Process Behavior Analysis with Focused Clustering

Isaac Amann

5/10/2024

Contents

1 Abstract

Windows programs interact with the operating system through functions provided by the Windows API. These are often abstracted through the use of other third party libraries, but programs still call them even if the programmer did not invoke them directly [4]. By counting the API calls made by processes, a data set can be created where process behavior can be analyzed. A motivating example of this would be ransomware. Ransomware programs would make many file access API calls since most files on the system would be opened in order to encrypt them. Other programs with a similar number of file access API calls may also be ransomware. In a large enough data set, it may be possible to group programs by their behavior.

2 Acknowledgments

I would like to thank Dr. Che for the opportunity to contribute to the Sow and Grow project, it has been an invaluable learning experience. Thanks also to Jacob Bowers and Jake Anderson, I enjoyed working on the Sow and Grow project with you both and your advice and critique of my senior project is greatly appreciated.

3 Introduction

3.1 Background

Programs running on Windows interact with the operating system primarily through the Windows API. Software written in languages like Java or using the .NET framework that rely on a runtime environment would still also end up invoking Windows API functions. These functions are stored in shared DLL files that can be loaded dynamically at runtime as needed. The Windows API provides many services including networking, file access, and memory management.

The main data analysis method intended for this project is cluster analysis using the Sow and Grow algorithm being developed by Dr. Che and DBSCAN with cosine similarity. Clustering suits the data set well as the data will not be labeled. The Sow and Grow algorithm is an alteration of DBSCAN where instead of clustering the entire dataset, a subset of the data are designated as seed points. Clusters are then grown from these seed points, leaving the rest of the data set unvisited saving large amounts of processing time. DBSCAN with cosine similarity is just regular DBSCAN using cosine similarity as its distance metric instead of euclidean distance.

The Sow and Grow algorithm fits my use case better since it allows seed points to be selected only for data points the user is interested in. For example in this project, we are not interested in web browsers or text editors and should not waste time discovering clusters of them. With this algorithm, known samples of malware can be selected as seed points to see if other malware samples with similar behavior are discovered by clustering with them.

DBSCAN with cosine similarity was also included as an alternative data analysis method since it will likely give better results with higher dimensional data. The system monitors around 50 Windows API calls creating a data points with around 50 dimensions. With euclidean distance, even if there was a strong relationship between two points in many dimensions, it would only take 1 outlier dimension to create a large distance that would exceed the minimum distance for the points to be clustered. Although Sow and Grow with cosine similarity would likely suit the project better, re-implementing Sow and Grow to use cosine distance is not a trivial task and is outside the scope of this project.

3.2 Motivation

The main motivation of this project is to create a tool set for collecting data on processes Windows API usage in order to reveal unknown malware samples that may be in the dataset. The system is also designed to support a large number of data collection clients, using a REST API to post collected data from an arbitrary number of clients. The data set itself is intended to be analyzed using cluster analysis through the Sow and Grow algorithm currently being developed by Dr. Che and DBSCAN using cosine similarity as its distance metric instead of euclidean distance to give better results with the high dimensional data.

3.3 Project Scope

The project was intended to deliver the following major features:

- User level Windows API monitor
  - Attach to other processes and keep count of the times certain Windows API calls were executed
  - Post these counts to the server through HTTP after a process exits
- Front-end web dashboard
  - Dataset explorer
  - Data analysis job submission / viewer
  - Admin dashboard for system management
- Back-end server
  - Provide a REST API for data collection / access
  - Authenticate data collection clients and users
  - Run and store results of data analysis jobs submitted by users

3.4 Problem Statement

The project is intended to provide an alternative method of malware detection to signature based detection. Instead of looking for similar code segments this method attempts to group processes by looking for similar behavior. Since this method does not rely on inspecting code, it would not be vulnerable to code obfuscation. Some other methods also rely on inspecting executables to see if they reference Windows API calls often used in malware. This method would not detect a suspicious volume of Windows API calls. For example, file access calls such as CreateFile, OpenFile, and WriteFile are used by most legitimate software. However, most legitimate software would not have a reason to iterate over every file in the file system. In the case of ransomware and other file encrypters, they would make an abnormal number of file access calls that could be detected with the analysis technique used in this project.

4 System Analysis

4.1 Functional Requirements

FR1 Server Data Collection		
Goal: Collection Server should collect Data Points from Client		
Collection Server should host an API endpoint that allows Client software to send Data Points collected while monitoring processes on the host system		
Origin:		
Version: 1.0	Date: 11/01/2023	Priority: high

FR2 Client Side Data Collection		
Goal: Client software should monitor running processes		
Client should monitor processes running on the host system and collect Data Points that include the number of Windows API calls made by the processes.		
Origin:		
Version: 1.0	Date: 11/01/2023	Priority: high

FR3 Process Information		
Goal: Data points should include identifying information		
In addition to the sums for the API calls made by the process, the data point should include additional information including run time, executable name, and possibly a signature of the executable. While the clustering algorithm only needs the unlabeled data, these labels will be needed to make use of the data after being analyzed.		
Origin: Meeting with Dr. Che		
Version: 1.0	Date: 11/15/2023	Priority: medium

FR4 Data Storage		
Goal: Collected data should be stored for later analysis		
Server should store received Data Points inside of a relational database. Clusters generated should also be stored in this database for later use.		
Origin:		
Version: 1.0	Date: 11/01/2023	Priority: high

FR5 Data Analysis		
Goal: Server should use Sow and Grow algorithm to analyze data		
The server should contain methods that allow the User to provide Seed Points to grow Clusters from the stored Data Set. Data Points that fall into the Clusters generated by the Seed Points should be similar to the Seed Points used.		
Origin:		
Version: 1.0	Date: 11/01/2023	Priority: high

FR6 Front End		
Goal: Server should provide a front end for accessing the system		
The server should interface with a web dashboard to allow authorized users to view data, diagnostic information, and analyze data.		
Origin:		
Version: 1.0	Date: 11/01/2023	Priority: high

FR7 API Key Generation		
Goal: The server should provide a method to allow new clients to be registered		
API calls should be provided to allow new clients to register and receive a API key used to authenticate when submitting data.		
Origin:		
Version: 1.0	Date: 11/01/2023	Priority: medium

4.2 Non-Functional Requirements

NF1 Backups		
Goal: Backups should be kept to prevent data loss		
The database containing the data set should be backed up regularly to prevent loss of data or to allow for rollbacks in the case bad data was submitted to the data base.		
Origin:		
Version: 1.0	Date: 11/01/2023	Priority: low

NF2 Client Stability		
Goal: The client software should not make the host system unstable		
The client software will need to intercept Windows API calls made by other processes. This should be done without causing processes to crash or significantly decreasing performance		
Origin:		
Version: 1.0	Date: 11/01/2023	Priority: low

NF3 Client Authentication		
Goal: Server should authenticate clients before accepting data		
The server should use API keys to authenticate clients preventing unwanted data from being submitted to the data set by bad actors. Data points should also be labeled with identifying information indicating which client they originate from.		
Origin:		
Version: 1.0	Date: 11/01/2023	Priority: medium

