Ollscoil Teicneolaíochta an Atlantaigh

Atlantic Technological University

Malware Detection using Dynamic Heuristic Analysis

By
Declan Kelly

for
Daniel Cregg

April 22, 2023

Minor Dissertation

Department of Computer Science & Applied Physics,
School of Science & Computing,
Atlantic Technological University (ATU), Galway.

B.Sc. (Hons) in Software Development

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

The word malware is a portmanteau of the words "malicious" and "software". It is used to describe software that can perform harmful actions. These actions include the destruction of data, the exfiltration (extracting data from a computer over a network) of sensitive information from a computer or holding it for ransom (meaning it encrypts files with a key and holds the key for ransom). Any one of these can have significant consequences for the victim. Malware is a never-ending threat to organisations and individuals, malware authors are getting more and more sophisticated, and they have begun to use new techniques to avoid detection from traditional anti-malware solutions.

For the Applied Project and Minor Dissertation module, I've been tasked with creating a significant software project. I decided to create a platform that can detect malware based on its behaviour, using dynamic heuristic analysis, which involves executing files in a sandboxed environment and observing what the file does during its execution, then comparing that behaviour to known malicious actions. When I say behaviour, I refer to the way software interacts with the operating system. Examples of this include reading and writing files. A sandbox is a way of isolating potentially harmful processes on your computer, limiting the resources it can access, sensitive files are an example of something you don't want malware accessing.

1.2 Literature

Traditional anti-malware scanning techniques, such as static analysis (which involves looking for known patterns of strings or bytes, in a file), can be easily bypassed. Malware authors often use techniques such as obfuscation [1] to purposely hide data in the file from analysers.

Packing is used to compress file contents, and have them decompress on execution, and sometimes malware is even encrypted to further hinder analysis, allowing the malware to have completely different static signatures while exhibiting the same behaviour. An example of a tool, that can do this is UPX, this will be explained below.

There is a lot of information about a program that isn't readily available just by analysing its static form. Once in execution, malicious software will carry out its malicious behaviour, which cannot be easily hidden [2]. This is why dynamic analysis is a much better strategy for detecting malware.

Avoiding detection from static analysis can also be done using tools such as UPX. UPX has the ability to compress executable files [3], hiding artefacts that would be used to identify malware.

1.3 Project Overview

This project will be targeting Microsoft Windows 10/11, it will feature a GUI application that can be installed, allowing users to automatically scan downloaded files in a sandboxed environment. Due to the dynamic nature of the platform (requiring files to be executed), worker computers will need to be set up, to handle scanning tasks. Administrators will be able to perform administrative tasks using a dashboard in their browser. These tasks will include: adding and removing workers, and adding malicious patterns to the database.

The novel aspect of this project is the use of dynamic heuristic analysis, this is an unconventional approach to malware detection, and, at times, can be much more effective than static analysis. This project will bundle everything together into a single platform, allowing users without prior knowledge of the field to use it. As soon as the installer is finished installing, the user can upload files to be scanned immediately.

The platform is designed to be used by enterprises, it can be installed on all the computers on the network. When a user downloads a file, they will be prompted to upload it to the platform for analysis. The user will see a real-time analysis of the file. If the file is deemed malicious, the user will be prompted to quarantine it.

There will be three main sub-projects included in the platform, these being the client, server and sandbox. The dashboard is closely integrated with the server, and although it is found in the server project, it is actually running on the client side during execution. The client is a native GUI application, designed for Microsoft Windows 10/11. To utilise the platform, users must install the client on their computer(s), once installed, it will automatically launch upon system boot.

When a user downloads a file (such as: downloading a file using their web browser), they will be prompted to upload it, to be scanned. The scanning process provides real-time updates on the status of the file, including a log box displaying the currently executing task and as well as a progress bar for scan progress. The client also has a quarantine feature, this is used to safeguard malicious files in a secure location, and includes an option for users to restore files to their original location.

The server project is responsible for establishing communication between the client and the sandbox analysis worker, while also offering administrators a dashboard for carrying out administrative tasks. The administrator will be able to view statistics by looking at the charts, found on the index page, and see the ratio of malicious to clean scans using the pie chart. They can also remove workers, provision new clients and add malicious patterns.

The sandbox analysis workers are responsible for running malware in a secure environment, and recording system call(s) for further analysis. It will also use static analysis as a fallback, the static analysis differs from my approach, in that, it doesn't actually execute the file, it just looks for known patterns of text and bytes, in the file and compares them to a list of definitions it periodically downloads.

1.4 Objectives

The objectives of this project are to:

- Develop a robust and reliable software platform capable of detecting malicious software based on its behaviour and quarantining if it's malicious.
- Ensure the platform's maintainability by producing clean and easy-to-understand code.
- Create a native GUI application for Microsoft Windows 10/11 that provides users with a simple and easy-to-use, way to scan files and monitor the scanning progress.
- Build a user-friendly dashboard for administrators to manage the platform's administrative tasks.
- Establish a server to facilitate communication between clients and the sandbox analysis instances using RESTful APIs.
- Create a secure platform that leverages state-of-the-art security measures to prevent unauthorised access to the platform.
- Produce code that adheres to good design principles, allowing easy maintenance and modification.

1.5 Sub-projects

1.5.1 GUI Application

The source code for the GUI application can be found in the `client/src` directory, it is implemented in Python, utilising the Tkinter library to create the graphical user interface, and using a threading library called "tkthread", to allow the application to do other tasks, without locking up the application's GUI.

The application comprises of three sections: "Main Page" (Caladium), "Quarantine" and "Preferences". The "Main Page" features a text label displaying the current directory being scanned, along with a manual scanning button. Since the execution of Python code requires Python to be installed, PyInstaller is used to bundle together all the necessary components to run the application on any computer.

Unfortunately, PyInstaller does not generate a single executable file, but it instead outputs a directory of files. I am using the "iexpress.exe" tool included in Windows to create an installer that outputs a single executable file. When executed it will install the Caladium application into the `Program Files` directory of the user's computer. The "Preferences" page in the application, offers an "Uninstall" button, offering users the option to remove the application from their computer.

1.5.2 Dashboard

The code of the dashboard can be found in the `server/src/static` directory. [1]

Administrators will be able to login into the dashboard and perform administrative tasks. They will also have the ability to change their password and add new workers to the system. The dashboard is written in JavaScript and is a single-page application. When you first navigate to the dashboard you will be presented with a login page, prompting you to enter your username and password, the default for these is "root" and "root". The password can be changed in the preferences menu. The dashboard will feature four different types of list pages, these are the **clients**, **patterns**, **tasks** and **workers**, pages.

1.5.3 Server

The code for the server can be found in the `server/src` directory. The server is responsible for bridging communication between the clients (GUI application or dashboard) and the sandbox analysis instances. It is also written in Python like the GUI application, using the Flask micro-framework for handling HTTP requests

[1]The dashboard may appear as if it is part of the server but it is actually on the client side, as it runs on the user's computer.

from clients. Data will be persisted using CouchDB, since the server is written in Python, it will be using the pycouchdb library to interface with the CouchDB instance. Data to be stored on the CouchDB instance will include records for each of the list pages in the dashboard listed above.

1.5.4 Sandbox Analysis

The code for the sandbox can be found in the `sandbox/src` directory. The main part of it is written in Racket (a functional programming language). Since this platform is targeting malware, that is designed to run on Microsoft Windows, the sandbox must run on Microsoft Windows.

It makes use of Sandboxie to isolate the malware from the host system, and a tool called Process Monitor is used to log the system calls. It will also use ClamAV to perform static analysis as a fallback, in the case of the dynamic analysis failing to detect malware.

1.5.5 Bot & Docs

The platform also features a promotional page, which gives new users an overview of the project. They are presented with a download button, as well as screenshots and descriptions of the various components featured in the platform. When you click the download button, it queries the GitHub API to fetch the latest GitHub release tag for the client.

The promotional page can be found here: https://g00378925.github.io/caladium/

There is also a chatbot called CaladiumBot, source code can be found in the `bot` directory, it makes use of OpenAI's GPT-3.5 model. [4] The bot can answer questions about the platform. Before talking to you, it has been given a copy of the `README.md`, and using that, it can then answer your questions. To chat with this bot, navigate to the promotional page and press the "Chat with CaladiumBot" button.

1.6 Document Overview

- The "Methodology" chapter will cover the way I developed the project, including my use of the Kanban board and using Git for source control.
- "Technology Review" contains a discussion on my initial research for selecting the technologies for the project and my rationale for choosing them.
- "System Design" will explain how I structured the platform, and talk about the various design decisions I made.

- The "System Evaluation" will be an evaluation on how well the project met the objectives set in this chapter.
- Finally I conclude with the "Conclusion" chapter.

1.7 GitHub URL

The GitHub repository for the project can be found at this URL (You can also find it in Appendix A):
https://github.com/G00378925/caladium

Upon visiting this link, you will be presented with the README, giving a description of the project.
At the bottom of the README, and in Appendix C, you will find instructions for building each component. In each of the directories, you will find each sub-project that was listed above.

A screencast of the project can be found at this URL:
https://www.youtube.com/watch?v=aYbQChGvz88

And this is a screencast of the promotional page and CaladiumBot:
https://www.youtube.com/watch?v=hMskBzCt6kU

Chapter 2

Methodology

This chapter will be discussing the methodology, that I used whilst developing my project along with the research that helped guide it. The primary objective of my research was to identify components, that would allow my platform to detect malware based on its behaviour. These components include Sandboxie and Process Monitor.

Git played an important role during the development of the project. It is also used to host my promotional page, and it uses GitHub Pages to automatically deploy it. I also used GitHub Actions to run a suite of tests on the API endpoints when code was pushed to the repository. To manage tasks and time effectively, I used a Kanban board on GitHub and adopted an agile-style methodology to ensure progress with changing requirements. And I used Docker to create a reproducible environment, to avoid having to install and set up dependencies for different machines. The server will be hosted on the Microsoft Azure Cloud.

2.1 Determining Requirements

I recognised that most anti-malware solutions targeting consumers come with a GUI application for performing scans. I wanted to do the same, which is why there is a GUI application.

There would be a server component and a database to persist data for the server. After some thought, I decided against performing scanning on the server. Instead, I opted to have the scanning done on separate analysis workers. This would isolate any potential malware from compromising the server and allows for multiple scans at the same time.

For communication between the clients and the server, I chose to use RESTful APIs. I chose this because it is now the de facto standard for web applications and is quite simple.

I also included a dashboard to provide a user-friendly interface for administrators, enabling them to perform actions on the platform as a whole without having to do it through a tedious command-line interface.

2.2 Research

When I figured out what I wanted to do my project on (that being malware detection), I conducted research on the various technologies that I could use. My initial focus was on tools that would enable me to log the system calls on Windows. Although I initially considered creating the logging software myself, I quickly realised that this would be too time-consuming, preventing me from creating a fully-featured platform in the time given.

My research led me to discover "Process Monitor" [5], a software by Microsoft that can be used to log system calls. I also explored open-source software that I could use to do static analysis. I was originally going to do only dynamic analysis but there is always the potential of it not detecting malware, and there are already definitions of known malware on the internet. I discovered ClamAV [6] and opted for its Windows-optimised version, ClamWin. This software periodically downloads definition signatures of known malware in the wild.

One of the biggest challenges I faced was finding a way to run potentially malicious software safely in an isolated environment without harming the computer running the software. After exploring the potential options, I settled for Sandboxie. [7] I initially considered running the software on a virtual machine, Sandboxie proved to be a more efficient option. This software works by hooking the operating-system, system calls. Sandboxie keeps track of what resources a running process has permission to access. It also has the ability to hide files from the running process.

2.2.1 Collecting System Calls

After choosing Process Monitor [5] as my tool to collect system calls, I needed to find a way to integrate it into my project. While most people use Process Monitor's user interface to click buttons and navigate using its interface, I needed a method of using it automatically without manually directing it each time. Thankfully, I discovered that Process Monitor supports some command-line options which can be viewed by typing "**procmon.exe /?**". This allowed me to start and stop Process Monitor as needed.

Through experimentation, I found a way to start Process Monitor from my Racket program. I would then spawn the program I was observing and record its behaviour. Once it was finished, I would then shut down Process Monitor and convert the data it collected into a usable CSV format. To start Process Monitor with a minimized window, I used the command "**procmon.exe /Minimized /BackingFile output-location.pml**", which produces a PML file. After the monitored process had finished, I passed the **/Terminate** parameter to Process Monitor to stop monitoring.

I struggled to find much documentation on the PML format, I then found that Process Monitor had a parameter to convert it to CSV format. By using "**/OpenLog output-location.pml /SaveAs csv-location.csv**", I could output a plain-text file with each system call on a new line. One final problem was that Process Monitor didn't have the ability to filter out processes I wasn't interested in, so I made a Python script that filters out everything that isn't the process and it's children.

2.2.2 Client Installer

For users to be able to install the client on their computer(s), I researched tools that could generate a Windows installer. I came across a tool called **iexpress.exe**, which is conveniently included in Windows.

iexpress.exe is primarily used through its GUI, I discovered that it can accept a parameter through the command-line, a SED file (**client/caladium.sed**) containing instructions for iexpress to build an installer.

Before creating the installer, I archived all the files generated by PyInstaller into a tarball (tar.gz) and included a script with the installer that extracts the archive into the `Program Files` directory on Windows. I also have it so it creates a shortcut in the startup directory, so it starts on system boot.

2.3 Development

For version control, I used Git and hosted the repository on GitHub. I added a `.gitignore` to prevent temporary files and build outputs from being pushed to the repository. Initially, I divided the project into three parts: the client, server, and sandbox, to maintain organisation. I then created a **src** subdirectory for each sub-project, containing only the core code for that project. For instance, the server subproject has a **tests** directory containing test code that is not part of the core code. Like the server, I moved the core code of the client to the **src** subdirectory, leaving the installer code in its original directory.

I worked mainly on Windows, but I occasionally switched between different operating systems. To avoid working on a stale version, I regularly committed changes to the GitHub repository and pulled the latest version of the repository to the current workstation. For text editing, I used Visual Studio Code, which provides text suggestions, and I occasionally used VIM to edit smaller scripts and files, especially when working on Racket code. I used Overleaf to write the dissertation and pushed the changes to the `dissertation` directory in the GitHub repository.

