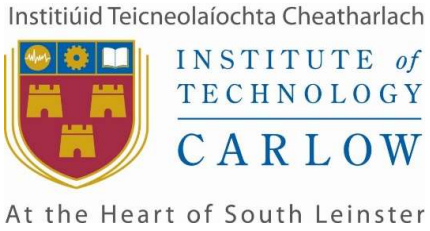

WINTRE: AN ADVERSARY EMULATION TOOL

Design Manual

Student: Martin Earls / C00227207
Supervisor: Richard Butler

1 Table of CONTENTS

2 INTRODUCTION

The application to be developed is a .NET C# desktop application for Windows operating systems. Users will be able to review and execute security tests, or emulate malicious user behaviour as defined in the MITRE ATT&CK framework. These techniques, coded as individual tests will help generate indicators of compromise in the form of logs created from system events or notable alerts from endpoint security products.

A key component in the design of the product, is to ensure the separation of security tests as separate, fully functioning executables. This means all techniques must be compiled in separate binaries that will then be called from the main application. This will be achieved by saving the source code of the associated techniques on disk with the main application which will then read and compile them at runtime.

The user will be able to selectively choose which tests they want to run or save a group of tests as a campaign. Tests can be executed one at a time or added to a queue system, the queue system will simply allow the user to collect and gather a preferred list of tests beforehand and then execute them.

This document outlines the design principles and choices made to facilitate the outlined deliverable, i.e., a user-friendly tool that can help run command line and WinAPI based security tests associated with the MITRE ATT&CK Framework.

3 PROJECT PLAN

The project plan outlined below covers all major milestones of the planned development phases. Development of the actual security tests, i.e., techniques is given priority over all other stages. Technique development comprises of "standard" command-line tests and "complex" WinAPI tests.

After core technique development has been completed, time is mainly allotted to quality assurance such as performance, unit and security testing.

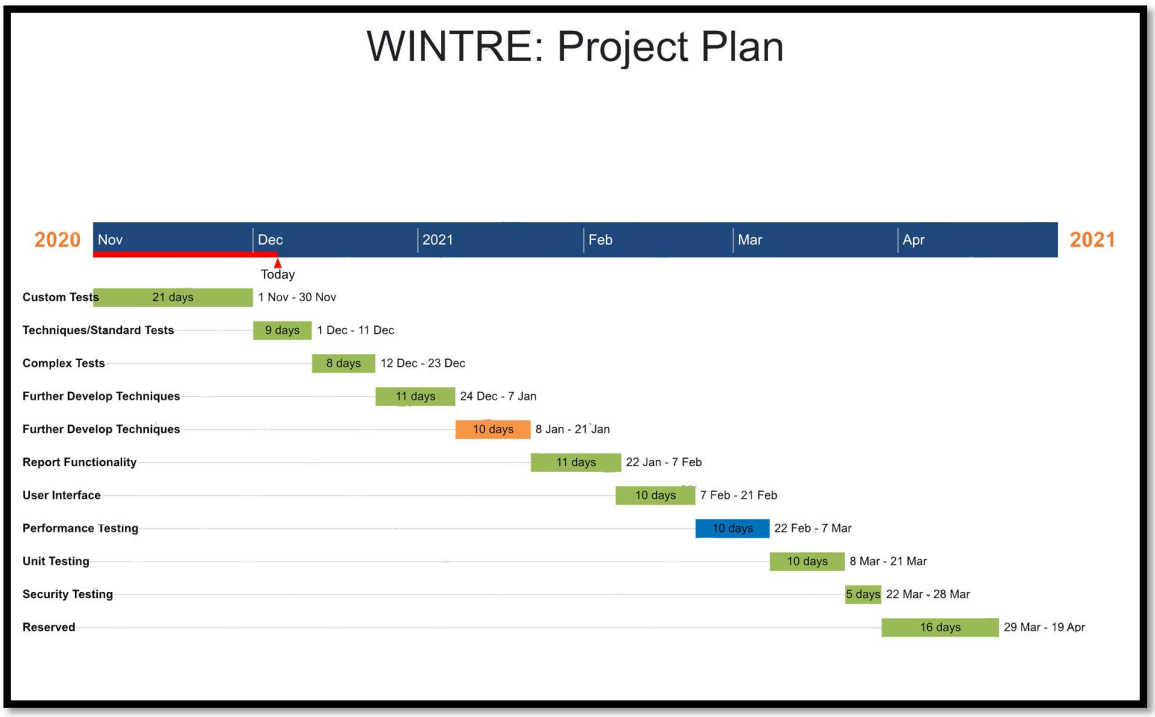

Figure 1. Project Plan

4 TECHNOLOGY STACK/REQUIREMENTS

4.1 C#

C# will be used as the main programming language as C# executables utilise the .NET framework, which is installed by default on all Windows 10 machines as of 2019 C# as a high-level language offers simplified ways of interacting with WinAPIs through the .NET framework that would've traditionally had to be accessed using a lower-level language such as C or Assembly. C# is object-oriented and type-safe, applications are highly scalable and easy to update and C# is packed with a rich plethora of libraries to enhance its interoperability. While not as fast as C or lower-level languages, compilation and execution speed are still sufficient for modern Windows applications. C# binaries do not require any 3rd party libraries to execute on modern Windows endpoints that have the .NET framework installed.

These reasons, while positive are also some of the main reasons threat actors have switched to using C#, to purposefully exploit its compatibility with modern Windows systems in order to develop malware. This matches our use case for the techniques needed to perform adversary simulation, many are essentially malware that will be running in a controlled and specific context. C# also reduces unnecessary, extra components that would be needed in a tool like this. Many similar applications require Linux based servers and have a much larger setup cost to an organisation. While a more complex infrastructure can benefit certain categories such as Command and Control for C2 testing, C# allows us to locally compile each individual test as its own executable. In more complex frameworks, this is not possible and the tester is forced to remove and update an agent each time they want to run a series of tests.

4.2 XAML

XAML syntax is self-explanatory and easy to modify, even after many UI elements are joined together. XAML UI design is directly integrated with Visual Studio along with C# and Windows Presentation Forms. UI can be transferred with ease. This is useful, if in the future an application was moving from a desktop to a web or mobile implementation, there are great cost savings for the developers both in time and resources needed to migrate the UI to a new platform.

4.3 POWERSHELL

PowerShell scripting can be utilised for practically every MITRE tactic or category of technique. PowerShell's intended usage is for automation and configuration management (Microsoft 2020). It provides a command-line shell to the end user from which they enter commands. This functionality is built upon the .NET Common Language Runtime. PowerShell is also extensible, allowing users to define cmdlets, this can be done directly using compiled code or scripts. PowerShell also has providers that provide direct access to important data stores. These data stores include the registry and file system, 2 very important and often targeted areas on a Windows computer. While intended for system administration, attackers may add malicious registry key entries or steal credentials from the registry.

C# binaries can also directly issue PowerShell cmdlets which will make testing offensive PowerShell more efficient. Frameworks such as PowerSploit written in PowerShell provide many modules for credential theft, process manipulation, discovery and data exfiltration. Red Canary's Threat Detection Report for 2020 shows that PowerShell is still a relevant technique for security testing that is being exploited despite many recent security controls developed for Windows by Microsoft.

4.4 WINDOWS COMMAND LINE SCRIPTING

Similar to PowerShell scripting, all supported versions of Windows rely on built-in Win32 console commands to execute instructions from a command line. One of the first things a threat actor may attempt when gaining access to a Windows system is to determine their integrity level and local privileges. The ability to easily retrieve file system and OS information is appealing to attackers, as this will likely aid in further compromise of the local or neighbouring systems.

4.5 C

C would be used in a limited capacity for evaluating a handful of techniques, where coding them in C may be more beneficial than C# as C has greater functionality to interact with the Windows OS at a lower-level. Payloads would be compiled using the Microsoft (R) C/C++ Optimizing Compiler.

4.6 C++

Similarly, to the C usage in this project, C++ would be used for evaluating select techniques where the C# equivalent may not be feasible or the C/C++ version may utilise different APIs more efficiently and therefore would be considered a sub technique and require separate detection analytics. Payloads would be compiled using the Microsoft (R) C/C++ Optimizing Compiler.

4.7 X86 ASSEMBLY

The primary usage of this assembly code would be to use WinAPIs to write shellcode in order to be cached and executed in a memory buffer for techniques such as process injection. This would be sufficient for demonstration proof of concepts in code execution techniques such as process injection. The shellcode will be included with the software and loaded or compiled at runtime depending on the technique.

4.8 JAVASCRIPT

JavaScript is a highly popular technique used to achieve code execution. As noted in the ATT&CK framework, at least 8 Advanced Persistent Threat (APT) groups are known to use JavaScript often to act as drive-by downloaders. JavaScript will be necessarily not for any architecture aspects but for related techniques that utilise it for code execution.

4.9 VISUAL BASIC FOR APPLICATIONS

Similarly, to JavaScript, Visual Basic for Applications which is used in Microsoft Office documents is a highly popular method of compromising users via phishing and is necessary to test for code execution.

4.10 RUNTIME COMPILATION

Note that all techniques are to be compiled at runtime and ran from the main program calling the selected EXE file based on the technique chosen. These may be C#, C or C++ source code files that must be kept external to the main project, dynamically loaded at runtime and compiled/executed based on user input. Pseudo code to accomplish this is included in a later section of this document.

4.11 JSON

JSON will be used to serialize technique-specific information and save it on disk, to be loaded when a user selects a technique in order to help them understand it and how to run it.
The Json.NET library has been chosen to fulfil this requirement.

5 JSON SCHEMA ENTITY RELATIONSHIP DIAGRAMS

WINTRE will not utilise a traditional database system but will instead utilise JSON to store metadata related to tests that can be loaded with great efficiency at runtime. This metadata includes whether or not a test requires elevated privileges, arguments a test may need, the name, description and ID of the test. Test metadata will come pre-packaged with WINTRE allowing users to load and define new techniques, which based on their input will generate new JSON files storing the data which can be loaded immediately to run their new technique.