NF4 Data Transfer Security		
Goal: Data from clients should be sent securely		
Data sent through API endpoint should not be sent as plain text. Data may include sensitive data including system information or API keys that could be used to send requests in place of the genuine client.		
Origin:		
Version: 1.0	Date: 11/01/2023	Priority: low

NF5 Rate Limits		
Goal: The server should not be vulnerable to brute force attacks		
The server should implement rate limits on API calls to prevent API keys or passwords from being leaked with a brute force attack.		
Origin:		
Version: 1.0	Date: 11/01/2023	Priority: medium

4.3 Use Cases

Use Case Name	Post Data Point
Related Requirements	FR1, FR3
Goal In Context	A Client posts a new data point to the Server through HTTP
Preconditions	Client is registered with the Server and possesses a valid API key
Successful End Condition	Data point is processed by the Server and stored on the Database. A success message is returned.
Failed End Condition	Server rejects data point and returns an error message in its response.
Primary Actors	Client, Server
Secondary Actors	Database
Trigger	Client makes a post data point request
Main Flow	1. Client makes a post data point request to the Server 2. Server authenticates the Client 3. Server creates a new data point object 4. Server pushes new data point to Database
Extensions	• None

Use Case Name	Register Client
Related Requirements	FR7
Goal In Context	A new Client registers with the Server and is provided an API key for making future requests from the server
Preconditions	Client is installed on host and Server is running
Successful End Condition	An API key is generated and returned to the Client. API key is also stored to Database along with other Client information.
Failed End Condition	New Client is rejected and an error message is returned.
Primary Actors	Client, Server
Secondary Actors	Database
Trigger	Client software makes a register request
Main Flow	1. New Client instance makes a register request to the Server 2. Server authenticates the new Client 3. Server creates a new registered client object 4. Server generates an API key for the new Client 5. Server stores new registered client object to Database 6. Server returns the API key in its response
Extensions	• None

Use Case Name	Start Process Monitor
Related Requirements	FR2, FR3
Goal In Context	Client begins monitoring Processes running on the host system
Preconditions	Client is running with elevated permissions
Successful End Condition	Client has access to a list of running processes along with their ID's
Failed End Condition	Client halts execution outputting an error
Primary Actors	Client, Host System
Secondary Actors	Process
Trigger	Client software started or refreshes its list of running processes on the host system
Main Flow	1. Client requests a list of processes from the Host System 2. Client checks for new processes
Extensions	• 2.1: Client attaches process if it is a new process • 2.2: Client detaches a process as it exits

Use Case Name	Get Process Info
Related Requirements	FR2, FR3
Goal In Context	Client retrieves info about a Process including a handle referencing it
Preconditions	Client is running and was able to successfully retrieve a list of process ID's
Successful End Condition	Client has a valid handle referencing a target Process
Failed End Condition	Client does not receive a valid handle to a target Process. Client discards the ID of the target Process
Primary Actors	Client, Process
Secondary Actors	None
Trigger	A new process is found when refreshing the running process list
Main Flow	1. Client requests process information from the Host System using process PID 2. Client receives a Handle referencing the target Process
Extensions	• None

Use Case Name	Attach Process
Related Requirements	FR2
Goal In Context	Client injects a DLL into the target process to reroute Windows API calls to include code for counting API calls made by the target process
Preconditions	Client has a valid handle referencing the target Process
Successful End Condition	Target Process is maintaining a list of counts for each monitored API call in its memory space and its execution is not disrupted
Failed End Condition	Client discards the reference to the target Process
Primary Actors	Client, Process
Secondary Actors	None
Trigger	A new process is found when refreshing the running process list
Main Flow	1. Client creates an object to store information about the Process 2. Client loads a DLL into the address space of the target Process
Extensions	• None

Use Case Name	Detach Process
Related Requirements	FR2
Goal In Context	Client removes previously injected DLL from target process and receives collected data from the target process
Preconditions	Process is about to close or has reached a monitoring time limit
Successful End Condition	Client receives data from the target Process
Failed End Condition	Client discards Process and logs error
Primary Actors	Client, Process
Secondary Actors	None
Trigger	Process attempts to close or reaches monitor time limit
Main Flow	1. Client receives a message from the Client containing its counted API calls 2. Client creates a new data point object 3. Client adds data point object to send queue 4. Client removes object tracking the Process 5. Process closes normally
Extensions	• None

Use Case Name	Verify Client
Related Requirements	FR7, NF3
Goal In Context	Server verifies Client sending a request before accepting data
Preconditions	Server is running and maintaining a table of API keys corresponding to individual Clients
Successful End Condition	API key is valid and the Client request is accepted
Failed End Condition	API key is invalid and the request is rejected
Primary Actors	Client, Server
Secondary Actors	Database
Trigger	Client makes a request that requires authentication
Main Flow	1. Server compares the passed API key to the corresponding key stored in the Database 2. Allows request if the keys match
Extensions	• None

Use Case Name	Process Data Point
Related Requirements	FR1, FR3, FR4
Goal In Context	Server stores a new data point after accepting a post data point request from a Client
Preconditions	Server accepted a post data point request
Successful End Condition	New data point is stored on the Database
Failed End Condition	Error is logged
Primary Actors	Client, Server
Secondary Actors	Database
Trigger	Server accepts a post data point request
Main Flow	1. Server checks that the data point is in the correct format 2. Server creates a new data point object 3. Server pushes new data point object to Database
Extensions	• None

Use Case Name	Run Data Analysis
Related Requirements	FR5
Goal In Context	Server accepts a request to start a new data analysis job
Preconditions	Request contained a valid API key and the request was valid
Successful End Condition	Data analysis job is started and a UUID corresponding to the job is returned
Failed End Condition	Error is logged and an error message is returned
Primary Actors	Server
Secondary Actors	Database
Trigger	Server receives a run data analysis request through its API
Main Flow	1. New Analysis Job object is created using the passed parameters 2. Server pushes the new Analysis Job to the Database 3. Server adds new job to the job queue 4. Server returns the UUID of the new job in its response
Extensions	• None

Use Case Name	Get Analysis Result
Related Requirements	FR5
Goal In Context	Server accepts a request for information about a analysis job
Preconditions	Request contained a valid UUID corresponding to a analysis job started on the server
Successful End Condition	Information is returned about the analysis job
Failed End Condition	Error is logged and an error message is returned in the response
Primary Actors	Server
Secondary Actors	Database
Trigger	Server receives a get analysis result request
Main Flow	1. Server looks finds Analysis Job object by the passed UUID 2. Server returns the Analysis Job object information in its response
Extensions	• None

