Web Security

Notes for Computer Security

Also part of exam revision

Contents

Web Basics

URLs

A web browser identifies a website with a *uniform resource locator*. They are follow this format:

```
Protocol://host:port/path?arg1=val1&arg2=val2#statement
```

- `Protocol`:
  - The protocol to access resource (`ftp`, `https`, `http`, etc).
- `host`
  - The domain/IP address of the server storing the resource.
- `path`
  - The path to the resource on the server, from the root.
- `args`
  - Resources can be static of dynamic, and dynamic content usually requires arguments to process

`HTTP`

After establishing a TCP connection to the web server, the browser sends `HTTP` requests to that server.

`HTTP` requests

A typical `GET` request may look like something of the following

```
GET / HTTP/1.1
Host: www.inf.ed.ac.uk
User-Agent: Mozilla/5.0
                        (X11; Ubuntu; Linux x86 64; rv:29.0)
                        Gecko/20100101 Firefox/29.0
Accept: text/html,application/xhtml+xml,
                application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://www.google.com
Connection: keep-alive
```

`HTTP` requests begin with a request line (`GET`/`POST`).

`HTTP` responses

```
HTTP/1.1 200 OK
Server: Apache
Cache-control: private
Set-Cookie: JSESSIONID=B7E2479EC28064DF84DF4E3DBEE9C7DF;
Path=/
Content-Type: text/html;charset=UTF-8
Date: Wed, 18 Mar 2015 22:36:30 GMT
Connection: keep-alive
Set-Cookie: NSC xxx.fe.bd.vl-xd=ffffffffc3a035...423660;
path=/
Content-Encoding: gzip
Content-Length: 4162

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
        Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/
        xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
        xml:lang="en" lang="en">
<head>
<title> Informatics home | School of Informatics </title>
...
```

HTML

The main body of a webpage is written in *Hypertext Markup Language* (`HTML`). `HTML` provides a structural description of a document using special tags. `HTML` also has a mechanism known as *forms* to allow users to provide input to a website.

Forms can submit data using the `GET` method (name-value pairs *encoded in the URL itself*) or the `POST` method (name-value pairs *encoded in the body of the message*).

Example `POST` form

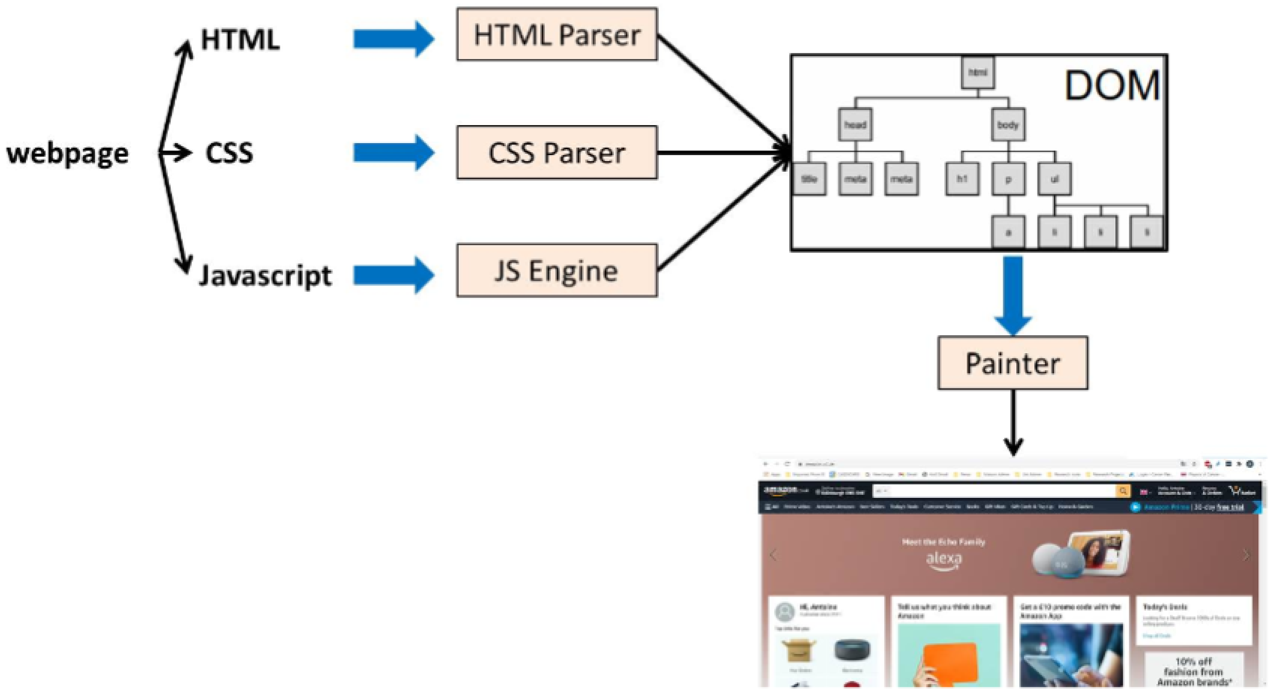
```
<form method="POST" action="login.php">
        FirstUsername: <input type="text" name="username">
        Password: <input type="password" name="password">
<input type="submit" value="submit"> </form>
```