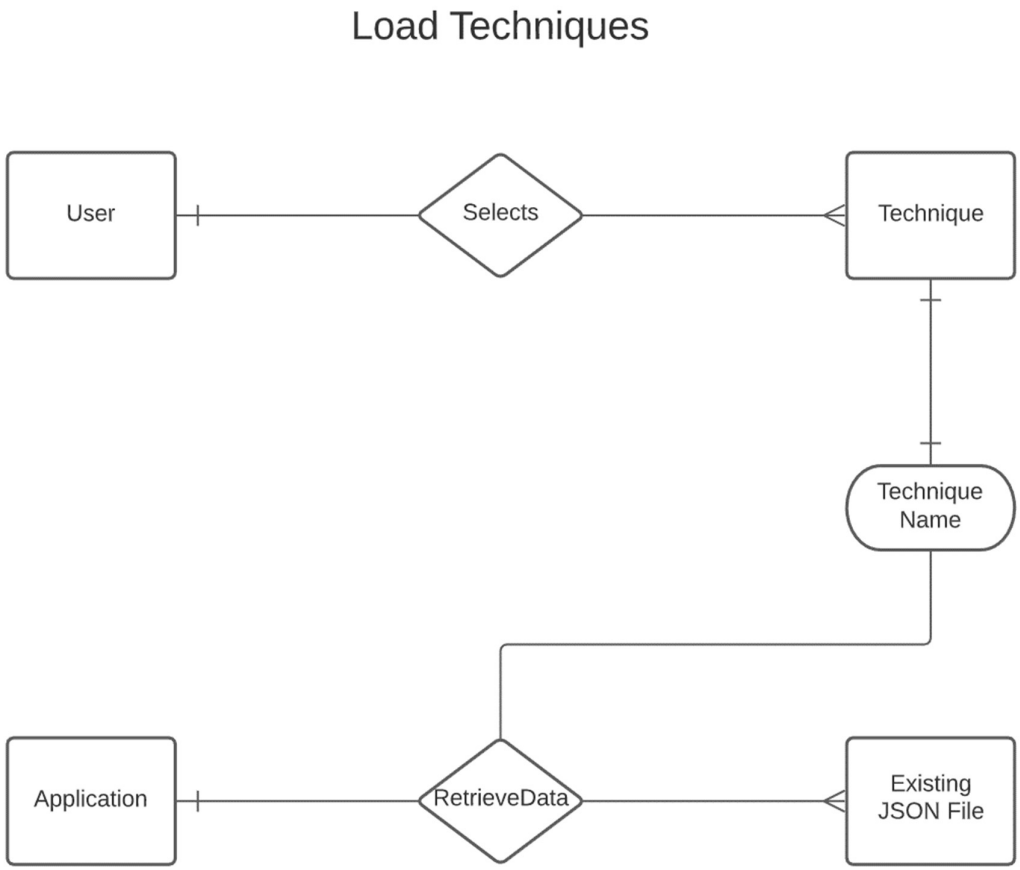

Figure 2. Loading Technique Information

When creating a technique, a user must enter several pieces of data to ensure another user running their test has sufficient information about what technique is being used, how it accomplishes this and any other necessary information.

A user must provide the following items to successfully create a custom technique:

1. Name
2. ID
3. Required privileges
4. Tactic
5. Commands
6. Description

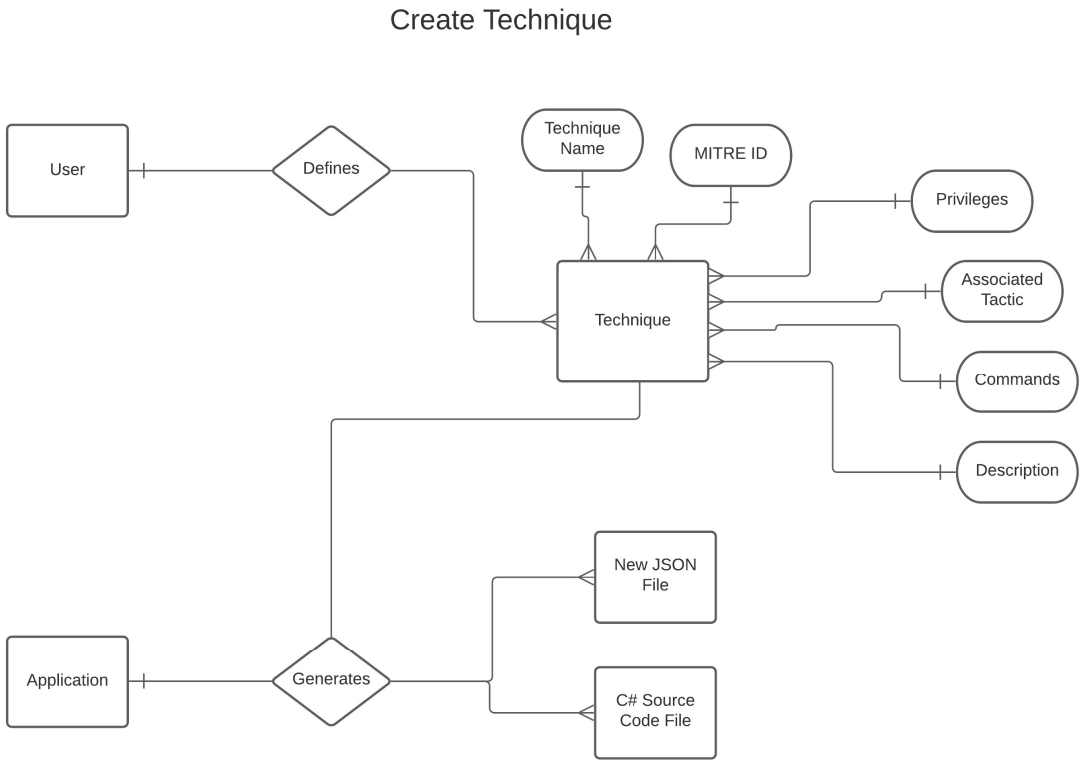

Figure 3. User Defining New Technique

5.1 JSON COMPONENTS

Each JSON file will contain information deemed necessary to run a test. Whether loading an existing technique or creating one, it is important from the application and user's perspective that they have sufficient information to know what the technique is doing. Strings can be serialized and saved easily in JSON format. The following JSON structures have been chosen in order to storm technique information.

Simple Test JSON Example:


```
{
  "name": "Process Discovery",
  "ID": "T1057",
  "elevated": "No",
  "tactic": "Discovery",
  "template": "CMD",
  "commands": "tasklist /v",
  "desc": "Adversaries may attempt to get information about running processes on a system. Information obtained could be used to gain an understanding of common software/applications running on systems within the network.",
  "hasArgs": "false",
  "arg1": "N/A",
  "arg2": "N/A",
  "arg3": "N/A"
}
```


WinAPI Test JSON Example:


```
{
  "name": "C# Reverse Shell",
  "ID": "T999",
  "elevated": "No",
  "tactic": "Command and Control",
  "template": "CMD",
  "commands": "N/A",
  "desc": "Establish a TCP reverse shell connection using streams to a remote server. Use Netcat to setup a listener and enter the relevant IP and Port as arguments.",
  "hasArgs": "true",
  "arg1": "IP Address",
  "arg2": "Port",
  "arg3": "N/A"
}
```


- Name: The name of the technique being tested.
- ID: The MITRE ATT&CK ID of the associated technique.
- Elevated: Whether or not the technique requires elevated privileges.
- Tactic: The associated category or "tactic" of the technique.
- Template: Whether or not the technique uses cmd.exe or powershell.exe to run.
- Commands: If applicable, what shell commands are ran to achieve this.
- Desc: Description of the technique, what it does and how.
- HasArgs: If a technique requires supplemental arguments.
- Arg1: Argument one (if applicable).
- Arg2: Argument two (if applicable).
- Arg3: Argument three (if applicable).

6 CLASS DIAGRAM

6.1 MAIN APPLICATION

Note that technique classes are external to the main application to make use of runtime compilation in order to avoid anti-virus detections from affecting the main program. Each technique will require its own class.

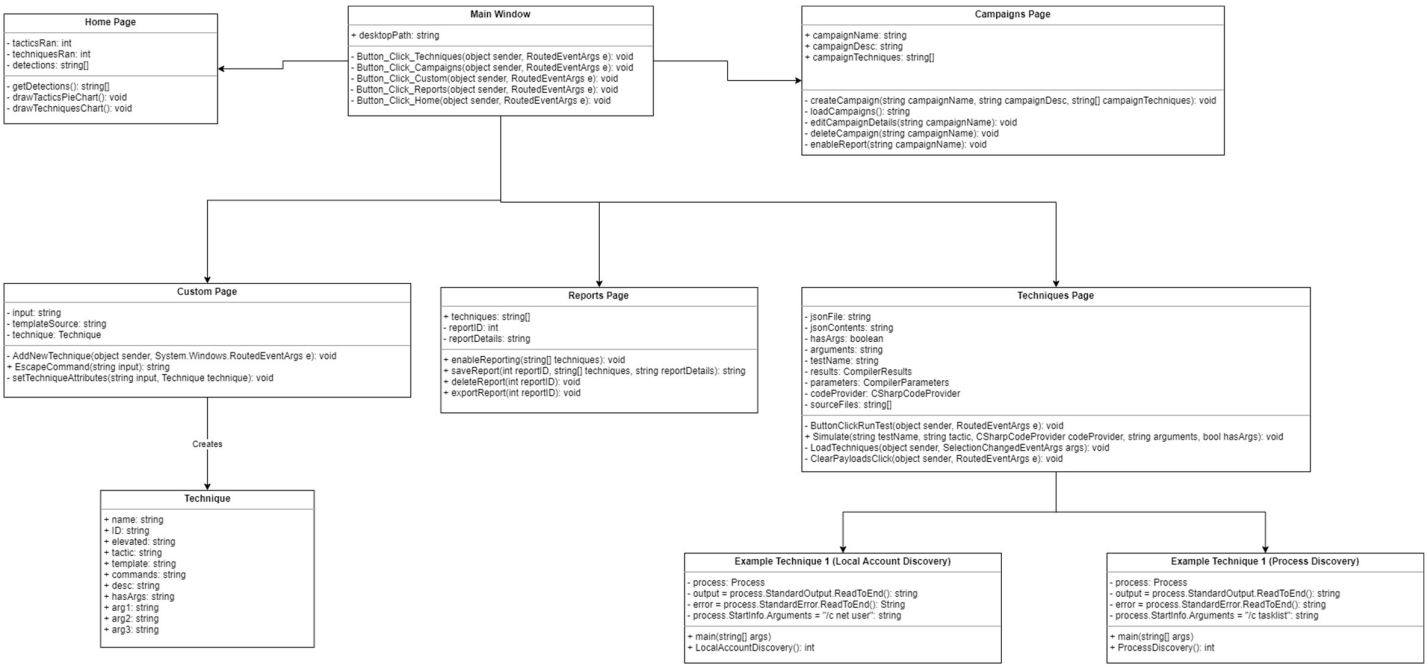

Figure 4. Overview of main classes

6.2 TECHNIQUE SOURCE CLASS EXAMPLES

6.2.1 Local Account Discovery

This example is based off a simple command-line technique where an attacker will use built in commands to enumerate local accounts they may want to target and pivot to.

```
using System;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Text;

namespace WINTRE
{
   class Program
   {
      static void Main(string[] args)
      {
         LocalAccountDiscovery();
      }
      static int LocalAccountDiscovery()
      {
         Process process = new Process();
         process.StartInfo.FileName = "cmd";
         process.StartInfo.Arguments = "/c net user"; //Simple command-line technique
         process.StartInfo.UseShellExecute = false;
         process.StartInfo.RedirectStandardOutput = true;
         process.StartInfo.RedirectStandardError = true;
         process.Start();
         string output = process.StandardOutput.ReadToEnd();
         Console.WriteLine(output);
         string err = process.StandardError.ReadToEnd();
         Console.WriteLine(err);
         process.WaitForExit();
         return 0;
      }
   }
}
```