4.4 Use Case Diagrams

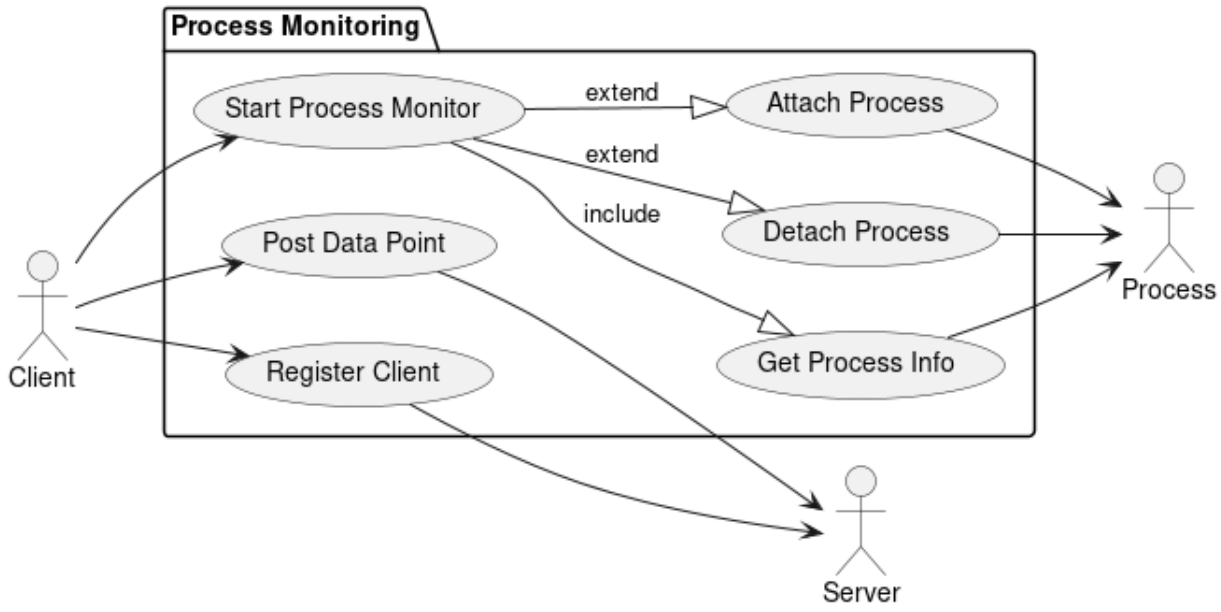

Figure 1: Use case diagram for Client

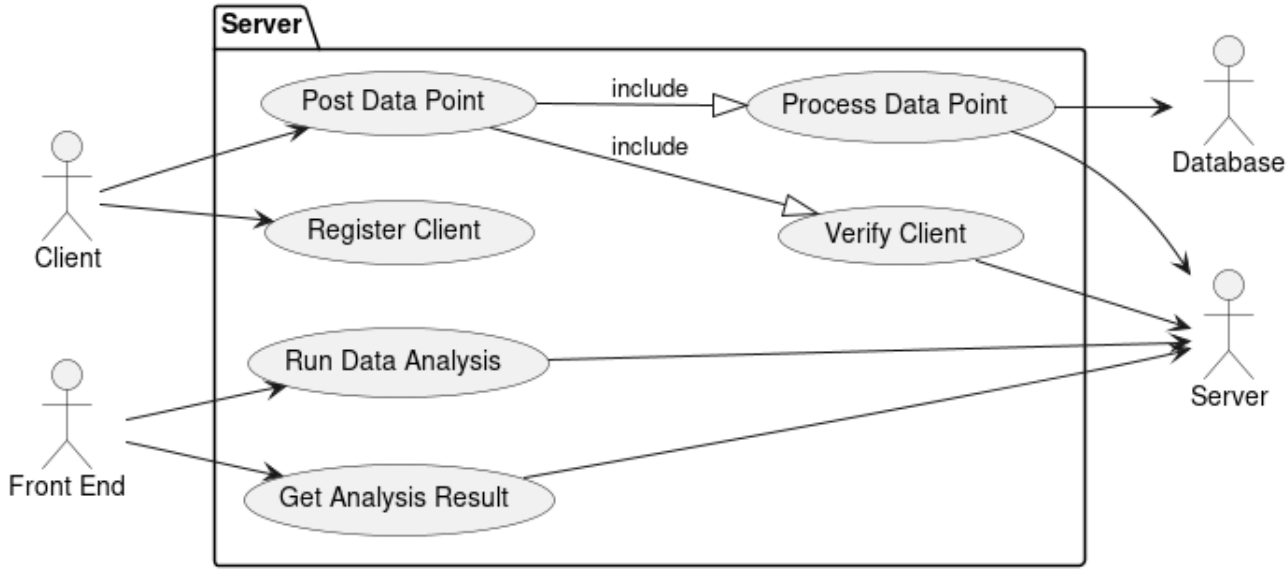

Figure 2: Use case diagram for Server

4.5 UML

4.5.1 Class Diagrams

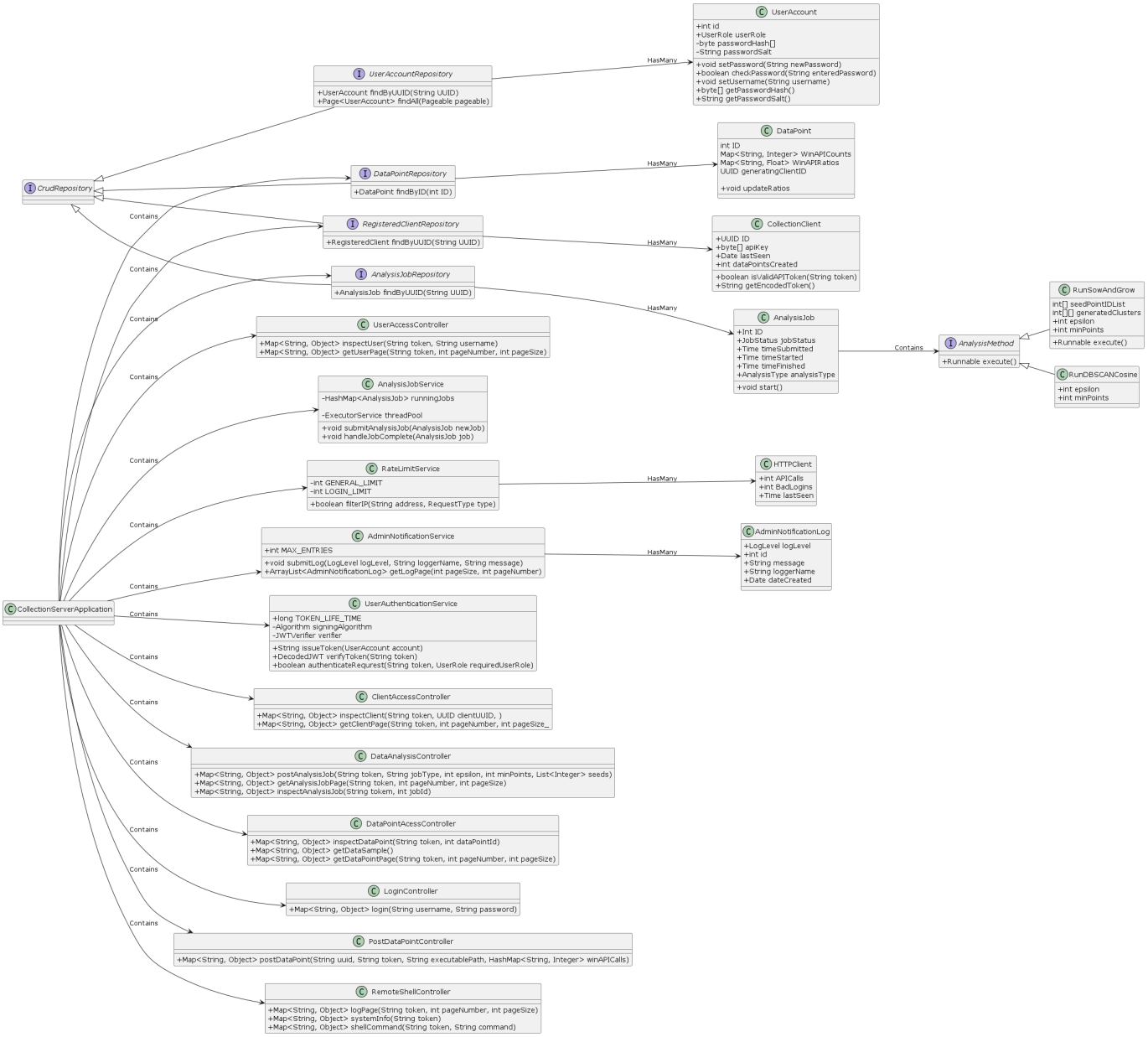

Figure 3: Server class diagram (Diagram is large, see submission folder for full image

The class diagram for the server did not change much from the project proposal. Classes were edited