CONJURE

ACE Research Project

Jun - Aug 2023

Research Team Members:

Veronica Vergara

Nolen Shubin

Quillen Flanigan

Dillon Shah

Alan Middleton

Elliot Viaud-Murat

Research Mentors:

Tom Henry

Matthew Rodrick

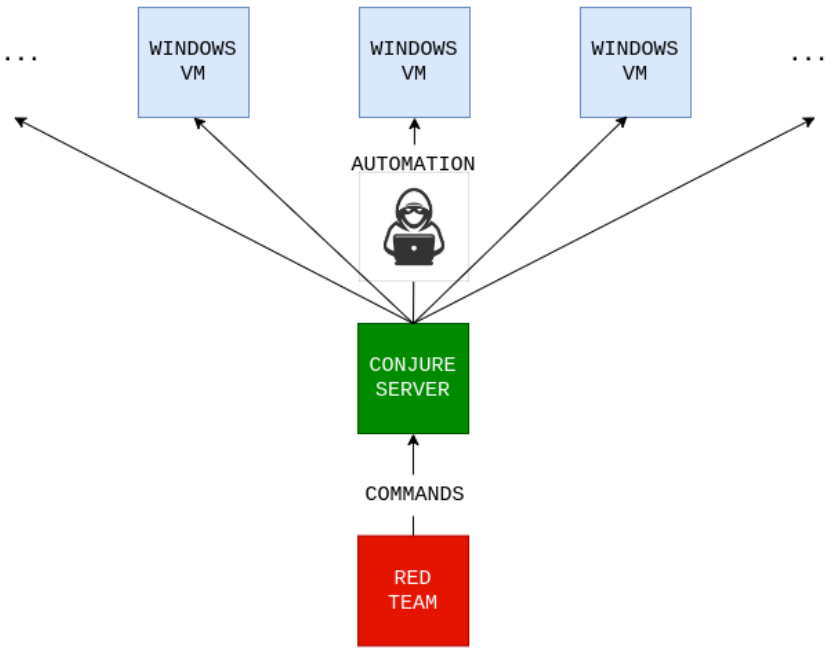
Research Supervisors:

Ryan Casa

1. SUMMARY

Conjure is a server that can send commands and run processes to several Virtual Machines (VMs) to simulate user activity and have the White Team facilitate Red Team requests to run operations on those machines. Such interactions include web browsing, running terminal commands, downloading files, and executing files. Logs of these activities across the VMs are consolidated and provided upon request. Our tool requires minimal installation on the local machine and an SSH connection to the conjure server. The Conjure server runs on the Ubuntu 22.04 OS image to command VMs running on Windows 10. [One or two sentences from Elliot/Dillon] The following documentation goes over our work from the weeks spent developing Conjure and how we came to the solutions we use in the final product.

