

POLITECNICO DI TORINO

**MASTER's Degree in DATA SCIENCE AND
ENGINEERING**



MASTER's Degree Thesis

**Automatic web crawler for malicious
websites classification**

Supervisors

Prof. MARCO MELLIA

Prof. IDILIO DRAGO

Prof. RODOLFO VALENTIM

Prof. CRISTIANO PANAZIO

Candidate

ALLAN BRUNSTEIN

MARCH/APRIL 2024

Summary

Introduction

Cyber crimes like phishing and cybersquatting are very dangerous and very common forms of scam in the internet. By impersonating target brands, criminals make use of copyrights and big brands reputation to extort victims and receive money via fraudulent messages. Their crimes are often more than only financial, since they are able to gain access to bank accounts, credit cards and they collect information about the victims, which might lead it to a data leak, identity theft or a series of different cyber crimes.

A FBI report over cyber crimes stated that in 2023 alone, phishing was responsible for a total loss of more than 50 million United States Dollars. A Google research on this topic with the University of South California showed that 45% of phishing attacks are successful, and that even the least performing ones were successful 3% of the times.

The current defense mechanism against phishing and cybersquatting is block listing the malicious websites. Internet providers, upon taking knowledge of new malicious domains, block access to them along the network, protecting their users from criminal activities.

However, this defence fails when the domain is new or undetected. Domains can stay under the radar for months or even years before authorities are alerted and block them. This method is even more inefficient against zero-day attacks, those that happen in the same day that a domain starts its malicious activities.

In this sense, it is necessary to constantly and consistently collect data from the largest amount of domains possible in the shortest amount of time, enabling Internet Service Providers (ISPs) to gather evidence and block malicious websites as quickly as possible.

The solution developed in this thesis is a proactive crawler that collects DNS records, security certificates, ownership information and takes a screenshot for each candidate of an input list of potentially harmful domains provided by the user. This kind of system is complex to engineer due to its need for efficiency, scale, portability and ability to manage incoming data from multiple sources.

This thesis is an effort to fill this gap in cybersecurity by constantly collecting consistent data from potentially dangerous domains, helping authorities to identify threats and block malicious websites as soon as possible, which is a fundamental step to avoid frauds and save victims all over the world.

Methodology

The web crawler collects the following information of monitored domains: Secure Sockets Layer (SSL) certificate, Domain Name System (DNS) information and WHOIS data, that shows who owns the domain, when it was registered and other information about its background. Finally, the crawler takes a screenshot of the home web page and saves all the results in a local folder.

The list of domains to monitor is served as an input via a CSV file. The crawler loads it and proceeds to parse the data inside a python script. This script then makes requests to DNS servers to acquire A (IPv4), AAAA (IPv6), MX (e-mail) and CNAME (redirects) records.

If this domain contains at least one A record or one CNAME record, in such way that it contains a visible home page, the server proceeds to collecting the SSL certificate and the WHOIS information. This criteria ultimately saves processing time, since the program will not try to screenshot websites that do not contain a home web page, which would be impossible.

Then, the python script executes an asynchronous call via web sockets to a .NET application called Screenshoter. This application uses a headless chromium browser to take a screenshot of the main web page of the monitored website and saves the image in the output folder. With this asynchronicity, the python script can continue processing new domains and making DNS, SSL and WHOIS requests while waiting for the screenshot of already processed ones, saving execution time.

In order to optimize the performance of a single run of the crawler and allow for the monitoring of the largest possible number of domains, the program works in multiple threads. The number of threads is defined by the user as a parameter when running the program.

Moreover, to facilitate horizontal scaling, the whole program was containerized using Docker. With this, the execution does not depend on any specific operational system or processor and can be orchestrated by simply loading the image and running it. The user can also split the list of domains to monitor across different machines, which contributes to the scaling of the whole process.

The final tool developed is a dashboard that allows for the visualization of the obtained results by listing them in a website containing a table and the screenshots. The visitor can then manually label the websites and these results can be further used to train an artificial intelligence model to automatically classify websites and

boost the process of blocking malicious domains.

Results

To test the functionalities of the crawler, I used a list of domains that are phonetically similar to target brands. This way, using a brand like *amazon*, the list suggests monitoring *amacon*, *amason* and *amazone*, to name a few. Approximately 2400 domains were monitored for two weeks in an Ubuntu machine with 4 cores and 8 GB of RAM memory. The results available in the website (<https://crawler.allanbr.net>).

Throughout the execution, the crawler successfully and consistently collected data from the input domains and websites. It is important to notice that although the initial list contains 2780 domains, not all of them are registered and some of them are already present in block lists due to malicious activities in the past.

Analyzing Amazon's case, it is possible to realize the consistency of the crawler. Since the 6th day, it collected A data from 156 websites (± 1), AAAA from 77 (± 0), MX from 64 (± 1) and captured screenshots from 144 (± 2) websites.

For the Apple brand, the initial list contained 101 similar domains, and in that same period the crawler could daily collect A records from 92 websites (± 1), AAAA from 17 (± 0), MX from 74 (± 1) and Screenshots from 89 (± 1). These results show consistency and that the crawler is reliable to resolve DNS and screenshot websites.

Moreover, the collected data from these websites was stored in a MongoDB cluster and is available in a dashboard website. By inspecting the obtained screenshots, I was able to identify that about 10% of the domains similar to Amazon (14 out of 145 screenshots) and Apple (11 out of 92) are parked domains. More than that, a big part of the parked domains with similar names are owned by the same registrar. This indicates that these registry companies might be mass buying domains that resemble famous brands in order to profit from their reputation.

The main issue with parked domains owned by domain registrars is that they can be bought at any given time and there is no way to ensure that the new owner will use it for legitimate purposes. Therefore, parked domains are extremely dangerous and need constant monitoring, since they can suddenly and unexpectedly become malicious websites.

All in all, this application aims to power up the fight against phishing and cybersquatting and prove a new technique of fighting malicious domains, applying data engineering concepts with the use of a crawler and paving the way to the use of machine learning techniques to more efficiently combat it in the future.

Acknowledgements

ACKNOWLEDGMENTS

Firstly, I'd like to thank my family for the unconditional support during my time abroad. To my parents, David and Diana. To my grandparents, José and Ágnes, Dora and Israel. To my brothers Rafael and Artur, to my cousins, aunts and uncles who actively participated in my development as a person and raised me with great values, including those of the persistence, resilience and showed me the importance of study and work to become a productive member of society and give back to the people.

I would like to dedicate this thesis and my work in memory of my late grandfather Israel Brunstein, that served University of São Paulo as a professor for more than forty years, becoming a reference in management engineering, economy, and a reference for me as a great person dedicated to creating a better world. I carried his memory along with me during all my time abroad, and in the hard times I always remembered his bravery when he moved to Stanford with my grandmother Dora and my uncle Leo in 1963, in a world where the only way of communicating with people abroad was with letters. He went studying in California to follow his dreams and his determination motivated me all along my trajectory to follow my own path with determination and passion.

I have no words to express my enormous gratitude to my family, a very united family that did everything possible, or even more, to raise me with good values and did not measure efforts to help me while I was abroad. I can not express my profound admiration for my grandparents trajectory or underestimate their advice. Ágnes and José came to Brazil as immigrants, from Hungary and Egypt, and with hard work and dedication they are now grandparents to a great family that admires them and looks up to them as an example for honesty, intelligence and values.

Dora and Israel were born in Brazil, as a second generation from Polish and Romanian families. From a very poor background, they proved that study, honesty, companionship and hard work can create roots to a successful family and a successful life.

David and Diana are my parents, to whom I owe infinite gratitude for showing me the right way in life to become a valuable member of society, for showing me how work and study dignifies a person and for making me interested in ever learning more and perfecting myself. In this sense, I must also acknowledge my brother for also making me interested in the most diverse topics, that developed my creativity and set my in my path to engineering.

I must, therefore, thank the Brazilian nation for receiving my immigrant family with open arms and providing us with all the condition to build our future in this wonderful land. The Italian nation, for receiving myself and giving me the opportunity to study in one of Europe's most prestigious universities.

In terms of people that helped me during this thesis process, I want to thank the professors at my home university (USP), in special Cristiano Panazio, and to the institution itself, that allowed me to board this opportunity, study in Turin and graduate via the Double Degree agreement.

In Italy, I must also acknowledge Rodolfo Valentim, Idílio Drago, Marco Mellia for their help during the preparing, developing and writing of this thesis. Their dedication and support was outstanding and I learned a lot from their expertise during this whole project.

In Hungary, I would like to acknowledge the help from a very special person that kept me motivated to work in this project as hard as I can, to give the best that I can, and that I have the privilege to call my girlfriend. Thank you, Abigél, for keeping me motivated and focused in this project through all the times.

To my friends in Brazil and the friends that I made during my time in Turin, from the Politecnico and from Granstudio. To my school professors that motivated me to follow a path in engineering - Evelyn and Vargas to cite a few - and to the assistants and support staff from my school, from my universities, from the International Office and every one that took part in this journey. You have all contributed to shape my personality, my talents and my way of seeing the world.

To all of you, a huge thank you from the bottom of my heart. You made every step of this process enjoyable and rewarding, and I am extremely happy to share this moment with you.

