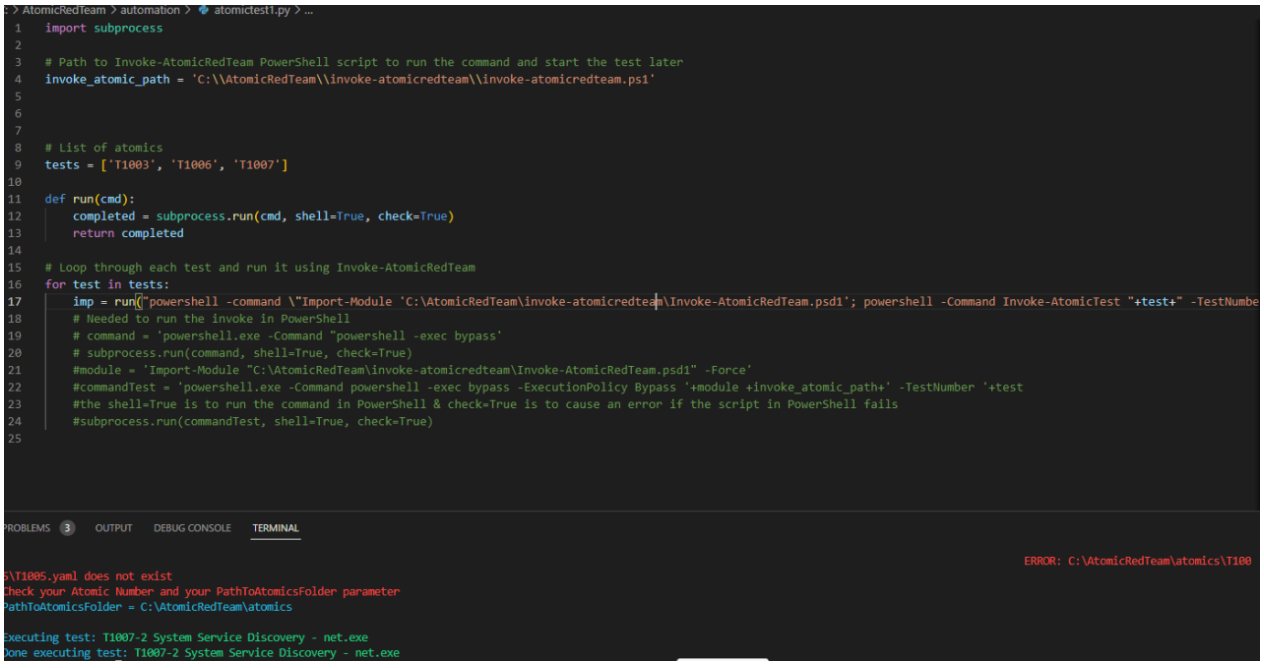
Script Description

My essential part of the project was to investigate and attempt to create an automated script with the Atomic Red Team library test. My first task was investigating the viability of automating the testing process, given that the tool's functionality and test execution relies on PowerShell. diligent research and exploration, I made a significant discovery: the existence of dedicated YAML files within the tool, specifically designed for automation purposes. I decided to use Python, a language I am well-versed in due to my academic studies and side projects, based on my expertise with automation. To begin the process, I created a short but functional script capable of running simple tests. This first step was to confirm the veracity of my study on the ART library's automation capabilities. One of the initial steps was to establish the default location for the atomics, which served as the repository from where the script could locate and execute these methods. Following that, I needed to indicate which atomics should be performed. However, there was a small bottleneck when it came to communication between Python and PowerShell. I conducted a significant study and sought advice from my mentors, which led me to discover the utility of the "run()" and "subprocess.run()" methods. These functions allowed me to run PowerShell commands straight from Python. With this newfound information, I successfully created the "Invoke-Atomic" command, which allows me to select the required method from a list of possible atomics. Despite initial challenges and a slow start, I persisted and completed my original set of goals. To guarantee constant communication and to keep my mentors informed, I swiftly updated them on my progress. Furthermore, we organized a meeting during which I demonstrated my accomplishments and engaged in productive talks about the next phases of my project. This collaborative session gave significant insights and assistance, laying the groundwork for the next phases of my work. After this achievement, the next thing that I had to focus on attempting and accomplish was to create a reporting system which would create text files containing information decoded as a report. This way, after a certain test is done, whoever uses it will be able to see the result and make changes if needed.