Finding time to work on the project was challenging, as I had other modules besides the dissertation. I mostly worked on the project during weekends, and whenever I thought of a new feature, I would then add it to the Kanban board. During development, I spent a considerable amount of time thinking about possible solutions to problems. For example, to avoid embedding ugly HTML in my dashboard single application's JavaScript, I created a library called **pantothenic.js** that can convert string-based Lisp expressions into DOM objects (you'll see more about this in the system design chapter).

2.3.1 Weekly Progress Reports

Every week during the 2 semesters, I exchanged my progress report, with my supervisor during our weekly meeting over Microsoft Teams.

Below you will find each week's report and a description of what I did, this would sometimes be research and/or development.

16th - 22th of October
I worked on the `DirChangeListener` class for the client, which is used to listen for changes in directories.

This will be used to detect new files added to the downloads directory. I researched methods of implementing the sandbox analysis, and have a basic prototype of it working.

23rd - 29th of October
I found out that the system call logger was collecting too many system calls. So I needed to filter out the ones I'm not interested in. I made a filtering program in Python to do this.

I also refactored the analysis code to make it cleaner.

30th of October - 12th of November
I started using the Kanban board on GitHub to keep track of tasks. I added code to the sandbox analysis service, so you can now connect to it over TCP, and pass in a JSON message object containing the instructions you want it to execute.

13th - 19th of November
This week I was working on the dashboard, and I added a stylesheet called Milligram, I enabled CORS, so I could now fetch data from the server without getting

an error message.

20th - 26th of November
I worked on setting up Docker and a server on Azure. Now I can run the server in a container, and it downloads all the dependencies I require.

27th of November - 3rd of December
I worked on setting up a page for adding workers in the dashboard application. I also added backend logic for logging in users.

4th - 10th of December
I added code that can begin a scanning task on the server, and added RESTful APIs for performing actions on this code.

11th of December - 28th of January
During these four weeks, I have:

- Added CouchDB to store records rather than storing them in memory.
- Added a window to the client to display the scanning progress.
- Made a test suite to test the server API endpoints.
- Added code to display pie charts and bar charts.
- Added a preferences menu to the dashboard, so admins can change their passwords.

29th of January - 4th of February
I have restructured the architecture of the client, requiring the client to be provisioned with the server, on the first startup. All requests from the client to the server, are now authenticated, and you can't do anything without being first provisioned. Files dropped into the downloads directory, now trigger a notification, asking the user if they want the files to be scanned. And I fixed some broken endpoints for deleting records, which the tests caught.

5th - 11th of February
During this week, I added a GitHub Action to automatically run the test suite, when new code is pushed to the repository. Fixed some problems in the client and server, and fixed the problem in the client that was causing the scan to fail. Clients can now auto-provision without requiring an admin to do it for them.

12th - 18th of February
This week, I successfully found a way to bundle the client code using PyInstaller. I also created an installer using the `iexpress.exe` tool found in Windows that creates a shortcut on installation. Added commentary to some of the code, I've restructured some of the code in the client to make it more smooth.

19th - 25th of February
I added a new preferences page to the client, so users could update the current scanning directory. Added a button to unprovision the client, I also added an icon to the executable, fixed the bar chart rendering bug in the dashboard.

26th of February - 4th of March
I implemented the uninstall functionality in the preferences menu, in the client. I added the code to do the static analysis with ClamAV. I created a promotional page in the `/docs` directory that will automatically be deployed to GitHub Pages, upon pushing to the repository. Removing all threading code from the client, making it all asynchronous.

5th - 11th of March
I worked on the promotional page, adding a download button to download the latest version of Caladium. When malware is detected the user is now prompted to quarantine the malicious files. I have now made a **v0.1** release of the software on GitHub along with a new feature, showing the current scanning directory on the main page.

12th - 18th of March
I made the scanning less likely to fail, by adding code to detect exceptions. Added a new button on the admin dashboard to disable dynamic scanning.

19th - 25th of March
I fixed the remaining bugs in the scanning process, finished the README and created a screencast.

Added a GPT-3.5 chatbot to the promotional page, the chatbot allows new users to ask questions about the platform.

Fixed the pie chart rendering for the clean-malware ratio in the dashboard. Found an XSS security vulnerability that has now been fixed. And set a timeout for infinitely running scans, to be killed after a minute in the dynamic analysis.

Added a 6th **v0.1.2** release of the client to GitHub.

26th March - 15th April
I finished writing a draft of the dissertation. I added a sound to the dashboard that plays when moving between pages, it can be disabled using a toggle in the preferences page. I recorded a screencast of the promotional page, which shows CaladiumBot answering questions about Caladium.

2.4 Agile Methodology

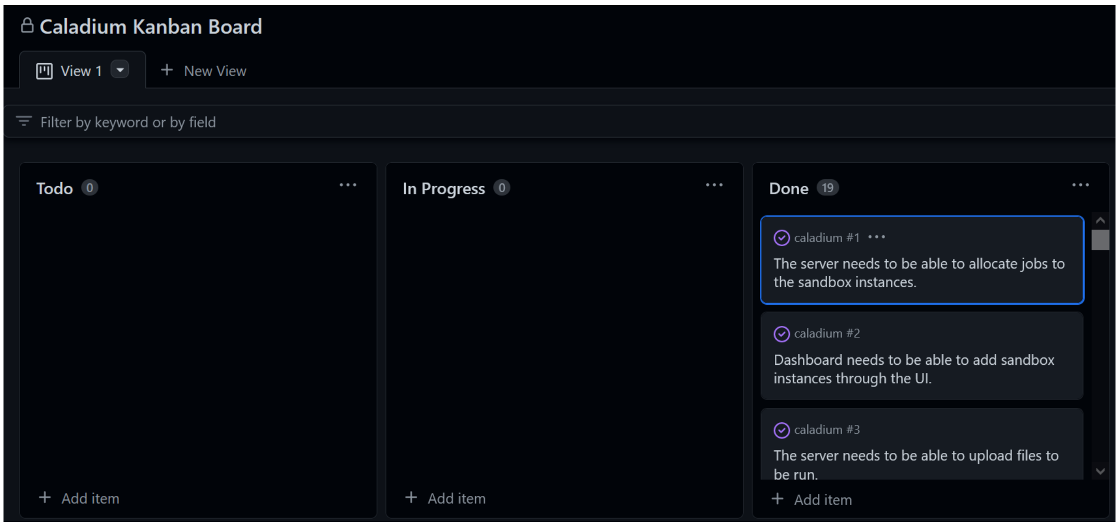
I decided on an agile-style methodology because I didn't have a complete understanding of everything I needed to do.

I didn't create a specification document in the beginning, because I hadn't fully decided on the components to use, for example, I didn't choose my database until December, but I proceeded with working on the other components.

One of the benefits of using Agile is that it provides developers with the ability to adapt to changing requirements, which allows for more flexibility during the development process. After running a suite of tests on the code, it can be deployed to production if it passes all of them.

In order to manage the tasks required for the project, I decided to break them down and create a Kanban board. I did consider using Jira, but I later found out that GitHub supported the Kanban board, which was much more convenient because it was integrated with GitHub. Throughout the development process, I continuously identified the tasks that needed to be completed.

Below you can find a screenshot of the Kanban board I used.



caladium #1
The server needs to be able to allocate jobs to the sandbox instances.

caladium #2
Dashboard needs to be able to add sandbox instances through the UI.

caladium #3
The server needs to be able to upload files to be run.

Figure 2.1: Kanban Board Screenshot

The Kanban board had three columns: "Todo", "In Progress" and "Done". When I thought of a task that needed to be done, I placed it in the "Todo" column. When a task was in progress, I moved it to the "In Progress" column. Once I finished working on a task, I then moved it to the "Done" column. Appendix B contains a list of all the tasks I had on my Kanban board, every one of them are now in the "Done" column.

Every couple of weeks in the 2nd semester, I created GitHub releases for the client. These releases included the installer. I followed the principles of semantic versioning, where the version numbers are formatted as major.minor.patch. By the end of the development, I had reached version `v0.1.2`, which was the 6th release.

2.5 Software Testing

I used a test-driven approach (TDD) to develop the server endpoints. I used the "unittest" Python library to write unit tests for the server endpoints.

I used the tests as a specification for the server endpoints. If the tests failed, it indicated I wasn't implementing the endpoints correctly. I could run the tests and it would verify if I had broken any of the endpoints.

The test files are prefixed with **test_**, these test the **clients**, **patterns**, **tasks** and **workers** endpoints, the endpoints are very similar, so I have each test file inherit code from the `setup.py`.

For example, every collection has an endpoint to delete all records of that type. So the only difference is the endpoint location.

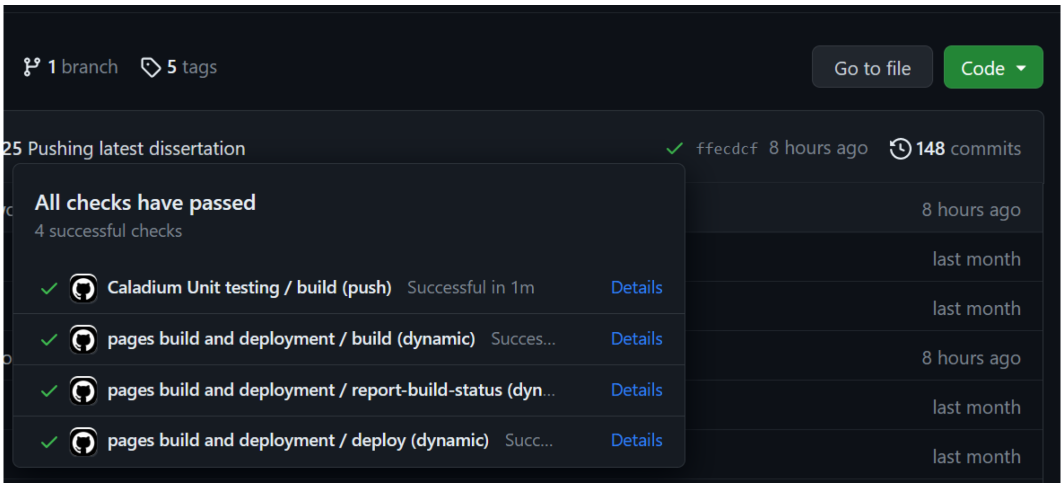
2.6 Continuous Integration and Deployment

After pushing the code to the GitHub repository, I wanted to automate the process of running a suite of tests.

So, I created a GitHub action that can be found at `.github/workflows/run_tests.yml`. The YAML file instructs GitHub to spawn an instance of the server, on their servers, sleep for 3 seconds, and then run the tests found under the `server/tests` directory.

To execute the server, a CouchDB instance is required. Since I have this instance on my Azure server and not on GitHub, I added the **COUCHDB_CONNECTION_STR** environmental variable through the GitHub UI. This variable is securely passed to the server, as having it leaked through logs would cause havoc to my database. This allowed me to test if any new code that was introduced or modified, broke the application.

Below, you can find a screenshot of the result of the GitHub action. The green checkmark indicates that all the tests ran correctly.



All checks have passed
4 successful checks

✓ Caladium Unit testing / build (push) Successful in 1m Details
✓ pages build and deployment / build (dynamic) Succes... Details
✓ pages build and deployment / report-build-status (dyn... Details
✓ pages build and deployment / deploy (dynamic) Succ... Details

Figure 2.2: GitHub Green Checkmark Screenshot

I set up a server on the cloud, to host the server, I did this by creating a Linux virtual cloud computer on Microsoft Azure Cloud and installed Docker on it. To test the server on the cloud, I simply **git pull**'d the latest version from GitHub to the Azure computer, and run the Docker container.

My Dockerfile can be found at `server/Dockerfile`. This file allows me to run my server code on any computer that has Docker installed. Building this Docker container fetches a minimalistic Debian bullseye version, updates the Debian package index, installs the latest version of Python, installs the required modules for the server (this includes Flask, pycouchdb), and copies the required files into the `/caladium` directory. When the container is run, it executes the Python `__main__.py` file.

Chapter 3

Technology Review

This chapter provides a technical review of the technologies used in the platform.

Detailing the reasoning behind selecting these specific technologies and how they contribute to accomplishing the objectives laid out in the introduction.

Also covering technologies specifically related to malware detection, these include Sandboxie and Process Monitor, and widely accepted standards like JSON for transmitting data over networks.

For the server section, it shows how the use of tooling like Docker can help with the creation of reproducible environments.

First, there will be a discussion on the technologies that are general to all the sub-projects, and what was used to create the chatbot for the promotional page. Then the technologies that were used in each sub-project.

3.1 JSON

In the platform, there will be multiple computers that receive and transmit data. There needs to be an agreed-upon format between these computers in which the data is encoded. The format must support dictionaries and lists to express data structures.

JSON has been chosen for this because the browser has native support for encoding and decoding JSON, Python supports it through the `json` module, and Racket also has it in its standard library. JSON stands for JavaScript Object Notation, a widely-used format for transmitting and storing data.

It provides a standardised way of serialising complex data structures or collections, making it easy to transfer data between different systems. JSON can be used to represent a wide range of data types, including lists, dictionaries, and primitive types such as numbers and strings. [8]

The standard document for JSON cited above provides developers with a specification to easily implement JSON encoding and decoding in their language. Also, its simplicity has made it popular amongst developers.

Below you will find an example of JSON being used in the browser to encode a dictionary:

```
const myData = {"1": 2, 3: 4.0};
console.log(JSON.stringify(myData));
```

And this is an example of it being used in Python, using the `json` module.

```
import json

my_data = {"1": 2, 3: 4.0}
print(json.dumps(my_data))
```

3.2 Python

Python [9] is an interpreted, high-level, general-purpose programming language. I chose to use Python because I wanted to use the same language for the client and the server. When I serialised data on the client side, I could use the same library, to deserialise it on the server.

Python is also a very simple language and can be run without compliation, this allowed me to quickly test code, without having to compile it every time.

Python comes with a package manager called `pip`, which is accessible on all operating systems, this allows me to easily install dependencies required by the server. Python is less verbose than languages like Java, allowing you to get a lot done with a small amount of code.