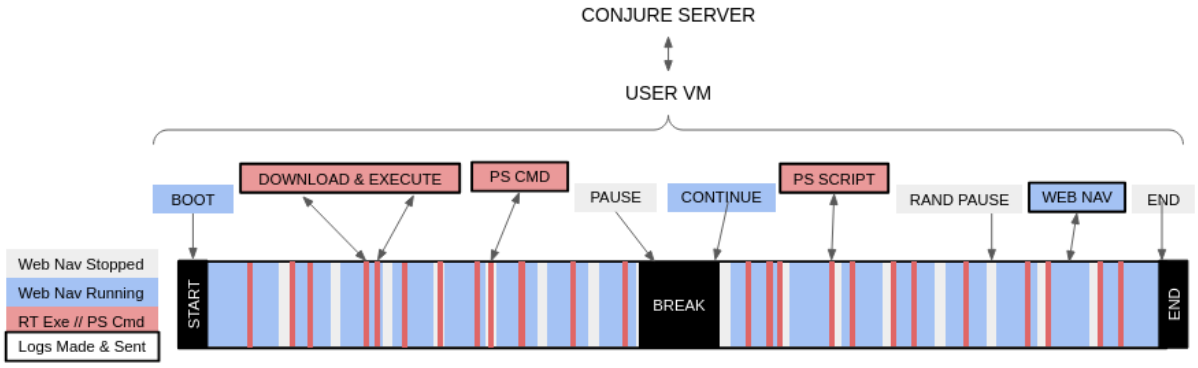
CONJURE



2. Conjure Capabilities

Every VM should have a Conjure client set up prepared before the initial boot on the day of the exercise. Conjure's interaction with each of the VMs involved the following primary actions:

- Tunnel from Conjure into a target VM
- User Automated Web Navigation
    - Start Web Nav on target VM at Boot
    - Access a random website
    - Scroll to a desired element on a website
    - Explore to other sites via hyperlinks
    - Randomly stop, wait, and restart
    - Pause and Continue Commands from Server
- Red Team Activity
    - Run Individual Powershell Commands or Scripts
    - Download files from a website to VM
    - Execute files (malicious payload) that are downloaded
- Logs for Web Nav, Downloading, Executing, and Commands
    - Success or Failure with Error messages sent to Server
    - User and IP Address associated with activity
    - Conjure automatically sends logs after the "pause"/"end" commands or requesting those logs via "log" commands



3. Set Up Requirements

The Conjure repository has a README file that explains how to set up both the server and the client VMs with Conjure. We hope that by the time we deliver this repository, there will be no need to connect to it from a vpn. We had issues cloning this repository at first because Git wanted to certify the domain we were cloning from. To avoid this issue, we set SSLVerify to false with the command git config --global http.sslverify false which should allow you to clone the repository from git.rsch.

In the repository, three sections divide the code inside the repository: server, client-vm, and conjure. The conjure directory holds the code that we developed during the research time that did not end up being used in the final product. Server has the code for a server to run the Conjure server while client-vm has the code to be installed on every VM inside the exercise. Conjure's README has more details about where to download the code in the VMs and the server.

3.1. Installations and Version Numbers

After the appropriate code is installed on the VMs and the Conjure server is set up, there are a few libraries and imports that must be installed for the programs to run correctly. The version numbers of these dependencies along with the services used by the VM programs are listed below.

- Visual Studio Code 1.78.2 (Used for testing)
    - Electron: 22.5.2
    - Chromium: 108.0.5359.215
    - Node.js: 16.17.1
    - V8: 10.8.168.25-electron.0
    - OS: Linux x64 5.19.0-50-generic
- Google Chrome: 113.0.5672.126 (64-bit)
- .NET (DOTNET) 7.0.9
- Pwntools 4.10
- Selenium 4.10.0/4.11.0
- Javascript 1!1.0.2

- Requests 2.25.1
- Requests.html 0.10.0
- Websockets.client 1.2.3
- Websockets 10.4
- Wget 3.2

The Conjure server plays a critical role and should exclusively exist on a White Team administrator's computer. While a single server should suffice, setting up multiple servers is an option to effectively manage different sets of machines. It is crucial to maintain an updated list of every Virtual Machine's (VM) IP address that Conjure will manage, as this enables the server to exert control over those specific VMs.

For optimal performance, it is recommended to download the client VMs in the Windows/System32 directory. While theoretically they can be placed anywhere, the System32 directory offers the advantage of having other services running, effectively masking Conjure's interactions with the VMs and running at boot. It's important to ensure that Conjure's client code is running on the VMs before initiating the exercise, as this guarantees that they will appropriately respond to commands from the server.

3.2. Working with C# Source Code

In order to work with and edit the C# source code for the user automation functions integrated into Conjure, some steps need to be taken in order to do so. These instructions are intended for VStudio and specifically the versions mentioned previously.