First successful automated testing

I wrote a script that can generate separate files to hold the contents of various TXT files. During this procedure, however, I ran into a problem with adding proper report information to the TXT files. Initially, the recorded data represented execution details rather than the anticipated end outcomes. To get around this problem, I invented a method that entailed overwriting the files. By introducing this feature, I ensured that the computer's storage was not filled with unnecessary and redundant data. This method not only solved the problem at hand but also improved the script's efficiency and resource management.

```
#command = 'powershell.exe -Command "powershell -exec bypass'
#Note: Before entering here it does not accept the bypass code
#Add powershell -command powershell -exec bypass; after the run(" will only execute the b
#Loop through each test and run it using Invoke-AtomicRedTeam
for test in tests:
    imp = run("powershell -command \"Import-Module 'C:\AtomicRedTeam\invoke-atomicredteam


# The Folder + .txt files
output_dir = os.path.join(os.getcwd(), "test_results")
if not os.path.exists(output_dir):
    os.makedirs(output_dir)



for test in tests:
    #result = art.run_test(test)
    filename = f"{test}.txt"
    filepath = os.path.join(output_dir, filename)
    with open(filepath, "w") as f:
        f.write(test)


#with open('/Users/User/Desktop/results.txt', 'a') as f:
#    for line in imp:
#        f.writelines(imp)
```

Report system problem

```
# Test Idea to devide both txt files in to two files - One is regarding the data that the test display
# which is the tests after they go through the prerequisites Check
# The txt files that are generated there are - 1: After execution ; 2: After prerequisites are chekced

    test_output_dir = os.path.join(os.getcwd(), "Test_Results")
    os.makedirs(test_output_dir, exist_ok=True)
    test_output_path = os.path.join(test_output_dir, f"{attack}_Test_Output.txt")
    with open(test_output_path, "w") as f:
        f.write(str(test_result.stdout.decode()))
```

Fixed report system with its own file

Initially, the files I received from the scans contained information about the scan's success or failure rather than the actual scan content. I used various dictionaries labelled "Successful Scan," "Unsuccessful Scan," and "Prerequisites" to extract the relevant scan details. These dictionaries were used to accurately identify the text material needed for insertion in the TXT document, which would provide the scan's full results. I ensured that the relevant information from the scans was correctly retrieved and appropriately included in the final document by structuring the data in this manner, offering a clear and simple representation of the scan findings. Following a productive meeting with my mentors, I gave them a thorough review of the project's prerequisites, including installation and setup processes. It became clear that certain atomics required the installation of extra specifications to work effectively. In the absence of these prerequisites, the preferred solution would simply indicate the need for extra material and display a notice outlining the necessary installations. This method allowed easy communication and effective troubleshooting, allowing for a rapid fix of any possible difficulties caused by missing requirements. I created a specific dictionary to ease the scanning process and assure the smooth execution of atomics with additional criteria. This dictionary is used to filter specific elements inside the outcome text, thereby identifying the atomics that require additional steps for initiation. In addition, I combined this dictionary with a function that automatically installs the appropriate prerequisites by utilizing the "-GetPrerequisites" command, which is performed easily from the Python script to PowerShell. This automated technique is consistent with my aim of establishing an optimized scanning process that does not consist of bottlenecks and failed atomic tests.