6.2.2 Process Discovery

This is another example of a simple command-line based technique, where an attacker will enumerate processes on a system. Process enumeration can be key to achieving compromise as it will let the attacker know any potentially vulnerable programs, anti-virus and local user applications that are running.

```
using System;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Text;

namespace WINTRE
{
   class Program
   {
      static void Main(string[] args)
      {
         ProcessDiscovery();
      }
      static int ProcessDiscovery()
      {
         Process process = new Process();
         process.StartInfo.FileName = "CMD";
         process.StartInfo.Arguments = "/c tasklist"; //Simple command-line technique
         process.StartInfo.UseShellExecute = false;
         process.StartInfo.RedirectStandardOutput = true;
         process.StartInfo.RedirectStandardError = true;
         process.Start();
         string output = process.StandardOutput.ReadToEnd();
         Console.WriteLine(output);
         string err = process.StandardError.ReadToEnd();
         Console.WriteLine(err);
         process.WaitForExit();
         return 0;
      }
   }
}
```

7 SEQUENCE DIAGRAM

7.1 HOME PAGE SEQUENCE DIAGRAM

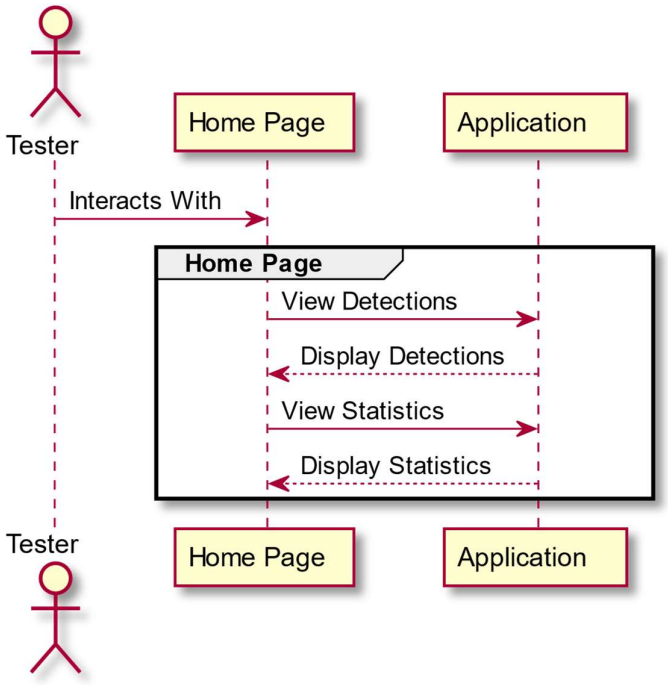

Figure 5. Home Page Interactions

7.2 TECHNIQUES PAGE SEQUENCE DIAGRAM

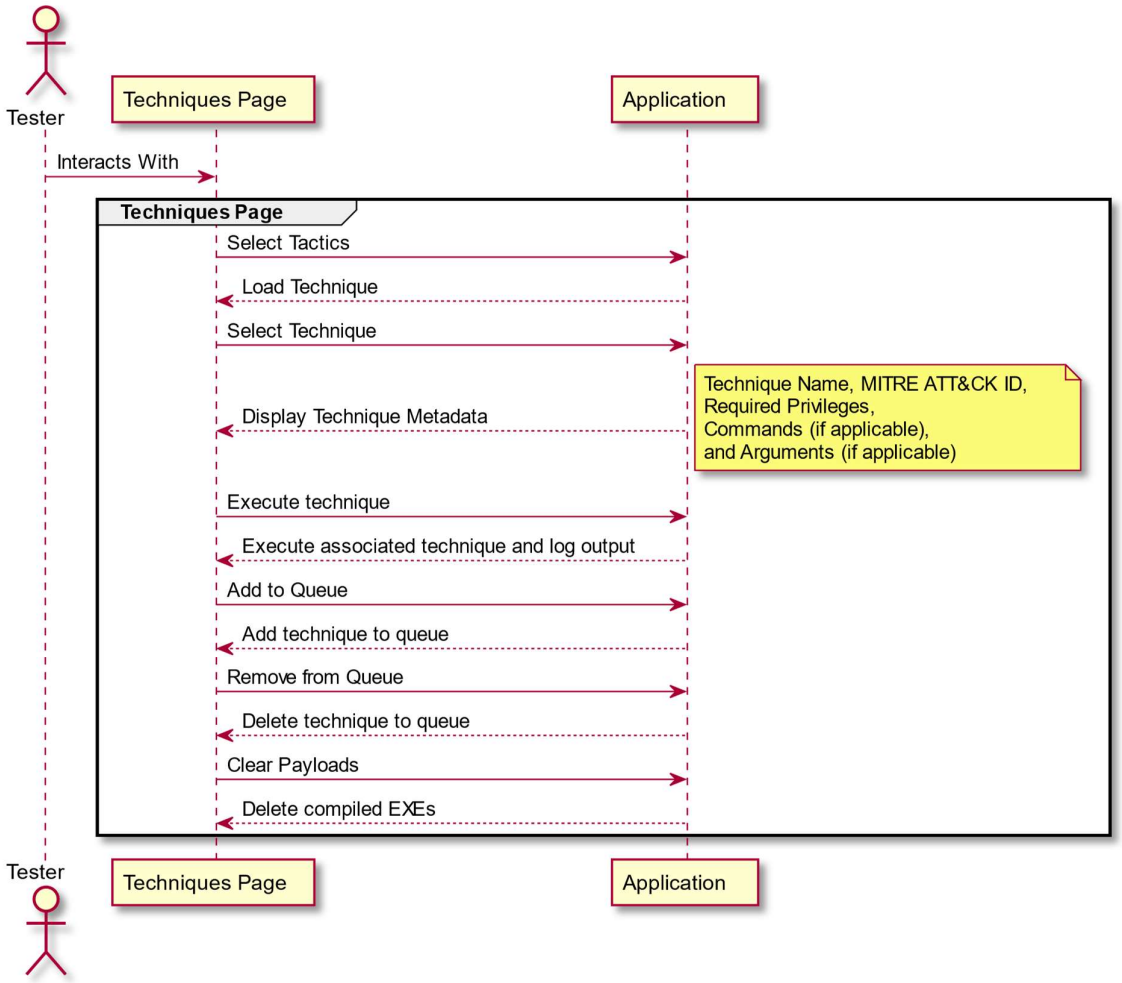

Figure 6. Techniques Page Interactions

7.3 CAMPAIGNS PAGE SEQUENCE DIAGRAM

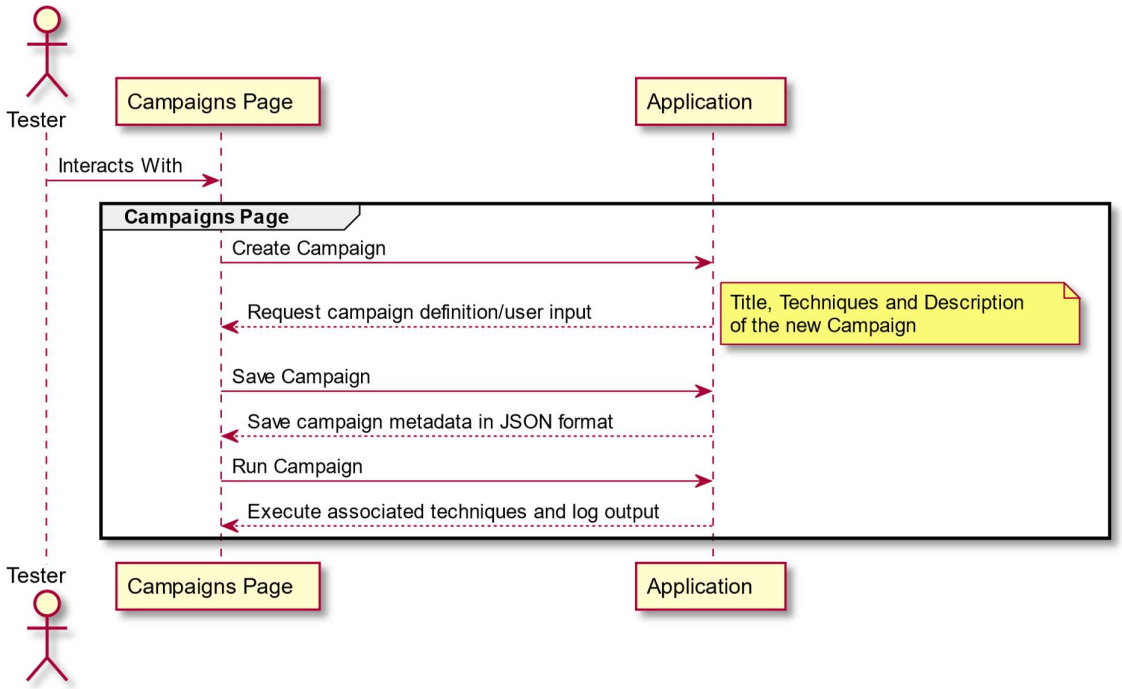

Figure 7. Campaigns Page Interactions

7.4 CUSTOM PAGE SEQUENCE DIAGRAM

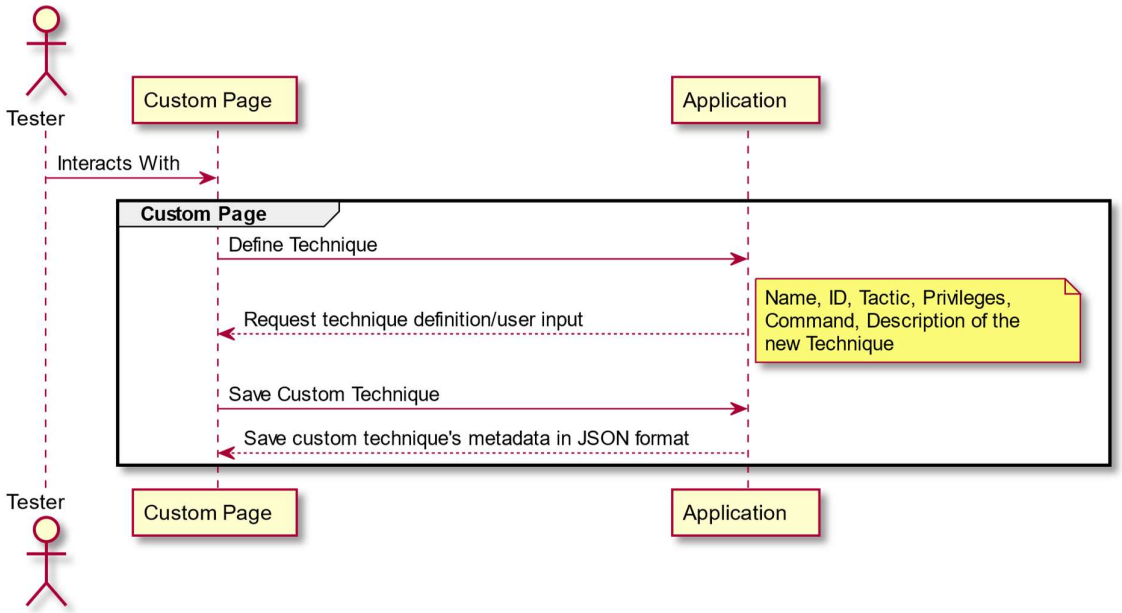

Figure 8. Custom Page Interactions

7.5 Reports Page Sequence Diagram

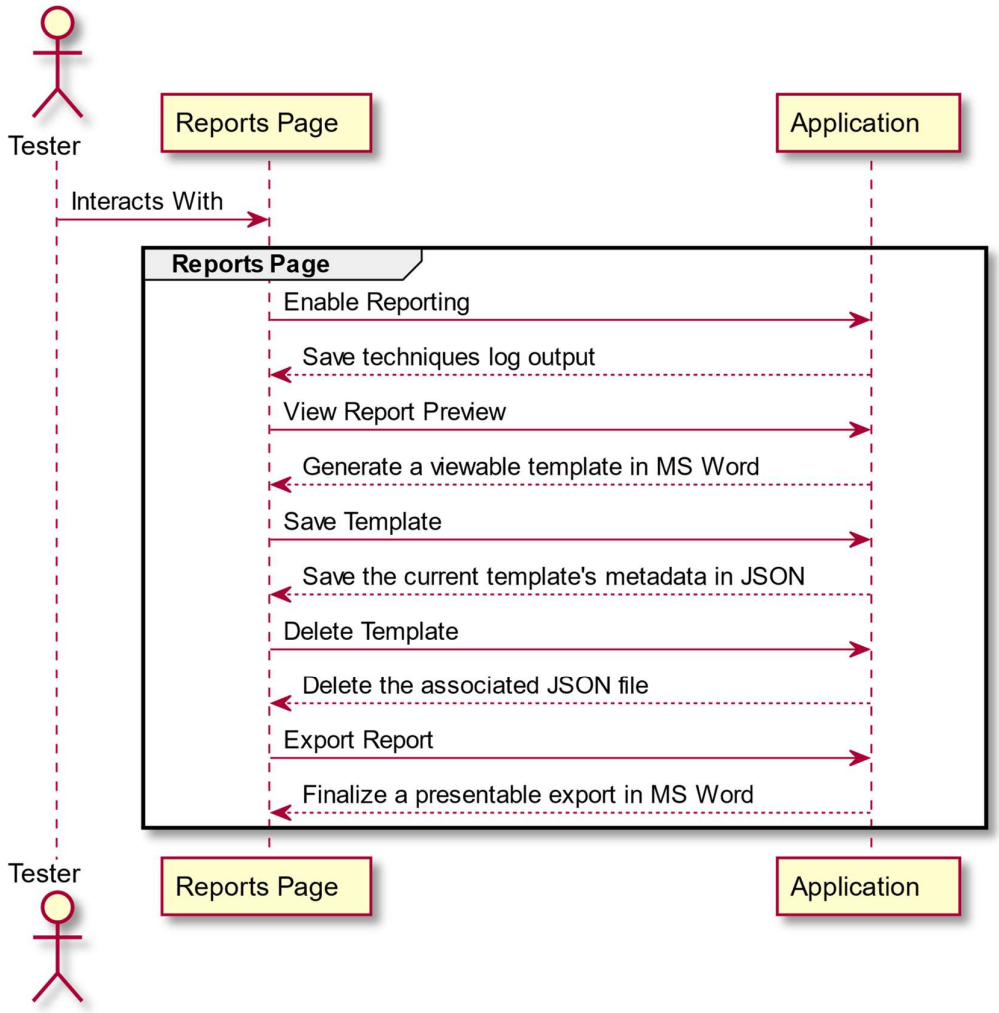

Figure 9. Reports Page Interactions