Table of Contents

Introduction	ii
Methodology	iii
Results	iv
List of Tables	X
List of Figures	XI
Acronyms	XIV
1 Introduction	1
1.1 Introduction	1
1.2 Objectives	4
2 Related Work	5
2.1 Generative methods for domain-based phishing	5
2.2 Other crawlers	6
3 Background	8
3.1 DNS	8
3.2 SSL	9
3.3 WHOIS	9
3.4 Selenium	9
3.5 Docker	10
3.6 Parked Domain	10
4 Methodology	11
4.1 Structure	11
4.1.1 Python Script	13
4.1.2 Screenshoter	15
4.2 Flexibility and optimizations	17
4.3 Docker	18

4.4	Dashboard	18
5	Results	19
5.1	Testing Dataset	19
5.2	Dashboard	19
5.3	Study cases	27
5.4	Further Considerations	31
5.4.1	Memory management	31
5.4.2	Performance	34
5.4.3	Synchronicity	36
6	Conclusions and Future Work	37
	Bibliography	39

List of Tables

2.1	Similar crawlers	6
5.1	Possible labels for websites	20
5.2	Number of outputs per day - Adobe (There were no CNAME nor AAAA entries)	27
5.3	Number of outputs per day - Amazon (There was only one CNAME entry all the days)	28
5.4	Number of outputs per day - Apple (No CNAME entries)	28
5.5	Labels for <i>Adobe</i> homophones, daily	30
5.6	Labels for Amazon's homophones in the last day of execution	30
5.7	Labels for Apple's homophones in the last day of execution	31

List of Figures

1.1	FBI IC3 report [1] - Number of phishing victims per year since 2018	2
1.2	Proposed work fit. The Web Crawler works as an intermediate between the Generator, that provides a list of domains, and the Classifier, that can be created to automatically label the websites.	3
4.1	High level abstraction of the model - The domains of interest are passed to the system via a csv file. For each candidate, the script captures DNS, SSL and WHOIS information. The data is then sent via TCP to the .NET application, that inserts the url into an asynchronous queue that captures a screenshot whenever a thread worker gets free.	12
4.2	Program's execution in steps: upon receiving the URL, the application resolves the DNS, then catching the SSL certificate and the WHOIS information before requesting the screenshot from the external application	14
4.3	Screenshotter abstraction - The domain received via TCP is added to the queue of candidates to be processed. The browsers receive these domains whenever they are free, sending the screenshot path back to the Python script via TCP	16
5.1	React website containing fetched results	21
5.2	The column <i>end_time</i> now appears in the table, showing how long it took for the website to be fully processed	22
5.3	Table results with example custom query	22
5.4	Modal with complete information of <i>adob.eu</i> , a parked domain	23
5.5	Modal with expanded information about the SSL certificate	24
5.6	Modal showing information about a not yet labeled parked domain	25
5.7	Screenshots displayed as a gallery in the crawler website	26
5.8	Memory usage running container with external screenshotter and 8 threads	32

5.9	Memory usage running container with external screenshoter and 24 threads	32
5.10	Memory usage running container with python screenshoter and 8 threads	33
5.11	Blocked site alert - amazen.mom	35

Acronyms

AI

Artificial Intelligence

BBC

British Broadcasting Corporation

CNN

Convolutional Neural Network

FBI

Federal Bureau of Investigation's

GAN

Generative Adversarial Network

ISP

Internet Service Provider

JS

JavaScript

SSL

Generative adversarial network

TCP

Transmission Control Protocol

UDP

User Datagram Protocol

Chapter 1

Introduction

1.1 Introduction

Cyberattacks continue to pose significant challenges for individuals, businesses, and organizations worldwide. One of the most dangerous forms of cyber threats is phishing, a method employed by malicious actors to deceive individuals into revealing sensitive information, like credit card numbers, bank account balance, home address, social security number etc.

Cybersquatting is one of the most insidious allies of phishing: the malicious registration of domain names to replicate legitimate brands further compounds the complexities of safeguarding the digital realm. When trusting victims mistype an institution name in the internet they are vulnerable to find themselves in a malicious website, that abuses the names and trademarks of legitimate organizations to deceive unsuspecting users into revealing their most precious data.

The profile of a malicious website usually follows a known pattern: it can be as simple as a website displaying fake virus messages - e.g. "Your computer has been infected - call xxx-xxxx") - or legitimate-looking like a real business replica, imitating the target company's website style and content. Since the user believes that they typed the correct address, they are likely to believe that they are in a legitimate position of concern.

The most recent report from the Federal Bureau of Investigation (FBI) [1] highlights the severity of these crimes: in 2022 alone, over 300,000 American citizens fell victim to phishing attacks, resulting in a staggering loss exceeding 52 million dollars. The pervasive nature of cyber threats is further emphasized by Google's research along with the University of South California: 45% of the phishing attacks are successful, which comes to show the hurry for innovative and robust tools to fight the existence of these activities.

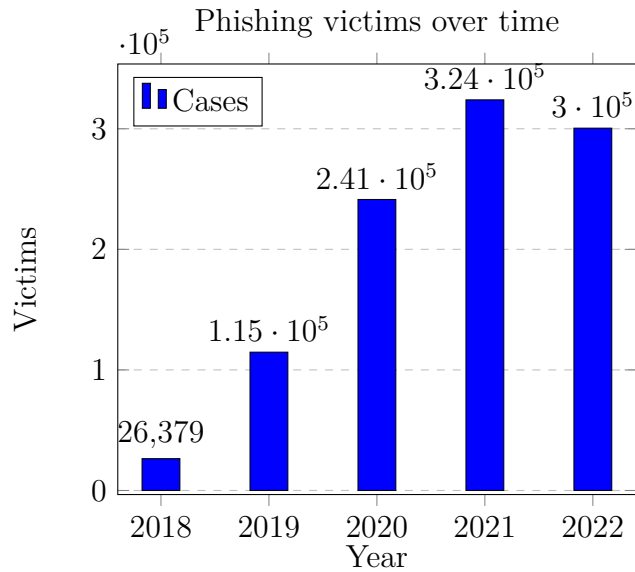


Figure 1.1: FBI IC3 report [1] - Number of phishing victims per year since 2018

The traditional mechanism of defence against those attacks consists in blocklisting, which means that internet providers and internet browsers block connections to the malicious website as soon as the fraudulent activity is discovered.

Block lists, however, are not effective against new domains, since they work by adding domains when a malicious actor is found. Therefore, websites can only be blocked after an individual reports their existence, which makes it possible for scamming domains to be unnoticed by authorities per months or even years.

To the best of our knowledge, there are still no tools that allow for constant monitoring and stopping cyber threats more quickly, in the day that they started. Zero-day attacks – the name of cyberattacks that occurs in the same date as the website has been published – pose a serious threat to internet users, and my goal with the thesis is to attack this problem, creating a tool that allows the monitoring of potentially risky domains in a daily basis, obtaining their content and the most significant information that can be used by a classifier to block them or trigger human validation as soon as they are up on the internet.

Traditional methods of obtaining information from websites can assist in the identification of threats, however, there are several limitations that makes it impossible to adequately monitor candidates only with their use. Some of these tools and their limitations are listed below.

Tools like *wget* [2] and *curl* [3] have been around for decades now, and they allow the retrieval of a webpage content via command line. While they excel at downloading static content, their most severe limitation is that they cannot run

JavaScript or other client-side scripts. Since most of the modern websites make extensive use of JavaScript, it is crucial to address this issue.

Selenium, on the other hand, is a more powerful tool in this context. It can be used to run web pages in the most commonly used Operational Systems (OS) and can be used together with the most used browsers as well. In this thesis, I opted to use Selenium’s C# (.NET 7) library along with Chromium web driver and Google Chrome browser. This system runs in an Ubuntu environment with a x64 processor architecture, and the end goal is to screenshot candidate web pages.

Therefore, the basis of this thesis is the creation of a web crawler, built on the top of Selenium, that receives a list of domains to monitor, retrieving essential information over their DNS, SSL and WHOIS status while simultaneously taking a screenshot of the visited web page.

The list of monitored domains comes from Valentim et al. paper from December 2021: the work proposed a Generative Adversarial Network (GAN) that was used to list more than 600,000 homophone domains to popular targets. In this list, for example, “google.com” is related to “gugle.com”, since the “oo” in Google has the sound of “/u/” in some languages.

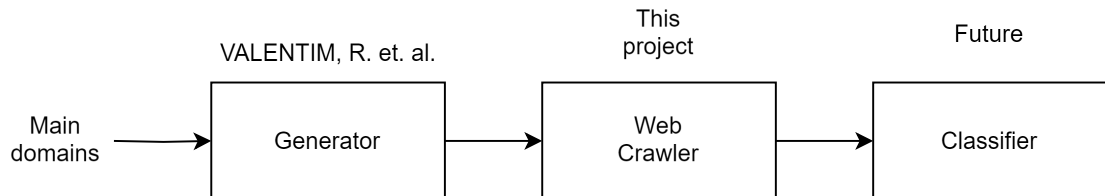


Figure 1.2: Proposed work fit. The Web Crawler works as an intermediate between the Generator, that provides a list of domains, and the Classifier, that can be created to automatically label the websites.

In terms of scalability, several users can run the application at the same time, dividing the number of domains in the input between different machines and horizontally scaling the application. This is possible because the output from a single domain can be treated as statistically independent from the output from another domain – there are no correlations between them. Therefore, in order to increment the number of monitored domains, it is sufficient to run parts of the list on separate computers or virtual machines.

What is more, the project is fully open source and available for everyone to contribute to its future improvement in GitHub [4]. Since this is a matter of interest of the whole internet community and the society in general, a collective effort can be made to stop phishing and cybersquatting for one and for all.

1.2 Objectives

The main objective of this work is to create a Web Crawler capable to consistently monitor a list of candidate websites, automatically collecting information and taking a screenshot, allowing for manual and automatically classification of potentially malicious websites.

- Collect DNS, SSL and WHOIS information from domains belonging to a pre-defined list
- Take a screenshot of viable websites, assuring that they are correct and consistent
- Create a visualization tool that allows labeling of the obtained data

The labeling of the data is essentially a proof that the collected information is sufficient to classify the websites, and validates the main objective of this thesis, that it is possible to use a Web Crawler to collect information and systematically detect phishing and cybersquatting domains.

Chapter 2

Related Work

2.1 Generative methods for domain-based phishing

The core of this project, as described in **Figure 1.2**, is the follow up step to the conference paper "Augmenting phishing squatting detection with GANs" [5]. This work introduces a novel approach to find phishing and squatting candidates, using a Generative Adversarial Network (GAN) to create a list of phonetically similar domains to a target brand.

The authors stated that this method generates more than 600,000 potential malicious domains and at least 1,175 of those were confirmed to be used for phishing. Moreover, an alarming share of 90% of the malicious domains evaded blocklists for more than a month, which emphasizes a critical the problem with current approaches to detect cyberthreats: they are unable to stop attacks with a reasonable speed and scams can last for months without being noticed.

Therefore, this list is the basis for this web crawler, serving as input for the domain monitoring. In other words, the generated domains are shortlisted and monitored every day by the crawler application. Collecting their main information is key to provide a future classifier with sufficient data to analyse and classify domains as legitimate or malicious, allowing for a safer internet and stopping scams as soon as they start.