3.3 GPT-3.5

In recent months, the popularity of large language models has exploded, primarily due to the advent of ChatGPT.

OpenAI, the creator of GPT, provides an API to their GPT-3.5 model [4]. This model has been trained on a large corpus of text from the internet and can make predictions based on inputs to the model.

Currently, this technology represents the state of the art in artificial intelligence and OpenAI's API makes it freely available to anyone.

To promote the platform, I have created a promotional page on GitHub pages and added a bot (CaladiumBot), that can answer questions about it. I fetch the latest version of the README.md file for this project, feed it into the OpenAI model as an initial prompt, and then used it to answer user's questions about the platform.

OpenAI provides a module for Python, below you will find a minimal example of how I'm using it. The full implementation can be found in bot/__main__.py.

```
import openai # Set OpenAI API key as environmental variable
openai.api_key = os.environ.get("OPENAI_API_KEY", None)

initial_prompt = """
You are CaladiumBot, you will answer questions
about the README listed below:
"""
# Append the README here

prompt = [{"role": "system", "content": initial_prompt}]
question = input("Your question: ")
prompt += [{"role": "user", "content": question}]

# This will output the GPT response to the command-line
print(openai.ChatCompletion.create(model="gpt-3.5-turbo", \
        messages=prompt).choices[0].message.content)
```

3.4 Technology in the Client Side

As stated in the objectives, there will be a native GUI application, this will be a Windows application so it will not run in the browser, the application will be written in Python, so I am using Python-specific libraries to develop it.

3.4.1 PyInstaller

A common reason why more people don't opt to use Python over a native language such as C/C++ is the fact you require Python to run the code. Some users of a particular software might want to be able to run the software without having to install Python first.

PyInstaller [10] is a module for Python that allows you to bundle your Python code with a Python interpreter, allowing it to be run on any computer that doesn't have Python installed. This allows your software to be self-contained and portable, users don't have to install Python on their computers to run your software.

As shown in the documentation you can easily convert a Python project into a bundle. [10] Pass the main file location of the Python project into PyInstaller: `pyinstaller __main__.py`

3.4.2 Tkinter

Tkinter [11] is the standard GUI library for Python, It is also cross-platform, allowing the client to run on Windows, Linux and macOS. It is stable and mature, with a plethora of documentation to be found on the internet.

Tkinter makes it very easy to create a GUI with a small amount of code, which allowed me to quickly build a prototype of the GUI application. The library features many GUI components including buttons, labels and input boxes, these are important for creating a native GUI application experience for Windows.

I will give an example below, of creating a simple GUI window.

```
import tkinter

# Creating the main window object
main_window = tkinter.Tk()
# Main loop of execution, polling for updates
main_window.mainloop()
```

Running this code will result in the window shown above. This serves as a stark reminder of the power of Tkinter and its ability to create interfaces quickly.

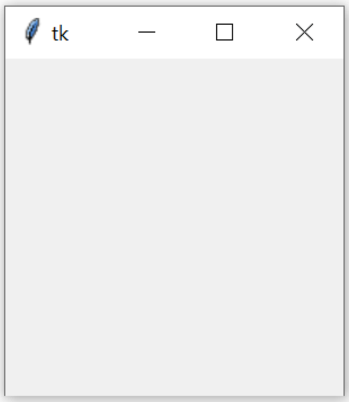


Figure 3.1: Tkinter Window Screenshot

3.4.3 tkthread

In Tkinter applications, there exists an infinite **mainloop** (as seen in the example above) that runs until the user closes the application.

Whenever a user clicks a button, it triggers an event that calls the corresponding callback function that was specified during the button's creation.

As this call blocks the **mainloop**'s execution, a function that takes a long time to complete can cause the GUI to freeze, causing the program to become unresponsive.

While a naive solution might involve threading, this approach requires thread synchronisation and can lead to race conditions and phantom threads, leading to even more problems than it fixes.

Instead, tkthread [12] provides a more elegant solution. tkthread is a third-party module for Python, that is non-affiliated with the developers of Tkinter.

In the example below, I'm showing how you can use tkthread to patch Tkinter to allow async operations, when the button is pressed it will spawn an async thread, and every so often it will give the UI a certain amount of time to draw itself.

This results in much cleaner code, and avoid complexities like using threading, instead running callback functions asynchronously.

```
import tkthread
# Patch tkinter to allow async operations
tkthread.patch()

import tkinter

main_window = tkinter.Tk()

def button_callback():
  # Spawn new thread
  @tkthread.main(main_window)
  def my_thread():
    while True:
      print("This won't block the GUI")

      main_window.update() # Allow UI to update
      main_window.after(1000)

# Create new button, that will call the function above
my_button = tkinter.Button(main_window,
    text="Test Button", command=button_callback)
my_button.pack()

main_window.mainloop()
```

3.4.4 JavaScript

JavaScript was used in the dashboard to add dynamicity, it is used to fetch data from the server using the RESTful API endpoints and display it on the screen.

I wanted to take full advantage of all the new JavaScript features that were brought in recent years, these include the ability to have classes and the new `fetch` function as an alternative to setting up the XMLHttpRequest object.

The 6th edition of the ECMAScript standard [13] (JavaScript standard) introduced the `class` keyword for creating classes, before this developers had to create classes through prototypes.

Classes are important in the dashboard because the list page classes all inherit from a common superclass allowing them to share code.

```
class Page {
    int x;

    constructor(newX) {
        this.x = x;
    }

    getX() {
        return this.x;
    }

    setX(newX) {
        this.x = newX;
    }
}
```

Here is the legacy syntax, as can be seen, it is quite verbose, and it isn't encapsulated in the class like above.

```
function Page(newX) {
    this.x = newX;
}

Page.prototype.getX = function() {
    return this.x
}

Page.prototype.setX = function(newX) {
    this.x = newX;
}
```

Another addition of the ECMAScript 2015 standard is the `fetch` function, prior to this, you'd have to use the `XMLHttpRequest` object, this is important in Caladium because it allows the dashboard to request data from the server through the RESTful APIs.

Below you can find an example of the `fetch` being used in the dashboard, `fetch` accepts a number of parameters including the resource path, the HTTP headers, in this case, the **Authorisation** token and a body containing the data being sent to the server.

```
const caladiumFetchParameters = {
    method: method, headers:
        {"Authorisation": localStorage["Authorisation"]},
    body: body ? JSON.stringify(body) : undefined
};
const resp = await fetch(path, caladiumFetchParameters);
// Returns when the response is resolved
return await resp.json();
```

3.5 Technology in the Server

The server is the main component of the platform, bridging communication between the clients and the sandbox instances, and providing RESTful APIs for the clients to interact with it.

The server needs to be able to persist data for the administrators, which is stored in CouchDB. Since the server is written in Python, it requires the pycouchdb library to communicate with the DB instance.

I have also created a `Dockerfile` for Docker that enables people to easily replicate the required environment for running the server.

3.5.1 Flask

I needed a way for the dashboard and the clients to be able to communicate with the server I decided to use Flask to create the HTTP endpoints.

Flask [14] is a micro web framework written in Python. It is classified as a micro-framework because it does not require any extra tools or libraries. This allowed me to create the HTTP endpoints for the RESTful API without having to worry about the underlying HTTP server.

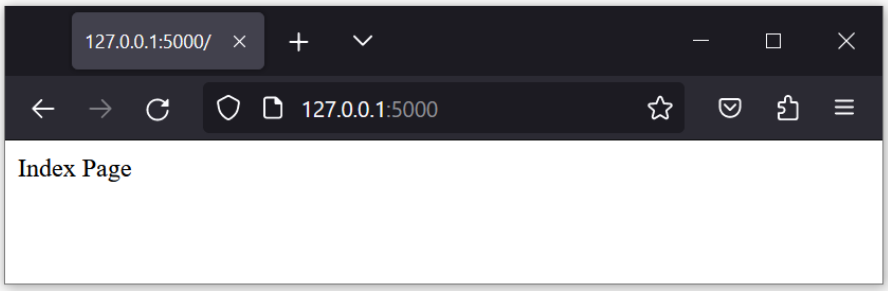
The simplicity of Flask allows for quick prototyping, which speeds up the development cycle. Many accept Flask as the de facto framework for creating web applications in Python. This led me to choose Flask as the framework for the server component.

Below you will find some example code to create a Flask server, with a GET endpoint that returns the text `"Index Page"` on request.

```
import flask

app = flask.Flask(__name__)

@app.get("/")
def root_page(): return "Index␣Page"

app.run()
```

When you run this code, it will spawn an instance of Flask, listing the IP address of the server, this usually being `http://127.0.0.1:5000`.

Below you can find a screenshot of the / route in the browser.



Index Page

Figure 3.2: Flask Server in Browser Screenshot

3.5.2 Representational State Transfer

I am using REST to model the HTTP endpoints for the server, REST was first described in Roy Thomas Fielding's dissertation [15] as a way of architecting APIs in a stateless manner to access resources over a network connection.

REST is a set of best practices for how systems should interact with one another. It promotes scalability and allows for caching of resources.

HTTP methods can be used to perform actions on a resource. For example, the **GET** method can be used to fetch data, which can also be cached. The **DELETE** method can be used to delete a record by its ID.

For example, if there existed a set of endpoints that performed actions on a database of tasks.

GET `/api/tasks`
Would return a JSON array, containing all the tasks.
POST `/api/tasks`
Would be used to create a new task, and you'd pass in the new task through the HTTP body.
DELETE `/api/tasks/0001`
Would be used to delete a task by its ID, in this case, it would be the task with ID `0001`.

3.5.3 CouchDB

CouchDB is a document-oriented NoSQL database. [16] An important feature of any platform is the ability to persist data.

In my case, I am using CouchDB to store the records for the **clients**, **patterns**, **tasks** and **workers** tables.

I find NoSQL quite convenient compared to SQL databases, because I don't have to write out the queries, and NoSQL allows you to easily append JSON objects to the tables.

I went with CouchDB because of the simplistic nature of a document database is appealing, allowing me to spend more time on more complex features. CouchDB is also scalable, allowing many instances of it, allowing it to increase with more users.

CouchDB provides an HTTP/JSON API, allowing any programming language with an HTTP library to interface with the DB instance.

3.5.4 pycouchdb

Python alone isn't able to communicate with the CouchDB instance, and requires a library, in this case, it's **pycouchdb**.

pycouchdb provides an API for developers to easily interface with CouchDB, without having to write the HTTP requests to do so.

The example below shows how to use pycouchdb, to create a table called "tasks" and add a record.

```
import pycouchdb

# Connect to the DB using the connection str in the env variable
couchdb_server = pycouchdb.Server(os.environ["COUCHDB_CONNECTION_STR"])
```

```
# Create the table, and create a new record with the dictionary
tasks_table = couchdb_server.create("tasks")
tasks_table.save({"name": "task"})
```

3.5.5 Docker

Docker is a tool used by developers to create reproducible environments. [17] Meaning you can run your software on different computers, and it would have the exact same stuff installed.

Most open-source projects come with a `Dockerfile`, it provides users interested in setting up the project on their own computer, with the same environment, the developers used.

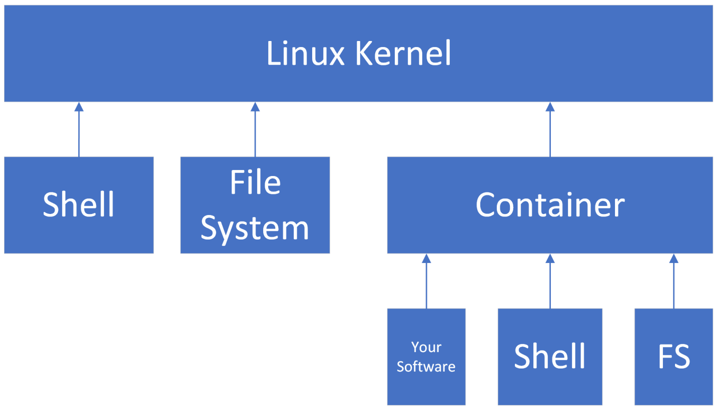


Figure 3.3: Docker Diagram

In the diagram above, you can see the host operating system on the left and the container on the right. Everything still goes through the Linux kernel, unlike a virtual machine that runs an entire operating system in isolation.

The container has its own file system, which contains all the utilities for that operating system. In this project, I'm using Debian for the container, and the host is using Ubuntu. Your software running in the container is unable to see what's going on in other containers or the host operating system due to isolation features offered by the kernel.

The `Dockerfile` contains instructions for the Docker daemon, this is a program that must be installed on a computer you want to use Docker on. You must then build containers using this file, you can have multiple containers on one machine, this is equivalent to having different computers with different Linux distributions.

You must then run these containers to execute the code of the project. These containers are isolated from the main system, and cannot edit files on the host system. I am using Docker to make it easy to test my project on different machines, allowing me to delete them easily, and not have to actually install the dependencies on my main host system.

The `Dockerfile` I am using in my project can be found here: `server/Dockerfile`

The Dockerfile begins by instructing the daemon, to install the Linux Debian bullseye version. Then refresh the apt-get package manager, and install Python and the Python pip package manager. Installs the required Python modules, and copies the server files into the `/caladium` directory.

Then when you execute the container, it runs the `__main__.py` on port 8080.

```
FROM debian:bullseye

RUN apt-get update -y
RUN apt-get install python3 python3-pip -y

RUN python3 -m pip install flask pycouchdb requests

WORKDIR /caladium

COPY src/ /caladium

ENTRYPOINT ["python3"]
CMD ["__main__.py", "8080"]
```

3.5.6 unittest

Unittest is a module that comes bundled with Python and allows you to create test cases for your code [18]. It takes inspiration from JUnit, which is a popular Java testing framework.

To create a test case, you must create a class that inherits the `unittest.TestCase` class. Each of the test methods in your test case class must be prefixed with `test_`. In each of the methods, you call `self.assertEquals` and check if the output of the function in question returns the intended output.

This allowed me to create tests for the endpoints and check if they returned their intended outputs.

3.6 Technology in the Analysis Side

In this section, I will be discussing the technology that facilitates the scanning of files and the Racket programming language I used to write the main part of the scanning side.