8 USE CASE DIAGRAM

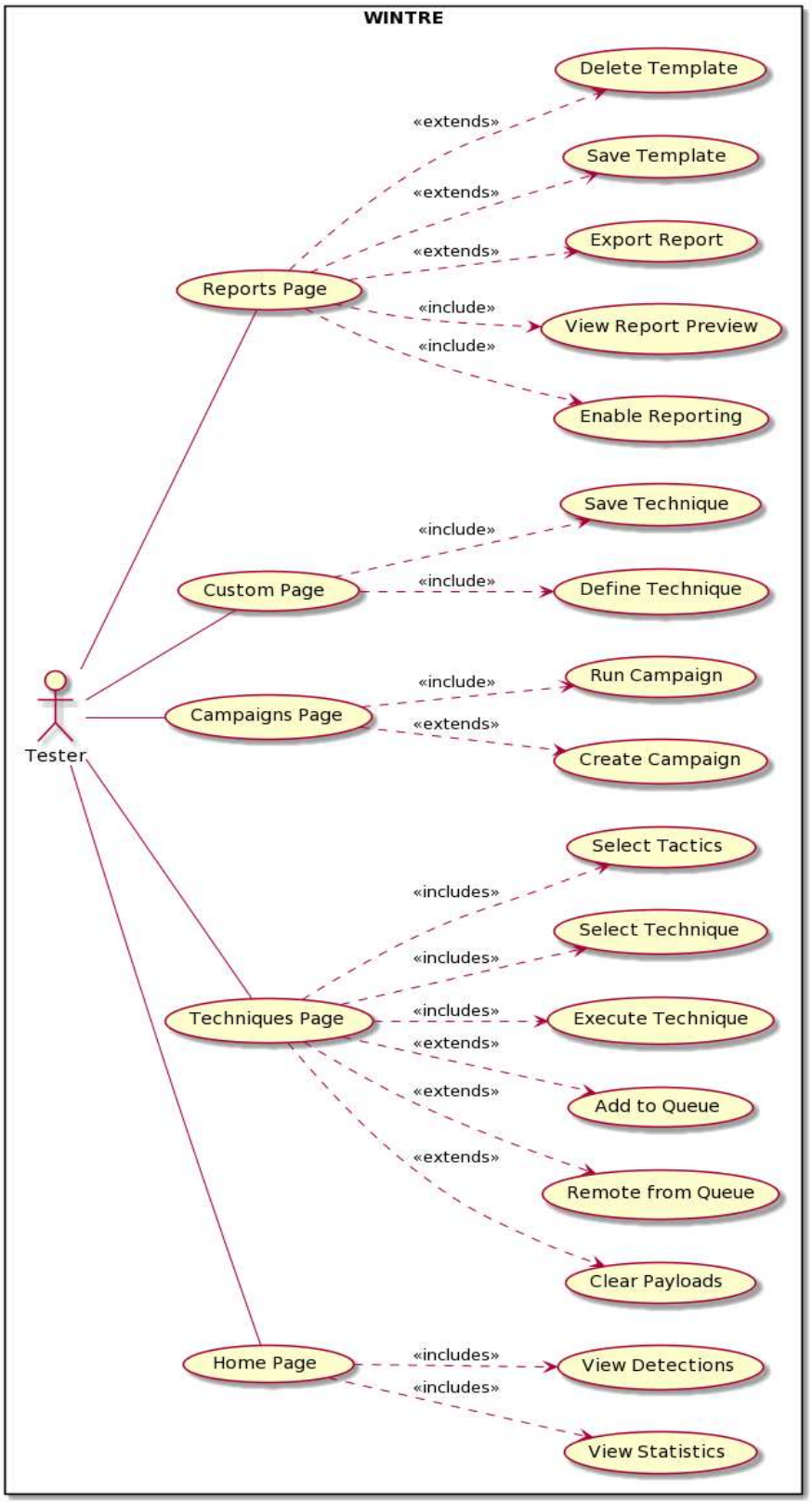

Figure 10. Primary Actor Use Case Diagram

9 DETAILED USE CASES

Running a series of techniques:

Primary Actor:

- Tester/Sys admin

Preconditions:

- None

Success Guarantee:

- The user can select techniques and execute them successfully.

Main Success Scenario:

- User starts the application.
- User selects the techniques page.
- User selects their preferred tactic/category.
- User can then select tests and review the test information.
- User can add the test to the queue or simply run the test.
- User can review the output of the test in the log output or run it from the queue on the sidebar where the output will also be send to the logs.
- User can then select new techniques and repeat the process to run them.

Alternative Scenarios:

1) User attempts to run a custom test with invalid syntax.
   a) Report standard error back to the user.
   b) Print the corrupted filenames.
   c) Allow the user to delete the custom test or review the source code.
2) User attempts to run a customizable test with invalid arguments.
   a) Run the test with the built-in default arguments.

Manage campaigns:

Primary Actor:

- Tester/Sys admin

Preconditions:

- None

Success Guarantee:

- User can create and execute a "campaign", i.e. a series of pre-saved techniques using the queue system.

Main Success Scenario:

- User goes to the techniques page.
- User selects each technique they want and reviews the technique information
- User then adds said technique to the queue system.
- User then navigates to the campaigns page.
- User selects to add the current queue to a campaign.
- User defines their new campaign (title and description).
- User can then review their campaign and choose to execute it.