Preceding the previous work is PhishGAN [6], which proposes the usage of GANs to generate homoglyph attacks. In this context, users often are unable to notice small changes in the domain name. Malicious domains often contain similar characters like "i" for "1", "w" for "vv", "o" for "0" or even their other alphabet counterparts: "l" for "1", latin "a" for cyrilic "a", which tricks victims eyes into believing they are in the correct webpage.

In fact, the academic paper cites a study by Dhamija et al. [7], that used a

website "bankofthevest.com" and 90.9% of participants did not realize the double "v's" were used instead of a "w" in "west".

One work that relates to this thesis and PhishGAN is "GlyphNet: Homoglyph domains dataset and detection using attention-based Convolutional Neural Networks" [8], that exemplifies how lethal can homoglyphs be in terms of cybersecurity, allowing malicious actors to invade networks, steal data and take control of organizations. In this project, domain names were generated using a Convolutional Neural Network (CNN).

2.2 Other crawlers

Several other crawlers have been already developed in the context of improving internet security. Some of the examples that inspired me for this project are listed below, as well as the missing feature in comparison to this project.

Tool	Main capability	Disadvantage
Puppeteer	Get PDF and Screenshots	Can't get SSL, DNS
Gospider	Find all subdomains, domain tree, robots.txt	No screenshot
Hakrawler	Gathers website, subdomains and JS data	No screenshot
CheckPhish AI	Screenshot, gets SSL, DNS	Paid service
OpenPhish	Screenshot, path, SSL	Paid service

Table 2.1: Similar crawlers

These crawlers are the State of the art in terms of modern security crawlers. Most of them are written in Go for the Kali operational system, a Linux open-source distribution aimed towards digital forensics, ethical hacking and penetration testing, among other cybersecurity features.[9]

While this work is very well related with the mentioned tools, none of them do exactly what I designed this application to do. With this crawler, the user can store main information regarding the website security (SSL certificate, DNS records) and origin (WHOIS data), allowing an automatic classification in the future. These features are not present in the above mentioned open-source crawlers, and the paid options are not viable due to the high cost to have access to their database.

In this context, State of the Art crawlers currently focus on XSS injections and other methods for server protection. These crawlers test the capability of websites to protect themselves against attacks, and their focus is not on actual data collecting on third-party servers.

Therefore, it is clear that these methods are not enough to obtain the necessary information to classify websites as potentially harmful for users or legitimate,

and the crawler I developed is a necessary tool in this sense, to allow for further classification of risky websites and make the internet a safer place for everyone.

Chapter 3

Background

Before proceeding with the work developed on the course of the thesis, it is crucial to define some of the terms that are extensively used in the following chapters and that are fundamental for the understanding of the system.

3.1 DNS

The Domain Name System (DNS), essentially, is the association mechanism between human-friendly URLs (e.g. `www.domain.com`) to computer-friendly addresses (IPs, e.g. `192.168.0.1`). Each domain on the internet publishes information about themselves in a distributed database system, whose nodes are called Name Servers (NS).

Each node stores records that are essential for correct operation of the internet, allowing other computers to easily locate resources such as IP addresses, e-mail servers addresses, aliases, subdomains etc.. The most important records for the understanding of this thesis are the A/AAAA, CNAME and MX:

- **A record**[10]: stands for Address. It stores the IPv4 address corresponding to the website. The A record helps to understand who is hosting the website, since many different domain names can point to the same IP address.
- **AAAA record**[11]: similar to A record, but it stores IPv6 addresses instead of IPv4.
- **CNAME record**[12]: stands for Canonical Name. It means that the queried domain (e.g. `page1.com`) is an alias for another domain (e.g. `page2.com`), therefore `page1.com` should use the same records as `page2.com`. Notice that a CNAME record holds relations between two domains, and not IPs.

- **MX record**[13]: stands for Mail Exchange. It shows the route for incoming e-mails and are always pointing to domains (not IPs).

3.2 SSL

SSL (Secure Sockets Layer) is an internet encryption-based security protocol [14]. A website in possession of a SSL certificate guarantees that the traffic is fully encrypted between the server and the client, assuring that the message can not be understood by a third party.

When browsing through the internet, a serious amount of data is transferred between the client and the server, which often involves sensitive data: credit card numbers, addresses, telephone numbers, medical data, private messages etc.

When the traffic runs unencrypted, every message can be intercepted and read as plain text, making it easy to steal sensitive data from unsuspecting users. However, one must notice that the presence of such certificate does not mean that the website is legitimate. It assures that the data transmission between the client and the server is encrypted, and that the current domain is associated with the current server. It does not certify, by any means, that one specific website can be trusted.

It is possible to instantly emit SSL certificates for free with providers such as Let's Encrypt and Cloudflare, which is one tactic that malicious websites can use to provide a feeling of safety to the victims. This step is important, since most browsers display a warning when the user tries to access a webpage without any certificate or with an invalid one. A clone website, for example, might have this certificate to go through this browser verification.

3.3 WHOIS

WHOIS is an internet database that provides records of ownership for internet domains. It contains information such as the domain's owner, e-mail, telephone, address, domain name registrar, expiry date, transfer date etc.

In the context of this thesis, the WHOIS is a very important data source. Domains used for cybersquatting and phishing are not associated to the target company, were recently registered or had their ownership changed. It can help the classification of malicious domains with more precision and work as another tool to detect fraudulent activity.

3.4 Selenium

Selenium [15] is an automated browser tool that is used as the backbone for this project. By using selenium, it is possible to use a *headless* web browser, that means,

without any user interface or human interaction.

In this thesis, the Chromium Webdriver (open source from Google [16]) is attached to the selenium, that accesses the desired websites and screenshots them through Selenium's C# library.

The advantage of Selenium, in comparison to tools like *wget* and *curl* that were mentioned in the Related Work section, is that it can run JavaScript and capture screenshots of the web pages, basically allowing for the attachment of most web drivers with their full functionality just like a human would use it's internet browser. Therefore, the image captured in the screenshots is exactly what a person would see when entering the same page.

3.5 Docker

Docker [17] is a virtualization tool that allows applications to run in isolated environments known as containers.

In this project, Docker was used to create an image of an Ubuntu operational system containing the Crawler's files and services installed. In practical terms, it is a new virtual machine containing only the necessary resources to run the software, making it more portable and prone to scale.

With Docker, the script can run in an isolated environment and be less susceptible to interference with services and other factors that would possibly harm the efficiency of the system. Therefore, given that the system has the necessary free RAM memory and CPU, any computer can run the application, which helps scaling further.

3.6 Parked Domain

A parked domain is essentially a web page that is hosted, but not being used for any specific purposes. Domain registrars often take hold of parked domains to resell them, or individuals for future development purposes.

Parked domain websites can often display messages stating that the website is "In development" or "Under construction", if the owner intends on using it, or it can show messages suggesting that the domain name is "For sale". Some of these pages might also display advertisements as a form of monetization.

Moreover, companies can also buy domain names which are similarly written or phonetically similar to their brand's name to prevent malicious actors from owning these domains and using them for phishing or cybersquatting.

Chapter 4

Methodology

4.1 Structure

The most fundamental abstraction of the application consists of two systems that communicate with each other. The first one is a Python script, that collects information about the domain's background, and the second one is a .NET application that runs on the background and takes screenshots of the web pages upon request, both of which communicate with via TCP sockets.

The .NET application is called Screenshoter, a tool developed in C# specifically for the purposes of this thesis. I projected it on the top of Selenium using a *TcpListener* handle to asynchronously listen to a socket port and communicate with other applications. The number of threads and the port that the screenshoter runs in is fully parameterized, and defined via an environment variable on the Docker image, that can be set when creating the container.

The Python script, on the other hand, processes the input domains and send the viable candidates to the screenshoter. This step aims to collect information from the website before actually taking the picture, in order to avoid unnecessary resource consuming and time wasting in the case that the website is known to be legitimate or if it is not hosted.

The Docker containerization is the step that provides reproducibility and allows the service to always run in the correct environment, with the original files and all services running as intended. Moreover, it makes the service portable and runnable in different machines with diverse operational systems (OS) - originally, I projected it to run in an Ubuntu 20.04 OS with the x64 architecture. When using Docker's virtualization, any x64 processor can run it, even if the host machine is a Windows or a Mac OS.

After the execution, the pipeline resumes outside of the Docker container. The output.ndjson file is uploaded to a third-party database service, and the screenshots

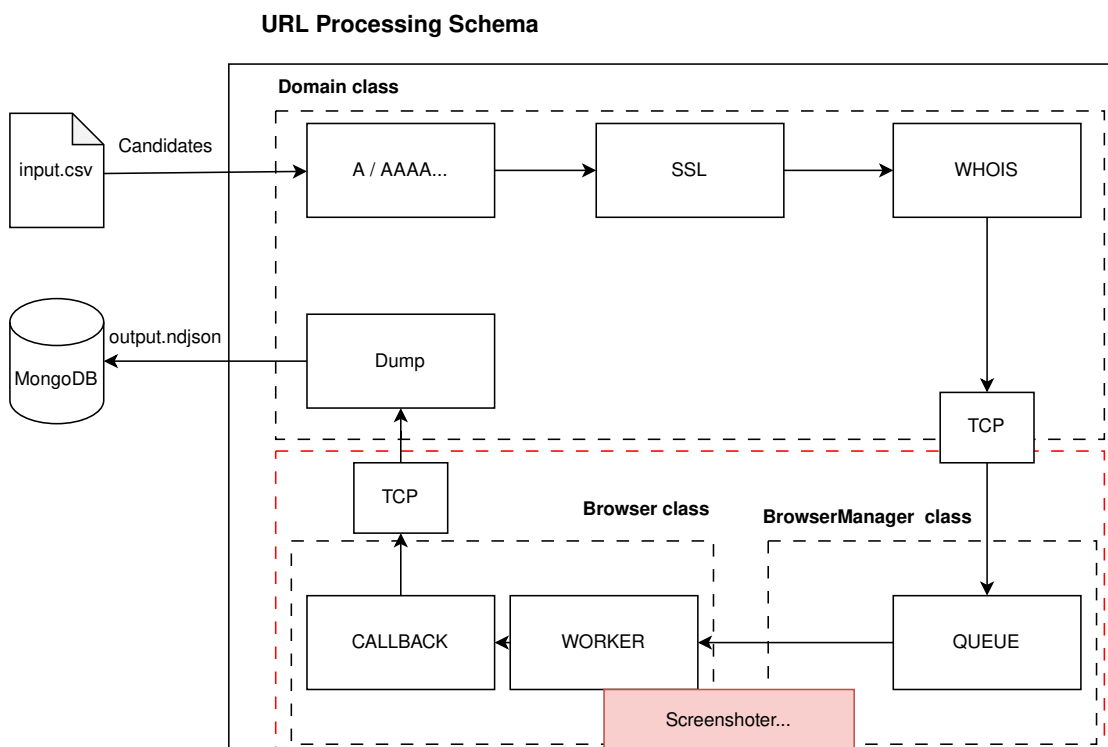


Figure 4.1: High level abstraction of the model - The domains of interest are passed to the system via a csv file. For each candidate, the script captures DNS, SSL and WHOIS information. The data is then sent via TCP to the .NET application, that inserts the url into an asynchronous queue that captures a screenshot whenever a thread worker gets free.

can be either saved in a cloud environment or saved in the database itself via Base64 encoding, for example.