Dynamic Content

`HTML` has no dynamic/interactive capabilities so scripting languages such as `JavaScript` were created to handle *dynamic content*. `JavaScript` is denoted between `<script></script` tags.

Document Object Model

The *Document Object Model* (DOM) is created from analysing the `HTML`, `CSS` (styling), and `JavaScript` on a webpage. A program called a *painter* is responsible for rendering the webpage visually.



Scripts can alter the content of a page by altering/updating it's DOM.

Frames

A *frame* is a method to embed an "inner" webpage *inside* an "outer" webpage. It uses the `<iframe src="URL"></iframe>` tag.

HTTP state

`HTTP` is *stateless* - when a client sends a request, the server sends a response based solely on that request, and doesn't store *info of previous requests*.

This presents a **problem**: in most web applications a client has to access various pages before completing a specific task and the client state should be kept along with those pages. How does the server know if two requests come from the same browser?
For example, if the user needs to log in, they shouldn't have to log in every single time - it should *remember*.

The **solution**: insert a *token* into the page when requested, and *keep passing it back and forth* between client and server to ensure state is maintained. There are two methods of maintaining state:

- Hidden Fields
- Cookies

Hidden Fields

Include a HTML form with a hidden field containing a session ID in all the HTML pages sent to the client. This hidden field will be returned back to the server in the request.

Advantage

- All browsers support HTML, so compatibility is not an issue.

Disadvantages

- Requires careful and tedious programming effort, as all pages must be dynamically generated to support this
- Session ends as soon as the browser is closed

Cookies

A cookie is a small piece of information that a server sends to a browser and is then stored inside the browser. A cookie has a name and a value, and other attributes such as domain and path, expiration date, version number, and comments.

The browser automatically includes the cookie in all its subsequent requests to the originating host of the cookie.

Cookies are only sent back by the browser to their originating host and not to any other hosts. Domain and path specify which server (and path) to return the cookie.

A server can set the cookie's value to uniquely identify a client. Hence, cookies are commonly used for session and user management.

Cookies can be used to hold personalised information, or to help in on-line sales/service (e.g. shopping cart). . .

When you make a request to a server for the first time, it'll send it's reply but will also send a special instruction `Set-Cookie`

Cookies have several attributes:

- `name=value`
- `expires`: (when to be deleted)
- `domain`: (where to send)
- `path`: (where to send)
- `Secure`: (only over `SSL`)
- `HttpOnly`: (only over `HTTP`)

Limitations

- Cookies are only sent back to the originating host
- Users can disable and delete cookies in their browser, which can break some applications if not anticipated for

Threat Model

There are 2 types of threats when focusing on web security:

Web attacker

This attacker controls the malicious website (e.g. `evil.com`) and has valid `SSL`/`TLS` certificates for it. The victim then visits `evil.com`

Network attacker

This attacker controls the whole network and can intercept, craft, and send messages. This is more powerful than a Web attacker.

Same Origin Policy

Scripts can manipulate the DOM of the page using an API. This API can allow scripts to potentially effect other websites you're tabbed into. Imagine you logged into `bank.com` and then `evil.com` - `evil.com` could run scripts that get information from `bank.com`!

The *Same Origin Policy* (SOP) restricts how scripts/resources loaded from one *origin* (e.g. `www.evil.com`) can interact with a resource from another (e.g. `www.bank.com`). SOP *sandboxes* each origin from the rest of the web, keeping it isolated.

JavaScript

JavaScript can load a cross-origin script. The browser executes this with the parent frame's/window's origin. It *cannot inspect* the source but *can call* functions.

Images

The browser *can render* cross-origin images, but the page *cannot inspect* it

iFrames

Frames *can be loaded* but the original page *cannot inspect or modify* it's content.

Cookie Policy

Scripts can manipulate the cookies stored in the browser using an API. This can allow scripts from malicious sites e.g. `evil.com` to access cookies used to authenticate users on e.g. `bank.com`

The *Cookie Policy* restricts how web servers and scripts can access the cookies in your browser.

