Continue) response(s).

2.2.7.2. Successful 2xx

The 2xx (Successful) class of status code indicates that the client's request was successfully received, understood, and accepted.

2.2.7.3. Redirection 3xx

The 3xx (Redirection) class of status code indicates that further action needs to be taken by the user agent in order to fulfill the request. If a Location header field

(Section 7.1.2) is provided, the user agent MAY automatically redirect its request to the URI referenced by the Location field value, even if the specific status code is not understood. Automatic redirection needs to done with care for methods not known to be safe, as defined in Section 4.2.1, since the user might not wish to redirect an unsafe request.

There are several types of redirects:

1. Redirects that indicate the resource might be available at a different URI, as provided by the Location field, as in the status codes 301 (Moved Permanently), 302 (Found), and 307 (Temporary Redirect).

2. Redirection that offers a choice of matching resources, each capable of representing the original request target, as in the 300 (Multiple Choices) status code.

3. Redirection to a different resource, identified by the Location field, that can represent an indirect response to the request, as in the 303 (See Other) status code.

4. Redirection to a previously cached result, as in the 304 (Not Modified) status code.

2.2.7.4. Client Error 4xx

The 4xx (Client Error) class of status code indicates that the client seems to have erred. Except when responding to a HEAD request, the server SHOULD send a representation containing an explanation of the error situation, and whether it is a temporary or permanent condition. These status codes are applicable to any request method. User agents SHOULD display any included representation to the user.

2.2.7.5. Server Error 5xx

The 5xx (Server Error) class of status code indicates that the server is aware that it has erred or is incapable of performing the requested method. Except when responding to a HEAD request, the server SHOULD send a representation containing an explanation of the error situation, and whether it is a temporary or permanent condition. A user agent SHOULD display any included representation to the user. These response codes are applicable to any request method.

2.2.8. HTTPS con SSL

2.3. Protocolos asociados al Correo electrónico (explicacion de todo lo que hace un servidor de correo, y sus protocolos)

2.3.1. ¿Que es el protocolo SMTP?

2.3.2. Recorrido completo de un mail

2.3.3. SMTP con SSL

2.4. Protocolos asociados a la consulta de un sitio (DNS)

2.4.1. ¿Que es el protocolo DNS?

2.4.2. Recorrido completo de un mail

2.4.3. SMTP con SSL

Capítulo 3

(Nuevo) Herramientas a Utilizar

En éste capítulo se verá por qué los protocolos http y smtp son inseguros, introducción al snnifin, spoofing, arp attack

3.1. El problema de los protocolos http y smtp

3.2. Conceptos básicos

3.2.1. Snoofing

3.2.2. Spoofing

3.2.3. Arp attack

3.3. Herramientas utilizadas

3.3.1. Kali Linux

3.3.2. Ettrcap

3.3.3. Wireshark

3.4. Casos de estudio

3.4.1. Caso de estudio: Sniffing de la red para obtener credenciales

3.4.2. Caso de estudio: Sniffing de la red para obtener mails internos/externos

Capítulo 4

Casos de estudio (Explotaciones y soluciones)

4.1. (Seccion va para Herramientas utilizadas)Conceptos básicos

4.1.1. Virtualización

4.1.1.1. Máquinas virtuales

4.1.1.2. Máquinas Docker

4.1.2. Certificación SSL

4.1.2.1. En qué consiste una Certificación SSL

4.1.2.2. Challenges en una Certificación

4.2. Mejorando la seguridad en la navegación - Alternativas

4.2.1. Self-signed Certificates

4.2.2. Internal CA

4.2.3. Estrategia utilizadas, ojala que con let's encrypt

4.3. Estrategia

4.4. CertBot para redes internas

Capítulo 5

Conclusiones y Resultados Obtenidos

Apéndice A

Glosario

A.1. Terminología

Término en inglés	**Traducción utilizada**
argument	argumento
argumentative system	sistema argumentativo
assumption	suposición
atom	átomo
backing	fundamentos
blocking defeater	derrotador de bloqueo
burden of proof	peso de la prueba
claim	afirmación

A.2. Simbología

Símbolo	**Página**	**Significado**
$\neg h$	103	negación fuerte del átomo h

Bibliografía

[1] Bondarenko, A., Dung, P. M., Kowalski, R., and Toni, F. An abstract argumentation–theoretic approach to default reasoning. *Artificial Intelligence 93*, 1–2 (1997), 63–101.

[2] Capobianco, M. El Rol de las Bases de Dialéctica en la Argumentación Rebatible. tesis de licenciatura, July 1999.

[3] Capobianco, M., Chesñevar, C. I., and Simari, G. R. An argumentative formalism for implementing rational agents. In *Proceedings del 2do Workshop en Agentes y Sistemas Inteligentes (WASI), 7mo Congreso Argentino de Ciencias de la Computación (CACIC)* (El Calafate, Santa Cruz, Oct. 2001), Universidad Nacional de la Patagonia Austral, pp. 1051–1062.

[4] Chesñevar, C. I. *Formalización de los Procesos de Argumentación Rebatible como Sistemas Deductivos Etiquetados*. PhD thesis, Departamento de Ciencias de la Computación, Universidad Nacional del Sur, Bahía Blanca, Argentina, Jan. 2001.

[5] Davis, R. E. *Truth, Deduction, and Computation*. Computer Science Press, 1989.

[6] García, A. J. La Programación en Lógica Rebatible: su definición teórica y computacional. Master's thesis, Departamento de Ciencias de la Computación, Universidad Nacional del Sur, Bahía Blanca, Argentina, July 1997.

[7] Haenni, R. Modeling uncertainty with propositional assumption-based systems. In *Applications of uncertainty formalisms*, A. Hunter and S. Parsons, Eds. Springer-Verlag, 1998, pp. 446–470.