Both approaches have their advantages and disadvantages, and overall I consider that the cloud hosting should be the best approach in the case of a standalone server, considering the fact that the Base64 encoding would also possibly expand the image size due to the conversion of its natural JPEG compression into a text encoded from a binary string.

Since I plan to keep this thesis' project hosted at a PoliTO cluster, I chose the to host the images at the cluster itself, in a cloud environment. This should save space overall and allow for a smoother execution.

4.1.1 Python Script

The Python script is the entry point of the whole process. Here, the *CSV* input file is processed and divided into multiple workers, according to the number of threads that the user chose when creating the Docker container.

Each worker creates a call to a function named *process_url*, whose arguments are the domain to be processed and the target brand's domain it is associated with.

In the *process_url* function, the aforementioned arguments will be the parameters to create a *Domain* class' instance, which is responsible for getting the main information from the domain as stated in the diagram from Figure 4.1.

The ndjson output format was chosen to allow simpler visualization and indexation of the results. The final document can be easily indexed in a collection and used as a document in a NoSQL context, like a MongoDB or an Elasticsearch environment.

The WebBrowser class contains a wrapper to manage a *Chromium* instance that will run in the background in a headless state - running unattended without any user interface, which is suitable for automation purposes. This integration was possible due to the *Selenium* library, which provides modules for using a web browser inside a python code. Moreover, the web driver *headless-chromium* can be downloaded directly from Google's open-source repository [16].

Domain class

The Domain class possesses methods for the most important functions: getting DNS records (A, AAAA, CNAME, MX), SSL certificate and WHOIS data.

The DNS requests are crucial. If the domain does not have at least one A or CNAME record, then no websites are hosted in that domain, nor it is used to redirect.

This filtering is crucial to avoid overloading the screenshotter. The biggest bottleneck in this project is that there is a minimum loading time for a website, depending on its overall size, the distance between the host and the client and the webpage loading time is by far the slowest part of the process. In fact, the DNS data gathering process takes between one and two seconds to finish (average: 791 ms, standard deviation: 1454 ms), but the screenshot itself is a long process since the whole webpage has to be processed, including its images, JS scripts and content, allowing for the best screenshot with the page as complete as possible.

Therefore, I added a two-second delay between accessing the website and taking the screenshot in the Screenshotter, so that most websites are able to load completely by the time that the program takes the picture. This value has been found empirically to be a good compromise between the application speed and the quality of the screenshots.

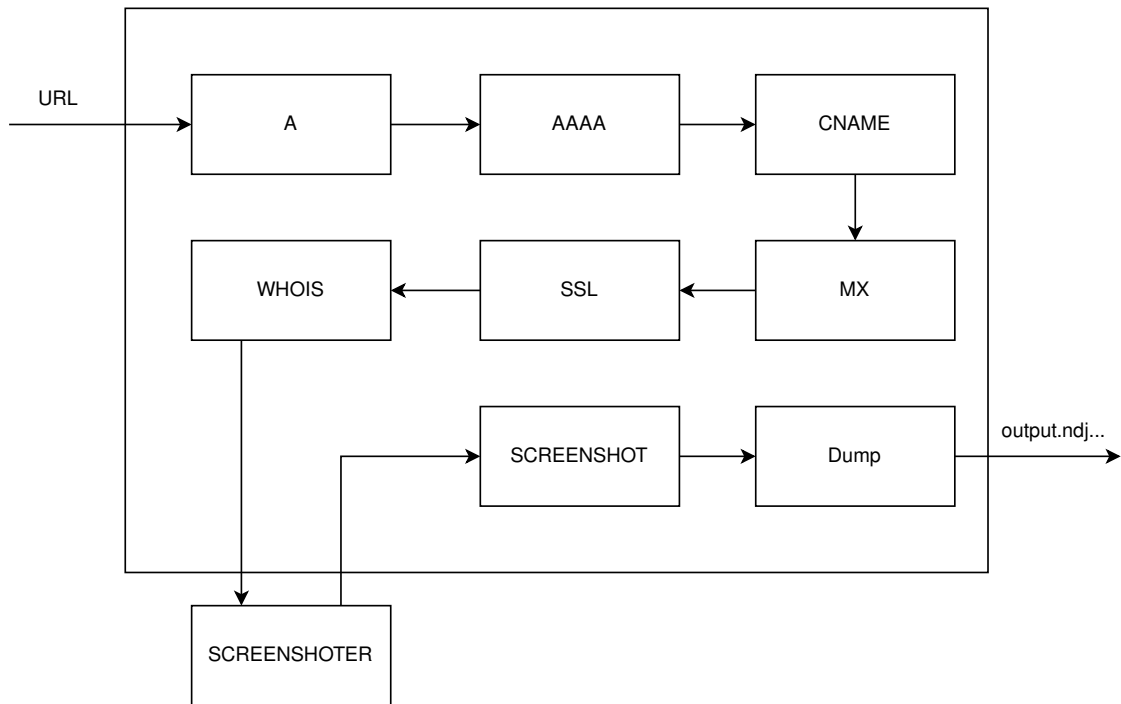


Figure 4.2: Program's execution in steps: upon receiving the URL, the application resolves the DNS, then catching the SSL certificate and the WHOIS information before requesting the screenshot from the external application

In order to find this value, I tested the application multiple times with smaller datasets, of about 500 websites, and I evaluated the amount of blank images and images of pages which were not fully loaded at the moment of the capture in the screenshot folder. The tested delay interval ranged from half a second up to five seconds, in steps of half a second. From this range, I found the two second delay to be the best compromise between speed and accuracy, since the number of unwanted images did not drop consistently during the following executions.

The diagram at Figure 4.2 shows the flux of actions since a new website starts its processing. In other words, it answers the question "how does the Domain class deals with a new domain?".

Firstly, the script asks a DNS question, in order to try to resolve the A address of the domain, observing if it is hosted somewhere and getting more information after: In order to further optimize the results and observe future correlations between DNS records and legitimacy, I also projected the script to collect AAAA (IPv6), CNAME (to check if the page redirects somewhere) and MX.

After all DNS records data is collected, a socket is opened at the domain's 443 port, in order to obtain its SSL certificate, check its legitimacy and who issued it.

Lastly, the instance tries to obtain the WHOIS records of the domain, in an effort to find out who owns the domain, date of acquisition, number of owners and other relevant information.

If the data is consistent, that is, if there is at least one A or CNAME record, the domain is sent for analysis in the Screenshooter Core application. It would not make any sense to analyze domains with no A or CNAME record, since they are not pointing anywhere.

4.1.2 Screenshooter

As I mentioned earlier, I built the Screenshooter application in C#, using the .NET 7 framework and on the top of the Selenium Webdriver library.

Several libraries were developed over the course of the years to allow web crawling. Some of the most famous ones include *wget* and *curl*, packages that date from 1996 and are still widely used especially in Unix-like systems, like Linux and Ubuntu. However, the main issue with these tools that disallow their use for this project is their lack of support for JavaScript (JS), which is crucial in modern websites.

With phishing websites, it is not different at all: they most probably use JS in some extent to show alerts, modals, banners, popups, track data, make requests etc. It is crucial that this web crawler sees the webpage exactly like an unsuspecting victim would.

Therefore, Selenium Manager comes in handy due to the option to choose which web driver and browser I want to use in my script, and this means that JS scripts will run as normal just as they would in a normal browser.

I chose the Chromium webdriver and the Chrome Web Browser [16] due to the fact that Google Chrome is the most used browser in the world, and using it as the application's browser would replicate at least 68% of the internet users' browser environment, according to the Cloudflare Radar from August 2023

[18].

The screenshooter is an event-driven program, which means that it updates only as a result of a call. In this case, the necessary call for the program to start running is executed in the form of sending a TCP message through the user-defined port.

There are two types of message that I configured the screenshooter to deal with. The first type is simply sending a domain name, and when it is received it is immediately pushed into the queue. The second one happens by sending the work "finished", which lets the driver know that no more domains are coming in and that the Screenshooter should finish as soon as its queue empties.