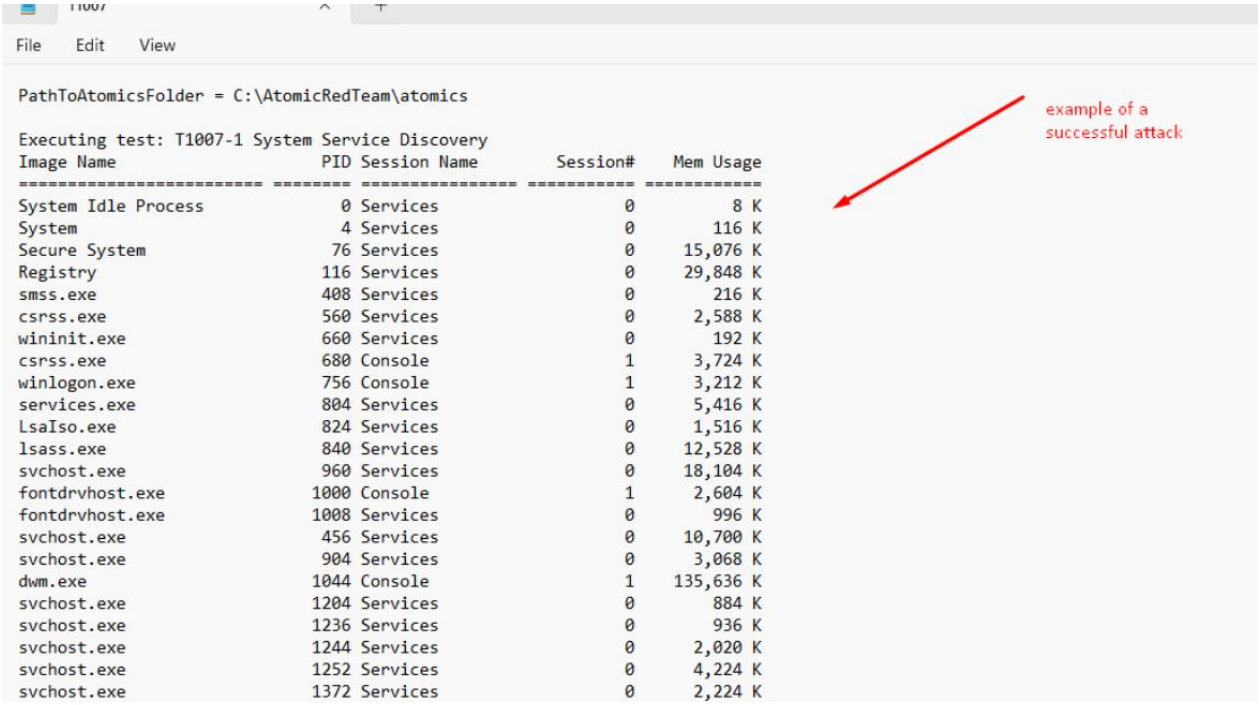
```
# Check if any prereqs are required
prerequisites_present = True
output_str = prereq_result.stdout.decode()
for word in prereq_dict:
    if word.lower() in output_str.lower():
        prereq_dict[word] += 1
        prerequisites_present = False
        break

# Output appropriate message based on prereqs
if prerequisites_present:
    f.write(f'<p>{attack} - Prerequisites required</p>\n')
    # Downloading the prereqs
    result = run("powershell -command \"Import-Module 'C:\AtomicRedTeam\invoke-atomicredteam\Invoke-AtomicRedTeam.psd1'; powershell -command Invoke-A
else:
    f.write(f'<p>{attack} - Success with prerequisites</p>\n')
```

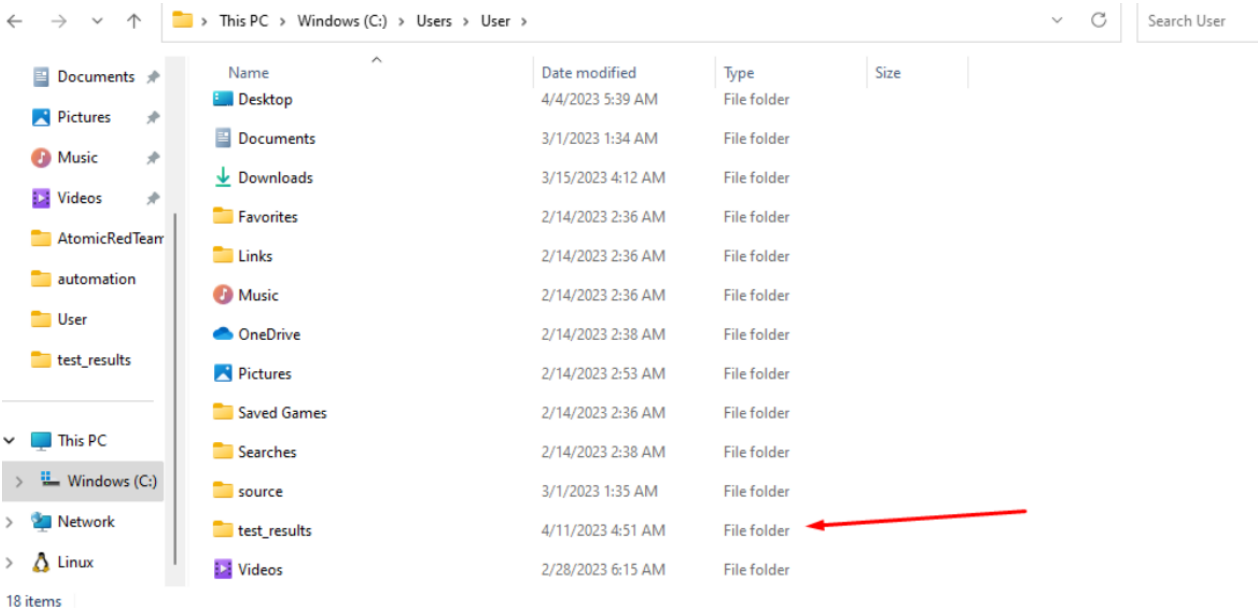
Prerequisites check and prerequisites download