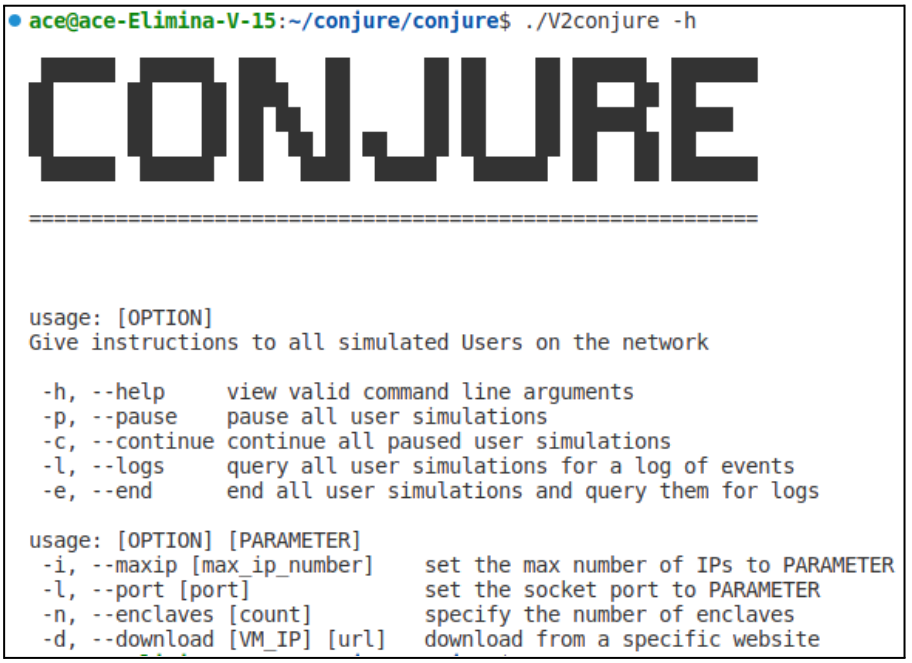
1. Install DOTNET with ***sudo snap install dotnet-sdk –classic***
2. In VStudio, go to the Extensions tab
3. Look for “vscode-solution-explorer” and install it
4. With Ctrl+Alt+L, the Solution Explorer panel should open
5. Create a new solution in Solution Explorer
6. Add a new project and selection “Console App” then “C#” and name the file/ folder
7. Expand the directories until you can see the references and packages
8. Right click and select Add Packages
9. Add the Selenium.WebDriver package
10. In your directory run ***dotnet new console*** then ***dotnet build*** after writing your code and finally ***dotnet run***

4. How To Use Conjure CLI

Help (*./V2conjure -h* –OR– *./V2conjure --help*)

Running *./V2conjure -h* will display the commands that a user can use with the conjure server. These are split into argument-free instructions & instructions that require some arguments. They can be run by typing

./V2conjure <flag> <parameters>



Pause (*./V2conjure -p* –OR– *./V2conjure --pause*)

The pause command will tell all running VMs to pause running. Each instance of conjure client will cease scrolling through webpages and instead continually wait for a new instruction

Continue (*./V2conjure -c* –OR– *./V2conjure --continue*)

The continue command will make all VMs continue running. Any paused instances of conjure client will continue scrolling through webpages. Any unpaused instances of conjure client will do nothing

```
Extracting IP addresses of all VMs
10.241.1.27
10.241.1.72
Sent: <pause> to <10.241.1.72:40148>
Socket closed
```

Logs (*./V2conjure -l* –OR– *./V2conjure --logs*)

The logs command will retrieve all log information from all conjure client instances. This will be outputted into a text file called allLogs.txt

End (*./V2conjure -e* –OR– *./V2conjure --end*)

The end command will close all conjure client instances. Once these have been closed, the client can only be reopened by restarting the machine it is on

```
Extracting IP addresses of all VMs
10.241.1.27
10.241.1.72
Sent: <continue> to <10.241.1.72:60768>
Socket closed
```