3.6.1 Sandboxie

Sandboxie [7] is a tool for Microsoft Windows designed for sandboxing, it provides a secure environment to run potentially malicious software.

Typical Windows applications, for example, your browsers like Chrome or Firefox, never actually have direct access to your system's hard drive. Instead, they have to go through the operating system, specifically the kernel, to gain access. If the requesting applications lack the required permissions the operating system can reject the request.

Sandboxie works by intercepting the system calls made by these applications, it then identifies the application that made the request, if the process is currently running under Sandboxie, it will fake a response, making the application think it was successful, allowing it to think it has more power than it actually has.

The main purpose of using Sandboxie is to provide an easy to use isolation software for users, to run software without worrying about it harming the system.

Sandboxie provides plenty of documentation on how to use the software, on its website, and the utilities that come with the software come with command-line parameters, allowing programmers to interface with it programmatically.

Below you will find an example command of how to run the Windows **notepad** in a sandbox:

```
"%ProgramFiles%\Sandboxie\Start.exe" notepad.exe
```

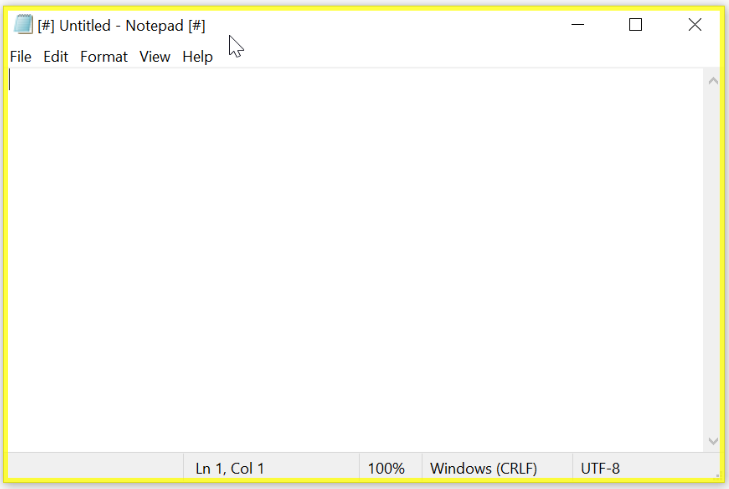


Figure 3.4: Notepad Running in Sandboxie Screenshot

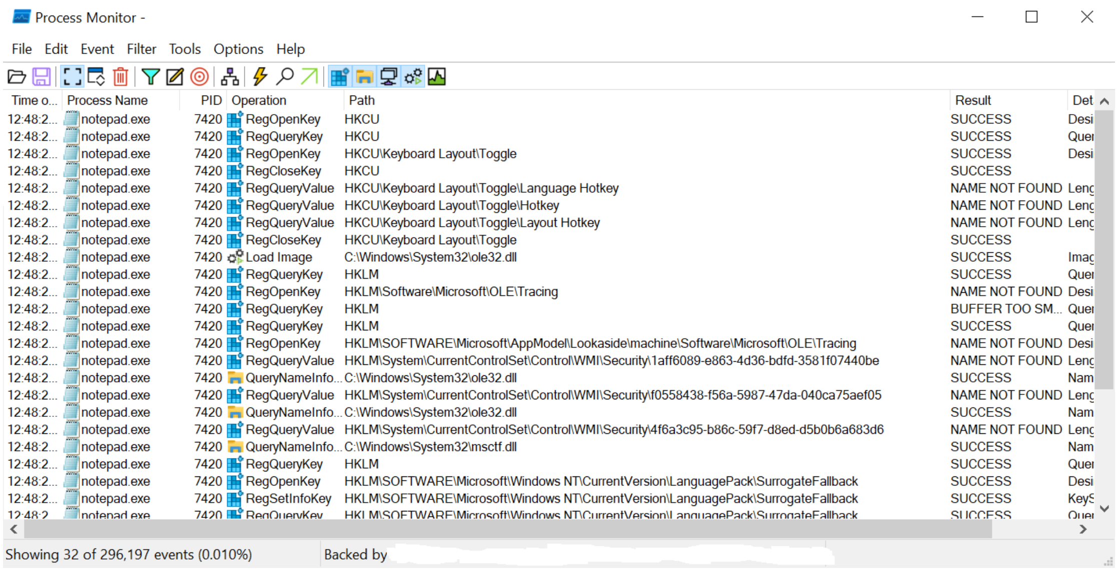
Above you can find a screenshot of `notepad.exe` running in Sandboxie, notice how it adds a yellow border to windows indicating they are being sandboxed.

3.6.2 Process Monitor

Process Monitor is a system monitoring program that allows you to log system calls. It was created by Microsoft to serve as a monitoring tool, giving power users the ability to see what exactly every process on their machine is doing, it produces a log, of all the actions performed and the name of the process that did it.

I needed a way of finding out what the malware is doing, so I decided to use Process Monitor to log system calls. I originally was thinking of creating a driver to log system calls, but I decided to use Process Monitor instead, to avoid reinventing the wheel.

As described by my research in the methodology chapter, Process Monitor features command-line parameters allowing it to be programmatically controlled by your code.



Time o...	Process Name	PID	Operation	Path	Result	Det
12:48:2...	notepad.exe	7420	RegOpenKey	HKCU	SUCCESS	Desi
12:48:2...	notepad.exe	7420	RegQueryKey	HKCU	SUCCESS	Quer
12:48:2...	notepad.exe	7420	RegOpenKey	HKCU\Keyboard Layout\Toggle	SUCCESS	Desi
12:48:2...	notepad.exe	7420	RegCloseKey	HKCU	SUCCESS	
12:48:2...	notepad.exe	7420	RegQueryValue	HKCU\Keyboard Layout\Toggle\Language Hotkey	NAME NOT FOUND	Leng
12:48:2...	notepad.exe	7420	RegQueryValue	HKCU\Keyboard Layout\Toggle\Hotkey	NAME NOT FOUND	Leng
12:48:2...	notepad.exe	7420	RegQueryValue	HKCU\Keyboard Layout\Toggle\Layout Hotkey	NAME NOT FOUND	Leng
12:48:2...	notepad.exe	7420	RegCloseKey	HKCU\Keyboard Layout\Toggle	SUCCESS	
12:48:2...	notepad.exe	7420	Load Image	C:\Windows\System32\ole32.dll	SUCCESS	Imag
12:48:2...	notepad.exe	7420	RegQueryKey	HKLM	SUCCESS	Quer
12:48:2...	notepad.exe	7420	RegOpenKey	HKLM\Software\Microsoft\OLE\Tracing	NAME NOT FOUND	Desi
12:48:2...	notepad.exe	7420	RegQueryKey	HKLM	BUFFER TOO SM...	Quer
12:48:2...	notepad.exe	7420	RegQueryKey	HKLM	SUCCESS	Quer
12:48:2...	notepad.exe	7420	RegOpenKey	HKLM\SOFTWARE\Microsoft\AppModel\Lookaside\machine\Software\Microsoft\OLE\Tracing	NAME NOT FOUND	Desi
12:48:2...	notepad.exe	7420	RegQueryValue	HKLM\System\CurrentControlSet\Control\WMI\Security\1aff6089-e863-4d36-bdfd-3581f07440be	NAME NOT FOUND	Leng
12:48:2...	notepad.exe	7420	QueryNameInfo...	C:\Windows\System32\ole32.dll	SUCCESS	Nam
12:48:2...	notepad.exe	7420	RegQueryValue	HKLM\System\CurrentControlSet\Control\WMI\Security\f0558438-f56a-5987-47da-040ca75aef05	NAME NOT FOUND	Leng
12:48:2...	notepad.exe	7420	QueryNameInfo...	C:\Windows\System32\ole32.dll	SUCCESS	Nam
12:48:2...	notepad.exe	7420	RegQueryValue	HKLM\System\CurrentControlSet\Control\WMI\Security\4f6a3c95-b86c-59f7-d8ed-d5b0b6a683d6	NAME NOT FOUND	Leng
12:48:2...	notepad.exe	7420	QueryNameInfo...	C:\Windows\System32\msctf.dll	SUCCESS	Nam
12:48:2...	notepad.exe	7420	RegQueryKey	HKLM	SUCCESS	Quer
12:48:2...	notepad.exe	7420	RegOpenKey	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\LanguagePack\SurrogateFallback	SUCCESS	Desi
12:48:2...	notepad.exe	7420	RegSetInfoKey	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\LanguagePack\SurrogateFallback	SUCCESS	KeyS
12:48:2...	notepad.exe	7420	RegQueryKey	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\LanguagePack\SurrogateFallback	SUCCESS	Quer

Showing 32 of 296,197 events (0.010%) Backed by

Figure 3.5: Process Monitor Screenshot

Above, you can find a screenshot of Process Monitor logging the system calls of `notepad.exe`. Each line reports the operation it performed, which can be reading/writing files or creating registry keys.

3.6.3 Racket

Racket is a functional programming language. [19] I decided to program the main part of the analysis in Racket because I've used a lot of Python in other components and wanted to use a different programming language.

Functional programming languages encourage immutability, this means once variables are set a value, then they can't be updated. If you want to update a variable then you must create a new one with the changes.

Racket is based on Lisp, which means List Processor, nearly everything in Racket is a list, Lisp stems from lambda calculus which is an area of mathematics for expressing computational models.

Below you can find a function written in Racket that adds two numbers, the list begins with "define", the seconds parameter is the function name and arguments, and finally, the body of the function.

```
(define (add x y)
    (+ x y))
```

Racket comes with a plethora of documentation which can be found on its website.

3.6.4 ClamAV

ClamAV is an open-source anti-malware solution that supports Windows, macOS and Linux. [6] This is the software I'm using to perform my static analysis.

Most anti-malware software are paid and/or closed source, but ClamAV is the exception, on its website, you can find a rich set of documentation.

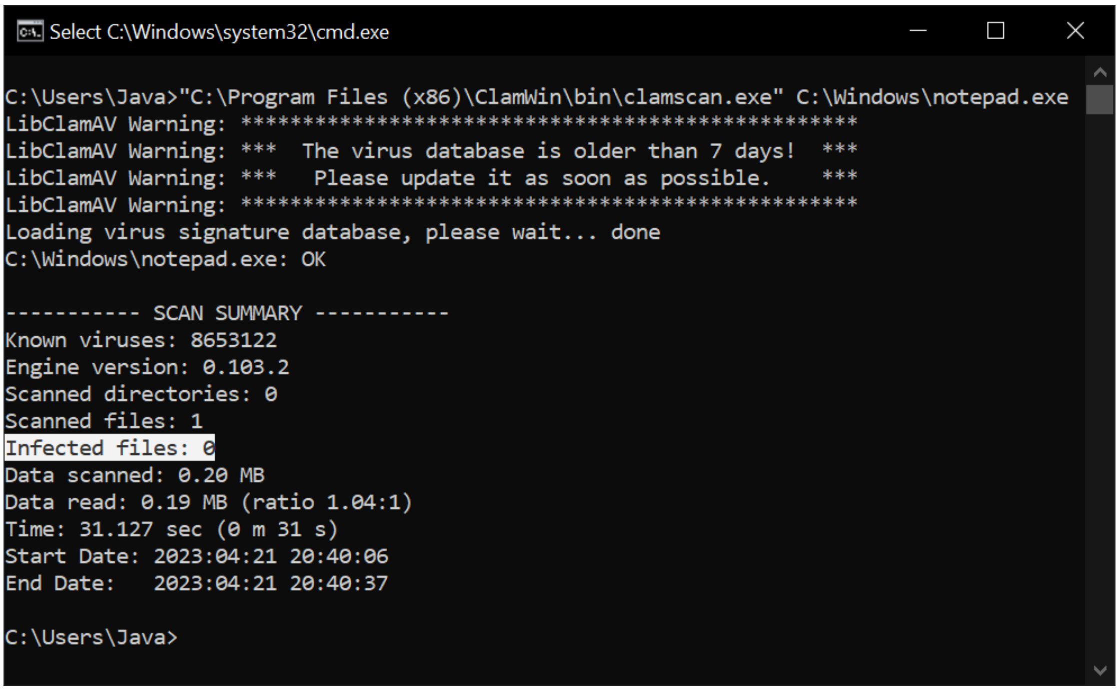
It comes with a number of command-line utilities. In my case, I was concerned with the **clamscan** command, and its parameters. Most anti-malware solutions don't provide low-level access to their software requiring you to use their menuing because that's who their target audience is.

ClamAV is targeted at power users, giving them the ability to modify the source code, and being able to add their own malware definitions.

I had some difficulties with setting up ClamAV on Windows, ClamAV is mostly targeted towards Linux users, but I found a modified distribution of ClamAV optimised for Windows called ClamWin.

ClamWin periodically downloads the latest malware definitions from its website, this all happens automatically.

Below you will find a screenshot of the use of the `clamscan` program, which comes bundled with ClamAV, to scan the `notepad.exe` file. Its malicious status is denoted by `Infected files: 0`. This shows how easy it is to use ClamAV programmatically to statically scan files.



Select C:\Windows\system32\cmd.exe

```
C:\Users\Java>"C:\Program Files (x86)\ClamWin\bin\clamscan.exe" C:\Windows\notepad.exe
LibClamAV Warning: ****************************************************
LibClamAV Warning: ***  The virus database is older than 7 days!  ***
LibClamAV Warning: ***   Please update it as soon as possible.    ***
LibClamAV Warning: ****************************************************
Loading virus signature database, please wait... done
C:\Windows\notepad.exe: OK

----------- SCAN SUMMARY -----------
Known viruses: 8653122
Engine version: 0.103.2
Scanned directories: 0
Scanned files: 1
Infected files: 0
Data scanned: 0.20 MB
Data read: 0.19 MB (ratio 1.04:1)
Time: 31.127 sec (0 m 31 s)
Start Date: 2023:04:21 20:40:06
End Date:   2023:04:21 20:40:37

C:\Users\Java>
```

Figure 3.6: ClamAV Scan Screenshot

Chapter 4

System Design

This chapter features a discussion on how the project was designed and on the design decisions used throughout. Diagrams and screenshots of the various components will be provided and so will implementation snippets of code.

