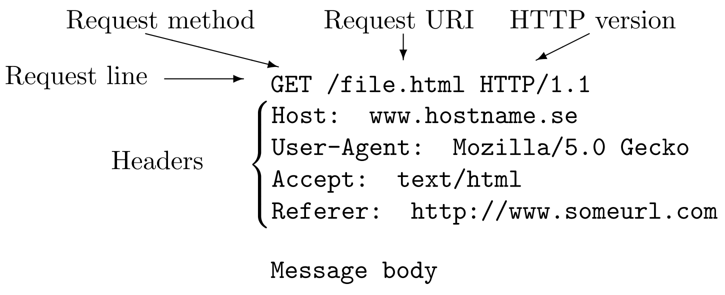
# Lecture Notes for Web Security 2019
Part 1 — HTTP Security

Martin Hell

## 1 The HTTP Protocol

HTTP stands for *HyperText Transfer Protocol* and is the most common protocol for transporting web pages, but also to submit information to web servers in e.g., forms. It is a client server protocol, so in every communication instance there is one party acting as client and one acting as server. The server, the one hosting the data, is called *origin server*, and the client is called *user agent*. The user agent is typically a web browser, but can also be e.g., a spider indexing web pages. The protocol does not require a specific transport layer, but the most common is the *Transmission Control Protocol* (TCP). There are two types of messages in HTTP. The client sends *request* messages to the server and the servers replies with a *response* message. The request message can use one of eight different methods defined in the standard, namely GET, HEAD, POST, PUT, DELETE, TRACE, CONNECT and OPTIONS.

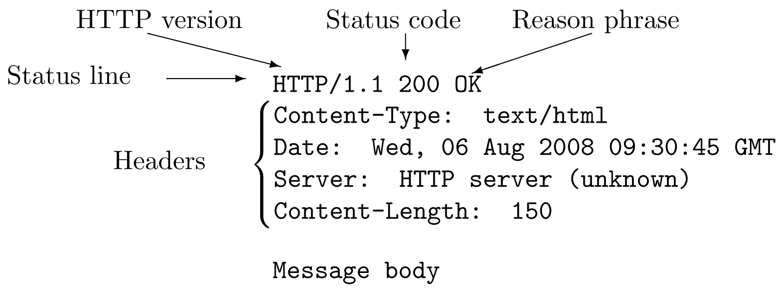
The most common are GET and POST, but it can be noted that only GET and HEAD must be implemented by servers, while the rest are optional. GET is used to retrieve information identified by the request-URI, e.g., a web page, an image or a pdf file. POST is used to submit data to a server and this data is to be handled by the resource specified by the request-URI. An example of a



```
GET /file.html HTTP/1.1
Host:  www.hostname.se
User-Agent:  Mozilla/5.0 Gecko
Accept:  text/html
Referer:  http://www.someurl.com

Message body
```

Figure 1: Example of an HTTP request sent from client to server.



```
HTTP/1.1 200 OK
Content-Type:  text/html
Date:  Wed, 06 Aug 2008 09:30:45 GMT
Server:  HTTP server (unknown)
Content-Length:  150

Message body
```

Figure 2: Example of an HTTP response sent from server to client.

GET request message sent from a client to a server is given in Figure 1.

The first line is the *request line*. It consists of three parts, the *request method*, the *request URI*, and the *HTTP version*. The method is one of the eight methods defined in the standard, the URI is the file on the server which is requested and the version states which HTTP version is supported by the client. The following lines are *headers*. After the headers, follows an empty line which indicates the beginning of the message body. The message body is data sent from the client to the server. It is not commonly used together with GET even though it would be possible.

An example of a response message is given in Figure 2. The first line is the *status line*. It states the *HTTP version* that is used in the response, a numeric *status code* together with the *reason phrase* which is the text representation of the status code. The status code is a 3-digit integer used to inform the client how the request was handled on the server.

## 1.1 Safe and Idempotent Methods

Idempotent methods are methods which can be sent several times with the same result as sending the method only once. There is a distinction between idempotent methods and safe methods in that a safe method should not have any side-effects at all. Sending a request with a safe method should only result in the server sending a response, without performing any action in the background. Note that by these definitions it follows that any safe method must also be idempotent. Examples of safe methods are GET and HEAD, while idempotent methods additionally include PUT and DELETE, OPTIONS and TRACE.

It is in particular important that application developers are aware of the distinction between GET and POST, as GET is both safe and idempotent whereas POST is neither. This is most apparent when information is sent to the server using one of the two methods. Just because GET is a safe and idempotent method does not mean that it cannot have side-effects on the server. Instead, it means that the application developer must make sure that data sent using GET does not have any side-effects on the server. Actually, to be more specific, it means that the application developer can not hold users responsible for any side-effects caused by (multiple) GET requests. On the other hand, if POST is used the web browser can notify the user that it is about to make a POST request and that this may have side-effects on the server, e.g., putting items in a cart, sending an email, signing in to a service etc.