Alternate Scenarios:

1) User enters invalid data.
   a) Do not save the technique.
   b) Inform the user of the incorrectly formatted data and clear that input field.
2) User wants to generate a report template from the campaign.
   a) Enable reporting in the background.
   b) Generate the template based on techniques saved with the campaign.

c) Export automatically when finished executing techniques and notify the user.

3) User wants to reset the queue.
   a) Clear out the array storing queue data.

Creating a custom technique:

Primary Actor:

- Tester/Sys admin

Preconditions:

- User has tested their custom technique's syntax is correct and functioning.

Success Guarantee:

- The user can create a custom test based on a command of their choosing to fulfil a specific MITRE technique.

Main Success Scenario:

- User enters a technique name.
- User enters a related MITRE ATT&CK ID.
- User selects whether their technique runs with elevated privileges or not.
- User select the relevant category/tactic of their technique.
- User chooses a template, either cmd.exe or powershell.exe to launch their technique.
- User enters their command(s) to execute their technique.

Alternate Scenarios:

1) User enters invalid data.
   a) Do not save the technique.
   b) Inform the user of the incorrectly formatted data and clear that input field.

Generating a report:

Primary Actor:

- Tester/Sys admin

Preconditions:

- User has started the application and has selected tests to run.

Success Guarantee:

- The user is able to successfully enable report generation and the report is dynamically generated as tests are ran. This means a table containing the tests and its details are generated along with additional fields relevant to the testing (such as time / date) for the tester to fill in.

Main Success Scenario:

- The user starts the application.
- The user runs some tests or adds some tests to a queue.
- The user enables the report generation feature.
- The user runs several tests.
- The user can then view the results generated from tests and manually mark off if a technique was detected.

Alternative Scenarios:

1) User saves the report template for later usage.
   a) Save the template on disk in JSON with a name and description.
   b) Allow the user to list and load available templates in the menu.
2) User exports a report
   a) Export the report template as a word document.
   b) Open the word document so the user can fill in their own notes and details from the techniques that were ran.

10 UI INTERACTION

10.1 TECHNIQUES PAGE

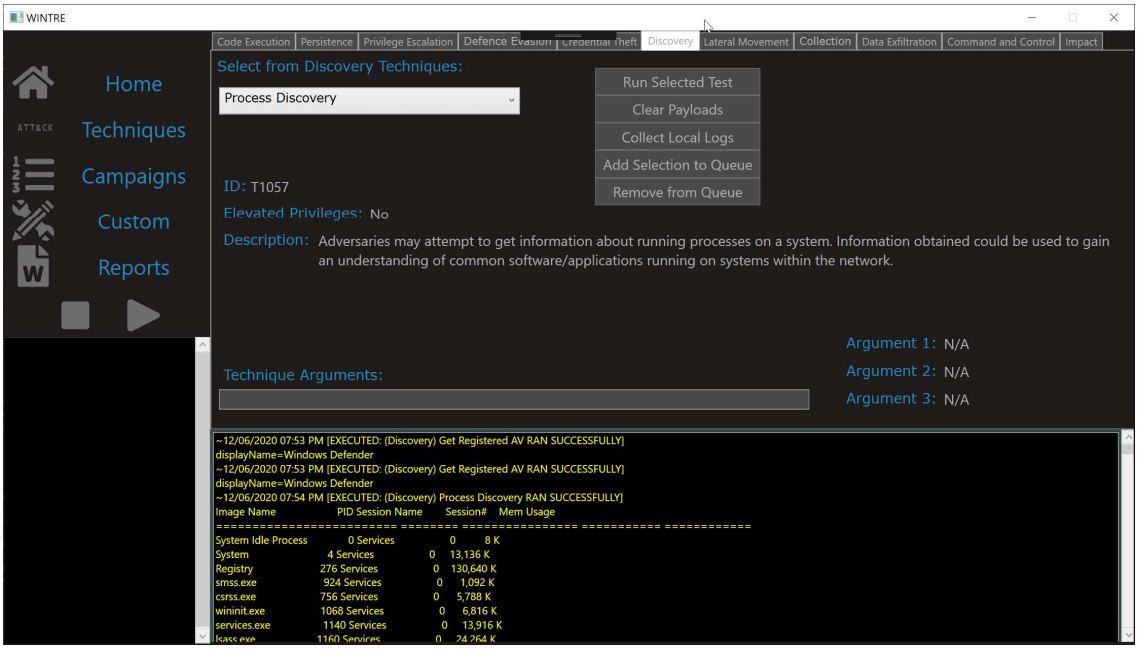

Figure 11. Techniques Page (in development) screenshot

From the techniques page, a user can have the following interactions:

- Select a tactic/category.
- Select a technique.
- Run the selected technique.
- Clear existing payloads (locally compiled EXE files).
- Collect local log files (Windows events, PowerShell logs).
- Perform queue actions (add/remove).
- Specify arguments for configurable techniques.

10.2 CUSTOM TECHNIQUES PAGE

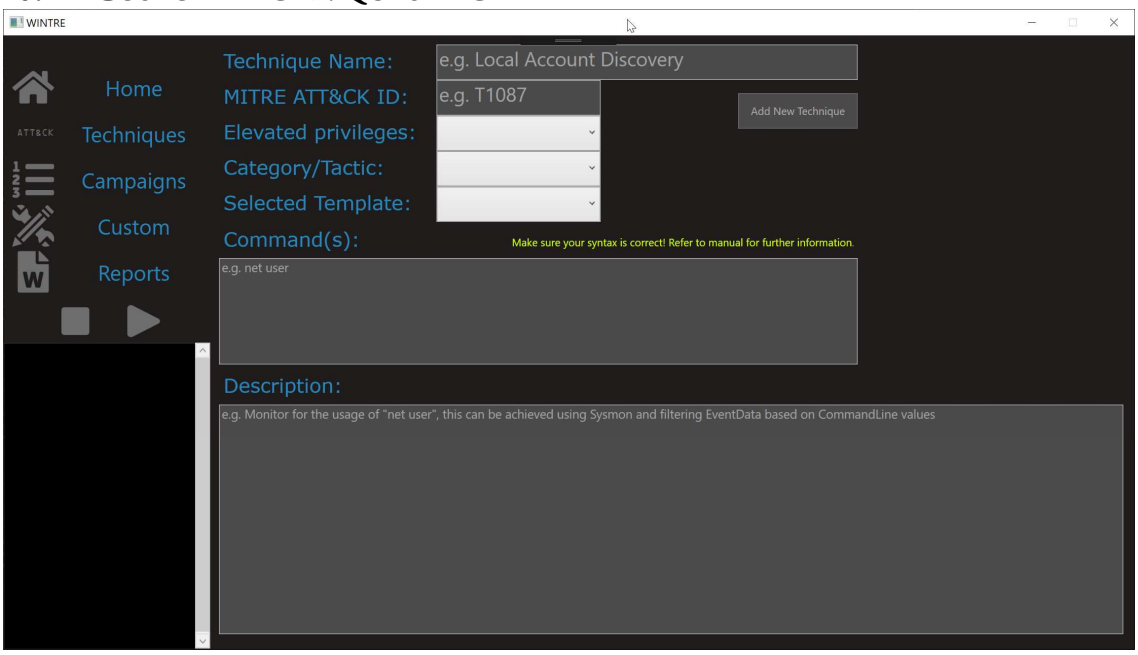

Figure 12. Custom Page (in development) screenshot

From the custom techniques page, a user can have the following interactions:

- Define a technique name.
- Define the associated MITRE ATT&CK ID.
- Define if the technique requires elevated privileges.
- Define the associated category/tactic.
- Define the template (cmd.exe or powershell.exe), i.e., the launcher used to run the technique.

10.3 HOME WIREFRAME

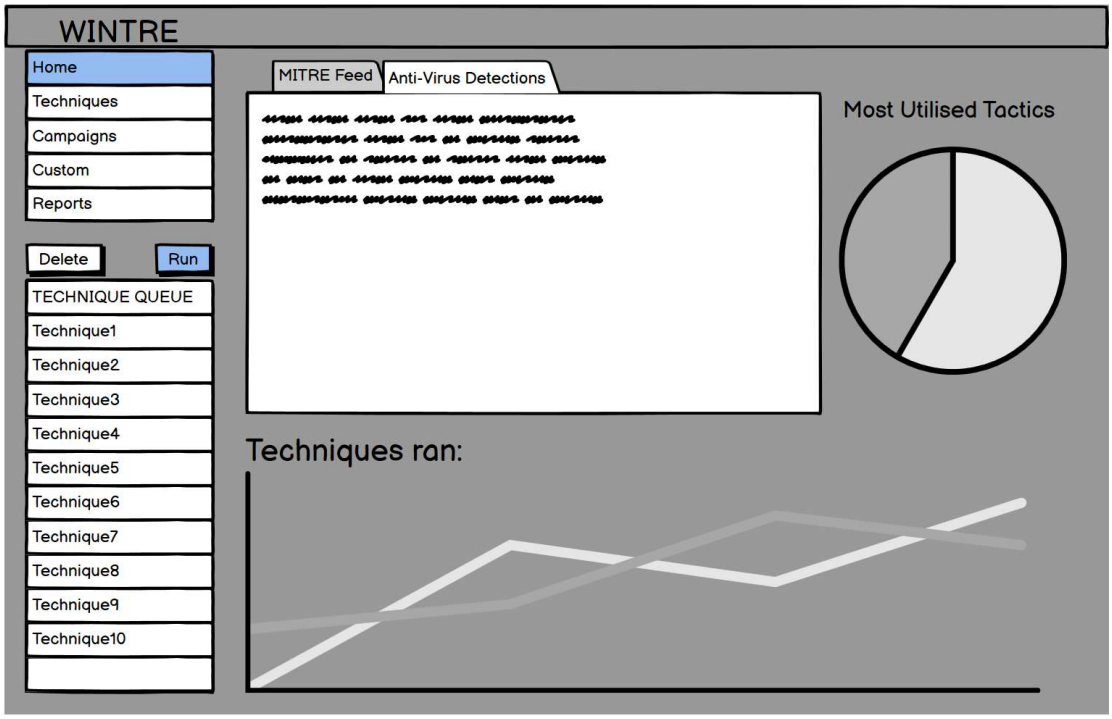

Figure 13. Home Wireframe

From the home page, a user can have the following interactions:

- View detections from Windows Defender.
- View technique statistics.
- View tactic statistics.
- View a MITRE news feed that could link to MITRE's Twitter or similar for updates/information on the framework.