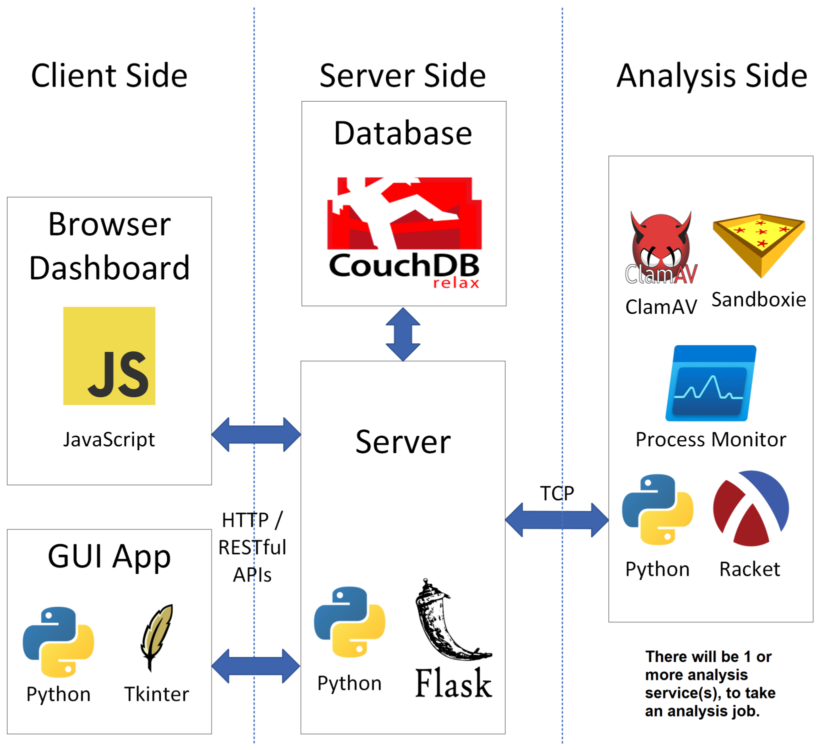


Figure 4.1: System Architecture Diagram

The system architecture diagram shown above provides a top-down view of the platform during execution.

It can be divided into three sections, each with the ability to run on different computers.

On the left side, there is the client software, which includes the GUI application and the dashboard that runs in the administrator's browser. In the middle, there is the server. On the right side, there are instances of the sandbox analysis workers. Multiple instances of these workers can be created.

4.1 Windows GUI Application

The GUI application runs on a user's computer and it runs on the client side.

Python files with the `frame.py` suffix contain Tkinter frame classes, and each of these files contains only one class that inherits the Tkinter `Frame` class.

This allows all code related to that frame, to be fully encapsulated, for instance in the `PreferencesFrame`, the buttons for changing the settings are kept inside the class and not accessible to other classes.

The main code of the application can be found in the `client/src/__main__.py` file, the purpose of this file is to setup up the Tkinter window and run the main loop of execution.

In order for the application to do anything with the server, it first needs to be provisioned. The requirement of provisioning prevents unauthorised clients from executing tasks on the workers.

Administrators can optionally enable auto-provisioning on the server, allowing clients to provision without manual intervention, manual intervention requires administrators to issue users with a token, and users will input this token into their application on the first load.

4.1.1 User Interface

The client post-provisioning features a Tkinter `Notebook`, with 3 sub-frames, **Main Page ("Caladium")**, **Quarantine** and **Preferences**. You can navigate through these by clicking on each label on the window. The `Notebook` Tkinter widget allows you to group frames together allowing navigation between them.

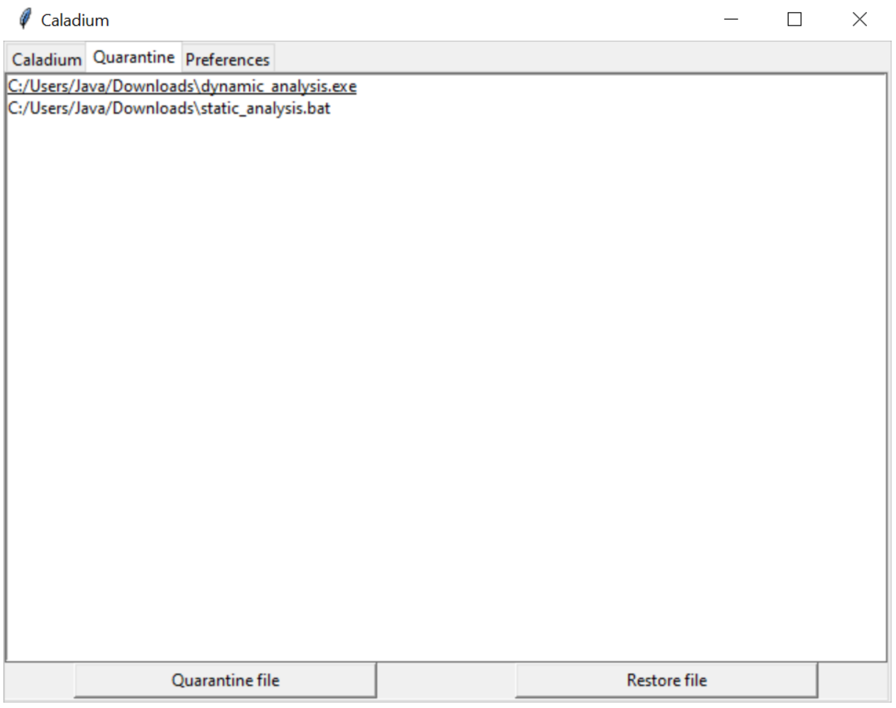
The Main Page in the frame features a button to manually scan a file, and a label indicating the current scanning directory.

Quarantine Page

The code for the quarantine page can be found in `client/src/quarantineframe.py`.

The quarantine frame displays all the files currently in quarantine, it shows the original location of each file, with the ability to restore each file, by clicking on the entry in the list box and then pressing the "Restore file" button.

A screenshot of the quarantine page can be seen below, you can also see the menu bar at the top, showing how you can switch between each frame.



C:/Users/Java/Downloads\dynamic_analysis.exe
C:/Users/Java/Downloads\static_analysis.bat

Figure 4.2: Quarantine Page Screenshot

To add files to the quarantine users can press the "Quarantine file" button, and then select the file in the file picker.

To allow users to pick files from their computers,
I'm using `tkinter.filedialog.askopenfile("rb")`, which returns a handle to a file when picked.

Preferences Page

The preferences page contains buttons, allowing users to adjust settings.

The "Uninstall Caladium" button will uninstall Caladium from your computer, it does this by dropping a script into your `%TEMP%` directory, the client will close, and shortly afterwards the script will execute. It will then delete the Caladium directory from the `Program Files` directory, and cleans up the shortcuts on the desktop and the startup directory.

Pressing the "Change Scanning Directory" button will open a directory picker, similar to the file picker shown above in the quarantine frame. Once the user has selected the new directory, it will change the current scanning directory to the one selected, and update the label on the main page to reflect this.

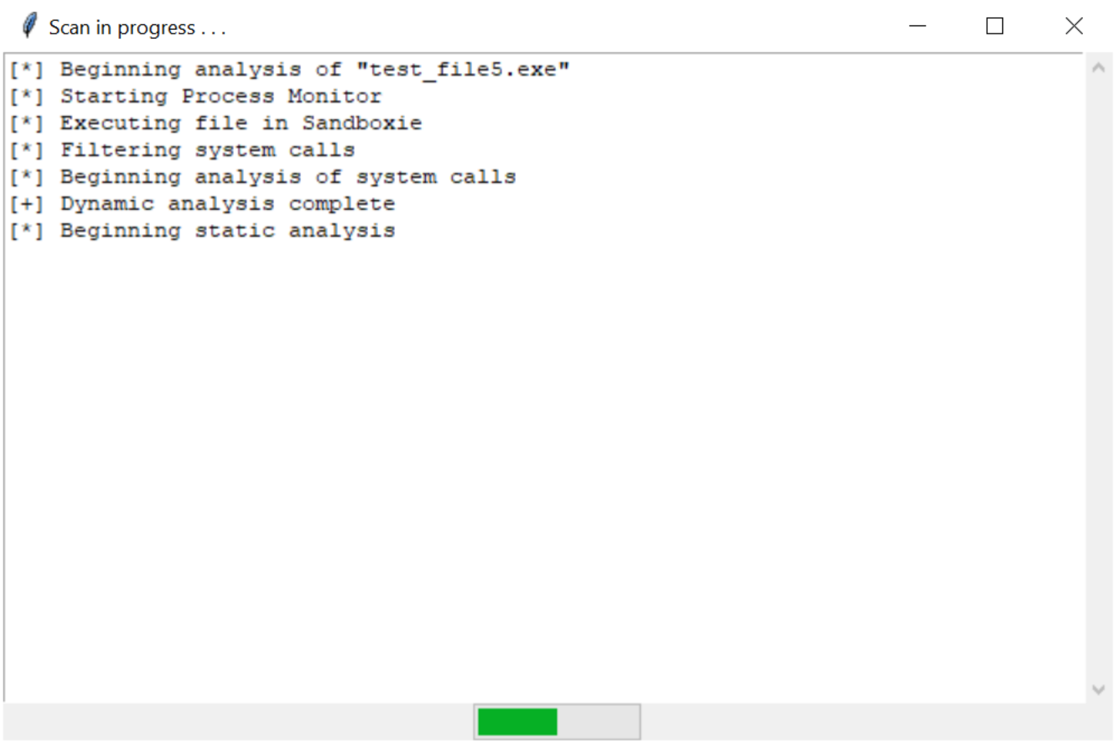
"Unprovision Caladium" will reverse the provisioning, and will close the application. When the user reopens the application, they will be prompted to re-provision.

Scanning Frame

The code that implements the scanning frame can be found in `client/src/scanningframe.py`.

When the scanning process begins, this window will open, it will have a text box containing real-time feedback from the server, and at the bottom will have a progress bar.

The progress bar is implemented using the `tkinter.tk.Progressbar` widget, the state of the bar is updated using its `value` attribute.



```
[*] Beginning analysis of "test_file5.exe"
[*] Starting Process Monitor
[*] Executing file in Sandboxie
[*] Filtering system calls
[*] Beginning analysis of system calls
[+] Dynamic analysis complete
[*] Beginning static analysis
```

Figure 4.3: Scanning Frame Screenshot

In the screenshot above, you can see the log messages coming back from the server, each message is prefixed with an error level indicator, `[+]` meaning success and `[-]` meaning failure. At the bottom, you can see the current progress stage, in this case, its `50%`.

4.1.2 Quarantine

When files are added to the quarantine, they are encrypted with an XOR cipher to avoid accidental execution of malware in the event that the user accidentally stumbles upon the location of the quarantine.

This is a snippet of the code from `client/src/quarantine.py`; it takes in input file data and its key and applies the XOR of the key to each byte, then returns the newly encrypted version.

```
def _xor_bytes(self, data_bytes, xor_key_bytes):
    # Convert bytes to list of ints
    data_bytes = [i for i in data_bytes]

    # XOR each one with the xor key
    for i in range(len(data_bytes)):
        data_bytes[i] ^= xor_key_bytes[i % len(xor_key_bytes)]

    return bytes(data_bytes)
```

4.1.3 Directory Scanning

The client has the ability to identify newly downloaded files, this is implemented in `client/src/dirchangelistener.py`. This functionality is achieved by scanning for new files in a directory.

This is implemented in a class called `DirChangeListener`, an instance of this class will be made, and will be passed a callback function, that will be called when a new file appears in the directory.

A **tkthread** thread is spawned which loops and checks for new files on every iteration. The current directory state is obtained using `os.listdir(dir_name)` and compared with the last state. To prevent the application from locking up, the Tkinter GUI is given some time to render on each iteration. This is explained in detail in the technology review chapter.

4.2 Server Dashboard

The dashboard is a single-page web application (SPA) and is fully written in JavaScript. This means when you navigate to another page in the dashboard it doesn't have to pull the entire page from the server, it will only fetch the data specific to that page, then renders it dynamically using JavaScript.

Below you will find a screenshot of the dashboard's index page, the dashboard lists all the sub-pages.

The `Clients`, `Patterns`, `Tasks` and `Workers` are all list pages, containing a list of records for that table.

The index page shows a statistical chart for each, the bar chart shown in green shows the distribution of records for each month, in this case, it only shows March.

Clicking on any of the titles on the index page will bring you to its list page.

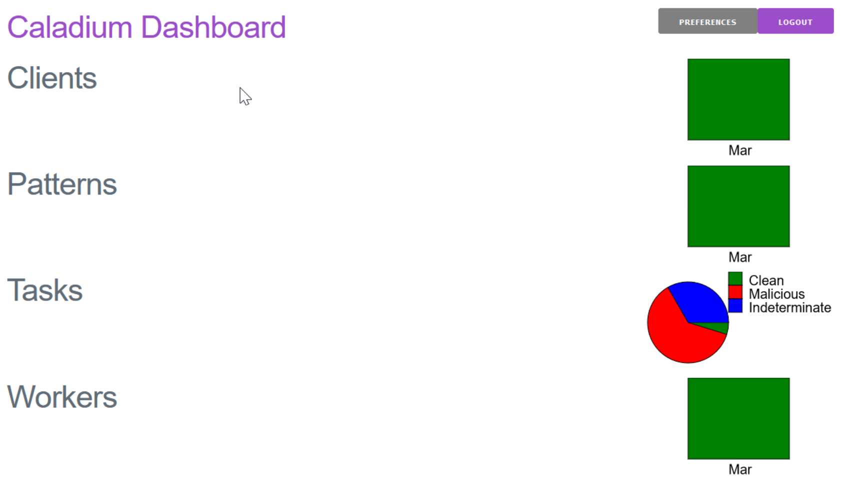


Figure 4.4: Dashboard Index Screenshot

4.2.1 Single-Page Application

A single-page application can update its contents without having to refresh the page, using JavaScript to dynamically fetch content from an API. [20] For example, in my dashboard, when a user clicks on a button, content such as a list of workers will be fetched from an API and then displayed on the screen.

Upon loading a new page, the contents of the page are dynamically generated using DOM Lisp expressions. The URL of the page is then updated using the History API.