A request using GET can be stored in browser history, be cached, bookmarked, and shared with others so it is usually a good idea to not allow these requests to have side-effects.

## 1.2 Headers in HTTP

There are many possible headers defined in the HTTP standard. Some request headers are

- Host. This specifies which host the request is directed to. It is possible that a single IP address is responsible for several host names and this header then specifies which one the request is targeting.
- User-Agent. The name of the user-agent. This field is primarily used to gather statistics about user-agents, but can also be used to show certain pieces of information to some user-agents only. Some web browsers may not support certain functionality and it is thus possible for the server to create the response based on this knowledge.
- Accept. Used to specify which media types the response is allowed to use. Typical examples are text/html, text/plain, image/jpeg, image/gif, etc. A quality factor, or qvalue, represented by a $q$ can be included to indicate which type is preferred if several are accepted. The value of $q$ should be between 0 and 1, with 1 as the highest priority.
- Accept-Language. A list of languages that are accepted in the response. A qvalue can be used for preference.
- Accept-Encoding. A list of encodings that are accepted. Identity, meaning "no transformation" is always implicitly accepted. Additionally it is possible to accept any of the compression formats deflate, compress and gzip. A qvalue can be used for preference.
- Accept-Charset. A list of character sets that are acceptable. A * can be used to include all character sets. Again, qvalue can be used for preference.

Some response headers are

- Transfer-encoding. How the message has been encoded. The value "chunk" means that the message has been divided into several chunks. In this case the server can start sending data even though all data is not available yet. Several transfer encodings can be combined, e.g., both gzip and chunk can be used at the same time. In that case the message is first compressed with gzip and then sent in chunks.

- Date. The date that the message was sent.
- Server. This contains information about the server that handles the request.

Two headers that will be covered in detail in Section 4 are the WWW-authenticate and the Authorization headers. Note that there are several other headers possible that are not part of the HTTP standard. Examples are the X-powered-by header which is used by some servers to send information about which version of PHP or ASP that has been used to generate the web page.

HTTP/1.0 is described in RFC 1945 [2]. HTTP/1.1 is described in RFC 2616 [5]. The current version is HTTP/1.1. Some important differences are:

- The request header "host" is required in HTTP/1.1, while in HTTP/1.0 it is optional.
- Persistent connections are used by default in HTTP/1.1. In HTTP/1.0 each request with corresponding response uses one TCP connection by default, even though it is possible to negotiate persistent connections. Since TCP uses a 3-way handshake, using persistent connections saves resources on both the client, the server and the network. It allows the client to make several consecutive requests to the server without waiting for each response. These are called pipelined requests. Note that even if the client uses pipelined requests, the server must send its responses in the same order as the corresponding requests were sent. Also note that non-idempotent methods, i.e., POST, should not be pipelined.

### 1.3 URL Encoding

A URL is a URI with the additional property that the URL specifies the location of a resource. The terms URI and URL are sometimes used interchangeably to refer to basically the same thing, and some confusion may arise. A URI (*Uniform Resource Identifier*) identifies a resource either by name or by location, or both. A URL (*Uniform Resource Locator*) is a special type of URI which always gives the location of a resource. The URIs used in HTTP are URLs since they point to an address where the resource can be found. Thus, both names can be used even though many technical specifications have adopted the more generic term URI. The term URL encoding will be used here as it is commonly used even though it applies to the general URIs as well.

When data is sent to the server in a GET request, the data is given as a part of the URL. Since there are several characters that have special meaning in URLs, there is a potential problem when these characters are sent as data. Should the server treat them as data or as a metacharacter in the URL? This problem is addressed by encoding these characters into characters that always are treated as data. This encoding scheme is called URL encoding or percent encoding since the encoding is done by replacing the special characters by %XY where XY is the ASCII value of the character in hexadecimal notation. Some examples of characters that need to be encoded are given in Figure 3.

| Examples | | | | |
|---|---|---|---|---|
| **Char** | **ASCII** | **Binary** | **Enc** | **URL meaning** |
| + | 43 | 00101011 | %2B | Whitespace character |
| / | 47 | 00101111 | %2F | Separating directories and subdirecories |
| ? | 63 | 00111111 | %3F | Separates the path and the query string in the URL |
| % | 37 | 00100101 | %25 | Start of URL encoded character |
| # | 35 | 00110101 | %23 | Start of fragment defining a bookmark in HTML |
| & | 38 | 00111000 | %26 | Separate parameters sent to server |
| <space> | 32 | 00110010 | %20 | A literal space should not be present in a URL |

Figure 3: Example of characters that need to be encoded in URLs.

Note that a whitespace character can be represented either using + or using the URL encoding %20. If the string "Alice & Bob?" is sent using URL encoding it is encoded as "Alice%20%26%20Bob%3F" or "Alice+%26+Bob%3F".