A step by step analysis of Figure 4.3 shows that:

1. The TCP listener receives the message containing a domain. The received domain is enqueued in an asynchronous queue (ConcurrentQueue [19]), which triggers an event (OnAnyQueueChanged)

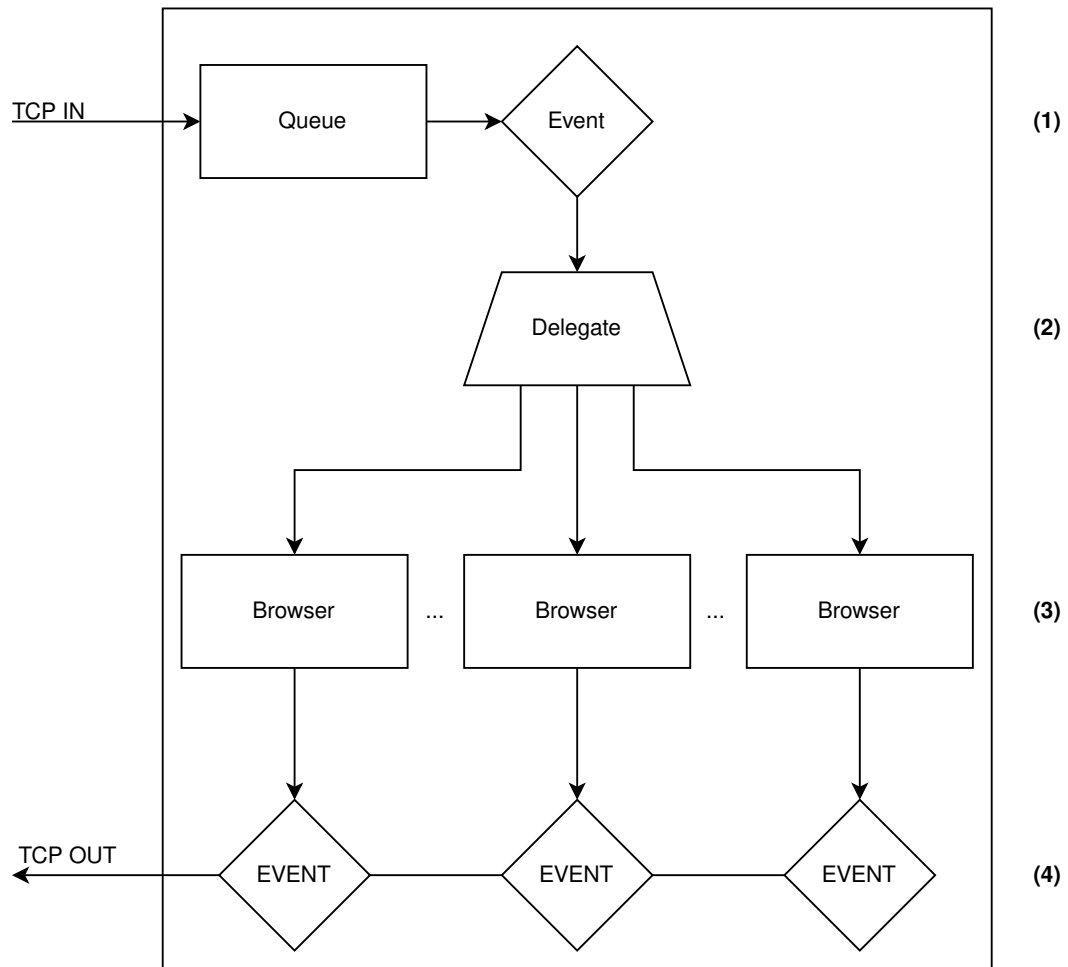


Figure 4.3: Screenshoter abstraction - The domain received via TCP is added to the queue of candidates to be processed. The browsers receive these domains whenever they are free, sending the screenshot path back to the Python script via TCP

2. The event delegate checks if there is any available instance of a Browser in the Browsers' ConcurrentQueue and any domain in the Domains' ConcurrentQueue.
 - If there are no domains in the queue, the program is not going to do anything. If the *finished* signal was also received, then the program understands that all domains have been processed and exits.
 - If there are no browsers available, the domain stands waiting in the queue and the program will not do anything as well.

- If there is a browser available, the first domain in the domains queue is dequeued and sent to the first browser instance in the browsers' queue for screenshotting. Proceed to (3).
3. The selected browser instance visits the sent domain's home page and screenshots it.
 4. In either case of success or failure, the browser is enqueued again in the Browser Manager's browsers concurrent queue and the *OnAnyQueueChanged* event is triggered once more. Repeat from (2)
 - If a screenshot was taken, its path is sent via TCP (TCP OUT). Otherwise, a failure message is sent back.
 - The python script receives the feedback and writes the obtained information into the output file (Dump method in Figure 4.2).
 5. If the TCP listener received the end signal, indicating that all the domains have been sent to the Screenshoter, the program waits for the domains' queue to empty and stops its execution.

4.2 Flexibility and optimizations

The number of parameters allow for the user to choose several information: time limit for screenshot execution, debugging flags, number of processing threads, DNS name servers and output domains. The goal of this flexibility is to facilitate the distribution of the code, making it possible for users to run it with a performance that is condizent with their machinery. Moreover, this is important since it also allows horizontal scalability, by not demanding expensive hardware and running in slower machines.

Many aspects of the script were chosen for practical reasons. Since it is crucial to handle the largest amount of websites as possible as quickly as possible, it is fundamental to reduce as much as possible the execution time. While it would be possible to make more complex calculations and functions, this would consume precious amount of memory and time, which would compromise the scalability. Therefore, a single screenshot of the web page is taken after it is fully loaded.

The most time-consuming and resource-consuming step of the algorithm is taking the screenshot. The page has to load completely, so that the image contains all the information that would be shown to a regular user visiting the web page. Therefore, a higher amount of threads is advised to make the execution smoother. The execution time is inversely proportional to the number of threads, given that they can run smoothly on the CPU.

4.3 Docker

In order to facilitate the implementation of the algorithm in virtual clusters, there is a ready Dockerfile in its repository, allowing its image creation and it can be easily run inside a container. This makes it so that every device can run the same program with different parameters, distributing even more the load of the algorithm. With the Docker virtualization, I could write all the applications targeted at the Ubuntu-x64 CPU architecture and it allowed me to use Linux-specific libraries instead of searching for cross platform ones and compile to multiple operating systems.

4.4 Dashboard

The final part of the methodology is the creation of a dashboard aggregating all the results achieved in the execution period. This step is an important part of the data visualization and in the classification process, since the interface should provide a way for manually labeling the images and helping find suspicious websites by inspection.

Since the data collected is in the format of a ndjson document, I decided that the best way to store all data is in a MongoDB database. The documents of this database are consolidated in a table, and each row represents a single execution for a domain.

When clicking on a row of the table, the user is presented with all the collected information for the selected row's website in the selected date. If there is a screenshot, it is displayed as well. With this feature, users can monitor daily activities of websites and try to spot a malicious domain by inspection.

Chapter 5

Results

5.1 Testing Dataset

I constructed a testing dataset considering the limitations of the machine dedicated to run it: an Ubuntu-based computer with 4 cores, 8 GB of RAM and 20 GB of total storage. Therefore, I prepared a small subset of the previously mentioned domain list, coming from Valentim et. al containing 46 target brands considering their number of related/homophone domains, the target business sector - criteria which I used to selected several names from the banking/financial sector and from retailers sector, since they are often targets for cyber criminals who abuse their name and reputation to obtain large financial returns.

Overall, the list contains 2780 domains, and each day about 2400 domains can have their information collected, according to testing results. The following section shows the results obtained with this dataset and goes into details for a couple of interesting target brands.

5.2 Dashboard

In order to visualize the results in real time and allow for more transparency, I decided to showcase the results that I obtained during testing in a dashboard, hosted in my private server, as a proof of concept.

I stored all the collected data in a MongoDB collection, and I uploaded all the images to my server. Therefore, whenever an user decides to check the individual result for a website, they can see how did it look like in the selected date.

The figures 5.1 to 5.6 display the website and the next paragraphs describe its functionality. Each caption contains a brief overview of a different functionality of the dashboard.

As seen in Figure 5.1, the user can see a list of columns in the top of the page.

By clicking on each column name, the user toggles its exhibition, allowing the visitor to choose only the data that is important for their purposes. This allows a quick visualization and comparison of punctual data, for example comparing SSL certificates or WHOIS data from various websites.

The next important feature is the Custom Search Builder (Figure 5.3). The user can filter the table with their own criteria and try to find, this way, potential risky websites, unlabeled websites or or any website belonging to a group of interest. Also, it is possible to export the current table's result in the format of PDF, CSV and print. This is due do the native JS DataTables extension, that powered this visualization tool.

Upon clicking a website on the table (a row), a modal appears containing all the information of the current website (Figure 5.4 and Figure 5.5), and if it is not labeled, the user can manually label it (Figure 5.6). This is an important step for the future, facilitating the development of an artificial intelligence classifier with the labeled data.

The possible categories that a website can be classified in are displayed on the Table 5.1.

Finally, the dashboard displays a gallery of currently filtered websites, a side-by-side picture comparison (Figure 5.7). This is a visualization tool that helps quick detection of risky websites, avoiding the need to open every single domain by hand. When a user clicks a screenshot, the website displays the same modal that it would display if the user interacted with a table row.

Label	Meaning
Legitimate	Website is from a legitimate company/author and poses no threats
Dubious	Uncertain, it is not possible to affirm that the website is legitimate and there are no signs of malicious use. Low chances of changing ownership in the near future
Suspicious	Strong evidences that the domain is used for malicious purposes
Whitepage/Error	Website returned an empty page or an error page (e.g. 404 not found, 403 forbidden, nginx error etc.)
Parked	A parked domain. This is the most common label in the dataset and it is extremely dangerous, since its ownership can change at any point in time

Table 5.1: Possible labels for websites

Phishing Crawler Home

Results

Toggle Columns:

timestamp main_domain domain ssl_cert a_info aaaa_info cname_info mx_info whois_info screenshot_file_path label timing end_tin

Custom Search Builder

Add Condition

Copy CSV Print Search:

timestamp	domain	screenshot_file_path	label
13/11/2023	adobes.de	2023-11-13 14-20-21 - adobes.de.jpeg	parked
13/11/2023	adobai.com	2023-11-13 14-20-22 - adobai.com.jpeg	whitepage/error
13/11/2023	adobes.ch	2023-11-13 14-20-26 - adobes.ch.jpeg	legitimate
13/11/2023	adobi.me	2023-11-13 14-20-29 - adobi.me.jpeg	null
13/11/2023	adobs.network	2023-11-13 14-20-22 - adobs.network.jpeg	whitepage/error
13/11/2023	adobit.com	2023-11-13 14-20-23 - adobit.com.jpeg	null
13/11/2023	adob.eu	2023-11-13 14-20-27 - adob.eu.jpeg	parked
13/11/2023	adob.de	2023-11-13 14-20-27 - adob.de.jpeg	null
13/11/2023	adob.xyz	2023-11-13 14-20-33 - adob.xyz.jpeg	parked
13/11/2023	alien.moe	2023-11-13 14-20-30 - alien.moe.jpeg	legitimate

Showing 1 to 10 of 29,621 entries 1 row selected Previous 1 2 3 4 5 ... 2,963 Next

Figure 5.1: React website containing fetched results

Results

Phishing Crawler Home

Results

Toggle Columns:

timestamp main_domain domain ssl_cert a_info aaaa_info cname_info mx_info whois_info screenshot_file_path label timing end_time