In the **server/src/static/js/index.js**, I have a dictionary called **routes** containing a list of all the routes and their corresponding page classes and their page titles. When a page is loaded, the routes dictionary is queried using the current URL in the browser, and the corresponding page class is then loaded.

Below you can find a code snippet of the routes found in `index.js`, for example, the "/" route, will load the `IndexPage` class, and sets the title to "Dashboard".

```
// These are all the defined pages
const routes = {
    "/": {
        "body": IndexPage, "title": "Dashboard"
    },
    "/clients": {
        "body": ClientsPage, "title": "Clients"
    },
    "/login": {
        "body": LoginPage, "title": "Login"
    },
...
```

4.2.2 Page Class Hierarchy

The dashboard is object-oriented, all of the main code can be found in the **server/src/static/js/index.js** file. This contains code for communicating with the server endpoints and the classes for each page.

Below you can find a diagram showing the hierarchy of the classes, all classes inherit from the `Page` class, as they share common functionality, this also upholds the DRY principle.

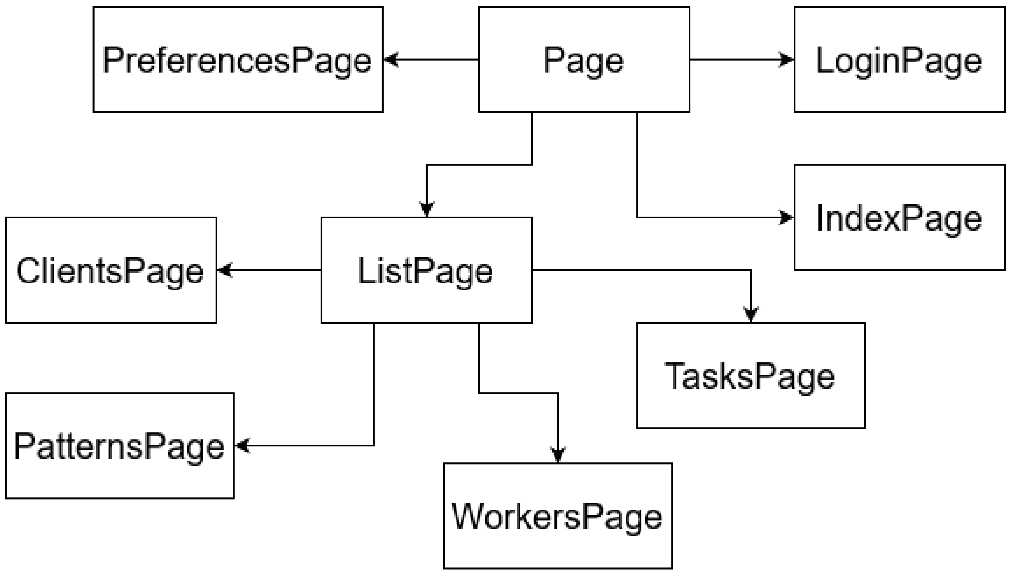


Figure 4.5: Page Class Hierarchy Diagram

Each page class must implement the **loadPage** method, which is called when the page is loaded. This function pulls data from the server, that is specific to that page.

4.2.3 DOM Lisp Expressions

In a single-page application (SPA), all page content is dynamically generated without needing to pull HTML from a server when navigating between pages.

Initially, I considered embedding raw HTML within the JavaScript code, but found it to be inelegant. As a solution, I created a domain-specific language (DSL) to represent the Document Object Model (DOM) as Lisp expressions.

My approach involved storing these Lisp expressions as strings within each page class, they can include parameters which can be replaced with their actual values during page loading. This approach allowed me to encapsulate the expressions within each page class.

This approach simplifies the creation of the visual components of each class. I developed a library, that can be found at **server/src/static/js/pantothenic.js**, which has a **generateDOM** function that takes a DOM Lisp expression and its parameters, and returns a DOM object.

The DOM is a tree of nested elements that represents a web page's HTML. By using Lisp expressions as a DSL to represent DOMs, I have created a more elegant solution for generating dynamic content without embedding ugly HTML.

Below you will find an example of a DOM Lisp expression. This is taken from the **Preferences** class in the **server/src/static/js/index.js** file. It describes the HTML components for that page, it includes an input box for a new password for the admin user and a button that toggles the auto-provisioning of new users. When the user clicks the button, it calls the **toggleAutoProvision** method within the class.

The navigation bar is also represented as a DOM expression, this can be found in the main `Page` class, that all pages inherit. This avoids repeating myself, allowing easy maintenance. The `navigationBar` parameter found below, is replaced with the actual navigation bar when loaded.

```
this.body = ` (div (hash "children"
                (list
                    navigationBar
                    (input (hash "type" "password" "id"
                        "newPassword" "placeholder" "New Password"))
                    (button (hash "onclick" changePasswordOnClick
                        "innerHTML" "Update Password"))
                    (hr)
                    (button (hash "onclick" toggleAutoProvision
                        "innerHTML" autoProvisionButtonText)))))`;
```

4.2.4 Login Page

Before you are able to perform any actions, you must log in to the dashboard, using the Login page as shown below, the default username and password are "root" and "root".

These can be changed on the preferences page once logged in, the password is hashed with SHA-256, so its the plain-text version cannot be retrieved in the event of a compromise.

Input these details into the username and password fields and press "Login" as seen in the screenshot below.

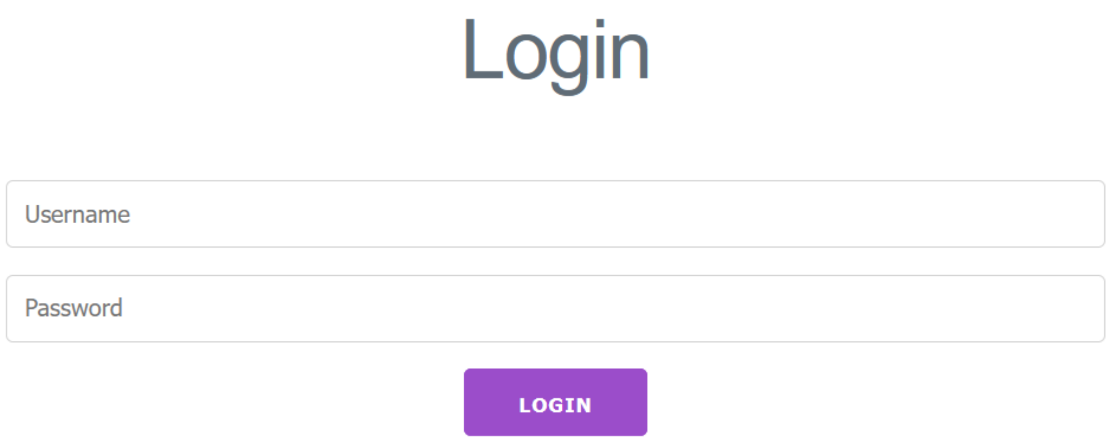
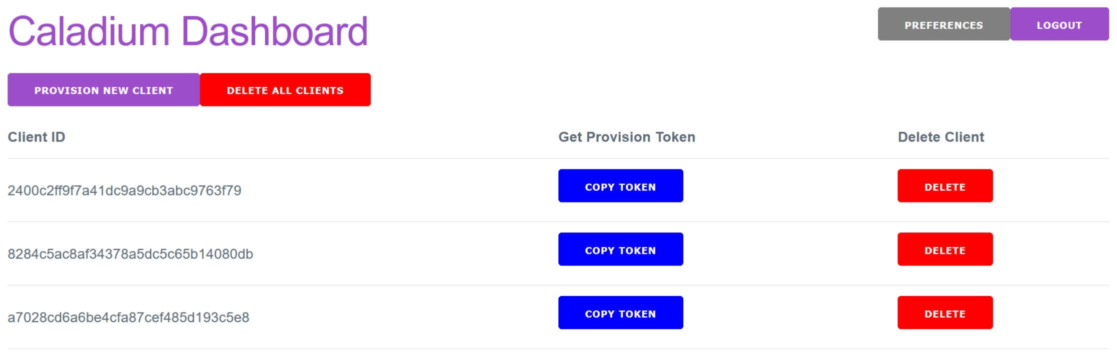


Figure 4.6: Login Page Screenshot

4.2.5 List Page

Below you can find an example screenshot of one of the list pages, the "Clients" page list all the clients in the client's table, each list page has a button at the top allowing you to delete all the records.

Each record has a number of buttons that can perform actions on that record, these are specified by the class that inherits the list page class.



Client ID	Get Provision Token	Delete Client
2400c2ff9f7a41dc9a9cb3abc9763f79	COPY TOKEN	DELETE
8284c5ac8af34378a5dc5c65b14080db	COPY TOKEN	DELETE
a7028cd6a6be4cfa87cef485d193c5e8	COPY TOKEN	DELETE

Figure 4.7: List Page Screenshot

4.3 Server Side

The server bridges communications between the clients and sandbox instances. Clients use HTTP to perform actions on the server. I'm using Flask on the server to facilitate this. The server also needs to be able to persist data, which will be done with CouchDB. Below, I will show how I abstracted the table records into a class called `DatabaseRecord`.

4.3.1 Server Endpoints

The server uses the Flask micro-framework to provide the RESTful API. I decided to categorise the API into four sections: clients, patterns, tasks and workers. Flask supports a feature called blueprints, which allows you to split your web application into multiple components. I used a blueprint for each of the four sections of the API, and each blueprint is stored in a separate file with the name of the API resource.

This is an example of a blueprint taken from the worker's blueprint.

```
...

import flask

...

workers = flask.Blueprint(__name__, "workers")

@workers.get("/api/workers")
def get_records_route():
    return database.get_caladium_collection("workers")
```

HTTP Methods and Descriptions	
Method	Description
GET	Fetches data from a server, without modifying it.
POST	Creates new data on the server.
DELETE	Delete a record by ID.

Table 4.1: HTTP Methods and Descriptions

...

Each of the endpoints is RESTful, the HTTP method name signifies the type of operation to be performed on that resource, the table above will show each method used in the server and a description of its purpose.

4.3.2 Database

Data was persisted using CouchDB, and the server communicates with it using the pycouchdb module.

Having to write out the pycouchdb queries became cumbersome, so I decided to abstract the records of the database away using a class called `DatabaseRecord`, this can be found in `server/src/database.py`.

Now If I wanted to fetch a record from a table I would do this:

```
task_record = database.get(TaskRecord, task_id)
```

To get/set attributes of the record, I would use the instance methods of `DatabaseRecord`.

In the case of the setting operations, the `DatabaseRecord` class would push the modified attribute of the record to the CouchDB instance.

```
print("This is worker ID:", task_record.get("worker_id"))

# Setting new worker_id
task_record.set("worker_id", "1")
```

4.3.3 Authentication

Before processing each call in the server, the `before_request` function is called, as seen in the `__main__.py` file. It checks the token contained in the request and verifies if the caller has the required permissions to execute the request.

4.3.4 Cloud Hosting

I'm using Microsoft Azure to host a Linux server in the cloud. During development, to test the latest version, I would pull in the latest version of the repository using Git and then build and run the Docker image using the `Dockerfile`.

I am also using Azure to host the CaladiumBot server part. To expose ports other than 80, which is the default for HTTP web traffic, I had to manually add the ports in the Azure dashboard.

My promotional page, which is hosted on GitHub Pages, communicates with this CaladiumBot server. But could only do this if the server had a valid HTTPS certificate. To get an HTTPS certificate you need your server to be linked with a domain name. So I purchased a domain name, and generated an HTTPS certificate, then I succeeded in getting it working.

4.4 Analysis Side

The main file of the analysis side can be found in `sandbox/src/main.rkt`, it is written in the Racket language. When it begins it will open a TCP port, ready to accept incoming requests

4.4.1 Communication with Server Side

When a client asks the server to analyse a file, the server needs to assign the job to one of its worker instances. To keep track of what the workers were doing, the server needed a way to communicate with them in real-time. I chose to use TCP for this purpose, a communication method that allows data to be sent and received between devices.

The main program for analysing the file is written in Racket, which has built-in support for TCP.

The server is written entirely in Python, which also supports TCP using the `sockets` library. But, I encountered a problem when trying to receive information from the workers. The server had no way of knowing how much data was being sent, so I looked for a solution and found a helpful answer on Stack Overflow. [21] The solution was to include the length of the incoming data at the beginning of the packet, so the server would know how much data to expect.

This code can be found in `sandbox/src/caladium_resp.py`. It initially sends the size of the data, it uses the struct module to convert the Python integer, into bytes form, and the $>$ denotes that it will be in big-endian. (most significant byte on the left). Then sends the actual data, the other computer listens for the initial size, then knows how much to read. I've included the code snippet that does this, below.

```
output_func(struct.pack(">I", len(json_obj_bytes)))
output_func(json_obj_bytes)
```

4.4.2 Scanning Process

When a file is passed to the analysis worker from the server, the scanning process executes as seen in the diagram below. Malicious patterns are also passed to the worker alongside the file to be scanned.

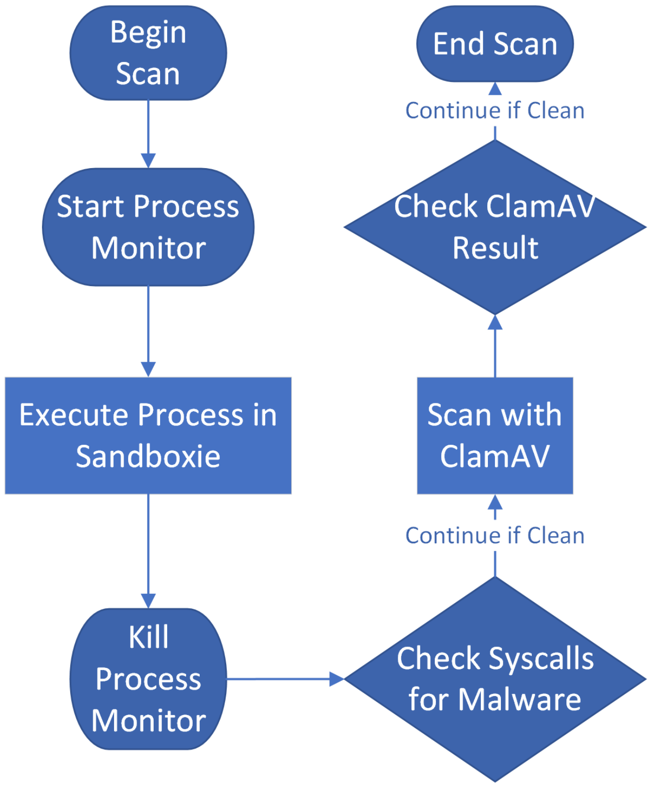


Figure 4.8: Scanning Process Diagram

The first part of the process is dynamic analysis. It begins by spawning Process Monitor, then the target file is executed in Sandboxie. While this is happening, Process Monitor is logging the system calls performed by the target file. Once the process is finished or a certain amount of time elapses (timeout), Process Monitor is killed. This will return a list of system calls in CSV format.