Sometimes characters outside the ASCII character set need to be used. Then the Unicode character set can be used. There are many characters in the Unicode set and since they can not be represented as one byte, a way of transforming these characters to bytes is needed. There are several standards for this and a variant of *Unicode Transformation Format* (UTF) is the standard way of doing this transformation. UTF-8 and UTF-16 are the most common, and UTF-8 is the one most widely used in networks. As the names imply, UTF-8 transforms the characters into a sequence of bytes, while UTF-16 transforms the characters into a sequence of 16-bit words. An advantage of UTF-8 is that it is backwards compatible to ASCII, i.e., these characters are represented in the same way in UTF-8 as in ASCII. The encoding rules of UTF-8 are given by

| | | | | |
|---|---|---|---|---|
| U000000 - U00007F: | 0xxxxxxx | | | |
| U000080 - U0007FF: | 110xxxxx | 10xxxxxx | | |
| U000800 - U00FFFF: | 1110xxxx | 10xxxxxx | 10xxxxxx | |
| U010000 - U10FFFF: | 11110xxx | 10xxxxxx | 10xxxxxx | 10xxxxxx |

## 1.4 Sending Data with GET and POST

Both the GET and POST methods can be used to send data to the server. However, data is sent differently depending on the method used. A common way to send data to the server is to use a *HTML form*. In HTML, the method is defined by the method argument to the form tag.

```
<form method="GET" action="submit.php">
First Name <input type="text" size="12" name="Fname"><br /><br />
Last Name <input type="text" size="12" name="Lname"><br /><br />
<input type=submit value="Send">
</form>
```

In the example above, the *method* argument is *GET*. Thus, when the form is submitted the browser will send the data in a GET request. By changing this argument to *POST*, the browser will use the POST method instead.

The resulting GET request, given below, does not contain a message body. Instead the data is sent as a part of the URL. The "?" is used to separate the path and the query string part of the URL.

```
GET /index.html?Fname=John&Lname=Doe HTTP/1.1
Host:  www.server.com
User-Agent:  Mozilla/5.0 Gecko
Accept:  text/html
```

In the POST request, given below, the data is sent in the message body.

```
POST /submit.php HTTP/1.1
Host:  www.server.com
User-Agent:  Mozilla/5.0 Gecko
Accept:  text/html
Content-Type:  application/x-www-form-urlencoded
Content-Length:  20

Fname=John&Lname=Doe
```

In the requests, the characters ? and & are used the way they are intended to in the HTTP protocol, and are thus not URL encoded.

## 2 Cookies

A cookie is a piece of text stored by the user agent. A server sends cookies in the response header and when a webpage is requested any cookie with the same domain as that web page is sent in the request header. It is also possible to use JavaScript to set and send cookies.

Cookies are mainly used for two reasons, session management and user tracking. First party cookies are cookies that belong to the same domain as the visited webpage. Third party cookies are cookies set by another domain. It is not possible to set a cookie for a domain which is not the one that the cookie comes from. Browsers are required to reject these cookies.

There are two types of cookies, persistent cookies and session cookies. A persistent cookie has an expiration date and the browser will store the cookie on disk and not delete it until the date has expired. It is of course possible to manually delete it in all common web browsers. Also, the server can delete the cookie by resending the same cookie value but changing the expiration date to a past date. A session cookie does not have an expiration date. Instead the cookie is deleted by the browser as soon as the browser is shut down. These cookies can be used to keep a user logged in to a site. At successful login the user receives a cookie and as long as this cookie is sent with requests the user stays logged in. It also allows the server to keep track of items put in a shopping cart when the application is a webshop.

## 2.1 Fields in a Cookie

The use of cookies was first specified by Netscape around 1995. This preliminary specification can be found at [12]. It was later proposed as a standard in RFC 2109 [10], updated in RFC 2965 [11] and again updated in RFC 6265 [1]. The HTTP response header *set-cookie* is used to send a cookie from a server to a client. The only required field in the set-cookie header is the name field. This field sets a name and a value of the cookie. Additionally, a *domain* field can be set. This defaults to the domain that is sending the cookie. Only a host belonging to a certain domain can set a cookie for that domain. In order to limit the cookie to only be sent to paths with a given prefix, the *path* field can be used. The last field that is common to all specifications is the *secure* field. If this field is set, the cookie will only be sent in an SSL session. There is also a field that determines when a cookie will be deleted by the browser. This field is what separates session cookies from persistent cookies. In the Netscape specification this is the *expire* field. It sets the date when the cookie expires, using the format `Wdy, DD-Mon-YYYY HH:MM:SS GMT`. An alternative field, introduced in the RFCs is *Max-Age* which sets the lifetime of the cookie in seconds. Finally, the field *HttpOnly* can be used to make sure that the cookie can only be accessed in the HTTP header, and not by e.g., a JavaScript.

Moreover, RFC 2965 defines the headers *cookie2* and *set-cookie2*. The set-cookie2 header includes an extended list of optional fields. The cookie2 header can be used by user agents to announce that it supports set-cookie2 cookies. However, these headers were again obsoleted by RFC 6265.