to reflect changes made during implementation and classes were added to show new features added. Most features are broken down into services that can be used by other classes. The REST API is implemented in functions within REST controller classes.

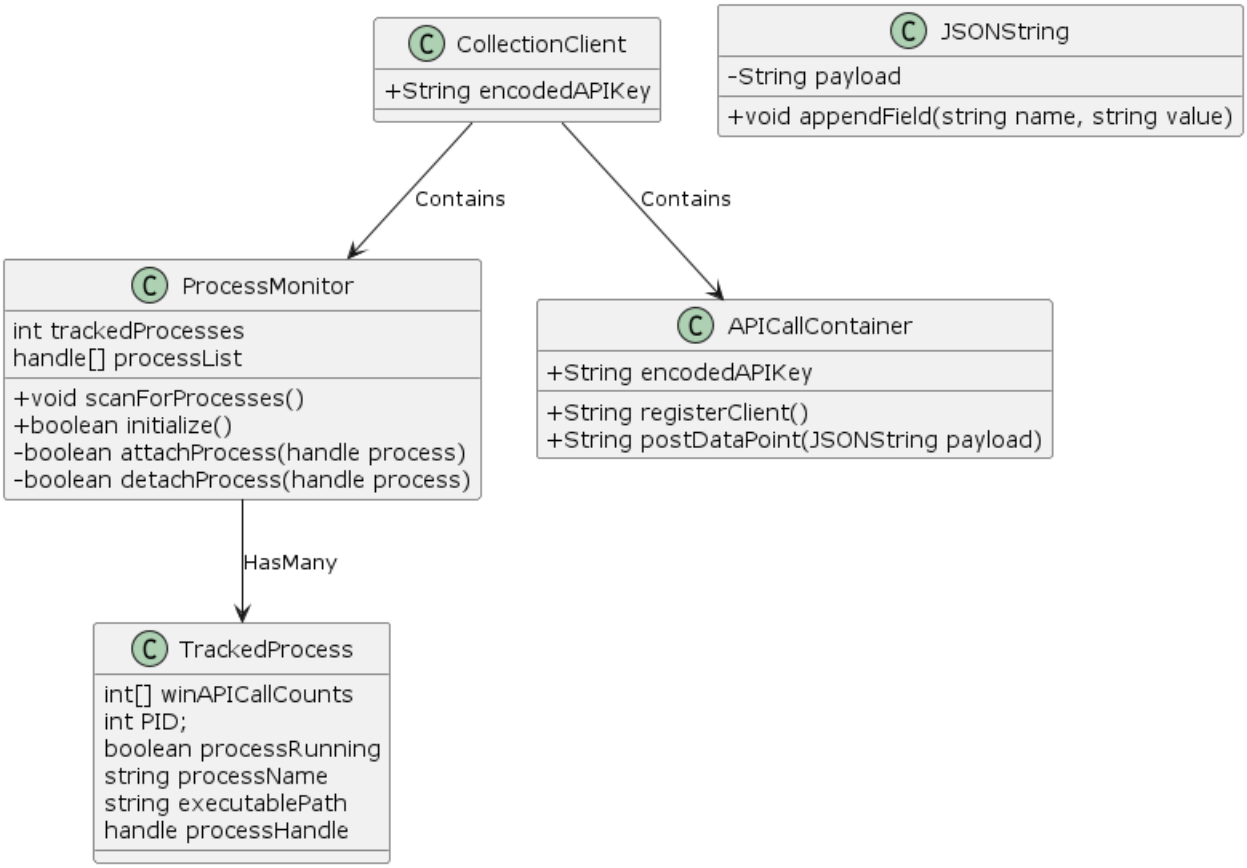

Figure 4: Client class diagram
The class diagram for the Windows API monitor did not change and it was followed for the most part during implementation