Then it must iterate through all these system calls, to see if there is a match with one of the patterns that was passed to the worker. If there isn't a match, it will continue to the next stage of the process, the static analysis.

This is done with ClamAV, which passes the location of the file to `clamscan.exe` as a parameter. It will check the output of this to see if it's malicious. If both strategies detect nothing, the scan will end, and the file can be deemed clean. Each state of the process gives updates back to the server along with text messages that will be presented in the GUI application's log box.

4.5 Promotional Page

In addition to the sub-projects above, the platform features a promotional page hosted on GitHub Pages.

It contains two buttons on the top of the page, the first downloads the latest build of the Caladium client from the GitHub releases. The second button "Chat with CaladiumBot", launches a chatbot that is powered by GPT-3.5.

4.5.1 Fetching Latest Release

To find out the latest GitHub release for the client, the page must query the GitHub API to get the latest release tag.

It will query `https://api.github.com/repos/g00378925/caladium/tags`, and this will return a JSON object that contains the latest tag. Setting the window location to this URL and setting the tag will download the file.

```
window.location = `https://github.com/G00378925/caladium/
    releases/download/${latestTag}/caladium-setup.exe`;
```

4.5.2 CaladiumBot

In order to use the OpenAI API to access GPT-3.5, I needed to use an API key, which was supposed to be private and couldn't be kept on the public promotional page.

To fix this, I created a shim API Flask server located at `bot/__main__.py`, similar to the main server above. This shim API provided a single API endpoint `/api/ask_question`.

When a user asks a question on the promotional page, it inserts the question into the HTTP body, when sending the request to the API. Upon receiving the question, the server sends a copy of the `README.md` as an initial prompt to the model. The model then responds with its text prediction, which the shim API sends back to the promotional page. The promotional page then dynamically adds this text, character by character, to the page using JavaScript, creating the effect of generating the text.

An excerpt of the code that creates this effect can be found below.

```
// Add text to the output box
function addTextToOutput(outputText) {
    document.getElementById("caladiumBotOutput")
      .innerHTML += outputText;
}

...

function addCaladiumBotMessageEffect(caladiumBotMsg) {
    ...

    // This adds a single character,
    // and pauses for a random amount of time
    addTextToOutput(caladiumBotMsg[0]);
    setTimeout(addCaladiumBotMessageEffect,
        Math.random() * 100, caladiumBotMsg.substr(1));
}
```

Chapter 5

System Evaluation

In this chapter, I will evaluate how well the objectives were met. I will also discuss the limitations of my platform, including its stability and scalability.

5.1 Detecting Malware

5.1.1 Dynamic Analysis

The main goal of this project is to detect malicious software. To achieve this, administrators can add malicious patterns through the dashboard. These patterns are then passed to the workers when a scan task is created. The software is executed, and its system calls are collected and compared against the malicious patterns. If any of the system calls contain any of the patterns, then they can be deemed malicious.

I tested the dynamic analysis, by compiling the C program provided below (using a compiler such as GCC or Clang). The program creates a file called "very_malicious.txt" when executed, and I added this file name to the malicious patterns before scanning the executable.

And I got the GUI application to scan this file using the manual scan option on the main page. **Caladium successfully detected the file as being malicious.**

```
#include <windows.h>

#define FILE_NAME "very_malicious.txt"

int main(int argc, char *argv[]) {
    DeleteFileA(FILE_NAME);
    CreateFile(FILE_NAME, GENERIC_WRITE, 0, NULL,
        CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

    return 0;
}
```

5.1.2 Static Analysis

The primary method of malware detection is dynamic analysis, it uses static analysis as a fallback, and uses ClamAV to facilitate this.

To test this, I am using the "EICAR Test File" [22], which is a standardised file that is universally detected as malware by all anti-malware solutions.

Caladium successfully detected the file as malware.

5.1.3 Quarantine

When the sandbox analysis worker detects the file as malware, it will send a message back to the client indicating "malware_detected".

The user in the client will then be prompted to quarantine the malware, and the scan will finish. **The quarantined file can be found in the quarantine frame, concluding that the quarantining feature works correctly.**

5.2 Code Quality

I am evaluating the code quality based on its cyclomatic complexity and its readability.

5.2.1 Cyclomatic Complexity

Cyclomatic complexity is a metric that measures how complex code is. This can indicate that there may be too much code inside a class or method.

`Radon` is a tool that you can use to calculate the code complexity of your code base. [23]

By using the `radon cc client -a` command, I was able to measure the Python code found in the client directory, and it analysed the code of 48 blocks, including classes, functions, and methods. The average complexity for the client was calculated as `A (2.3125)`.

I also ran the tool on the server and the average complexity was calculated as `A (1.7567567567567568)`. They both got grade A for complexity, meaning the complexity is evenly distributed.

While these are good scores for code complexity, it does not necessarily indicate good code readability. Readability can be improved by using good variable names.

5.2.2 Readability

Throughout the development process, I have made efforts to assign meaningful names to variables that reflect their purpose. I have also created functions to break up functionality, which helps to keep the cyclomatic complexity low, as explained above.

In the GUI application, I have organised each frame into its own class, such as the `QuarantineFrame`, which contains GUI-specific functionality. The actual code for handling quarantine logic is placed in a separate `Quarantine` class.

I believe that following these measures has improved the overall code readability of the application.

5.3 User Experience

A user-friendly interface is provided to both users and administrators through the GUI application and the dashboard.

5.3.1 Windows GUI Application

To install the software, users just need to execute the installer, which will automatically install and provision the software, and start it on system boot. It will immediately begin scanning the downloads directory, and users can easily remove the software through the preferences frame.

The quarantine frame makes it easy for users to understand which files are quarantined and allows them to easily quarantine new files. Users also have the ability to modify the current scanning directory.

When users scan a file, they are given real-time feedback on the scan process, and a progress bar showing the percentage level of completion, users can easily stop the scan process by closing the scan window.

I believe these features, create a rich and friendly experience for users.

5.3.2 Dashboard

Administrators are prompted to log in to the dashboard and are given the option to change their password in the preferences menu once logged in.

Administrators can disable dynamic analysis in case it breaks, and the system will then solely rely on static analysis.

The dashboard is a single-page application, which allows seamless switching between pages without loading times. The list pages allow administrators to easily add and remove clients or workers.

5.4 RESTful APIs

Clients communicate with the server using RESTful APIs. Each of the endpoints is designed to be stateless, meaning that the server does not need to maintain context. HTTP methods such as `GET` and `POST` are used to specify the type of action to be performed.

In the tasks endpoint, I am using the HTTP `DELETE` method to delete records by their ID.

I have developed a suite of tests to test the endpoints when the code is pushed to the repository. Currently, all tests are passing. And I have also confirmed their functionality as the GUI application is able to successfully scan files, which relies on the APIs functioning correctly.

GitHub shows a green check mark, indicating all tests are passing.

5.5 Security

As stated in the introductory chapter, the platform needs to be secure, it would be rather hypocritical to ignore security in a platform targeted at identifying malware

5.5.1 Authentication

To get into the dashboard you must log in with a username and password. The password is hashed using the SHA-256 algorithm. If the server is compromised, the admin's password won't be readable in its original form.

The diagram below illustrates how data can be easily hashed, but is difficult to reverse the process. Passwords can be securely stored after being hashed, this makes it infeasible to retrieve the original password from the hash.

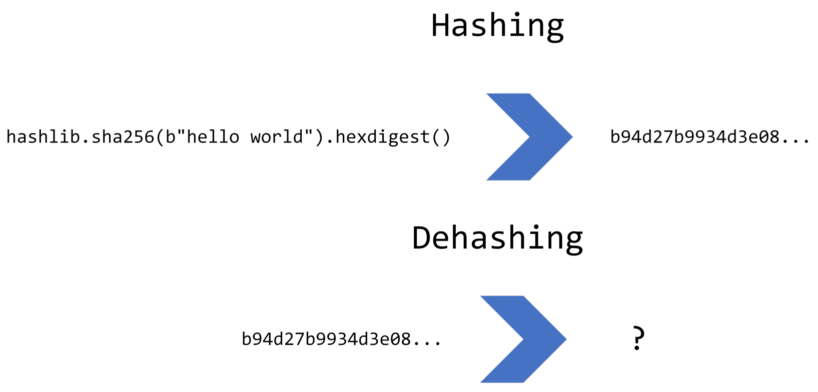


Figure 5.1: Hashing Algorithm Diagram

When you log in you are given a session token, which is stored in the browser's local storage. All RESTful API calls require a valid session token to be sent with the request.

In the `__main__.py`, you can find a function called `before_request`, this gets called before every request, it checks the path being requested and if the requester has the correct token to access the resource.

5.5.2 Cross Site Scripting (XSS)

Cross-site scripting is a type of attack, where attackers can inject code into other users' browser sessions. [24] A field may display data on the screen, and attack input data may contain HTML that is not sanitised.

In the screenshot below, you can see how the `"hello <script>alert(1)</script>world"` string can cause a problem. The script tags are being processed literally, executing the JavaScript and showing the alert box.

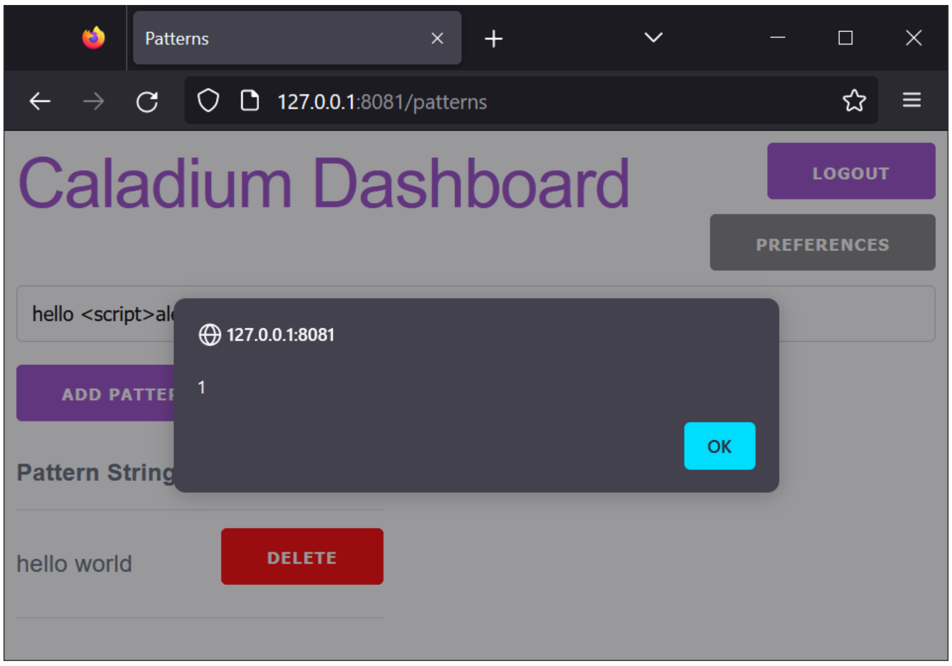


Figure 5.2: Dashboard XSS Screenshot

In my project, I modified my code in the dashboard after discovering this security bug, to convert HTML tags to entity tags.

I added the following code, which replaces the < and > characters with their safe equivalents.

```
if (newHash["innerHTML"]) // Disable XSS
    newHash["innerHTML"] = newHash["innerHTML"]
    .replace('<', "&lt;").replace('>', "&gt;");
```

5.6 Design Principles

I set out from the beginning with the objective of ensuring that the code in my project adheres to good design principles.

Throughout development, I tried to follow the DRY (Do Not Repeat Yourself) principle as much as possible.

This is evident in the Page hierarchy found in the dashboard, where each of the list pages shares common functionality, this being: displaying a list of records and showing a navigation bar. I refactored this functionality into a class called `ListPage` that all the list pages inherit.

In the GUI application, I encapsulated the frame code into its own classes, these classes are the ones with the `Frame` suffix, `QuarantineFrame` for example.

5.7 Stability

Functionally, the platform works as intended, but sometimes unexpected problems can occur related to the networking aspect.

When designing a platform that makes use of different languages and computers, getting everything to work seamlessly is quite a challenge.

To try and fix this, I've added timeouts to network requests in the GUI application when making HTTP requests. I've also added code to catch exceptions and present the relevant error message.

This is code from the GUI application that creates a new scan task. Notice the timeout parameter; this triggers an exception if it goes beyond 20 seconds.

```
resp_obj = provisioning.caladium_api("/api/tasks", method="POST",
            data=file_data, timeout=20)
```

5.8 Scaling

The server is capable of handling multiple clients, the platform was designed to make use of only one server, but it is possible to have multiple instances of the server, and have them all use the same CouchDB instance.

In the case of one server crashing, clients using a different instance of the server would still have the ability to scan files.

The platform can handle multiple scans at the same time, as multiple workers are supported.

The platform can scale by adding more analysis workers.

5.9 Time Complexity

When a file is executed by the worker it produces a log of all the system calls it produces, a list of patterns is passed to the worker in order to be checked against the system calls, if there is a match then the file can be deemed malicious.

This is a sample of the code from `sandbox/src/syscall_analysis.py`, before optimisation it had a O(n * m) time complexity, n is the size of the syscalls and m being the size of the malicious_patterns.

```
def analysis_worker(syscall_queue, malicious_patterns):
  while True:
    # Fetch a syscall from the queue
    try: syscall = syscall_queue.get(timeout=1)
    except: break
    for pattern in malicious_patterns:
      # Check for a pattern match
      if pattern in syscall:
        # Detection . . .
```

Before optimisation, if the malicious_patterns list grows significantly this would quickly become a bottleneck, so the solution I found is doing the checking in parallel, each of the system calls would be added to a queue, and a worker thread would fetch a syscall and then compare it against all the patterns.