## 2.2 User Tracking with Third Party Cookies

Using ads on a website is very common and can help financing the website. To be as efficient as possible an ad should reflect the interests of the viewer. Tracking users across websites using third-party cookies is a common way to record user behaviour. It requires that the tracking, or advertising, company has some content, e.g., an advertisement, on each website that is used as a basis for tracking, see Figure 4. It does of course not have to be the same ad on all websites, the only requirement is that the same user is sending an HTTP request to the same company (www.ad.com), with the same cookie, from each

Webpage 1
Ad from www.ad.com
Webpage 2
Ad from www.ad.com
Webpage 3
Ad from www.ad.com

Figure 4: An advertisement from www.ad.com

different web page. If the request includes information about the webpage the advertisement company can collect this information and map it to the cookie sent by the user.

## 2.3 User Tracking with First Party Cookies

As third party cookies are mainly used for tracking users, and users have no particular interest in being tracked by companies they have never heard of, more and more people block third party cookies in their web browser. However, the tracking is not only used to display ads, it can also be used to help developers understand which part of their own website is most often viewed. Maybe some important information is difficult to find because of a complex website structure, then tracking user behaviour can help clarify this information. Using first party cookies is a simple way to gather this information. However, two problems still remain. First, what if the website is split over several domains? Then the same cookie can not be used for all domains. Second, it is not possible for third parties to do this tracking. Small companies may not have neither the money nor the knowledge to implement this themselves. It would be more cost-efficient to hire this tracking service from a third party.

It turns out that first party cookies can be used in both situations. The solution is to use a JavaScript that sets and reads the cookies. The involved parties are the user who is being tracked, the web server who wants to track its users and a third party analyzer that provides the tool for tracking the user. User tracking with first party cookies can then be summarized in the following steps, see Figure 5.

1. The third party analyzer develops a JavaScript that includes functionality for setting and reading cookies and collecting user information. This JavaScript file is sent to the web server.

2. A user makes an HTTP request for a resource on the web server.

3. The web server sends a response to the user, returning the requested resource. Included in the response is the JavaScript.

4. The user's web browser executes the JavaScript. Since the JavaScript comes from the web server, it has the ability to set a first party cookie. If a cookie is already set it is read and the request can be linked to other

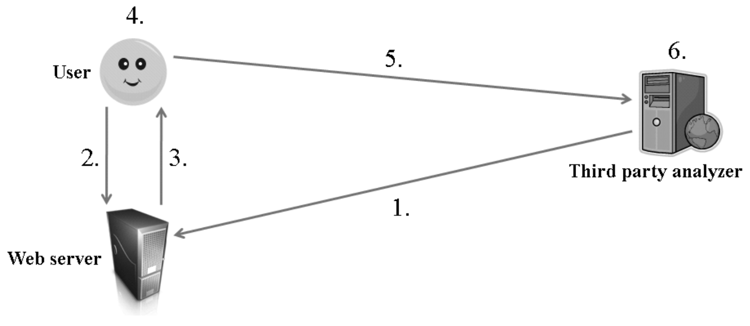


Figure 5: User tracking with first party cookies.

requests from the same user. The JavaScript can also gather other information from the user, e.g., which web browser that is used or which language settings that are used.

5. The information gathered is sent to the third party analyzer. This is achieved by requesting a resource from the third party, e.g., a $1 \times 1$ pixel image, using GET. The information is sent in the URL as described in Section 1.4.

6. The information is collected, analyzed and sorted by the third party analyzer. An easy to read summary is ready to be presented to the web server administrators.

This tracking can also be done over several domains provided that the administrator(s) of the domains include cookie information in the links between the domains.

## 3 Base64 Encoding

Not all ASCII characters are printable. In some situations there is a need to transport data over a network in a printable form. One example could be when a digital certificate is to be distributed.

The most common type of encoding used in this situation is base64. It is described in RFC 4648 [9]. As the name indicates, the bytes sent over a network are transformed into 6-bit chunks, ($2^6 = 64$). Each 6-bit chunk is then written as one of 64 characters. The alphabet used in base64 is

```
ABCDEF...XYZabcdef...xyz0123456789+/
```

In other words, the 6-bit string `0000000` is encoded as `A`, the string `000001` is encoded as `B` and so on.

| | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Text** | c | | | | | | | | a | | | | | | | | t | | | | | | | |
| **ASCII** | 99 | | | | | | | | 97 | | | | | | | | 116 | | | | | | | |
| **Binary** | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| **6-bit index** | 24 | | | | | | 54 | | | | | | 5 | | | | | | 52 | | | | | |
| **base64** | Y | | | | | | 2 | | | | | | F | | | | | | 0 | | | | | |