4.5.2 Sequence Diagrams

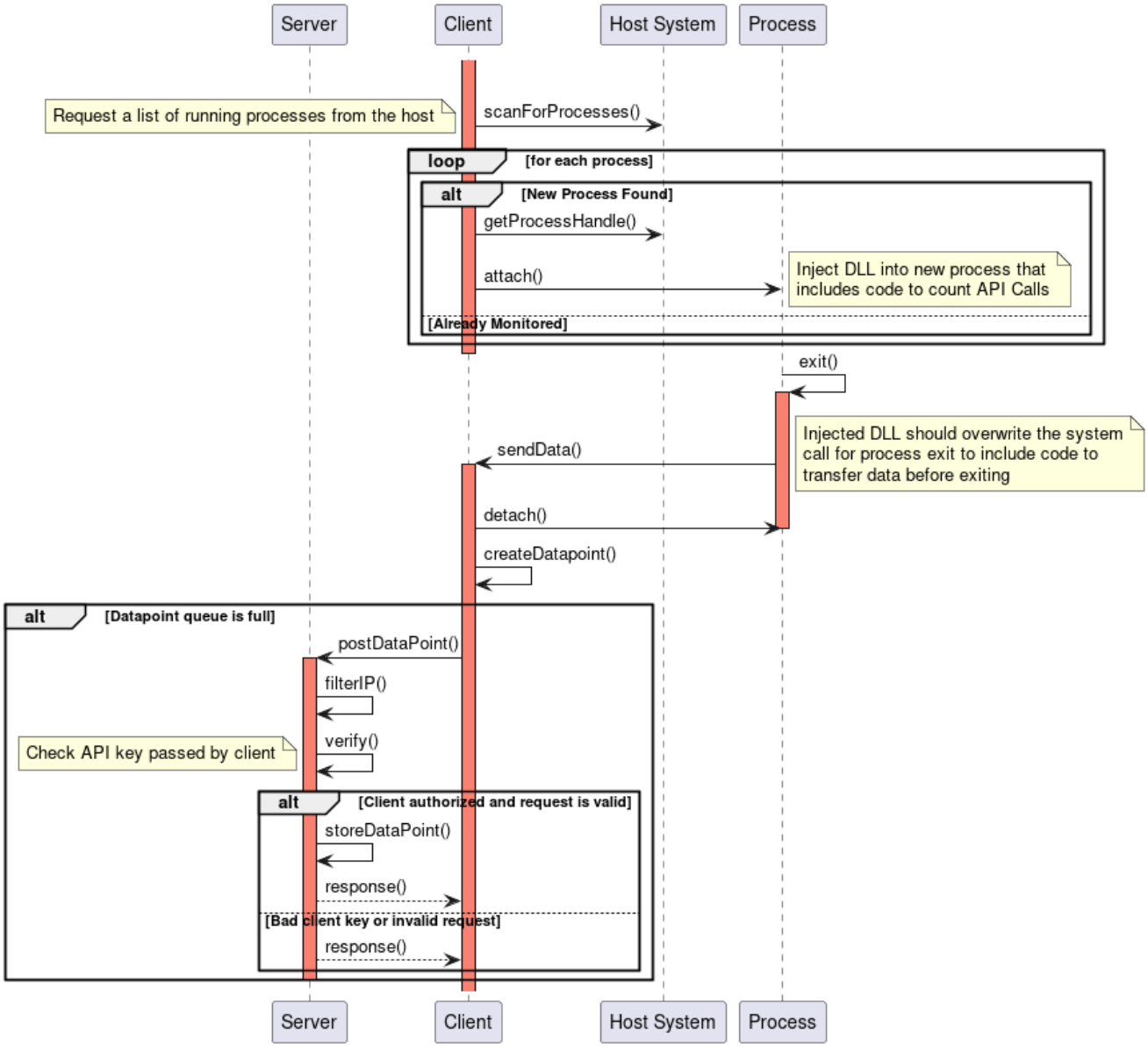

Figure 5: Sequence diagram for data collection

4.6 System Architecture

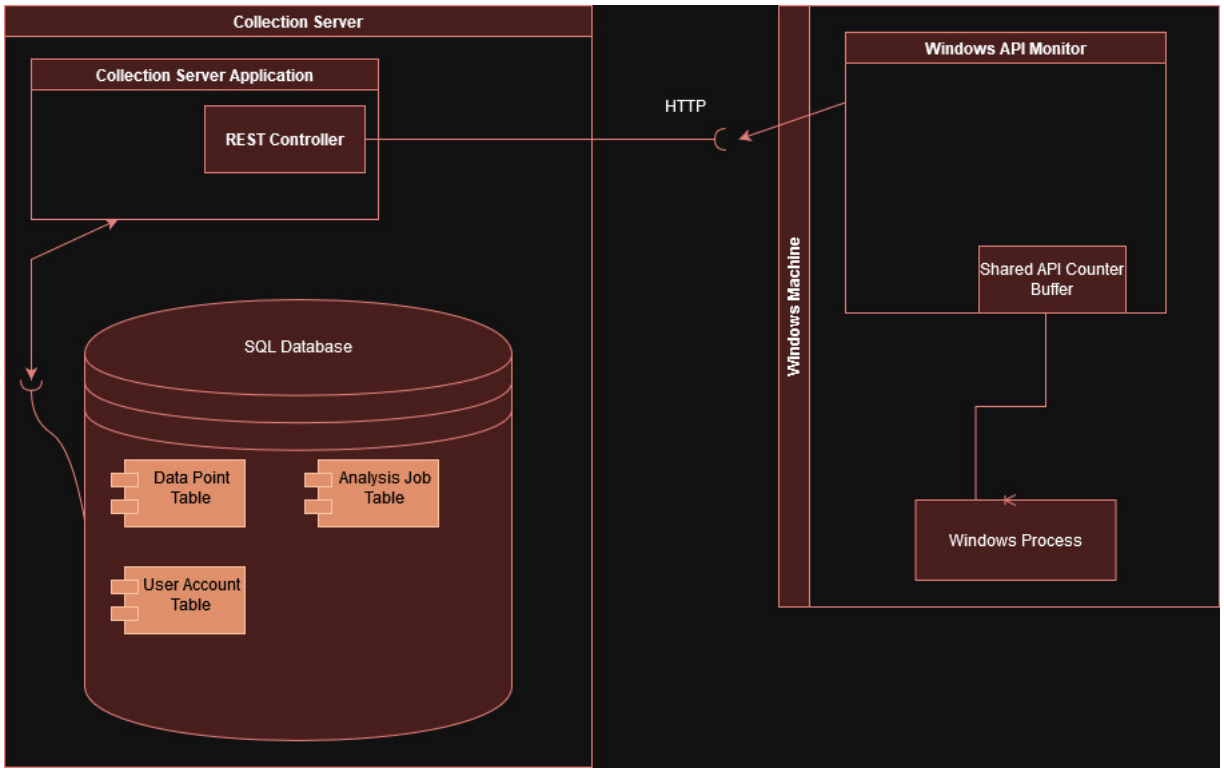

Figure 6: Sequence diagram for data collection

5 Implementation

5.1 Tools

5.1.1 Spring Boot

The back-end server was implemented using Spring Boot with Java. This was selected because of my previous experience with it. Java also has a feature that allows a thread to run a command on the host system. This is important as it allows the C++ implementation of the Sow and Grow algorithm and the python script for DBSCAN to be executed from the back-end server.

5.1.2 React

React was used to create the front-end web dashboard. React makes adding dynamic content to pages relatively simple and it has many well supported component libraries that simplify creating visually appealing user interfaces. The Material UI React framework was also used to create many components including the data set explorer, system monitor, and different navigation components.

5.1.3 HTTP / REST API

A REST API was implemented to allow data points to be posted to the server and to allow for data access and system management from the front-end dashboard. The Windows API Monitor used Curl to make HTTP requests to the server

5.2 Techniques

5.2.1 DLL Injection

The process monitor uses DLL injection in order to run code within a monitored process's address space. This is necessary since Windows does not provide a direct way of monitoring Windows API usage, so

processes need to be modified at run time. One of the most common and direct method of DLL injection was used. It involves using opening a thread in the target processes address space and then using LoadLibrary to load the DLL [3]. This can be seen in the following code segment:

```
//Load dll into process
void TrackedProcess::attach()
{
  HANDLE threadHandle;
  void* injectedLibAddress;
  HMODULE hkernel32 = GetModuleHandleA("Kernel32");

  //Allocate memory in target process
  injectedLibAddress = VirtualAllocEx(this->processHandle, NULL,
      sizeof(libPath), MEM_COMMIT, PAGE_READWRITE);

  //Write DLL path name to memory
  WriteProcessMemory(this->processHandle, injectedLibAddress, (void
      *)libPath, sizeof(libPath), NULL);

  //Create new thread and load dll into process
  threadHandle = CreateRemoteThread(this->processHandle, NULL, 0, (
      LPTHREAD_START_ROUTINE)GetProcAddress(hkernel32, "LoadLibraryA
      "), injectedLibAddress, 0, NULL);
}
```

5.2.2 Function Hooking

Since the Windows API does not provide a method for creating an event for API calls or other means of monitoring calls, the monitored Windows API calls need to be directly modified in the target process's address space to modify its behavior to include updating a count of the number of times it has been called. This is achieved with a hook and trampoline function. The first 5 bytes of a target function are stored in a buffer and replaced with a 1 byte machine code JMP instruction followed by a four byte integer representing the distance from the target function to the hook function. A trampoline function would also be created that would contain the first five bytes that were replaced followed by another JMP instruction. Another 4 byte integer would follow this JMP instruction representing the address of the target function + 5 bytes to avoid running the altered code again. The hook function would then run any extra desired code or modify arguments before calling the created trampoline function to return control to the original target function[2]. A diagram of this process can be seen in figure 5 and figure 6.