Custom Search Builder

Add Condition

Copy CSV Print Search:

timestamp	domain	screenshot_file_path	label	end_time
13/11/2023	adobes.de	2023-11-13 14-20-21 - adobes.de.jpeg	parked	1.9193005561828613
13/11/2023	adobai.com	2023-11-13 14-20-22 - adobai.com.jpeg	whitepage/error	3.300568103790283
13/11/2023	adobes.ch	2023-11-13 14-20-26 - adobes.ch.jpeg	legitimate	4.226350545883179
13/11/2023	adobi.me	2023-11-13 14-20-29 - adobi.me.jpeg	null	5.254292726516724
13/11/2023	adobs.network	2023-11-13 14-20-22 - adobs.network.jpeg	whitepage/error	5.293579339981079
13/11/2023	adobit.com	2023-11-13 14-20-23 - adobit.com.jpeg	null	5.455219984054565
13/11/2023	adob.eu	2023-11-13 14-20-27 - adob.eu.jpeg	parked	7.2385900020599365
13/11/2023	adob.de	2023-11-13 14-20-27 - adob.de.jpeg	null	5.9908061027526855
13/11/2023	adob.xyz	2023-11-13 14-20-33 - adob.xyz.jpeg	parked	8.973571300506592
13/11/2023	alien.moe	2023-11-13 14-20-30 - alien.moe.jpeg	legitimate	7.846379041671753

Showing 1 to 10 of 29,621 entries 1 row selected Previous 1 2 3 4 5 ... 2,963 Next

Figure 5.2: The column `end_time` now appears in the table, showing how long it took for the website to be fully processed

Custom Search Builder (5) Clear All

main_domain Equals adobe

ssl_cert Equals null

ssl_cert Contains Let's Encrypt

ssl_cert Contains Cloudflare

label Equals null

Add Condition

Copy CSV Print Search:

timestamp	domain	screenshot_file_path	label	end_time
13/11/2023	adobi.me	2023-11-13 14-20-29 - adobi.me.jpeg	null	5.254292726516724
13/11/2023	adobit.com	2023-11-13 14-20-23 - adobit.com.jpeg	null	5.455219984054565
13/11/2023	adob.de	2023-11-13 14-20-27 - adob.de.jpeg	null	5.9908061027526855
14/11/2023	adobs.network	2023-11-15 00-22-49 - adobs.network.jpeg	null	1.3263249397277832
14/11/2023	adob.de	2023-11-15 00-22-52 - adob.de.jpeg	null	2.0456619262695312
14/11/2023	adobi.me	2023-11-15 00-23-02 - adobi.me.jpeg	null	5.317889928817749
14/11/2023	adobit.com	2023-11-15 00-22-56 - adobit.com.jpeg	null	5.67396092414856
15/11/2023	adob.de	2023-11-15 15-13-31 - adob.de.jpeg	null	1.7631218433380127

Figure 5.3: Table results with example custom query

Details ✕

[-]

timestamp:
13/11/2023

main_domain:
adobe

domain:
adob.eu

ssl_cert:
[+]

a_info:
[+]

aaaa_info:

cname_info:

mx_info:

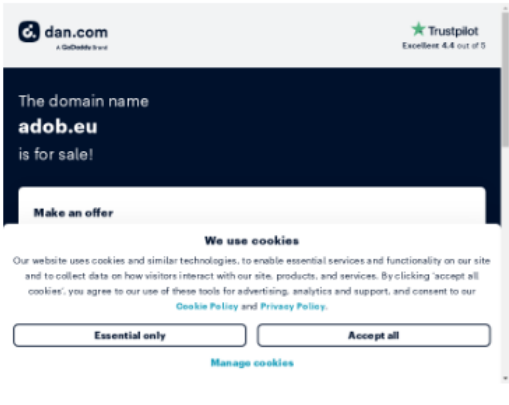
whois_info:
[+]

label:
parked

labeled_by:
179.209.46.202

timing:
[+]

end_time:
7.2385900020599365



Label: parked
Labeled By: 179.209.46.202

Figure 5.4: Modal with complete information of *adob.eu*, a parked domain

Details ✕

[-]

timestamp:
13/11/2023

main_domain:
adobe

domain:
adob.eu

ssl_cert:

[-]

subject:

[-]

commonName:
adob.eu

issuer:

[-]

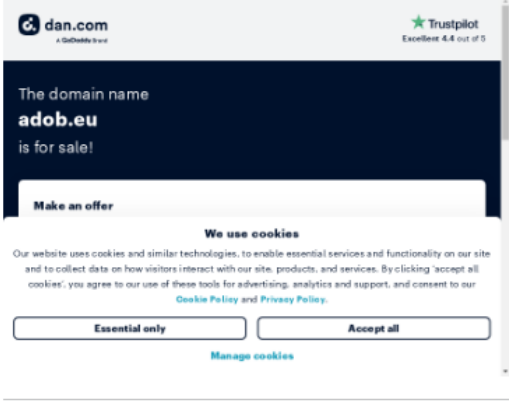
countryName:
US

organizationName:
Let's Encrypt

commonName:
R3

version:
3

serialNumber:
03AF17323E01085D49CC2CAA61270C2A



Label: parked
Labeled By: 179.209.46.202

Figure 5.5: Modal with expanded information about the SSL certificate

Details

[-]

timestamp:
13/11/2023

main_domain:
allianz

domain:
allien.co

ssl_cert:
[+]

a_info:
[+]

aaaa_info:

cname_info:


mx_info:

whois_info:
[+]

label:

timing:
[+]

end_time:
2.4718165397644043



How would you classify this website?

- Legitimate
- Dubious
- Suspicious
- Whitepage/Error
- Parked

Close

Figure 5.6: Modal showing information about a not yet labeled parked domain

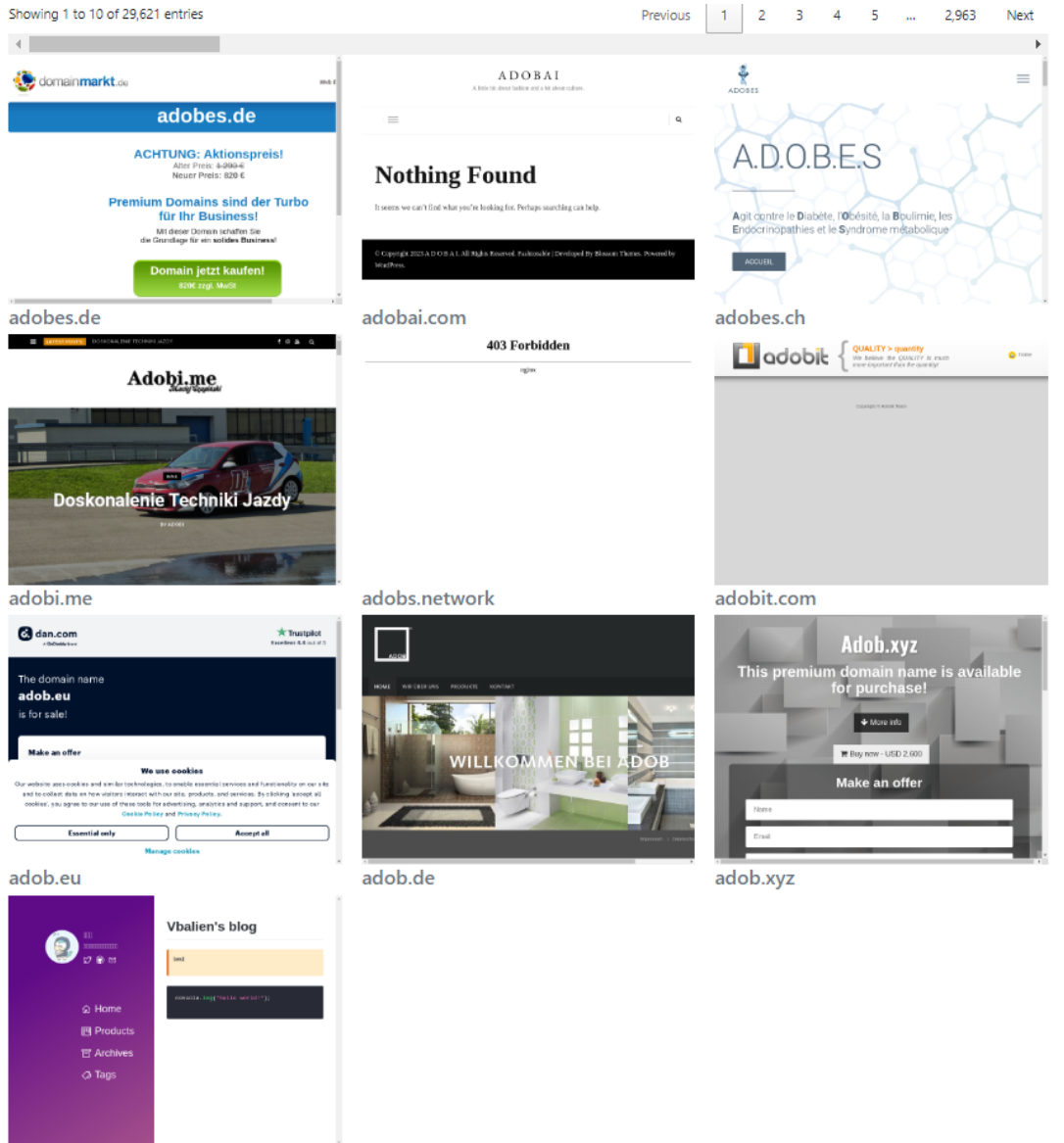


Figure 5.7: Screenshots displayed as a gallery in the crawler website

5.3 Study cases

As a study case, the crawler was used to monitor 12 homophones of the "*adobe.com*" website, 177 homophones of the "*amazon.com*" website and 101 homophones of "*apple.com*". During thirteen days of execution, the algorithm consistently obtained screenshots from the monitored websites and information on their SSL certificates, DNS records and WHOIS status. The aggregated results are shown in the Table 5.2 for Adobe, Table 5.3 for Amazon and Table 5.4 for Apple.