Figure 6: The base64 encoding of “cat”

| | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Text** | s | | | | | | | | *padding* | | | | | | | | *padding* | | | | | | | |
| **ASCII** | 115 | | | | | | | | −− | | | | | | | | −− | | | | | | | |
| **Binary** | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **6-bit index** | 2B | | | | | | 4B | | | | | | −− | | | | | | −− | | | | | |
| **base64** | c | | | | | | w | | | | | | = | | | | | | = | | | | | |
| **Text** | s | | | | | | | | ? | | | | | | | | *padding* | | | | | | | |
| **ASCII** | 115 | | | | | | | | 63 | | | | | | | | −− | | | | | | | |
| **Binary** | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **6-bit index** | 2B | | | | | | 51 | | | | | | 60 | | | | | | −− | | | | | |
| **base64** | c | | | | | | z | | | | | | 8 | | | | | | = | | | | | |

Figure 7: The base64 encodings of “cats” and “cats?”

In the encoding, the data is divided into chunks of 24 bits since the least common multiple lcm(8,6)=24. The 24-bit chunk is divided into four 6-bit chunks, which are each independently encoded using base64. As an example, the encoding of the word “cat” is given in Figure 6.

Of course, when the base64 encoded characters are transmitted over the network, or saved in a file, they are saved as bytes. This results in an overhead of at least 33%, (4/3). As long as the data to be encoded is a multiple of three bytes, the procedure given above works well. However, if the length of the data is not a multiple of three bytes, then this has to be indicated in the encoded string. Just padding the data with zeros will not suffice. We would not be able to distinguish between “a”, “a NULL” and “a NULL NULL”. Instead, the data is padded with zeros and the character “=” is used at the end of the base64 encoded string to indicate the number of padded bytes in the data string. As an example, the last four base64 characters of the two strings “cats” and “cats?” are given in Figure 7.

As shown, the string “cats” is encoded as “Y2F0cw==” and the string “cats?” is encoded as “Y2F0cz8=”. In these two cases the overhead is larger than 33%. In the first case it is 100% and in the second it is 60%.

## 3.1 Related Encoding Schemes

Base64 is not the only proposed encoding scheme. Variants with smaller alphabets have been proposed. Base32 and base16 are very similar to base64, but consists of 32 and 16 characters respectively, increasing the overhead.

OpenPGP, described in RFC 2440 [3], specifies an encoding scheme called Radix-64. This is the same encoding scheme as used in base64, but with an additional 24-bit checksum.

# 4 Authentication in HTTP

The HTTP/1.0 specification, given in RFC 1945 [2], includes a specification for a built-in authentication scheme. This is called *Basic Access Authentication* and has a serious weakness in that the password is sent in clear. An improved authentication scheme was given in RFC 2069 [7], called *Digest Access Authentication*. This was introduced together with the first version of the HTTP/1.1 specification, RFC 2068 [4]. When the HTTP/1.1 specification was updated in RFC 2616 [5], the Digest Access Authentication was also updated including several new features. Both Basic Access Authentication and Digest Access Authentication, intended to be used in HTTP/1.1, are described in RFC 2617 [6]. This section will describe these authentication methods. Both variants of the Digest Access Authentication will be given in order to highlight the differences and to motivate the new components added in RFC 2617.

## 4.1 Basic Access Authentication

Basic Access Authentication provides a method to authenticate users to a web server. The functionality is included in the HTTP protocol and is very simple to enable in e.g., Apache. Basic Access Authentication is very insecure since the password is sent to the server in plaintext. This is the main reason not to use this type of authentication unless an underlying secure protocol is used, e.g., SSL.

If a directory on a webserver requires authentication before access is granted, this information is sent in an HTTP response with status code 401 and reason phrase "Authorization Required". The type of authentication is indicated by the *WWW-Authenticate* header as shown below.

```
HTTP/1.1 401 Authorization Required
Content-Type:  text/html
Date:  Wed, 06 Aug 2008 09:30:45 GMT
Server:  HTTP server (unknown)
WWW-Authenticate:  Basic realm=Private"
```

The string "Basic" in the WWW-Authenticate header simply indicates that it is Basic Access Authentication that is required to access the directory. The realm, also included in the WWW-Authenticate header, is used to distinguish between directories in case there are different password files corresponding to different directories. The webserver can e.g., allow one set of users access to directory `/www/dirA` and another set of users access to directory `/www/dirB`. A user with access to both directories may have different passwords for each. In this case, the realm can be used to indicate to the user which password to submit.