Download (*./V2conjure -d <IP of client> <address of file>* –OR– *./V2conjure --download <IP of client> <address of file>*)

The download command will download and execute an online file from <*address of file*> onto the client vm at <*IP of client*>

Maxip (*./V2conjure -i <max_ip_number>* –OR– *./V2conjure --maxip <max_ip_number>*)

Maxip determines the maximum number of IPs allowed to be stored within the conjure server

Port (*./V2conjure -l <port>* –OR– *./V2conjure --port <port>*)

Port will set the port number of the server to <*port*>. **Make sure to set the clients to use the same port**

Enclaves (*./V2conjure -n <count>* –OR– *./V2conjure --enclaves <count>*)

Enclaves will set the number of enclaves that conjure can access

5. Design Choices

Originally, we started to develop the code that Conjure's client VMs would run in Python as a proof of concept. However, we found significant performance and compatibility issues using Python as Conjure's server side needed to run on C. Because of this, we converted our Python scripts to C# to enhance the Conjure client's performance and increase the compatibility with its server. We also use PowerShell scripts to run the terminal commands from the Conjure server to the client VMs.

Because Conjure is designed for the Defense Cyber Marvel (DCM) training exercise, we wanted to conceal Conjure's interactions with the client VMs. We pursued replacing one of the VM's DLL files with our own that would proxy to the original while giving Conjure concealed access to the VM. These DLLs would run at boot so the DLL would make a connection to Conjure pre-exercise to reduce the Blue Team's chances of finding Conjure during the time of exercise.

5.1. DLL Proxying

Conjure employs DLL Proxying as a strategic approach to operate covertly, concealing its automated nature.

The operational mechanism is outlined as follows:

- Standard Program Calls Proxy DLL
- Proxy DLL initiates Conjure code in a new thread
- Proxy DLL forwards function calls to actual DLL

To ensure seamless integration, we replace the genuine DLL with our proxy DLL within the system32 directory. Consequently, when a program seeks the authentic DLL, the proxy DLL is invoked. The genuine DLL is then renamed and/or positioned in a concealed location referenced by the proxy DLL.

Upon DLL loading (during process attach), the Main() function is executed in a new thread, thereby executing Conjure's code. (As of now, the DLL proxy simply runs a conjure executable)

Given that the proxy DLL could be loaded by multiple programs concurrently, a mutex is employed to enforce singular instance execution of Conjure. The mutex creation follows these steps:

- Attempt to create a mutex with a predetermined common name.
- If the mutex already exists, introduce a sleep interval and retry after a specified duration.
- If the mutex does not exist, initiate the Conjure program.

Utilising DLL proxying ensures that Conjure is executed solely as a thread within an authentic process. It represents a highly effective approach for several reasons:

- Detection of automated web browsing is challenging, as no standalone process is created.
- Identifying the origin of execution is intricate, necessitating precise identification of the malicious DLL amid numerous loaded DLLs within a program.
- Removal is complicated:
    - Terminating a suspected process hosting the Conjure program results in another program initiating the conjure code.
    - Eliminating the proxy DLL requires locating and restoring both the proxy and genuine DLL in system32 across affected machines, followed by system reboots.
    - Conjure's resilience increases through potential implementation of multiple DLL proxies for frequently used DLLs, rendering eradication more daunting.

5.2.1 Current Limitations

While the DLL Proxy is fully operational, it's constructed in C++, while Conjure is developed in C#. Notably, C# mandates execution within the .NET framework (managed), whereas our C++ proxy functions in an unmanaged environment.

Presently, the Proxy DLL launches a separate Conjure executable, undermining key benefits of DLL proxying:

- Conjure becomes apparent as a distinct process, rendering identification and removal of the Conjure executable relatively straightforward.