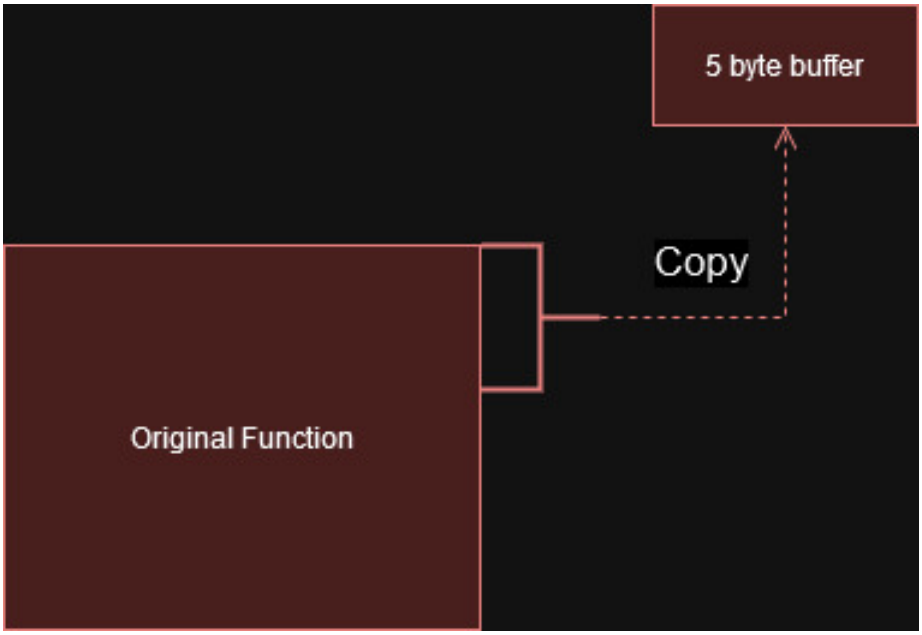

Figure 7: First 5 bytes of the target function are first copied to a buffer

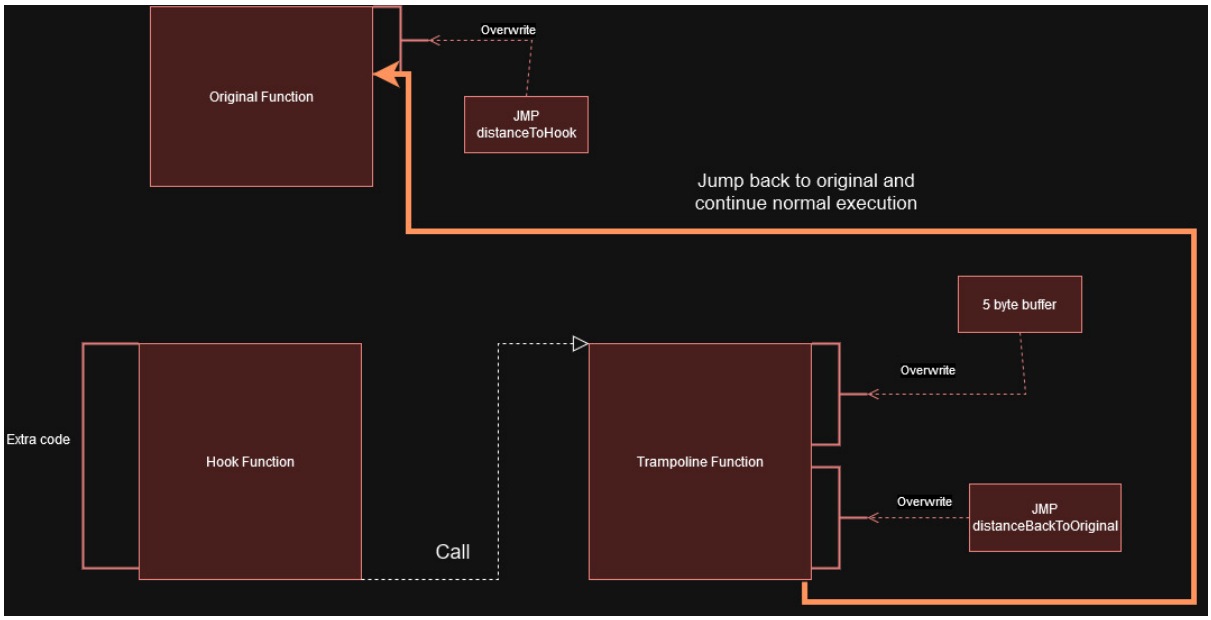

Figure 8: The first 5 bytes are overwritten with a relative jump instruction to the hook function. The hook function then calls a trampoline function that executes the borrowed instructions before returning execution to the original function

This method was initially going to implemented manually without the use of libraries. However, the basic example shown previously only works on 32 bit systems as the relative jump instruction only accepts 4 bytes as a parameter. Since pointers in 64 bit machines are 8 bytes, it would often be the case that not enough distance in memory could be jumped with a 4 byte number. Since creating a stable function hooking framework could be a large scale project in itself, Microsoft Detours was used to implement function hooking. A code segment showing the process of hooking a function in Detours can be seen below:

```
//Create a function pointer to the original API function
LPVOID(WINAPI* origVirtualAlloc)(LPVOID lpAddress, SIZE_T dwSize,
    DWORD flAllocationType, DWORD flProtect) = VirtualAlloc;

DetourTransactionBegin();
DetourAttach(&origVirtualAlloc, hookVirtualAlloc);
DetourTransactionCommit();

//Create a hook function that matches the parameters and calling
    convention (WINAPI or __stdcall) exactly
LPVOID WINAPI hookVirtualAlloc(LPVOID lpAddress, SIZE_T dwSize,
    DWORD flAllocationType, DWORD flProtect)
{
    //Modify parameters and add any additional code here
    if (PRINT_CALLS)
        std::cout << "Called VirtualAlloc\n";
    counterMap.at("VirtualAlloc")->incrementCall();
    //Call the original unmodified function using the function
        pointer created earlier
    return origVirtualAlloc(lpAddress, dwSize, flAllocationType,
        flProtect);
}
```