When the user-agent receives the "401 Authorization Required" response, it provides the user with a dialog box and acquires the name and password. This is submitted in a new HTTP request, similar to the original request. The new request includes the *Authorization* HTTP header containing the word Basic together with the string *name:password* encoded with base64, as shown below.

```
GET /protected/index.html HTTP/1.1
Host:  www.server.com
User-Agent:  Mozilla/5.0 Gecko
Accept:  text/html
Authorization:  Basic bmFtZTpwYXNzd29yZA==
```

Upon receiving the name and password, the webserver opens a password file, or queries a database, containing a list of names and the corresponding passwords. If there is a match, the user is authenticated and the requested webpage is returned. The passwords are stored in the password file in hashed form, using a hash algorithm chosen by the server. This procedure is very similar to the login functionality used by operating systems such as Linux and Windows. However, there is a fundamental difference in the way the passwords are submitted to the verifier. The password in Basic Access Authentication is sent over a public network, possibly to the other side of the world, passing many routers on the way. An eavesdropper could be located on any of these routers. If the user is connecting to the network using an unencrypted wireless connection, additionally anyone nearby with a wireless network card can listen to the traffic.

Encoding the name and password with base64 does not provide any cryptographic protection and in the context of confidentiality it can be seen as the string is sent in cleartext. An eavesdropper that intercepts the message will immediately recover the name and password and can use these to impersonate the user.

After authentication, when the user requests additional webpages within the same realm, the correct Authorization header can be sent pre-emptively so that the user does not have to enter this again.

## 4.2 Digest Access Authentication, RFC 2069

The first version of Digest Access Authentication was given in RFC 2069. The protocol intends to address the problem of sending the password in cleartext. Due to a number of discovered weaknesses, the Digest Access Authentication protocol was updated. The improved version is described in Section 4.4.

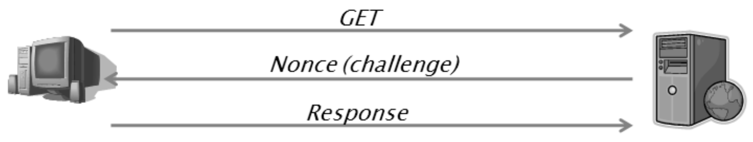


Figure 8: A challenge response protocol as used in HTTP authentication.

Digest Access Authentication is a challenge-response protocol, see Figure 8. In its most simple description, the server sends a random challenge to the client. The client takes the challenge and the secret password and computes a new string and sends the result back to the server. The server, knowing the password, computes the same string and verifies that the answer is correct.

Similar to Basic Access Authentication, the protocol is initiated when a client sends an HTTP request, without Authorization header, requesting a resource in a protected directory. The server responds with a “401 Authorization Required” and WWW-Authenticate header as shown below. The header consists of the word “Digest” followed by the challenge. The challenge consists of the mandatory parameters *realm* and *nonce.* Additionally, the optional parameters *domain, opaque, stale* and *algorithm* can be used. For more details about the optional parameters, refer to RFC 2069. It should be noted though, that if algorithm is not specified, the “MD5” hash function is assumed. The realm is used to inform users which username and password to use, but it is also used as a salt to the username and password when these are stored in hashed form on the server. The nonce is a random string generated by the server. An example of an HTTP response is given below.

```
HTTP/1.1 401 Authorization Required
Content-Type:  text/html
Date:  Wed, 06 Aug 2008 11:39:03 GMT
Server:  HTTP server (unknown)
WWW-Authenticate:  Digest realm="Private",
     nonce="ab99827112d52f0e8411d0f632af0ed62f"
```

Upon receiving the “401 Authorization Required” response, the user-agent prompts the user for the username and password and then sends another request with an “Authorization” header. This header consists of the word “Digest” followed by the digest-response. The digest response includes *username, realm, nonce, uri* and *response.* Additionally it can include a *digest* field which can be used to authenticate the body, e.g., a POST body. If the *opaque* field is used by the server this has to be returned by the client. An example of a request is given below.

```
GET /private/index.html HTTP/1.1
Host: www.server.com
User-Agent: Mozilla/5.0 Gecko
Accept: text/html
Authorization: Digest username="William",
     realm="Private",
     nonce="ab99827112d52f0e8411d0f632af0ed62f",
     uri="/private/index.html",
     response="f736c93279242ba4e4988ee2a7c1276d"
```

The response field is computed as

$$H(H(A_1) : \text{nonce} : H(A_2))$$

where $H$ is the MD5 hash function, unless something else has been negotiated, and $A_1$ and $A_2$ are given by

$$\begin{aligned} A_1 &= \text{username} : \text{realm} : \text{password} \\ A_2 &= \text{method} : \text{URI} \end{aligned}$$

Assuming that the password in the example above is "Wallace", the response would be computed as

$$\begin{aligned} A_1 &= \text{William:Private:Wallace} \\ A_2 &= \text{GET:/private/index.html} \\ \text{response} &= \text{MD5}(\text{MD5}(A_1)\text{:ab99827112d52f0e8411d0f632af0ed62f:MD5}(A_2)) \end{aligned}$$

The server stores the value MD5($A_1$) and uses this to compute the expected response. If the received response equals the expected response the authentication is successful and the requested URI is returned in an HTTP response. Using this protocol, the password is never sent in clear over the network. The password is also not stored in clear text on the server, only a hash is stored. However, it is important to note that it is possible to impersonate a user without knowing the password. Only MD5($A_1$) is needed to authenticate, i.e., the value stored on the server. This problem is somewhat compensated by the fact that the realm is included in the hashed string. Even if a user has the same username and password on another server, the realm will most likely be different, and thus also MD5($A_1$). Still, Basic Access Authentication does not suffer from this problem.