10.4 CAMPAIGNS WIREFRAME

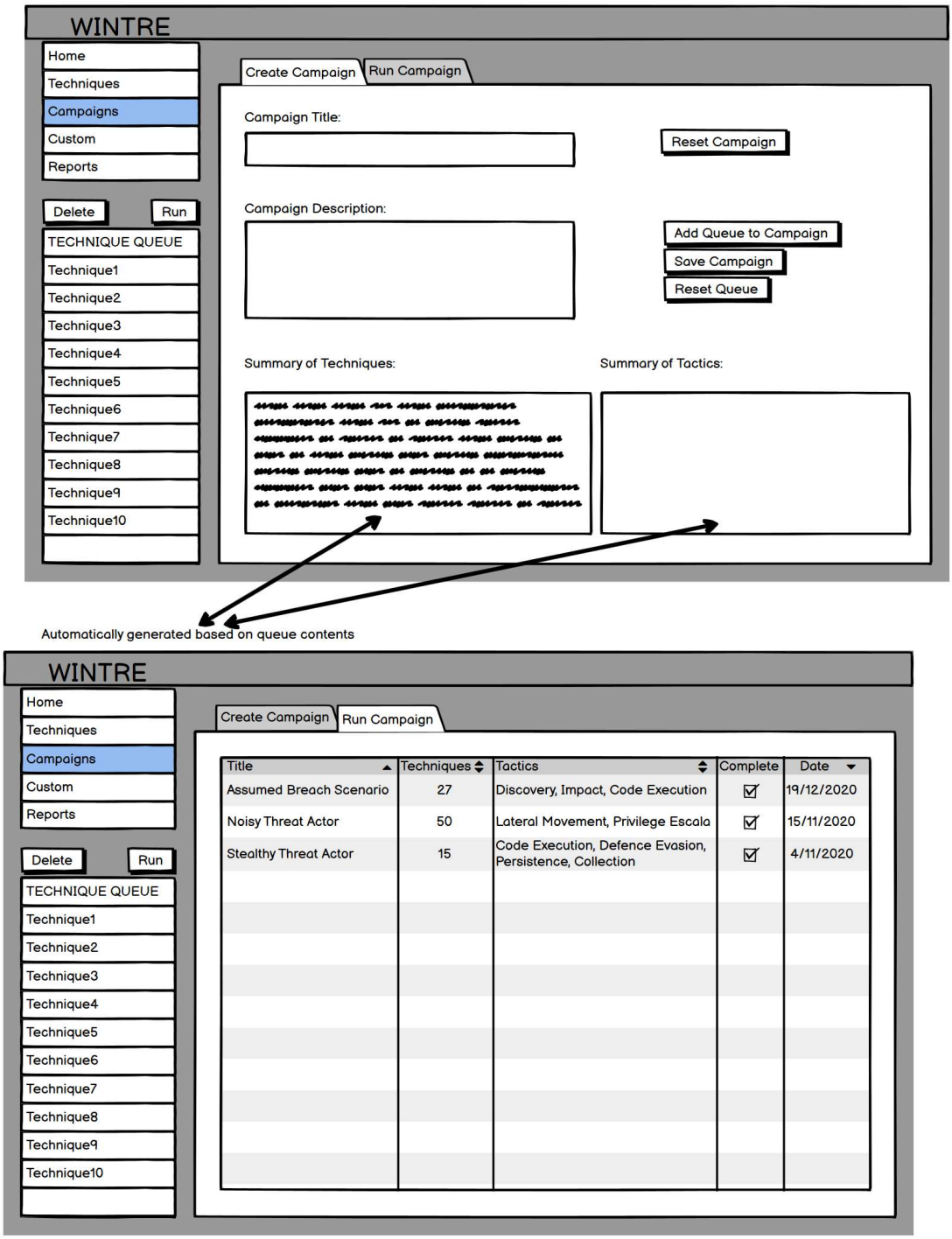

Figure 14. Campaign Wireframes

From the campaigns page, a user can have the following interactions:

- View all previously ran campaigns and their outcomes.
- View descriptions and techniques of previously ran campaigns.
- Choose to run a newly defined campaign.
- Create a new campaign, defining a title, description and techniques using the queue system.

10.5 REPORTS WIREFRAME

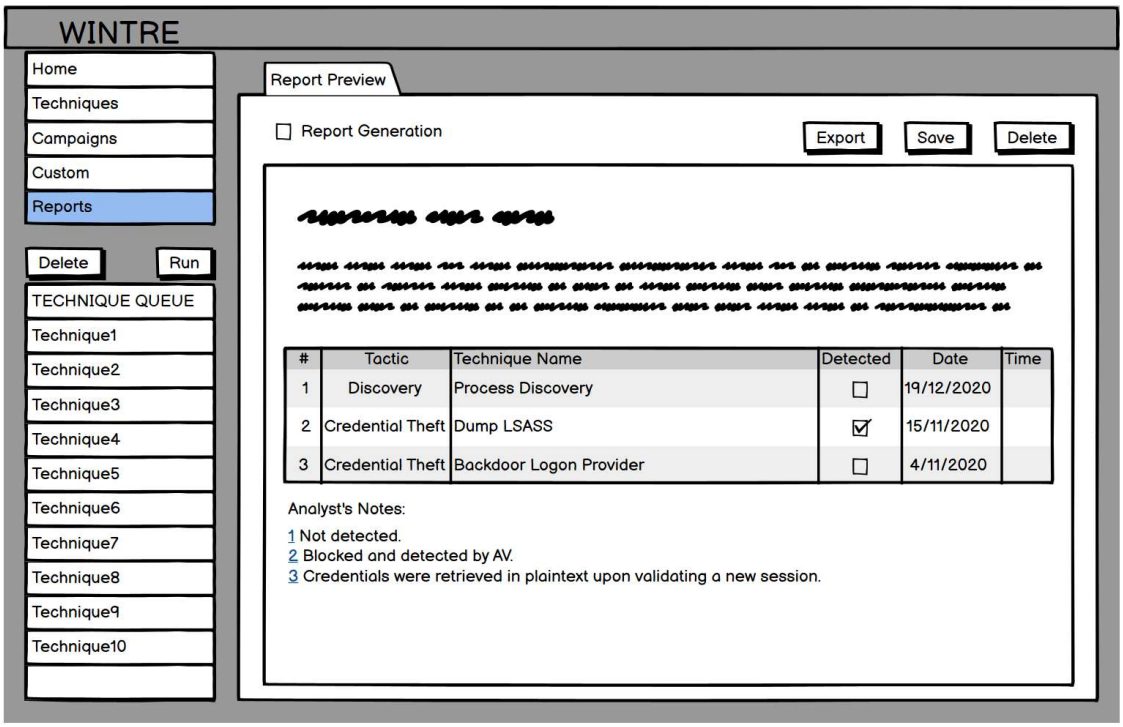

Figure 15. Reports Wireframe

From the reports page, a user can have the following interactions:

- Enable report generation, this will begin generating the report for any techniques that are ran.
- Ideally generate the report regardless of whether a technique is manually launched or ran from the queue.
- If a user runs a campaign, the report should also be generated.
- Be able to export the report to a word document.
- Save the template for later.
- Delete the current template.