A cookie has several attributes:

```
Set-Cookie: value[; expires=date][; domain=domain][;
path=path]
                                 [; secure][; HttpOnly]
        expires : (whentobedeleted)
        domain : (whentosend)   \
                                                         |>
scope
        path : (whentosend)     /
        secure : (onlyoverSSL)
        HttpOnly : (onlyoverHTTP)
```

The `scope` of a cookie is set by the server in the *header* of it's response.
The `domain` should be a suffix of the webserver's hostname.
The `path` can be anything, really.

Cookies are automatically sent back to the server by the browser if the URL is *in scope* - a cookie with `domain` and `path` will be sent to all URLs of the form: `http://*.domain/path*`

JavaScript (Cookies)

The browser applies the Cookie Policy and not Same Origin Policy for JavaScript.

`HTTPonly` Cookies

If a cookie is `HTTPonly` scripting languages cannot access nor manipulate the cookie. Combined with the Same Origin Policy this means even *Google Analytics* cannot see some cookies!

`Secure` cookies

A cookie with the `Secure` attribute is only sent to the server with an encrypted request over *`HTTPS`*, not *`HTTP`*.

Session hijacking

Session hijacking is exploiting a valid computer session to gain unauthorised access to information/services in a computer system. There are multiple methods to achieve this:

- predictable tokens, like Cookies
- a mix between `HTTP` and `HTTPS` elements, resulting in unencrypted sensitive communication
  - Although secure cookies solve this
- Cross-Site Scripting (XSS)
- Cross-Site Request Forgery (CSRF)

Cross-Site Scripting

Cross-Site Scripting (XSS) is an attack in which malicious scripts are injected into otherwise trusted websites. It attempts to beat the Same Origin Policy.

The malicious site (e.g. `evil.com`) provides a malicious script, and an attacker tricks a vulnerable server (e.g. `bank.com`) to send the script to the user's browser. The browser now believes the script originates from `bank.com` and will run it, with `bank.com`'s access privileges.

Stored XSS attacks

The script is *permanently stored on the target servers* and accessed, such as in a database, message forum, visitor log, comment field, etc

Reflected XSS attacks

The more common type of XSS, in a reflected attack the script is *reflected* off the web server. This could be through an error message, or more commonly through a phishing email - which directs to `evil.com`, and then redirects to `bank.com` with the attack request

The key to a good reflected XSS attack is finding a server that will *echo* user input back in the `HTML` response.
Imagine we visited `https://bank.com/search.php?term=...`

```
<html>
        <title>
                Search results
        </title>
        <body>
                Results for: $term
                ...
        </body>
</html>
```

We could replace `$term` with a script that e.g. sends the `bank.com` cookies to `evil.com`, which is fine according to the browser, because it's all done through `bank.com`.

Preventing XSS

Escaping input

By escaping the user input, we can preserve the character meaning without including any actual data.
e.g. `< → &lt; > → &gt; & → &amp; " → &quot; remove any <script>, </script>, <javascript>, </javascript>`

Input Validation

Check that inputs (headers, cookies, query strings, etc) are of expected form (through *whitelisting*)

CSP

This is when the server provides a *whitelist* of scripts that are allowed to appear on the page

`Http-Only` Attribute

If enabled, scripting languages cannot access or manipulate the cookie.
This will not prevent all exploits!

Cross-Site Request Forgery

While XSS exploits a user's trust of a specific website, *Cross-Site Request Forgery* (CSRF) exploits a website's trust of a specific user. CSRF attacks execute a *legitimate but malicious* request on behalf of a legitimate user.

CSRF target *state-changing* requests as the attacker has no (proper) way to see the response to the forged request.

CSRF Example

1. *Alice* logs into `bank.com` and gets a *valid session cookie*.

2. In another tab/window - with that session cookie *still valid* to make requests in that browser element - she visits `evil.com`
3. `bank.com` has the following form to request transfers:

```html
<form action="https://bank.com/transfer.php" method="GET">
        <input type="text" name="acct" />
        <input type="text" name="amount" />
        <input type="submit"/>
</form>
```

4. On `evil.com`, the following `HTML` exists:

```html
<form name="attack" action="http://bank.com/transfer.php"
method="GET">
        <input type="text" name="acct" value="Eve"/>
        <input type="text" name="amount" value="100000"/>
</form>
<script> document.attack.submit(); </script>
```

5. *Alice's* browser, upon reading the `HTML`, will run this form in the background without her being aware of it happening.
6. Her session cookie *is sent alongside* the `GET` request.

Preventing CSRF

CSRF is difficult to defend against, but there are some common techniques used.

Checking the referrer field

`HTTP` requests can contain a *referrer* header specifying the context in which this request was issued. A server could use this to ensure the `HTTP` request comes from *the original/a trusted (whitelisted) site*.