## 4.3 Security Remarks

While Digest Access Authentication as described in RFC 2069 is more secure than Basic Access Authentication, it has a few shortcomings.

Consider a replay attack, in which an adversary replays an old message. In Basic Access Authentication this attack is devastating as a replay implies that the adversary has access to the data that was sent, i.e., the password. This in turn implies that any webpage in the protected area of the server is accessible to the attacker. In Digest Access Authentication, this is not the case. The string $A_2$ included in the computation of the response consists of method and URI. Hence, a response is unique for a certain webpage. If the attacker can capture the response for later replay, it is reasonable to assume that the message body, i.e., the web page content, in the HTTP response can also be captured. As GET is an idempotent method a replay should not have any consequences on the server side and the attacker should not get any additional information. However, it is possible that the content of the web page has changed between the original request and the replay. Moreover, if the method POST is used, a replay can have serious unwanted consequences also on the server side. It could e.g., be possible to replay a money transaction. Replay attacks can be prevented by only allowing a nonce to be used once. In fact, the word *nonce* is a short variant of *number used once*. This will require the server to remember which nonces have been used before, causing overhead. A simpler variant is to include a timestamp in the nonce. Then the replay is only possible until the nonce expires.

Next, consider a man-in-the-middle attack. In this attack scenario, it is assumed that the adversary has the ability to intercept traffic, modify it, and then send to the recipient. One possible way to mount a man-in-the-middle attack is to offer a free proxy service to users. A powerful attack in this scenario could be that the adversary intercept the "401 Authorization Required" HTTP response sent by the server. The WWW-Authenticate field is then changed from Digest to Basic, which will result in the password being sent in clear from the client to the adversary. If the user knows that Digest Access Authentication should be used this can be detected by the user, provided that the user agent informs the user that Basic Access Authentication is taking place. The user agent can also remember if Digest Access Authentication has previously been used on a particular server, and warn the user if it uses Basic Access Authentication at a later time. Another variant of a man-in-the-middle attack is that the adversary changes the nonce. If the nonce can be chosen by the adversary, a large amount of the input to the digest function can be known in advance. It would e.g., be possible to mount a time-memory tradeoff attack [8] on the password together with a few thousand possible combinations of usernames, realms, methods and URIs.

## 4.4 Digest Access Authentication, RFC 2617

The version of Digest Access Authentication described in RFC 2617 include a few security enhancements targeting the potential problems discussed in Section 4.3. New fields in both the WWW-Authenticate header in the server response and the Authorization header in the client request are added. In the WWW-Authenticate header, the field *qop* (quality of protection) is added. Two different types, *auth* and *auth-int* are defined. The auth value indicates authentication, while the auth-int value indicates authentication with integrity protection. The server announces which option(s) it supports. The same field is used in the Authorization header, but in this case the client determines which option it decides to use. The strings $A_1$ and $A_2$ are computed as

$$A_1 = \text{username : realm : password}$$
$$A_2 = \begin{cases} \text{method : URI} & \text{if qop=auth} \\ \text{method : URI : MD5(entity-body)} & \text{if qop=auth-int} \end{cases}$$

Two other fields that are used in this version are the *cnonce* (client nonce) and the *nc* (nonce-count) fields. The cnonce is a random string generated by the client and the nc is a counter, starting at 1 and incremented for every request. The response is computed as

$$H(H(A_1) : \text{nonce} : \text{nc} : \text{cnonce} : \text{qop} : H(A_2)).$$

This improved version of Digest Access Authentication is backwards compatible with the one proposed in RFC 2069. This is achieved using the *qop* directive. If the server does not send a qop directive in the WWW-Authenticate header, cnonce and nonce-count must not be used by the client. If the client responds with a qop value, it must also send a cnonce and a nonce-count.

## 4.5 Security Remarks

While a replay attack can be detected to a certain extent using the first version of Digest Access Authentication, the protection is more explicit in this version. The server basically only needs to keep track of the nonce-count and make sure that this value is incrementing for each request. Since the value is included in the computation of the response it can not be altered without detection. Thus, the protection against replay attacks is better in this version.

Next, consider time-memory tradeoff attacks used to recover a password. If most information about the input to the response computation function is known, then this attack can be considered feasible. The cnonce completely prevents this attack as this value is generated by the client and used in the computation of the response.