Potential solutions to this challenge and future work include:

1. **Load .NET assembly into C++ Proxy**
    - Create .NET environment and inject the .NET assembly into running process
    - Useful repository: https://github.com/med0x2e/ExecuteAssembly/tree/main
    - Not tried yet, but very likely to work if correctly implemented.
2. **C++ Rewrite for Conjure**:
    - Rewrite Conjure in C++ and implement it directly in the DLL proxy Main() function. However, selenium does not exist for C++ which will make this process cumbersome.
    - This approach remains untested.
3. **C# Implementation within a C++ Proxy**:
    - Develop Conjure as a C# DLL callable from the C++ Proxy DLL.
    - This will hide where the execution of Conjure is coming from.
    - C++ being unmanaged, interoperation with C# is complex but feasible.
    - A critical consideration is preserving the C++ proxy's unmanaged nature, as managed DLLs don't function as proxy DLLs.
    - Prior attempts of this have yielded inconclusive results.
4. **C# Proxy DLL Conversion**:
    - Reconfigure the C++ Proxy DLL to C#; however, feasibility is limited.
    - Running a C# DLL as an entry point from .NET poses challenges, and exporting functions in C# is not as straightforward as in C++.
    - Prior attempts of this have yielded inconclusive results.

5.2. Conjure Server

Conjure server services should be structured as such:

conjure.py

V2conjure

/0

| - networkDetails.txt

| - envVar.txt

/…

| - networkDetails.txt

| - envVar.txt

/N

| - networkDetails.txt

| - envVar.txt

5.2.1. Conjure Server C2

The conjure server is reliant on two text files that store the necessary information for it to function: networkDetails.txt (a file that stores all the IPs of the VMs within an enclave) and envvar.txt (a file that stores environment variables such as the port number). These must both be readable by the server or else it will not be able to access any VMs. The structure for these files must be:

Conjure will iterate through enclaves with a for loop, so enclave folders must be an integer from 0 to N where N is the *number of enclaves - 1*.

V2conjure will issue instructions to all conjure clients currently trying to establish a connection with it; the specific instructions to do so can be found in 4. HOW TO USE CONJURE CLI. Be advised, most commands within V2conjure will communicate with **every** registered conjure client. This is to stop any team having an unfair advantage. The only exception to this rule is the *download* command as that will only act one one device in one enclave.

The pause/continue commands are intended to stop unnecessary use of bandwidth from conjure clients - all conjure clients will cease querying the web while paused, and continue again upon receiving continue.

It is recommended to use a lull in the exercise to *pause* and call the *logs* command. The logs command will send **every** log from **every** machine to the server. Obviously, this is

not very stealthy. However, the logs function **does** work if run without pausing the conjure clients first. These logs will be printed to both the stdout as well as a file.

Download is a **red team** tool that will download a given file from a given page to a specified VM, it will then run this file. The use-case for this is establishing/re-establishing initial access to a victim by downloading and running an implant.

End is a means of **killing not pausing** all conjure clients. If the end command is called before logs are extracted **they will be deleted**.

5.2.2. Conjure Server "Shell"


```
Red Team Interface

>>> help
help                          View possible commands
====                          ======================
ls                            View items in current directory
whoami                        View current user
ip                            View current VM IP
mkdir [dirName]               Make directory [dirName]
rmdir [dirName]               Remove directory [dirName]
echo [msg] > [filename]       Write [msg] to file [filename]
cd [dir]                      Change directory to [dir]
run [scriptName]              Run [scriptName] from PowerShell (include the .ps1)
change [IP]                   Change the workstation you're on
```

As well as the conjure C2 server, there is a pseudo shell for the red team to utilise. These have some built in commands for the red team to use to enumerate or execute powershell scripts (for example, the red team could use *download* with Conjure C2 Server to download a powershell file, then execute it with Conjure Server Shell.

Any command run on the Conjure Server Shell will send a command/list of commands to the associated VM. This can help the red team enumerate, run powershell scripts on their current victim or change the victim they are on.