Referrer fields can be used for tracking and can be a privacy violation, and some browsers can have this disabled.

CSRF tokens

CSRF attacks are dependant on the request format *being predictable*. In the above example, only the session cookie is used to authenticate the request. We could store a CSRF token in a hidden field and check this on every request. This means `evil.com` would have to forge this token to create a valid request. This should be *different* in **every** server response to avoid predicting this via a *replay attack*.

`SameSite` cookie attribute

Cookies have a `SameSite` which prevents them being sent in cross-site requests. If `bank.com` enabled these for the session cookie, *Alice's* browser *will not include* any cookies for `bank.com` when the request is issued from `evil.com`.

Injection Attacks

Injection attacks are a method to execute arbitrary code by sending specific data to a server. This occurs when commands are issued based on the data provided by a user, and can be prevented by sanitizing/validating this data.

Command Injection

Command injection is a class of attacks that occur when the system passes the data to a command which is passed into a *system shell*.

Example: `whois`

The `whois` command can be used with a URL to get information about IP addresses, domain names, owners, etc.

The budding `example.com` wants to implement an online version of this command and has the following form:

```
http://www.example.com/whois/content.php?domain=google.com
```

In this `content.php` is the following code:

```
<?php
        if ($_GET['domain']) {
                <? echo system('whois '.$_GET['domain']); ?>
        }
?>
```

However, *Eve* wants to delete all files on this server so provides the malicious argument `www.example.com; rm *` resulting in:

```
http://www.example.com/whois/content.php?


domain=www.example.com; rm *
```

resulting in the following PHP that executes two commands:

```
<? echo system('whois www.example.com; rm *'); ?>
```

This will run:
`whois www.example.com` - the actual `whois` command
`rm *` - **deletes all files**.

Input Escaping

We can avoid this by *escaping the input*. For example, using the `escapeshellarg()` command will add single quotes around a string and *escapes* any existing single quotes. If we used this, the command executed above would be

```
<? echo system('whois'.escapeshellarg(www.example.com; rm
*)); ?>
```

and would run `whois 'www.example.com; rm *'` (returning error, obviously).

SQL Injection

SQL Primer

SQL is a commonly used database query language that can be used to return a set of records from a database.

username	password
alice	01234
bob	56789
charlie	90210

```
SELECT password FROM user_accounts WHERE username='alice'
```

would return `01234`

```
DROP TABLE user_accounts
```

would delete the entire `user_accounts` table.

Semicolons can be used to separate commands.

```
INSERT INTO user accounts VALUES (’eve’, 98765);
SELECT password FROM user accounts
                                WHERE username=’eve’
```

would return `98765` when inserted on one line (line break added for clarity here)

`--` can be used to denote comments

```
INSERT INTO user accounts VALUES (’eve’, 98765);
SELECT password FROM user accounts
                                WHERE username=’eve’ --get
eve's password
```

SQL injection example

The website `example.com` uses this following script to send a user to their respective control panel

```
$conn = pg_pconnect("dbname=user accounts");
$result = pg_query(conn,
                                   "SELECT * from user
accounts
                                   WHERE username = " ’.$
GET[’user’].’"
                                   AND password = ’".$
GET[’pwd’]."’;");
if(pg query num($result) > 0) {
        echo "Success";
        user_control_panel_redirect();
}
```

If an attacker entered the `username` as `admin';--` and password as `f` (this could be anything, but form validation rules usually make it have atleast something there), the server would run the following query

```
SELECT * from user_accounts
WHERE username = 'admin';
--' AND PASSWORD = 'f';
```

Since SQL doesn't do the password validation itself, it just grabs the admin's control panel.

If the attacker entered `username` as `admin'; DROP TABLE user_accounts;--`, the server would run the following query:

```
SELECT * from user_accounts
WHERE username = 'admin'; DROP TABLE user_accounts;
--' AND PASSWORD = 'f';
```

(line-breaked for clarity again, both of these would be one line)

SQL Injection Defences

Sanitised inputs

Most languages contain functions that strip the input of *dangerous* characters. For example, if the `example.com` programmers used `PHP`'s `mysql_real_escape_string`, the attacker's query would look something like:

```
SELECT * from user_accounts
WHERE user = 'admin\'; DROP TABLE user_accounts;'
```

Prepared statements

This is the idea of sending the query and data separately to the server. We do this by creating a *template* of the SQL query in which we substitute the data values. This could result in the following `PHP`

```
$result = pg_query_params(
                        conn,
                        SELECT * from user accounts WHERE
username = $1
AND password = $2,
                                                   array($ GET[’user’], $ GET[’pwd’]));
```