Day	A	MX	Screenshot	Total
1	11	6	11	11
2	11	7	11	11
3	11	7	11	11
4	11	7	11	11
5	11	7	10	11
6	11	7	11	11
7	11	7	11	11
8	11	7	11	11
9	11	7	11	11
10	11	7	11	11
11	11	7	11	11
12	11	7	11	11
13	11	7	11	11

Table 5.2: Number of outputs per day - Adobe (There were no CNAME nor AAAA entries)

By inspection of the tables above, it is clear that the Crawler yields consistent results. The number of records obtained after each execution for these websites suggests that the tool is indeed functional and able to collect correct and consistent data from the provided list of websites.

After I obtained these results by running the crawler, I stepped up to manually label a couple of domains. In order to track daily evolution, I tracked *Adobe's* homophones and labelled its data during the whole period. I also labeled *Amazon's* and *Apple's* homophones' data for the last day of execution.

Upon labeling, I found that Amazon is the owner of a large number of homophone domains that redirect to their own website. This shows that they have been taking action against possible phishing attacks, avoiding cybersquatting by owning multiple similar domains. Unfortunately, not every brand has the same consciousness of this issue and their users might inadvertently fall victims to malicious activities.

Labeling is an exhaustive process, since there are about 2400 new screenshots every day and most legitimate web pages do not change significantly their content

Day	A	AAAA	MX	Screenshot	Total
1	147	66	56	129	147
2	156	75	63	136	156
3	155	74	58	131	155
4	155	75	62	136	155
5	151	74	58	132	151
6	156	77	65	144	156
7	156	77	65	144	156
8	156	77	65	145	156
9	156	77	64	142	157
10	156	77	65	144	156
11	157	77	65	144	157
12	156	77	65	142	156
13	155	77	63	145	155

Table 5.3: Number of outputs per day - Amazon (There was only one CNAME entry all the days)

Day	A	AAAA	MX	Screenshot	Total
0	87	16	65	80	87
1	88	16	71	76	88
2	86	17	67	74	86
3	90	16	72	79	90
4	90	16	72	76	90
5	92	17	75	88	92
6	92	17	75	89	92
7	92	17	73	89	92
8	92	17	74	90	92
9	92	17	74	89	92
10	92	17	75	88	92
11	91	17	73	88	91
12	92	17	75	89	92

Table 5.4: Number of outputs per day - Apple (No CNAME entries)

from one day to another. In order to label the entire result dataset, of about thirty thousand domains, it would take a collaborative effort.

Therefore, it would be of great use a tool that compares websites' information from one day to another, especially WHOIS and screenshot data. A supervised learning tool can be developed in the future making use of the current available data and future labels, complementing this work by automatically classifying them - blocking the malicious domains more quickly than current defense strategy and allowing for zero-day phishing defence.

It is extremely difficult to catch phishing websites on the act since they are also protected against crawlers. Most of them are going to display malicious information after user interaction only, something that invalidates automated actions. This way, phishing websites can "disguise" themselves and protect against automatic detection methods.

Moreover, phishing websites are also well-hidden behind parking domains. Parked domains are one of the most dangerous and common websites used for malicious context. Since they are available for sale, they are extremely volatile (their ownership can suddenly change) and they need to be constantly monitored. Zero-day attacks often come from parked domains that turn themselves into a phishing website for a single day, then going back to its parked domain status.

The results for the monitoring of the Adobe target phonetically similar domains over the course of 13 days are shown on the Table 5.5.

The table shows consistent results for the analyzed time, and it is important to notice that during the observations one website changed labels from Whitepage/Error to Legitimate, and in the tenth day one website left the parked label to the whitepage/error category for a single day.

In this case, the website *adob.com* had a failure on the *nginx* application for a single day. However, it would be very much possible as well that malicious actors tried a single-day attack in the website, thus the importance of constantly monitoring. The page could have changed, this day only, into a page that uses Adobe's reputation for phishing purposes.

In general, domain registrars offer plans starting from 28 days, a monthly bill. Therefore, domain ownership can change very suddenly and daily monitoring is necessary.

In Amazon's case (Table 5.6), it is noticeable that most domains are classified in the Legitimate category. Besides Amazon's effort and the fact that Amazon is the legitimate owner of a great number of similar domains to their name, I must state that a legitimate company, *AMAZONE*, an agricultural machinery manufacturer, also contributes to this positive result, since they own 71 domains from the 96 legitimate ones, while Amazon owns 23 and there are 2 other legitimate businesses running under a name similar to Amazon.

However, Apple's case is very different from Amazon's one. The 19 domains

Day	Legitimate	Dubious	Suspicious	Whitepage/Error	Parked
1	2	1	1	3	4
2	2	1	1	3	4
3	2	1	1	3	4
4	2	1	1	3	4
5	2	0	1	3	4
6	3	1	1	2	4
7	3	1	1	2	4
8	3	1	1	2	4
9	3	1	1	2	4
10	3	1	1	3	3
11	3	1	1	2	4
12	3	1	1	2	4
13	3	1	1	2	4

Table 5.5: Labels for *Adobe* homophones, daily

Day	Legitimate	Dubious	Suspicious	Whitepage/Error	Parked
13	96	15	9	11	14

Table 5.6: Labels for Amazon’s homophones in the last day of execution

that I classified as legitimate businesses (those that exist for a long time and there is little risk of the domain changing ownership) are all from different organizations except for *aple.ro* that redirects to Apple’s official website. The full results are in 5.7, and must be interpreted in relation to the results table present in the Crawler website.

In Apple’s case, there is a seriously high number of dubious domains (23 domains). Due to their recent registry, lack of proper SSL certificate and lack of information, they can not be deemed as malicious or legitimate, and I classified them in this category for continuous monitoring in the future.

It is important to notice that legitimate domains can be monitored less often, since they already belong to trustworthy organizations that exist for decades and there is a very low probability that they will become a malicious website in the near future. Therefore, the main source of concern for them is to guarantee that the company will keep its ownership after the expiry of the current registration.

Day	Legitimate	Dubious	Suspicious	Whitepage/Error	Parked
13	19	23	8	27	11

Table 5.7: Labels for Apple’s homophones in the last day of execution

5.4 Further Considerations

I considered the possibility to use a single, monolithic python script to run the whole project, and while it was possible to construct the whole project in a single program with object-orientation and multi threading, the two-application approach presented better performance and less memory management issues.

5.4.1 Memory management

In the early versions of the project, I used a monolithic Python code to serve the application. The main issue is that the flow control was difficult to manage and the memory management was very poor when using workers for multi-threading.

Spacial complexity of the algorithm is supposed to be constant, which means that the memory usage should not rise over time and remain constant on average for the whole program execution. Stochastically, the actual memory consumption is expected to float around a value, since there are pages that require more storage, with more images, fonts, ads and more information overall. However, statistically, it is expected that on average this value should not systematically increase during the execution.

Empirically, I experienced that the equivalent code used in Python rises unbounded until memory exceptions occur and breaks the execution. This is a serious threat for the application consistency and could compromise data integrity and correctness.

For a complete program running with 8 threads and the .NET 7-based screenshotter (Figure 5.8), the Docker container accused around 1.3 GB of memory usage, while 2.65 GB for 24 threads (5.9).

The same pattern can not be observed in the Python execution, as the Figure 5.10 shows. Instead of keeping a constant memory usage, it grows over time and eventually might exceed the expected RAM memory usage of the container, causing crashes and making it impossible to run the whole project.

Although these images refer to single runs, I observed this pattern in hundreds of testing runs, with the monolithic code always exceeding memory usage and the final version being consistent in terms of memory usage.

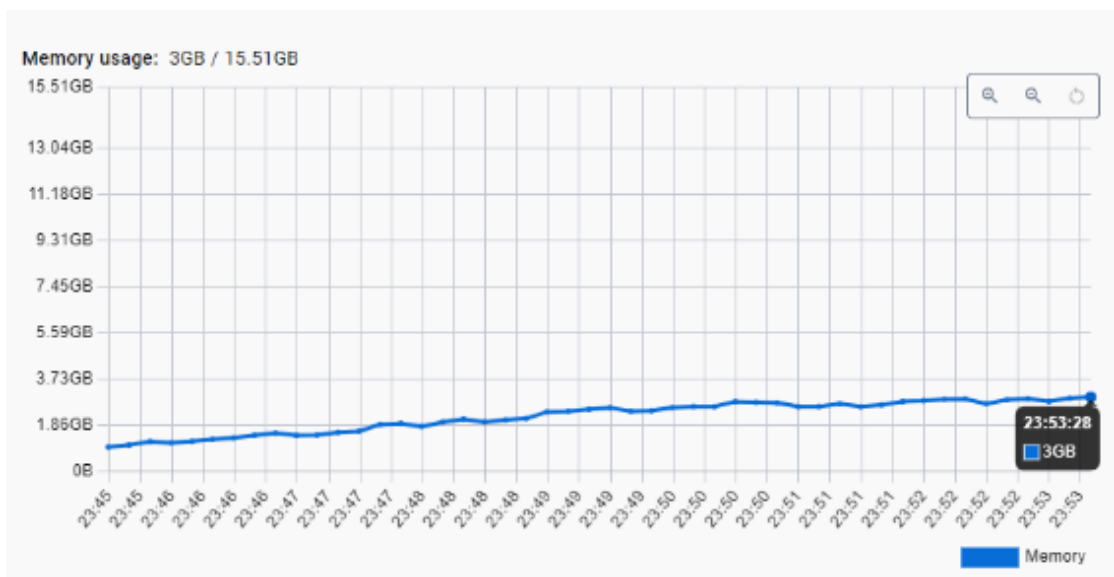


Figure 5.10: Memory usage running container with python screenshotter and 8 threads

5.4.2 Performance

It is worth mentioning that using a larger number of threads does not necessarily mean that the program is going to perform better. Several factors limit the execution time, including but not limited to CPU availability, CPU speed, threading, internet speed, internet bandwidth, servers availability, firewall, cache etc.

In fact, the usage of higher number of threads (more than 50% of the available threads) is prejudicial to the execution. Empirically, I noticed that the best number of threads to run the application is about half of the total number of threads on the CPU. I believe that for a larger number of threads, they start interfering with each other, overflowing cores of the CPU and reducing performance. It might be the case that packets are thrown away due to instability and the screenshotter will not be able to work as intended.