5.3. User Activity

The current version of the user activities is implemented in C# using the DOTNET framework. DOTNET (.NET) is an integrated software framework that offers multilingual operability, allowing us to use multiple programming languages and functions within our user-simulation programs. We mainly used this .NET framework with our main programs using C# along with incorporating javascript executors to increase the effectiveness of our user-automation.

Selenium and .NET

Within this .NET structure, the main program that facilitates our user-automation uses the Selenium automation library. Selenium allowed us to create instances of web drivers that could be used for automating the ability to open browsers, search and click on links, and scroll to a certain element. This functionality, along with Selenium's multi-browser compatibility made it an effective tool for testing while also being able to simulate a real user as closely as possible. While testing on our Linux machines, we used Chrome as our default browser used for random web navigation. We attempted using Firefox but due to significant performance issues we do not recommend using it. Once we began testing on the Windows VMs, we used the Edge browser to limit the dependencies our product would need to function. Within the main VM-client code, there are options to use either Chrome or Edge as the browser to create the Selenium web driver instance. Both are available to the user, along with the options to hide the "controlled by automation software" flag already being set and used in the code.

5.3.1. Automated Web Navigation

The main functionality of the user-automation code is broken down into web navigation and downloading specific files as directed by the RT. The main function is first

called and determines which action path to direct based on the commands sent from the server. The program is installed within the boot directory of the VMs, meaning once the VMs begin to run the program will run instantly. It will direct the program to execute the 'WebNavigation()' function, which first chooses a random website domain name from a hard-coded list. We have the ability to read in a list of websites from a file named 'websites.txt' already within the code, but we chose to hard-code the list of websites for now to limit the dependencies on the VM. It then creates a web driver with the chosen web browser and opens the randomly chosen website. We then enter a while loop which gathers every link on the current web page and filters to make sure it is clickable by the web driver. The selenium web drivers are particular about what they can/cannot click, so we filter the list of all links to make sure they are enabled, visible, and have a non-zero size. We then click a random link out of the filtered list of links and the program navigates to this new page, and the process repeats while using the scrolling functionality to scroll near each element, further simulating true user activity. The program will then randomly exit this loop by choosing a random number. This randomness along with random wait and scroll times allow us to further simulate user randomness within automation.

While this web navigation function is running, it checks the ExecuteClient() function to see if the server is sending any commands. The server will send 'pause', 'continue', 'end', or 'logs', forcing the program to pause activity, continue from a pause, exit the program entirely or send the created logs. These logs are created from a ConjureLog class, containing a URL, time, user, type, success, message, ip, and destination directory for the download execution.

The full URL with the "https://" at the beginning must be included to run successfully. Otherwise, the program will log an Invalid URL response back to Conjure. To execute the downloaded file on the VM, the full path from home (/home/<user>) is needed that leads to the file but cannot have the ".exe" file extension included in the path. Use *powershell.integratedConsole.showOnStartup* as False so the command prompt does not pop up in the User VM

5.3.2. Download and Execute

The server can also send the command ‘download’, with two arguments to specify the desired behaviour with the destination directory to download to, and the web link of the desired file to download. If the download command is received, the program will then call the DownloadFromWeb() function, which navigates to the specified URL and downloads the file within the path to the desired destination. The download function attempts several different commands, including HTTP clients, web clients, DownloadFile(), WriteAllBytes(), etc, all while catching any errors and recording them within the logs.

Once successful, the OpenDownload() function will be called to open the file from the destination using a powershell command within the Windows VMs. We used the *System.Diagnostics.Process.Start(DESTINATION_PATH)* command to download to a specific location.

5.3.2. PowerShell Commands

Conjure can execute Powershell files that exist on the client VM. Conjure cannot execute individual PowerShell commands from the server side to the client, but custom PowerShell scripts can be uploaded to VMs via the download and execute features of Conjure.