11 FLOW DIAGRAMS

11.1 TECHNIQUES PAGE

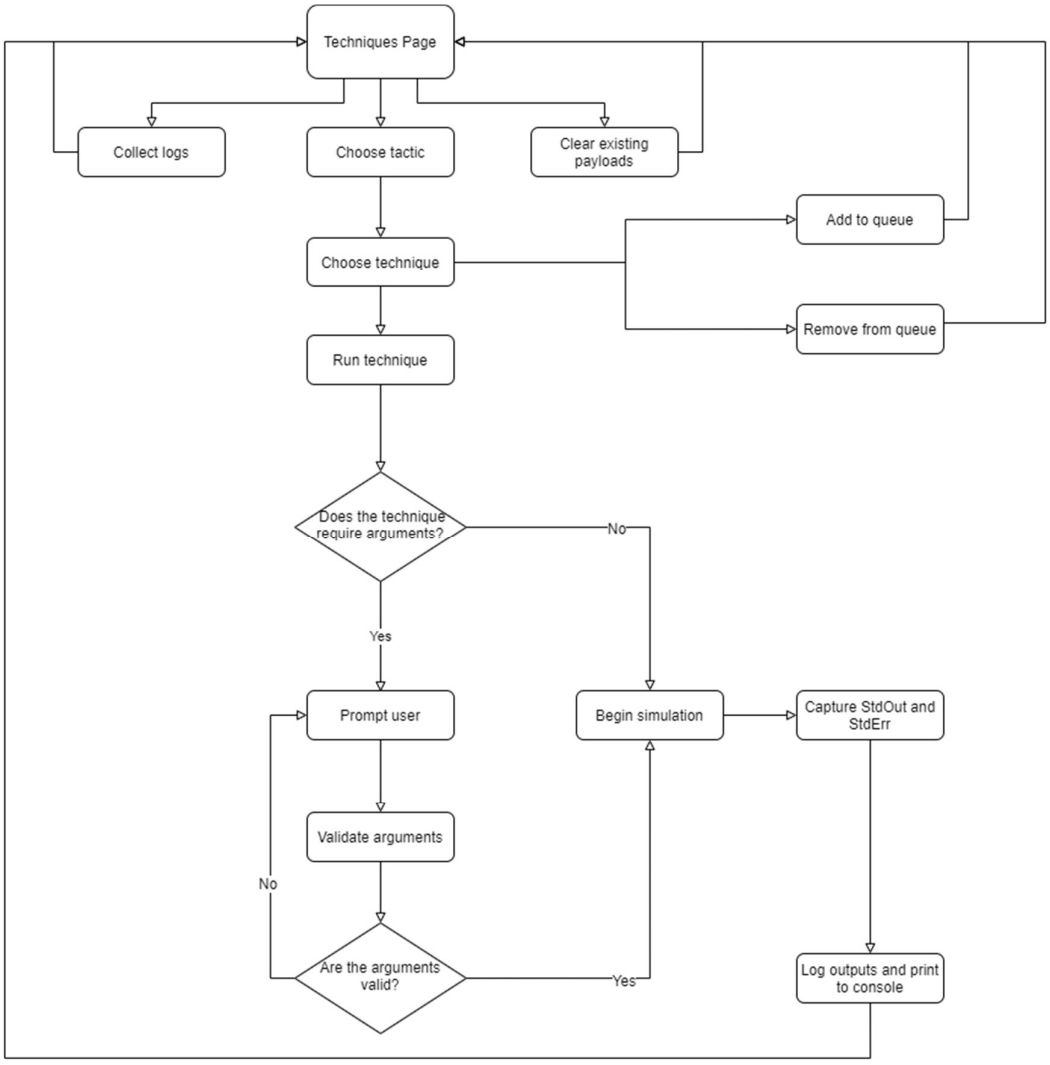

Figure 16. Techniques Page Flow

11.2 CAMPAIGNS PAGE

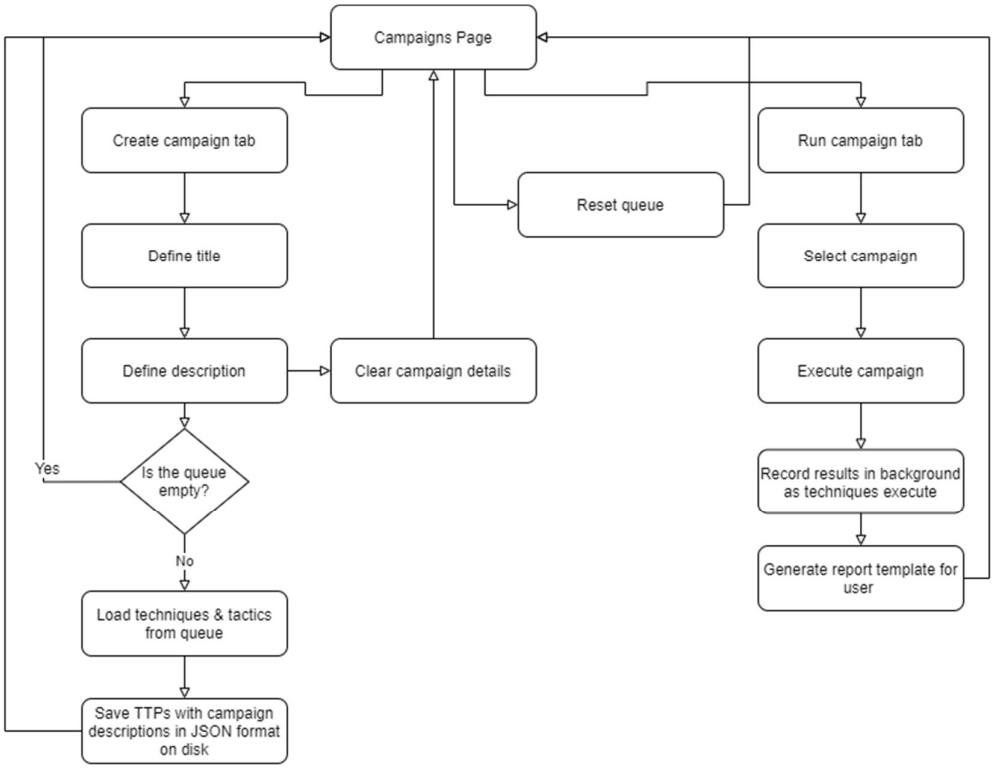

Figure 17. Campaigns Page Flow

11.3 CUSTOM PAGE

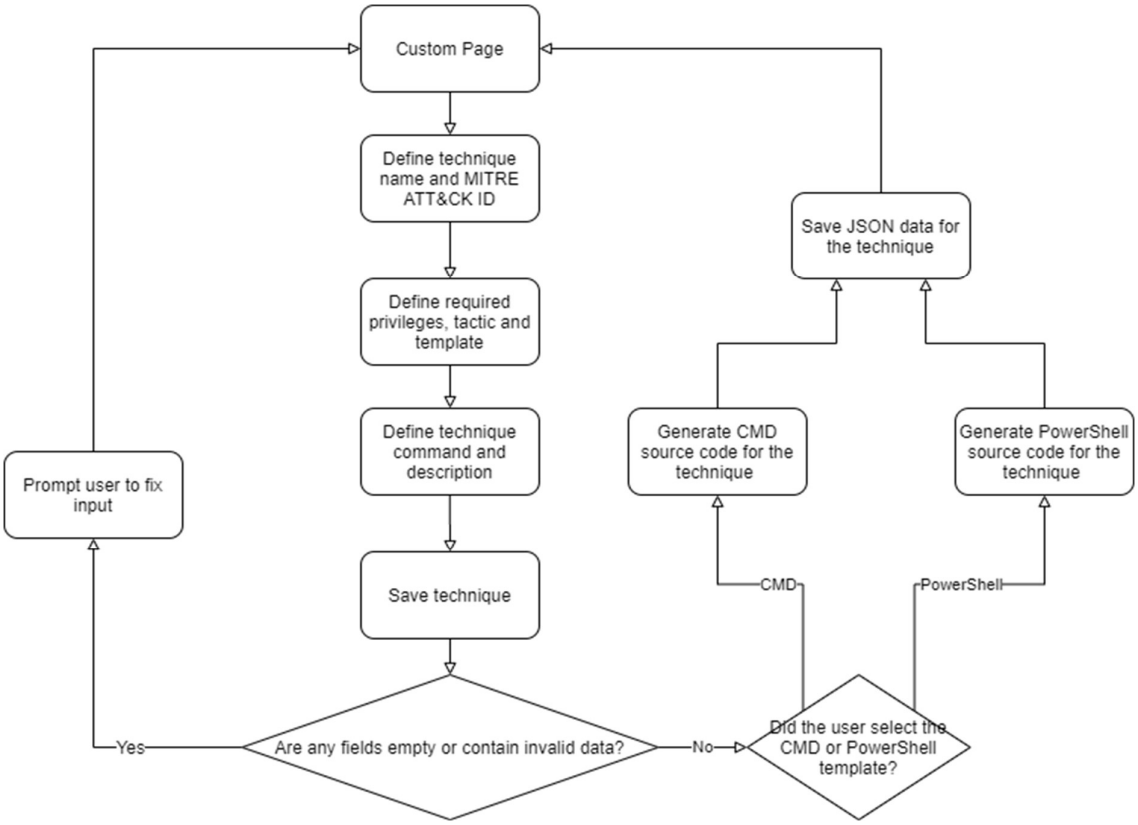

Figure 18. Custom Page Flow

11.4 REPORT PAGE

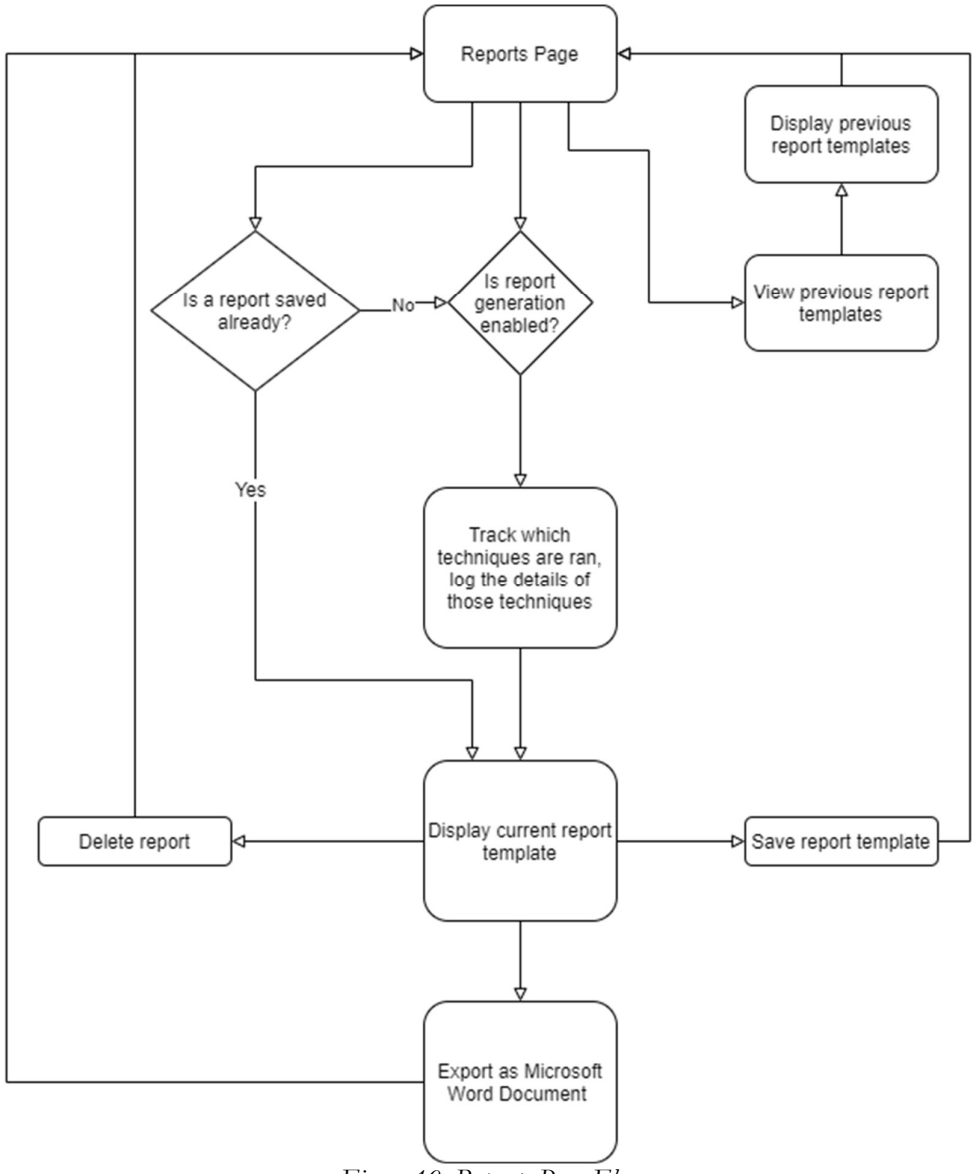

Figure 19. Reports Page Flow

12 FILE STRUCTURE

Due to the nature of this type of security testing (adversary simulation), it is necessary to store malicious payloads as source code so that they can be compiled later. A file structure similar to the following will suffice in allowing the application to separate the following to avoid anti-virus detection quarantining the main application, but also for user convenience when reviewing artifacts as a result of writing to disk:

1. Techniques - contains the source code of technique files and their matching JSON file which contains their metadata.
2. Payloads - a payloads folder should contain executables compiled from Techniques, any "helper" objects that will be called, e.g., JavaScript files that are called from a technique and outputs such as memory dumps (LSASS memory dump files).
3. Campaigns - save technique queues in JSON but also individual campaign JSON files.