Even if the system is isolated, running the program in a personal computer or home network can be prejudicial, since the usage of the computer for daily activities also interferes with the performance. Tasks like writing reports, filling spreadsheets, navigating in the internet, combined with other routine tasks of the OS, like antivirus, system scanning, clocks, e-mails and regular services consume CPU cycles and use memory that affect the performance of the crawler. Therefore, ideally the project should run in a dedicated cluster, which can lead to optimal CPU usage, network consumption etc.

Another factor affecting the results is that some of the domains might not be screenshotted in due to the restrictions imposed by the Internet Service Provider (ISP). In the test runs, hundreds of domains were already blocked in the crawler's network due to phishing activities, and I was able to find out this information when running the program in a different network from another ISP.

While some ISPs block the malicious domains without further explanation, by simply bouncing back the request and thus not fulfilling it, other providers actually send a response informing that a given domain is now blocked because of its history of being used for phishing or other malicious activities.

Figure 5.11 shows an example of phishing alert. One ISP displayed an alert when the crawler tried to access a blocked domain, whereas other ISP just blocks the website without giving further information.

The message is in Portuguese and can be translated as follows: "INFECTED WEBSITE", "This website was **blocked because it is a phishing webpage**. We recommend you to leave this page", followed by a "Exit" button or a link allowing the user to "Proceed anyway".

The fact that dozens of domains yielded this same message indicates that ISPs are constantly working to block malicious websites and are committed to stop phishing and reduce cybernetic crimes. More importantly, it corroborates with the argument that it is possible to detect phishing websites with a web crawler

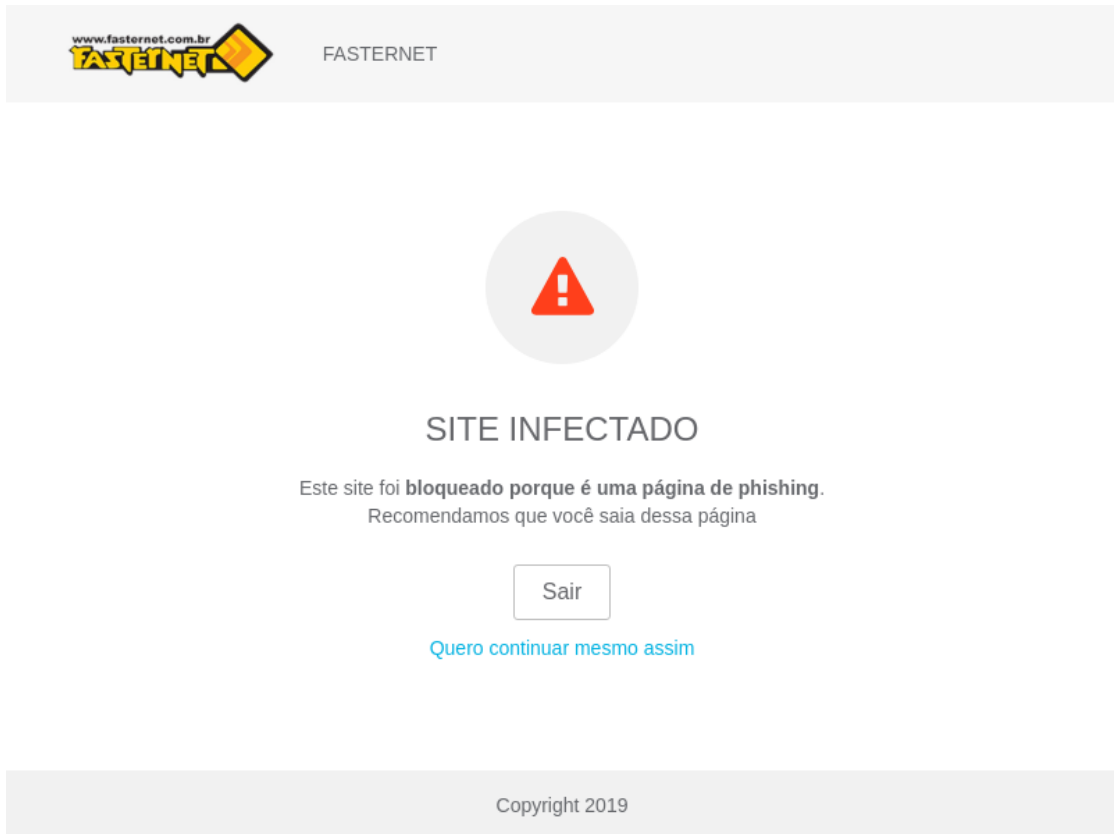


Figure 5.11: Blocked site alert - amazen.mom

and that the use of a GAN to generate candidates of cybersquatting is a valid and working approach.

5.4.3 Synchronicity

Finally, I would like to comment that using a monolithic Python script for all the execution would delay the process of obtaining the DNS, SSL and WHOIS information from the domains. Since the screenshot is the operation that takes the longest to occur, the bottleneck of the whole project, it is crucial that it takes place asynchronously from the data gathering thread. Therefore, having separate queues between the data collection and the screenshot allows for a faster execution and a larger separation of functions, asynchronous processing and a better performance overall.

Due to all the reasons mentioned above I decided that the screenshotter external application is indeed the way to go in this project and it's code is also fully available in GitHub [4] for transparency reasons. I strongly believe that this project is of sum importance in cybersecurity terms, and I believe that putting it in an open-source license to allow contributions from all over the world is indeed crucial to scale the project and fight cyber crime as efficiently as possible.

Chapter 6

Conclusions and Future Work

Finally, the main conclusion that I take from this project is that **it is possible** to monitor phishing and cybersquatting candidates using a web crawler.

The tool and the overall system that I developed for the purpose of this thesis has the potential to be of great use against phishing targets by collecting useful data, being another step towards the process of automatically finding, fetching and classifying malicious websites and protecting internet users from scams.

This data collection generates important proof that can be used to make a case against phishing candidates, allowing internet service providers to insert malicious domains in their block lists with more agility. Moreover, by doing this, the system has the potential to save victims from losing their savings to cyber criminals, which is the most important contribution of this work.

Phishing and cybersquatting are very serious cyber crimes that create immeasurable losses to victims all over the world, financially, emotionally, in safety terms, and also harm people and companies reputations. They move an extremely large amount of money every year by stealing money and data from victims and they manage to expand their activities every year.

By using this tool, I successfully and consistently collected information from thousands of websites, proving the effectiveness of the crawler. Data analysis from the last chapter corroborates with the argument that it is possible to automatically and systematically generate candidate domains, collect information about them and detect phishing targets.

Future work can start from this point and develop a machine learning algorithm on the top of these results to complete the cycle of generating, crawling and detecting risky websites toward the common goal of making the internet a safer environment.

Bibliography

- [1] Federal Bureau of Investigation. *Internet Crime Report (2022)*. 2023. URL: https://www.ic3.gov/Media/PDF/AnnualReport/2022_IC3Report.pdf (cit. on pp. 1, 2).
- [2] *wget*. URL: <https://www.gnu.org/software/wget/> (cit. on p. 2).
- [3] *curl*. URL: <https://curl.se/docs/manpage.html> (cit. on p. 2).
- [4] Allan Brunstein. *Crawler GitHub repository*. 2023. URL: <https://github.com/allanbru/thesis-repo> (cit. on pp. 3, 36).
- [5] Rodolfo Valentim, Idilio Drago, Martino Trevisan, Federico Cerutti, and Marco Mellia. «Augmenting Phishing Squatting Detection with GANs». In: *Proceedings of the CoNEXT Student Workshop*. CoNEXT-SW '21. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 3–4. ISBN: 9781450391337. DOI: 10.1145/3488658.3493787. URL: <https://doi.org/10.1145/3488658.3493787> (cit. on p. 5).
- [6] Lee Joon Sern, Yam Gui Peng David, and Chan Jin Hao. «PhishGAN: Data Augmentation and Identification of Homoglyph Attacks». In: *2020 International Conference on Communications, Computing, Cybersecurity, and Informatics (CCCI)*. 2020, pp. 1–6. DOI: 10.1109/CCCI49893.2020.9256804 (cit. on p. 5).
- [7] Rachna Dhamija, J. Doug Tygar, and Marti Hearst. «Why Phishing Works». In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2006, pp. 581–590 (cit. on p. 5).
- [8] Akshat Gupta, Laxman Singh Tomar, and Ridhima Garg. *GlyphNet: Homoglyph domains dataset and detection using attention-based Convolutional Neural Networks*. 2023. arXiv: 2306.10392 [cs.CR] (cit. on p. 6).
- [9] Wikipedia contributors. *Kali Linux — Wikipedia, The Free Encyclopedia*. [Online; accessed 25-November-2023]. 2023. URL: https://en.wikipedia.org/w/index.php?title=Kali_Linux&oldid=1186771493 (cit. on p. 6).
- [10] Cloudflare. *What is a DNS A record?* URL: <https://www.cloudflare.com/learning/dns/dns-records/dns-a-record/> (cit. on p. 8).

BIBLIOGRAPHY

- [11] Cloudflare. *What is a DNS AAAA record?* URL: <https://www.cloudflare.com/learning/dns/dns-records/dns-aaaa-record/> (cit. on p. 8).
- [12] Cloudflare. *What is a DNS CNAME record?* URL: <https://www.cloudflare.com/learning/dns/dns-records/dns-cname-record/> (cit. on p. 8).
- [13] Cloudflare. *What is a DNS MX record?* URL: <https://www.cloudflare.com/learning/dns/dns-records/dns-mx-record/> (cit. on p. 9).
- [14] Cloudflare. *What is SSL (secure sockets layer)?* URL: <https://www.cloudflare.com/learning/ssl/what-is-ssl/> (cit. on p. 9).
- [15] *Selenium*. URL: <https://www.selenium.dev/> (cit. on p. 9).
- [16] 2023. URL: <https://chromium.googlesource.com/chromium/src/+/master/headless/README.md> (cit. on pp. 10, 13, 15).
- [17] *Docker*. URL: <https://www.docker.com/> (cit. on p. 10).
- [18] Cloudflare Data Insights Team. *Browser Market Share Report for 2023 Q2*. <https://radar.cloudflare.com/reports/browser-market-share-2023-q2>. [Accessed 14-11-2023]. 2023 (cit. on p. 15).
- [19] URL: <https://learn.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentqueue-1?view=net-7.0> (cit. on p. 15).