I improved the process even more to improve clarity for my mentors by adding a new feature. This function makes it easier to create distinct TXT documents for each unique atomic, integrating the content obtained from successful scans. This thorough documentation guarantees that the scan data are thoroughly recorded and arranged, allowing for thorough analysis and evaluation. By automating these necessary stages, I have not only reduced the workflow but also improved the scanning process's overall efficiency and accuracy. Recognizing the need for an improved summary for the performed atomic methods, my company mentors asked me to add a new feature into the automated process. Following the successful creation of the folders holding TXT files, I started with creating an HTML summary file. This complete summary would offer a consolidated picture of the performed atomic methods, categorizing them based on success, failure, and whether or not prerequisites were necessary. I wanted to present a visually appealing and user-friendly summary of the scan findings by incorporating this functionality. The HTML summary file would be a handy reference for stakeholders, allowing them to immediately comprehend the

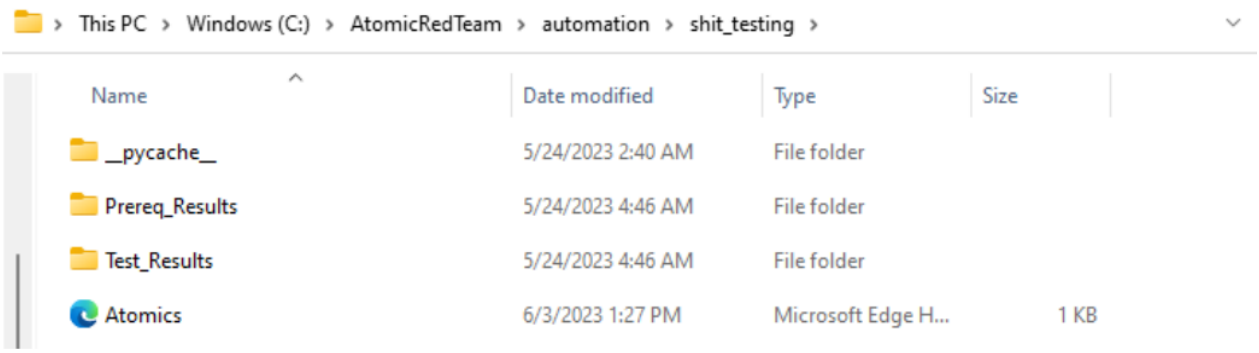
results of the scan, identify any possible difficulties, and analyze the impact of requirements on atomic execution. This extra feature not only highlights the automation script's adaptability and extensibility, but it also exhibits a proactive approach to meeting the increasing needs of my work environment.