5.3.4 Logs

- Creating *Conjure Log*
    - These logs communicate the success/error handled of automated and Red Team-requested processes on the VMs.
    - These logs come from automated browsing, downloading files, and executing those files.
- Set the ip address using *Dns.GetHostEntry*
    - Needs to be first ip in address list of the ip addresses retrieved from checking those associated with the host name
- Set the username with *Environment.UserName*

- To create a new log, the syntax should be *ConjureLog NAME = new ConjureLog();*
- Log Components
    - **Url** – Intended only for WebNav and Download
    - **Dest** – Intended only for Download and OpenDownload
    - **Type** – Specifies which functionality the log is for (web nav, download, run)
    - **Time** – Date and time the function took place
    - **Success** – "Success"/"Failure"
    - **User** – The user logged onto the machine
    - **Message** – Should be blank in successful, but will hold an error message if a failure is logged

Failed/Replaced Functionality:

1. **Functions to acquire links from webpage**
    - Originally statically acquired links and stored the CSS_SELECTOR of chosen elements from a webpage in a yaml file.
    - Had to be done manually, provided less flexibility, not adaptable to larger systems and adding in new websites to scan
    - Instead moved to acquiring the web elements dynamically and filtering all links to avoid webdriver errors
2. **Functions to aid in clicking on links**
    - Using element.click() was often unsuccessful and raised many "not clickable" and "not viewable" Selenium errors
    - We tried using the 'WebDriverWait' and 'Expected Conditions' libraries to wait for the web element to fully load and become visible but they didn't provide the desired functionality
    - Eventually used javascript commands within the webdriver to execute
    - "arguments[0].click()"
        - Helped a lot with clicking errors
3. **Functions to aid in automatic scrolling through the webpage**
    - Used the Action Chain (AC) library to create an instance of a scroller, used the coordinates of the chosen element to scroll to

        - scroller = AC(driver)
        - scroller.move_to_element(pathLink).perform()
        - action = ActionChains(driver)
        - action.move_by_offset(x,y).click_and_hold().move_to_element(target) .release().perform()
    - Didn't scroll smoothly to element, would bring back the clickable errors
    - Tried using scrollers in the AC library and scrolling with the scrollTo() function at specific points
        - Would instantly move to the element instead of smoothly scrolling like a user

4. **Attempts to execute downloaded files within VMs**
    - We used the Process.Start() function within the System imported library in C# to execute our downloaded files, such as Sliver implants.
    - We tried the statement Process.Start("cmd.exe", <downloaded_filename>), but the cmd.exe function was not executing the downloaded files properly.
    - We then tried to use the powershell.exe command instead of cmd.exe, and were able to execute the downloaded files and receive a callback on Sliver, as we were testing with a Sliver implant.
5. **Functions for sending/receiving commands to the server via socket connection**
    - We tried to send lists of logs all at once, but the server was not able to divide the strings effectively, so instead send each string one by one, waiting to receive a "next" command from the server to move on to the next string.
    - We then receive an "ack" from the server to move on to the next Log until all the logs have been sent, in which case the VM code will send the message "done", signalling that the VM has finished sending messages.

6. Future Work

- Create a functioning C# DLL instead of having to use a DLL proxy to execute the full code
- Use cmd.exe instead of powershell.exe within the Process.Start() function when executing downloaded files
- Find a more adaptable, versatile replacement to socket connections for the server and VMs to communicate
- Hide the terminal once code is executed (have an untested line of code within the VM program)
- Update EdgeOptions to remove automated software flag in the edge browser (already removed for chrome)

Appendix: Week By Week Breakdown

Week 2:

Everyone split off after a week of research to complete tasks we set for Conjure to accomplish.

- Dillon, Quillen, and Veronica worked on automating web interactions using Python and the WebDriver library.
- Elliot and Alan worked on the server-client interactions needed for Conjure to communicate to the VMs.
- Nolen worked on automating PowerShell commands as well as executing files found on the VMs.