# Exercises

**Exercise 101** *Why is the "host"-header mandatory in HTTP/1.1?*

**Exercise 102** *In HTTP, is a safe method always necessarily idempotent? Why or why not?*

**Exercise 103** *What is the UTF-8 encoding of the Unicode character '€' with (hex) number 20AC?*

**Exercise 104** *If you decode the Base64 encoded string*

"QWxsIHlvdXIgYmFzZSBhcmUgYmVsb25nIHRvIHVzLg==",

*how many bytes will the resulting string be?*

**Exercise 105** *What is the Base64 encoding of "Pear"?*

**Exercise 106** *Which word has the Base64 encoding "QXBwbGU="*

**Exercise 107** *How is a cookie sent from the server to a client?*

**Exercise 108** *Why must a cookie be sent to the client before the starting html tag (<html>) is sent?*

**Exercise 109** *What is the difference between first and third party cookies?*

**Exercise 110** *How does user tracking with first party cookies work? What are the limitations? How can this be done over several domains?*

**Exercise 111** *The realm has no security purpose in the Basic authentication of HTTP. In which aspect does the realm increase the securiy in Digest authentication?*

**Exercise 112** *Assume that we want to create tables for a time-memory tradeoff attack on Digest Access Authentication as defined in RFC 2069. The precomputation time $P$ is given by $P = N$, where $N$ is the total search space. The tradeoff curve is given by $N^2 = M^2T$, where $M$ is the memory and $T$ is the realtime phase computation time. Assume that an attacker can control the nonce that is sent by the server.*

*a) The attacker wants to recover the password of user "Murron" in the realm "Scotland". A GET request to* `/index.html`*, together with the Authorization header, has been obtained from this user. The attacker assumes that the password is at most 8 characters (uppercase, lowercase and/or numbers). The amount of memory that is available to the attacker is 128 GB and 32 Bytes are needed for each pair of startpoints and endpoints. Determine the precomputation time and the realtime complexity for the attack.*

*b) What would be the complexity of a brute force attack in the scenario above? What is the usefulness of the attack above?*

*c) Another attacker wants to cover* 256 *usernames and* 128 *realms by the tables. The attacker assumes that a GET request to the resource* `/index.html` *will be very common, and that at least some users have a password of* 6 *characters or less. The attacker has 1 TB of memory this time. Determine the precomputation time and the realtime complexity for the attack.*

*d) What would be the complexity of a brute force attack in the scenario above? What is the usefulness of the attack above?*

*e) If a cnonce is used as specified in RFC 2617, what will happen to the attack complexities in a–d?*

**Exercise 113** *One header defined in the HTTP/1.1 standard is the Content-MD5 header. It can be used to include an MD5 hash value of the message body. An MD5 hash of the message is also included if the qop type auth-int is used with Digest Access Authentication (RFC 2617). Compare the integrity protection given by the two alternatives.*

**Exercise 114** *When a client has submitted a valid authorization header in Digest Access Authentication the server responds with the requested web page. This response includes an authentication-info header which can optionally include the field rspauth. This is computed by the server in the same way as the computation of the response made by the client, with the only difference that $A_2$ is given by ":URI" instead of "method:URI". What is the purpose of the rspauth field?*

## References

[1] A. Barth. HTTP State Management Mechanism. RFC 6265 (Proposed Standard), April 2011. Available at: *http://www.ietf.org/rfc/rfc6265.txt.*

[2] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945 (Informational), May 1996. Available at: *http://www.ietf.org/rfc/rfc1945.txt.*

[3] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. OpenPGP Message Format. RFC 2440 (Proposed Standard), November 1998. Obsoleted by RFC 4880, Available at: *http://www.ietf.org/rfc/rfc2440.txt.*

[4] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2068 (Proposed Standard), January 1997. Obsoleted by RFC 2616, Available at: *http://www.ietf.org/rfc/rfc2068.txt.*

[5] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266, Available at: *http://www.ietf.org/rfc/rfc2616.txt.*

[6] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617 (Draft Standard), June 1999. Available at: *http://www.ietf.org/rfc/rfc2617.txt.*

[7] J. Franks, P. Hallam-Baker, J. Hostetler, P. Leach, A. Luotonen, E. Sink, and L. Stewart. An Extension to HTTP : Digest Access Authentication. RFC 2069 (Proposed Standard), January 1997. Obsoleted by RFC 2617, Available at: *http://www.ietf.org/rfc/rfc2069.txt.*

[8] M.E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory*, IT-26(4):401–406, July 1980.

[9] S. Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648 (Proposed Standard), October 2006. Available at: *http://www.ietf.org/rfc/rfc4648.txt*.

[10] D. Kristol and L. Montulli. HTTP State Management Mechanism. RFC 2109 (Historic), February 1997. Obsoleted by RFC 2965, Available at: *http://www.ietf.org/rfc/rfc2109.txt*.

[11] D. Kristol and L. Montulli. HTTP State Management Mechanism. RFC 2965 (Historic), October 2000. Obsoleted by RFC 6265, Available at: *http://www.ietf.org/rfc/rfc2965.txt*.

[12] Netscape. Persistent client state http cookies. Available at: *http://curl.haxx.se/rfc/cookie_spec.html*.