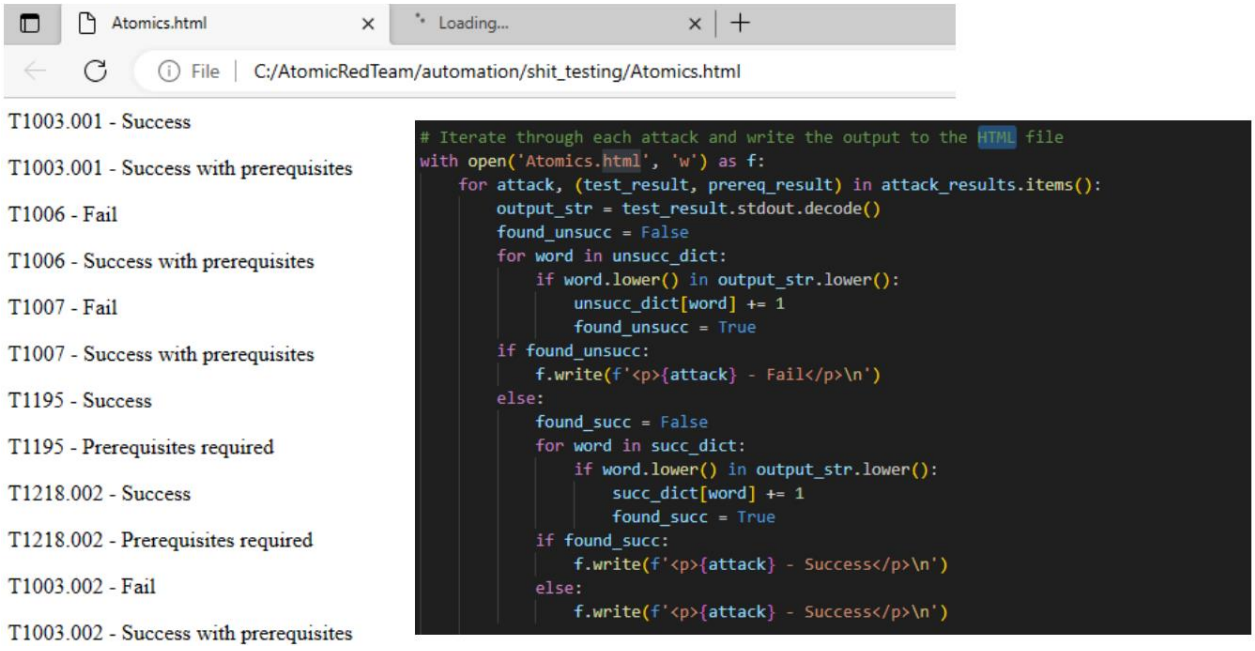
Example of one successful attack



Initial folder before the final update with prerequisites folders



Separate folder where the script is located with Prerequisites folder; Test results and HTML file with summery of the tests



HTML file with the summery from the executed atomics & the code that shows the iteration through each attack.

Further to my development, after another meeting with my mentors regarding progress, I came up with an idea that was from my research. Because the library is constantly updated, I came up with the idea that before execution of the automated script and initiating the automatic scan, I wanted to reinstall the Atomic Red Team library. The reason is that by installing it again this way, the library will be with the updated content and new extra features.

```
# Download the ART zip file
url = "https://github.com/redcanaryco/atomic-red-team/archive/refs/heads/master.zip"
response = requests.get(url)

# Creating a folder to extract the zip
folderPath = "C:\\AtomicRedTeam"
os.makedirs(folderPath, exist_ok=True)

# Extract the contents from the zip file to the specified folder folder
zip_file_path = os.path.join(folderPath, "atomic-red-team.zip")
with open(zip_file_path, "wb") as f:
    f.write(response.content)

with zipfile.ZipFile(zip_file_path, "r") as zip_ref:
    zip_ref.extractall(folderPath)

# Rename the extraction folder
extracted_folder_path = os.path.join(folderPath, "atomic-red-team-master")
target_folder_path = os.path.join(folderPath, "atomic-red-team")
if os.path.exists(extracted_folder_path) and not os.path.exists(target_folder_path):
    os.rename(extracted_folder_path, target_folder_path)

# Delete the zip file
os.remove(zip_file_path)
```

Installation of the ART before the execution

I developed a method that gets the necessary files in a ZIP package from the Git repository. The ZIP package is then extracted, and the resulting folder is renamed to improve readability and organization. To maintain a streamlined procedure and avoid any program disturbances, the ZIP archive is then removed immediately, removing the chance of duplicated data or unwanted clutter. This method not only optimizes file management but also ensures that future phases in the automation process are executed cleanly and efficiently. I considerably improved the overall resilience and stability of the automated script by taking these changes, allowing for flawless and uninterrupted operations. There was an minor issue with the overwriting the folders which I fixed with the deleting the zip file function.