This allows the analysis to quickly check all the syscalls.

As a test, I scanned common programs that come with Microsoft Windows. In the table below, you can find the name of the executable and the time it took to analyse the syscalls.

Executable Name	Second(s) Taken to Analyse
notepad.exe	0.84
SnippingTool.exe	0.89
Defrag.exe	1.2

Table 5.1: Analysis Times

Chapter 6

Conclusion

6.1 Brief Summary

The goal of this project was to develop a platform capable of detecting malicious software based on its behaviour.

For this to be achieved, files needed to be executed, these files were potentially malicious, so they needed to be isolated from the host system. Sandboxie was used for isolation, and Process Monitor was used to log the actions/system calls of the software being run.

Users can download and install a GUI Windows application to their computer. When a user downloads a new file, the client will detect it and prompt the user to scan it. The file will be sent to the server and then will be passed to a sandbox analysis worker, and the client will display the real-time feedback of the scan. If the file is found to be malicious, the user will then be prompted to quarantine it.

The clients communicate with the server through a RESTful API, the dashboard also uses this. The dashboard was designed to be user-friendly, allowing administrators to easily perform administrative tasks such as adding and removing workers.

In the objectives, it was stated that security would be given top priority. In the evaluation chapter, I showed that the administrator's password is hashed, and in the event of the dashboard being compromised, the plain text password would never be known. When auditing the code I did find some security vulnerabilities, this was the XSS which allowed unsanitised HTML to be injected. I then fixed this by substituting the HTML < and > characters with their entity equivalents.

I aimed to create clean and readable code and use good practices, such as the DRY (Do Not Repeat Yourself) principle whilst developing the software.

Using an agile methodology, I broke down features into tasks and added them to the Kanban board (can be found in Appendix B). I created tests for the endpoints which can be found in the `server/tests` directory, and these are automatically executed when the code is pushed to the repository. Every couple of weeks in the 2nd semester, I made releases of the client, which can be found in the releases section in the GitHub repository. I am currently on my 6th release.

6.2 Findings

6.2.1 Outcomes

- Can detect malware using dynamic and static analysis and quarantine if deemed malicious.
- Created a comprehensive platform, that is well-organised and has clean and readable code.
- Created a native GUI application for Windows that allows users, to scan their files and shows the scan progress.
- The dashboard allows administrators to perform administrative actions on the platform such as removing and adding workers.
- Clients can communicate with the server using the RESTful API, and has been tested using the test suite found in `server/tests`.
- The password for the administrator user is hashed to prevent unauthorised access in the case of compromise, and the XSS vulnerability that was found, has been patched.

6.2.2 Format Agnosticity

One of the more interesting applications of the dynamic scanning feature is its format-agnostic capability. This means that it can scan any file, regardless of whether it is an executable file or not.

For example, you can scan an image file to see how it affects the image viewer in Windows. Even though the image isn't an executable file, it could be crafted in such a way as to exploit a vulnerability in the image viewer resulting in code execution.

The dynamic scanning feature can be used to scan the files of other programming languages. For example, if Java is installed, you can scan a Java JAR and observe the syscalls that Java would perform.

6.2.3 Cross Site Scripting (XSS)

Administrators can input data into various fields on the dashboard, such as adding new malicious patterns.

I discovered that my dashboard was not sanitising the inputs, which could allow a malicious user to input raw HTML, that would be displayed in its raw form resulting in a vulnerability known as the XSS.

This is a common oversight made by many developers, as it requires in-depth knowledge of how browsers display content, which may be lacking, especially for developers who come from a server development background with less exposure to the web.

6.2.4 Network Reliability

When designing software that utilises networking technology, it is important to avoid making naive assumptions based on the comfort of developing on a local machine.

The reality is that developing for networks is challenging, as there are numerous factors to consider, including latency, potential delays before data reaches its destination, and unexpected network interruptions that may be beyond one's control.

During the development of Caladium, I had to strike a balance between reliability and functionality. I made sure that it was working at a minimum between my Azure instance and my local computer and decided to put the rest of my time into working on the remaining functionality.

This serves as a lesson, to avoid making naive assumptions, by putting more time into researching technologies that are trusted in the industry, that facilitate communication between distributed systems, like this project.

6.2.5 Opportunities Identified for Future Investigation

To determine if a file is malicious, I use a heuristic that involves checking if the system calls contain any of the known malicious pattern strings.

Regular expressions may have been more appropriate for this purpose, but I decided against using them, due to them being computationally expensive and potentially slowing down the scanning process.

Another possible approach is to assign weights to certain patterns, and if the total malicious score exceeds a certain threshold, then the file can be considered malicious.

In my system, I use an absolute approach, if a syscall contains any of the known patterns, it is considered malicious. This could be a potential application of fuzzy logic, where there is an aspect of uncertainty.

Large language models like GPT-4 could also have been used, it could take in a sample of the system calls and be instructed to return a string of text indicating if they are malicious or not. I also decided against this, as it's quite expensive at this time.

6.3 Conclusion

Overall, the project has been a massive success, because I achieved all my objectives set out in the introduction.

The platform provides a user-friendly experience to users who can download and install the client to their computer.

It will automatically start on boot, similar to typical anti-malware solutions, and begin scanning for newly downloaded files. When a new file is detected, users are prompted to scan the file.

Upon scanning, the file will be run on a sandbox analysis worker, providing in-depth feedback back to the client, and will then use ClamAV as a fallback for static analysis.

The most challenging part of the project was implementing the dynamic analysis, as I did not have any libraries that could do this for me. I had to research and combine different components to achieve this. I used a variety of languages and libraries to make Caladium possible.

Due to this being a significant project with multiple components, I learned about good software practices and applied methodologies such as agile. I also created tests for the endpoints, that were tested upon pushing to the repository.

I gained knowledge about the malware side of cybersecurity by researching malware detection and dynamic analysis. [2] Developing software requires knowledge not only in software development but also in neighbouring fields.

I found an opportunity to use GPT-3.5 in my project, it was used in the CaladiumBot, which answers questions about the platform. The model is given the `README.md` as an initial prompt, this is how it's able to answer questions about Caladium.

When I started the project, I wanted to learn more about functional programming, which is why I used Racket to write the main part of the sandbox analysis. Functional programming can help promote cleaner code, and this can be applied to other programming languages such as Python.

Bibliography

[1] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430, 2007.

[2] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. Dynamic malware analysis in the modern era—a state of the art survey. *ACM Comput. Surv.*, 52(5), sep 2019.

[3] The UPX Team. Upx: the ultimate packer for executables. Last accessed on 16 Apr 2023. `https://upx.github.io/`.

[4] OpenAI. Gpt-3.5 - openai. Last accessed on 16 Apr 2023. `https://platform.openai.com/docs/models/gpt-3-5`.

[5] Microsoft. Process monitor - microsoft. Last accessed on 16 Apr 2023. `https://learn.microsoft.com/en-us/sysinternals/downloads/procmon`.

[6] Cisco. Clamav® is an open-source antivirus engine for detecting trojans, viruses, malware other malicious threats. Last accessed on 21 Apr 2023. `https://www.clamav.net/`.

[7] Xanasoft. Sandboxie - xanasoft. Last accessed on 16 Apr 2023. `https://sandboxie-plus.com/sandboxie/`.

[8] ECMA International. The json data interchange syntax, 2017. `https://www.ecma-international.org/publications-and-standards/standards/ecma-404/`.

[9] Python Software Foundation. Python programming language. Last accessed on 21 Apr 2023. `https://www.python.org/`.

[10] PyInstaller. Using pyinstaller. Last accessed on 16 Apr 2023. `https://pyinstaller.org/en/stable/usage.html`.

[11] Python Software Foundation. tkinter. Last accessed on 21 Apr 2023. https://docs.python.org/3/library/tkinter.html.

[12] Roger D. Serwy. tkthread. Last accessed on 21 Apr 2023. https://pypi.org/project/tkthread/.

[13] ECMA International. Ecmascript® 2015 language specification. Last accessed on 16 Apr 2023. https://262.ecma-international.org/6.0/.

[14] Pallets. Flask - web development one drop at a time. Last accessed on 21 Apr 2023. https://flask.palletsprojects.com/en/2.2.x/.

[15] Roy Thomas Fielding. Architectural styles and howpublished = Last accessed on 16 Apr 2023, note = https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[16] CouchDB Team. Couchdb. Last accessed on 16 Apr 2023. https://cwiki.apache.org/confluence/display/couchdb/introduction.

[17] Docker Inc. Docker: Accelerated, containerized application development. Last accessed on 22 Apr 2023. https://www.docker.com/.

[18] The Python Software Foundation. unittest — unit testing framework. Last accessed on 17 Apr 2023. https://docs.python.org/3/library/unittest.html.

[19] PLT Inc. Racket - plt inc. Last accessed on 16 Apr 2023. https://racket-lang.org/.

[20] Madhuri A Jadhav, Balkrishna R Sawant, and Anushree Deshmukh. Single page application using angularjs. *International Journal of Computer Science and Information Technologies*, 6(3):2876–2879, 2015.

[21] chqrlie. Simple method to read a packet from a tcp socket in c, October 2022. https://stackoverflow.com/a/74089089.

[22] Trend Micro Incorporated. Eicar test file. Last accessed on 16 Apr 2023. https://docs.trendmicro.com/all/ent/de/v1.5/en-us/de_1.5_olh/ctm_ag/ctm1_ag_ch8/t_test_eicar_file.htm.

[23] Michele Lacchia. Radon. Last accessed on 16 Apr 2023. https://radon.readthedocs.io/en/latest/.

[24] The OWASP® Foundation. Cross site scripting (xss). Last accessed on 16 Apr 2023. https://owasp.org/www-community/attacks/xss/.

Appendix A

The GitHub repository for the project, can be found at this URL:
https://github.com/G00378925/caladium

A screencast of the project can be found at this URL:
https://www.youtube.com/watch?v=aYbQChGvz88

And this is a screencast of the promotional page and CaladiumBot:
https://www.youtube.com/watch?v=hMskBzCt6kU

Appendix B

Card No.	Task
1	The server needs to be able to allocate jobs to the sandbox instances.
2	Dashboard needs to be able to add sandbox instances through the UI.
3	The server needs to be able to upload files to be run.
4	Get Docker setup and write a Dockerfile.
5	The single-page application needs to be able to generate HTML dynamically.
6	Add a preferences menu to the dashboard so admins can update settings.
7	Clients need to be able to pass files to the server to be scanned.
8	The dashboard needs to be able to ping workers to see if they are alive.
9	Add code to check if system calls are malicious.
10	Rewrite syscall analysis in Python because it is too slow in Racket.
11	Add piecharts and barcharts to the index page on the dashboard to display statistics.
12	Make variable names consistent.
13	When a user installs the Caladium client, the client must request the provisioning token.
14	GitHub Action to automatically run the test suite when a commit is pushed to the repository.
15	Add an uninstall function to the Caladium client.
16	Add the static analysis code to the worker.
17	Setup GitHub pages for the promotional page.
18	Allow administrators to be able to set custom IPs for the server when building clients.
19	Add a label to the main page of the client, showing the currently scanned directory.
20	Fix the auto-provisioning bug where it doesn't provision after inputting the profile JSON.

Appendix C

The following are instructions, for building each component of Caladium:
First, retrieve the latest build of Caladium, you can do that with this command:
`git clone https://github.com/G00378925/caladium.git`

Python is a requirement for all sub-projects, you can download and install, the latest version for your system here.

C.1 GUI Application

You must make sure you are on Microsoft Windows for this part as **iexpress.exe** is required. Run the following commands in **Command Prompt**, making sure you are in the Caladium root directory.

```
cd client
python -m pip install -r requirements.txt

rem You will be prompted to enter the IP address of the Caladium server

build.cmd
```

This will result in a `caladium-setup.exe` in the `dist` directory, this is the installer that will be given to users.

C.2 Server

To build and run the server, enter the following commands into your terminal, this should work on Linux, macOS and Windows.

You need the address of your CouchDB instance this needs to be put in the environmental variable `COUCHDB_CONNECTION_STR`. If on Windows use `set COUCHDB_CONNECTION_STR=admin:root@0.0.0.0:5984`, if on Linux or macOS use `export COUCHDB_CONNECTION_STR=admin:root@0.0.0.0:5984`.
With `admin:root@0.0.0.0:5984` being the CouchDB connection string.

Make sure to swap out ‘python3‘ with ‘python‘ if executing on Windows.

```
cd server/src

python3 -m pip install flask requests
python3 __main__.py
```

The address of the instance, will be printed to the terminal.

Using Docker

Instead of having to set up an environment manually, you can use the supplied `Dockerfile` to set up a reproducible environment, replace the `0.0.0.0:5984` in the commands below like above.

Make sure you have Docker installed on your system, and type the following into your terminal.

```
cd server
sudo docker build --tag caladium .
sudo docker run -e COUCHDB_CONNECTION_STR=^
    "http://admin:root@0.0.0.0:5984" -p 80:8080 caladium
```

C.3 Sandbox Analysis

To set up the analysis service on a computer, you must first install Racket and Sandboxie. Create a `SysinternalsSuite` directory at the root of your drive, this is usually the `C` drive. Place `Procmon64.exe` in there, you can download it here.

Alternatively, you can use the commands below, to download Procmon to that location.

```
mkdir /l %SystemDrive%\SysinternalsSuite 2>null
curl https://live.sysinternals.com/Procmon64.exe
    --output %SystemDrive%\SysinternalsSuite\Procmon64.exe
```

ClamAV is required for the static analysis, I am using a distribution of ClamAV called ClamWin, which can be downloaded here, ClamWin will automatically download the latest malware definitions.

Run a new Command Prompt as administrator, and run `cd sandbox && start_sandbox.cmd`. The analysis service will now poll for tasks, in the case of it crashing, it will automatically restart. Note the IP address and port of the service, in the format `0.0.0.0:8080` and add it to the administrator dashboard.