13 CODE SNIPPETS

The following code snippets help further define different parts of the necessary functionality for pages within the application, such as the Techniques and Custom pages. These can be used to further understand the control flow of the application and its ideal implementation. Note these are mostly based off pseudocode and are not fully developed.

13.1 RUN TEST CODE

ButtonClickRunTest() prepares arguments for the current technique selected, so it can call Simulate().

```
private void ButtonClickRunTest(object sender, RoutedEventArgs e)
        {
            CSharpCodeProvider codeProvider = new CSharpCodeProvider();
            string testName = ComboBoxTechniques.Text;

            //Read in JSON file of selected technique
            StreamReader reader = new StreamReader(Directory.GetCurrentDirectory() +
"\\TTPs\\" + GetTabString() + "\\" + testName + ".json");

            string jsonFile = reader.ReadToEnd();
            dynamic jsonContents = JsonConvert.DeserializeObject(jsonFile);
            bool hasArgs = false;
            string arguments = "";

            //Does this test have arguments?
            //If there's arguments, process them
            if (jsonContents.hasArgs == "true")
            {
                hasArgs = true;
                arguments = ArgumentsTextBox.Text;
            }

            //Must choose TTP to run
            if (testName == "")
            {
                //Display warning label
            }
            else
            {
                Simulate(testName, GetTabString(), codeProvider, arguments, hasArgs);
            }
        }
```

13.2 SIMULATE CODE

Simulate() handles the local compilation and capturing the standard output of a technique being ran.

```
        public void Simulate(string testName, string tactic, CSharpCodeProvider
codeProvider, string arguments, bool hasArgs)
        {
            //Run test based on name, map the name to the test's src code
            string readContents;
            string currentDirectory = Directory.GetCurrentDirectory();
            //Read in the source code
            using (StreamReader streamReader = new StreamReader(currentDirectory +
"\\TTPs\\" + tactic + "\\" + testName + ".cs", Encoding.UTF8))
            {
                readContents = streamReader.ReadToEnd();
            }

            CompilerParameters parameters = new CompilerParameters();
            //Add necessary system PEs for technique compilation
            parameters.ReferencedAssemblies.Add("System.dll");
            parameters.ReferencedAssemblies.Add("System.Core.dll");
            // True - memory generation, false - external file generation
            parameters.GenerateInMemory = false;
            // True - exe file generation, false - dll file generation
            parameters.GenerateExecutable = true;
            //Name
            parameters.OutputAssembly = currentDirectory + "\\Payloads\\" + tactic +
"\\" + testName + ".exe";
            parameters.TreatWarningsAsErrors = false;
            CompilerResults results =
codeProvider.CompileAssemblyFromSource(parameters, readContents);

            if (results.Errors.Count > 0)
            {
                foreach (CompilerError CompErr in results.Errors)
                {
                    Console.WriteLine("Line number " + CompErr.Line + ", Error Number:
" + CompErr.ErrorNumber + ", '" + CompErr.ErrorText + ";");
                }
            }

            Process TTP = new Process();
            TTP.StartInfo.FileName = currentDirectory + "\\Payloads\\" + tactic + "\\"
+ testName + ".exe";
            TTP.StartInfo.UseShellExecute = false;
            TTP.StartInfo.CreateNoWindow = false; //Have to leave this off for some
GUI based ones (PS Cred Prompt). Can add additional argument to JSON "hasGUI" like
"hasArgs"
            TTP.StartInfo.RedirectStandardOutput = true;
            TTP.StartInfo.RedirectStandardError = true;
            if(!hasArgs)
            {
                TTP.Start();
            } else
            {
                TTP = Process.Start(TTP.StartInfo.FileName, arguments);
                //Need to consider if a user enters bad arguments
            }
```

```
            //Read the output (or the error)
            string standardOutput = "";
            string standardError = "";
            try
            {
                standardOutput = TTP.StandardOutput.ReadToEnd();
                standardError = TTP.StandardError.ReadToEnd();
            } catch
            {
              //Handle if neither can be retrieved (process failed to start)
            }

            Debug.WriteLine(standardOutput);
            Debug.WriteLine(standardError);
            TTP.WaitForExit();

            //Verbose output for each technique
            if (standardError.Length == 0)
            {

                LogOutput.Text = LogOutput.Text + "~" +
DateTime.Now.ToString("MM/dd/yyyy hh:mm tt ") + "[EXECUTED: " + "(" + tactic + ") " +
testName + " RAN SUCCESSFULLY]";
                LogOutput.Text = LogOutput.Text + Environment.NewLine +
standardOutput.Trim() + Environment.NewLine;
            } else
            {
                LogOutput.Text = LogOutput.Text + "~" +
DateTime.Now.ToString("MM/dd/yyyy hh:mm tt ") + "[EXECUTED: " + tactic + testName + "
FAILED TO RUN]";
                LogOutput.Text = LogOutput.Text + Environment.NewLine +
standardError.Trim() + Environment.NewLine;
            }


        }
```

13.3 LOAD TECHNIQUES

LoadTechniques() is used to load all available techniques on disk when the user selects a different tactic.

```
        //When we switch tactic tabs, load techniques of the selected tactic
        private void LoadTechniques(object sender, SelectionChangedEventArgs args)
        {
            //Clear out previous technique details
            valueID.Text = "";
            valuePrivs.Text = "";
            valueDesc.Text = "";

            //Get the techniques based off source code available on disk
            string[] sourceFiles = Directory.GetFiles(Directory.GetCurrentDirectory()
+ "\\TTPs\\" + GetTabString(), "*.cs", SearchOption.AllDirectories);
            string[] testNames = sourceFiles;
            for (int i = 0; i < testNames.Length; i++)
            {
                testNames[i] = Path.GetFileName(testNames[i]);
                testNames[i] = testNames[i].Remove(testNames[i].LastIndexOf(".cs"));
//Remove the .cs at the end for displaying
            }
            ComboBoxTechniques.ItemsSource = testNames;
        }
```

13.4 CLEAR PAYLOADS

ClearPayloads() works similar to LoadTechniques(), simply storing a list of compiled executables from previous testing and allowing the user to delete these executables. Not necessary for every session but may be used for clean-up when switching between test scenarios for different security controls.

```
        private void ClearPayloadsClick(object sender, RoutedEventArgs e)
        {

            string[] sourceFiles = Directory.GetFiles(Directory.GetCurrentDirectory()
+ "\\Payloads\\", "*.exe", SearchOption.AllDirectories); //Get paths of compiled EXEs

            for(int i = 0; i < sourceFiles.Length; i++)
            {
                File.Delete(sourceFiles[i]);
            }
            LogOutput.Text = LogOutput.Text + Environment.NewLine + "~" +
DateTime.Now.ToString("MM/dd/yyyy hh:mm tt ") + "[INFO: DELETED " + sourceFiles.Length
+ " COMPILED EXEs]" + Environment.NewLine;
        }
```

13.5 TEST SELECTED

TestSelected() reads a technique's metadata and displays it on screen to guide the user.

```
        //Event for when tests are selected, load details from matching JSON file
        private void TestSelected(object sender, SelectionChangedEventArgs e)
        {
            //The Technique's JSON file name
            string selectedTechnique = "";

            //Moving between tabs the selected test becomes NULL, need to handle this,
so that UI update/JSON retrieval only starts if a valid test is selected.
            try
            {
                selectedTechnique = ComboBoxTechniques.SelectedItem.ToString() +
".json";
            } catch
            {
                //Exception as combo box will equal NULL
            }

            if(selectedTechnique.Length <= 5) //Must be <=5 chars due to ".json"
            {

            } else { //Continue with loading JSON elements onto UI
                string[] sourceFiles =
Directory.GetFiles(Directory.GetCurrentDirectory() + "\\TTPs\\" + GetTabString() +
"\\", selectedTechnique, SearchOption.AllDirectories);
                string jsonFile;

                using (StreamReader reader = new
StreamReader(Directory.GetCurrentDirectory() + "\\TTPs\\" + GetTabString() + "\\" +
selectedTechnique)) //Read in JSON file of selected technique
                {
                    jsonFile = reader.ReadToEnd();
                    dynamic jsonContents = JsonConvert.DeserializeObject(jsonFile);
                    //IMPORTANT: All JSON files must follow a consistent set of values

                    //Update UI accordingly
                    valueID.Text = jsonContents.ID;
                    valuePrivs.Text = jsonContents.elevated;
                    valueDesc.Text = jsonContents.desc;
                    arg1.Text = jsonContents.arg1;
                    arg2.Text = jsonContents.arg2;
                    arg3.Text = jsonContents.arg3;
                }
            }
        }
```

14 UI/UX CONSIDERATIONS

- Simplicity: The application has been considered in order to make interaction as simple as possible. A minimal number of pages and components have been chosen to allow the user the necessary functionality to accomplish their tasks.

- Colour Scheme: The chosen colour scheme is consistent and easily readable. All UI elements need to be legible and certain colours will correlate with particular UI elements. For example, yellow will be used consistently to represent logging output and red will be used to display warnings etc. The colour scheme will be user friendly in regular and low lighting due to the chosen contrast and palette of colour.

- Responsiveness: The UI is responsive and the user is able to quickly switch between pages and change operating modes utilising the efficiency of XAML.

- Informational Components: Several informational features such as tool tips and watermark text will be used to try answer questions a user may have without having to refer to the manual. For example, on the Custom Techniques page the user will immediately recognise the format required for a MITRE ATT&CK ID. This ID will also utilise hyperlinks in the Techniques page from which they're ran, so a user can quickly refer back to the official MITRE technique reference.

- Design Hierarchy: The design hierarchy, i.e., the layout which the user is expected to click through. Most relevant buttons and controls are placed near UI elements that will be interacted with the most often.

Several additional components of the application are also based entirely around improving the user experience. Namely, the campaign, custom and reporting features.

Campaigns will allow users to save a series of tests in a queue for later usage. The user will be able to close the application and re-open at a later date to resume date. These campaigns will also have recordable results, so a user can periodically review efficiency rates of techniques and compare them over time. If an organisation wants to test the same series of techniques repeatedly until most are being detected, campaigns grant them this visibility.

Custom techniques incentivize users to try out techniques they may be interested in implementing themselves or ones specific to their organisation. The user manual will also strongly guide this process flow but it has been simplified so that a user may enter the relevant information and create new techniques that will be added to the list of all existing techniques.