5.2.3 Inter-process Communication

Since monitored processes keep counts of calls to API functions in their address spaces, this data needs to be sent to the process monitor. Initially a message passing model using a named pipe was used. The monitor would create a named pipe for each monitored process and the monitored process would just open it like a file and write messages to it. This approach works, but it relies on a fixed buffer in the monitor and a file write call to the pipe can block if the buffer becomes full. This would cause a program to either freeze or fail since it may not have been designed to be blocked after calling a monitored Windows API call. After switching to a shared memory model, several system instability problems were solved. How shared memory was implemented in the system can be seen in the code segment below:

```
// ******** DLL main that runs inside target process ******** //
    //Open shared memory from monitor process

    //Used a base name of Local/APIMonitor
    //The path for each process is this base name with the PID appended
    std::string pipeName = pipeBaseName;
    pipeName.append(std::to_string(GetCurrentProcessId()));
    std::wstring temp = std::wstring(pipeName.begin(), pipeName.end());
    LPCWSTR fullString = temp.c_str();
    //Open a handle to the file mapping created by the monitor
    sharedMemoryHandle = OpenFileMappingW(FILE_MAP_ALL_ACCESS, FALSE,
        fullString);
    //Catch the case where the handle is null, would cause a crash on
        attached process
    if (sharedMemoryHandle == NULL)
    {
        return true;
    }
    //Use MapViewOfFile to map the file to a pointer so it can be
        accessed like a normal memory location. Casting the pointer to a
        C struct that contains integers for counts for all monitored
        API calls
    callCountContainer = (CallCountContainer*)MapViewOfFile(
        sharedMemoryHandle, FILE_MAP_ALL_ACCESS, 0, 0, sizeof(
        CallCountContainer));
// ******** Called in TrackedProcess constructor******** //
    //Create shared memory to pass counts from tracked process
    std::string pipeName = pipeBaseName;
  pipeName.append(std::to_string(PID));
  std::cout << pipeName << std::endl;
  pipeHandle = CreateFileMappingA(INVALID_HANDLE_VALUE, NULL,
      PAGE_READWRITE, 0, sizeof(CallCountContainer), pipeName.c_str());
  if (pipeHandle == NULL)
  {
    _tprintf(TEXT("Could not create file mapping (%d)\n"), GetLastError
        ());
  }
    //Map to pointer
  callCountContainer = (CallCountContainer*)MapViewOfFile(pipeHandle,
      FILE_MAP_ALL_ACCESS, 0, 0, sizeof(CallCountContainer));
```

5.3 Test Cases

5.3.1 Unit Tests

Project Name: Process Behavior Analysis	
Test Case 1	
Test Case ID: User Login	Test Designed by: Isaac Amann
Test Priority (Low/Medium/High): High	Test Designed Date: 3/29/2024
Module Name: User Authentication Service	Test Executed by: Isaac Amann
Description: Show that user authentication only issues token when given correct credentials	
Pre-conditions: System connected to Database with a test user already created	

Step	**Test Steps**	**Test Data**	**Expected Result**	**Actual Result**	**Status Pass/Fail**	**Notes**
1	Call login API function	Correct username and correct password	System provides a JWT for the target user	System provided a JWT for the target user	Pass	none
2	Call login API function	Correct username and bad password	System returns bad username or password error	System returns bad username or password error	Pass	none

Project Name: Process Behavior Analysis	
Test Case 2	
Test Case ID: Request Authentication	Test Designed by: Isaac Amann
Test Priority (Low/Medium/High): High	Test Designed Date: 3/29/2024
Module Name: User Authentication Service	Test Executed by: Isaac Amann
Description: Show that restricted API functions can only be called when provided a valid token	
Pre-conditions: System connected to Database with test users for each user role	

Step	**Test Steps**	**Test Data**	**Expected Result**	**Actual Result**	**Status Pass/Fail**	**Notes**
1	Call authenticateRequest function	Signed JWT with matching user role to the requiredUserRole parameter	authenticateRequest returns true	authenticateRequest returns true	Pass	none
2	Call authenticateRequest function	Signed JWT with mismatching user role to the requiredUserRole parameter	authenticateRequest returns false	authenticateRequest returns false	Pass	none
3	Call authenticateRequest function	JWT signed with the wrong key	authenticateRequest returns false	authenticateRequest returns false	Pass	none
4	Call authenticateRequest function	Malformed JWT	authenticateRequest returns false		Pass	none

Project Name: Process Behavior Analysis						
Test Case 3						
Test Case ID: Collection Client Authentication			Test Designed by: Isaac Amann			
Test Priority (Low/Medium/High): Medium			Test Designed Date: 3/29/2024			
Module Name: User Authentication Service			Test Executed by: Isaac Amann			
Description: Show that only collection clients can only post data points using valid API tokens						
Pre-conditions: Test client created with generated ID and API token						
Step	**Test Steps**	**Test Data**	**Expected Result**	**Actual Result**	**Status Pass/Fail**	**Notes**
1	Call isValidAPIToken on test client	Correct token	isValidAPIToken returns true	isValidAPIToken returns true	Pass	none
2	Call isValidAPIToken on test client	Incorrect token	isValidAPIToken returns false	isValidAPIToken returns false	Pass	none
3	Call isValidAPIToken on test client	Malformed token	isValidAPIToken returns false	isValidAPIToken returns false	Pass	none

5.3.2 System Tests

Project Name: Process Behavior Analysis						
Test Case 4						
Test Case ID: Datapoint Posting			Test Designed by: Isaac Amann			
Test Priority (Low/Medium/High): Medium			Test Designed Date: 3/29/2024			
Module Name: PostDataPointController			Test Executed by: Isaac Amann			
Description: Show that registered clients can post data points to the database through HTTP						
Pre-conditions: Server running and client software installed on Windows machine						
Step	**Test Steps**	**Test Data**	**Expected Result**	**Actual Result**	**Status Pass/Fail**	**Notes**
1	Execute system monitor on client machine	Valid credentials with a data point containing at least 1 counted API call	code 200, datapoint posts to database	code 200, datapoint posts to database	Pass	none
2	Execute system monitor on client machine	Valid credentials with a data point containing all 0's	code 200, datapoint posts to database	code 500, server error	Fail	Server also stores ratios of called Windows API functions. Passing 0 causes a division by 0 throwing an exception
3	Execute system monitor on client machine	Invalid credentials	code 200, Failed to authenticate	code 200, Failed to authenticate	Pass	none

Project Name: Process Behavior Analysis						
Test Case 5						
Test Case ID: Frontend User Login			Test Designed by: Isaac Amann			
Test Priority (Low/Medium/High): Medium			Test Designed Date: 3/29/2024			
Module Name: LoginController			Test Executed by: Isaac Amann			
Description: Show that registered users can login from the webpage						
Pre-conditions: Server running and client connected through web browser						
Step	**Test Steps**	**Test Data**	**Expected Result**	**Actual Result**	**Status Pass/Fail**	**Notes**
1	Submit login form	Valid login credentials	Login form closes and page state changes to indicate login	Login form closes and page state changes to indicate login	Pass	none
2	Submit login form	Invalid login credentials	User not given access	User not given access	Pass	Need to display error message on form to indicate the wrong credentials entered. Should also clear the form

5.4 Results

The API monitor is able to attach to processes and keep counts of their Windows API usage before posting the created data point to the server after the process exits. While the system instability created by the monitor has been greatly reduced, some programs still do not interact well with it. These include Firefox and the Windows Explorer search bar. Debugging this is difficult since the programs are unlikely to throw an error or create an error log in the Windows event viewer. Users of the system are also able to access the data set through the web dashboard and submit data analysis jobs.