This week, we demonstrated that both the VMs and the server function individually and established a connection between them to begin interactions between the two. With the server and VMs running, we also developed code to inspect html elements from a website and travel to other links found on the site. In addition, Nolen did not present his code to the mentors, but assured its successful execution.

As a whole, we accomplished the tasks we set out for ourselves this week and are ahead of the prescribed schedule for this project.

Week 3:

Building upon the previous week's accomplishments, we emphasised optimising and fixing bugs found in our code as well as documenting our work in Gitlab. This week provided our team with reduced hours to work on the project, but we have successfully pushed our current code to our shared Git repository and begun the documentation process.

- Nolen set up the Git repository while modifying his code to travel to and from directories to execute files from certain directories.
- Quillen, and Dillon improved their web browsing tool to reduce falling into a loop of following the same links forever.
- Veronica developed searching through sitemaps to ident
- verify downloadable links, navigating to each, and downloading them
- Dillon also worked automating the ssh connecting process.

- Alan worked on creating logs for the interactions that occur from the VMs whether it's web browsing, running commands, etc.
- Elliot started working on sending files, commands, and credentials over a SFTP connection.
- Q changed the web element scanning from static to dynamic

Week 4:

- Nolen added a .gitignore file to clean up while files were sent into the repository while adding functionality to his script to unzip compressed folders than execute the files within them
- Q improved web-navigation program using javascript commands within the webdriver for clicking, scrolling
    - Helped reduce "element not clickable links" errors
- Q implemented functions to check if a web element has a certain size and is visible for the webdriver. Filtered all links for only the clickable elements
    - Helped reduce "element has no size" and "element not clickable links" errors
- Q changed the options of the webdriver to get rid of the "browser being controlled by automated test software" flag
- Q added in more random wait times throughout the program to aid in the human-like appearance of the code
- Nolen experimented with sending emails automatically from victim's VM but requires user logged into machine's default mail

Week 5:

- Nolen started experimenting with executing files downloaded/created on a certain date and testing the execution of those files
- Q implemented scrolling functionality for the automatic-web navigation
    - Improves the human-like appearance
- Q added in functions to check if a link has been clicked before
    - Reduces the likelihood of repeatedly clicking the same link
    - Improves human-like functionality

- Veronica and Q combined random web-navigation with download functionality so we can now scan for all clickable links and all downloadable files within the website in the same program

Week 6:

- Veronica created functions for downloading and running
- Q improved user simulation of web navigation
- Veronica & Q tested downloading a user-specified path using a sliver-implant on an https local server
- Q, Veronica, Alan worked on uploading the web-nav and download files onto the server
- Alan integrated the download and web-nav into one program, action to take can be specified by user
- Nolen deconflicted and merged git branches to keep repository up to date and achieved file execution from files in the Downloads folder
- Elliot set up the first working DLL sideloading code, can run any code we want through the the nslookup DLL

Week 7:

- Q converted random webNav code from python to C# so we can use DLL sideloading
    - Removes dependency, improves performance
    - Had to alter the edge options to “AddExcludedArgument” to get rid of the automated test software flag
    - Had to create a JavaScript executer from the web driver for the clicking and scrolling functions
- Veronica converted the download code from python to C# for the same purpose as mentioned above
- Veronica integrated the webNav and download functions into one program that calls them respectively from main
- Nolen debugged powershell, assisted in translating Python scripts to C#, and made diagrams to be used in future research presentations and for internal team use.

Week 8:

- Nolen implemented error/success logs into the web browsing and downloading functionality of Conjure
- Alan and Q implemented Socket connectivity between C2 Conjure server and the user-simulation code within the VMs.
    - Able to send/receive commands (pause, continue, end) from server and client can acknowledge
    - Can send logs over the Socket to the server which the server stores in a file

Week 9:

- We fixed any bugs of connecting the VMs to the Conjure server and receiving commands
- Can execute downloaded files automatically from the VMs
- Organised documentation
- Download from web, must pass FULL url (https://<link>)