5.5 Validation and Verification

Almost all of the requirements were satisfied with all medium and high priority requirements being fulfilled. Non-functional requirement 4 was not implemented, using SSL would require reconfiguring the build for the API monitor to include openssl in addition to other time consuming configuration for the host server, back-end application, and front-end application. If this project was deployed in a real world scenario for a long period of time, this would be necessary for security.

Non-functional requirement 1 was not implemented due to time constraints. However, it is likely possible to implement without altering the back-end code by creating a bash script that would just take snap shots of the database periodically.

Non-functional requirement 2 was not completely satisfied. Some programs still do not respond well to the API monitor and can freeze up. This is acceptable for a research tool to be used on dedicated test machines, but not for running in the background on an end user's machine. The original vision of the API monitor was that it could run in the background collecting data without affecting the user experience. In its current state, this is not the case but it could be improved in the future.

6 Conclusion

The system developed provides a tool set for collecting Windows API usage data from a large number of machines. Although the system can reliably collect Windows API usage data, it is too slow to create enough data to get good results from clustering in its current state. This could be improved

by creating automated tests that would continually run known malware samples on a virtual machine without constant supervision.

6.1 Future Work

6.1.1 Kernel Mode Driver

The system monitor was implemented as a user mode program. Re writing the monitor as a kernel mode driver would give it higher permissions, making it more difficult for malware at a lower permission level to avoid. Kernel mode drivers are also able to intercept new processes before they begin execution. This would allow functions to be hooked before the program begins, allowing fewer API executions to go uncounted and possibly improve stability.

6.1.2 API Monitoring VM Memory Inspection

The instability issues caused by the monitor are likely rooted in that the monitor is modifying code in other processes memory at run time. This also makes the monitor easy to detect and avoid. More advanced malware may include code to make it more difficult to analyze, either preventing monitors from attaching or failing to run if it detects a monitor running on the system. Another direction to take is to create a monitor that would inspect the memory of a virtual machine. This virtual machine would also be setup to allow its CPU to be stepped by the monitor, allowing the monitor to inspect the value of the program counter register to determine what function is being called by a process. My proposed system architecture can be seen in the following figure.

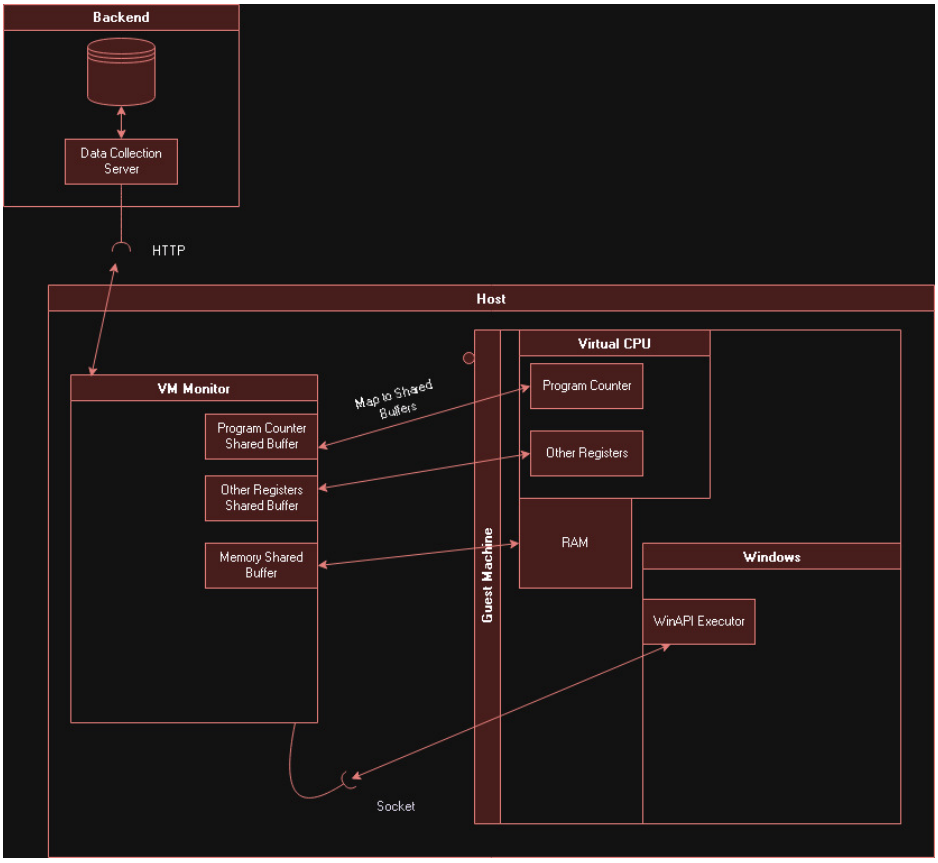

Figure 9: Proposed system architecture of alternative API monitoring system

In my reading on the topic of virtual machine introspection, I found that a similar system has already been developed. Dinaburg et al. provide a proof to show that their system is undetectable by malware running on the guest machine on page 53[1]. This is very intuitive as no components of their system are accessible to programs running on the host barring any virtual machine busting exploits. My proposed

architecture would not completely satisfy this as I intend for it to have a WinAPI Executor process running on the guest that would allow the monitor to execute code on the guest OS.

7 References

[1] Artem Dinaburg et al. "Ether: malware analysis via hardware virtualization extensions". In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS '08. Alexandria, Virginia, USA: Association for Computing Machinery, 2008, pp. 51–62. ISBN: 9781595938107. DOI: 10.1145/1455770.1455779. URL: https://doi.org/10.1145/1455770.1455779.

[2] Jayson Hurst. *Basic Windows API Hooking*. "https://medium.com/geekculture/basic-windows-api-hooking-acb8d275e9b8". 2021.

[3] Red Team Notes. *DLL Injection*. "https://www.ired.team/offensive-security/code-injection-process-injection/dll-injection". 2018.

[4] Andrew Steane. *Quick Introduction to Windows API*. https://users.physics.ox.ac.uk/~Steane/cpp_help/winapi_intro.htm. 2009.

8 Appendices

8.1 Progress report for Milestone 1

The monitor program is mostly complete and work on the back-end/collection server has been started. Some basic user authentication and data collection API calls have been implemented on the back-end server. The client is also able to post data points after a process has exited.

In addition to work on my project, I have worked on a couple utility programs for work on the Sow and Grow algorithm as well as started exploring solutions for load balancing between threads. These two utility programs include a cluster visualizer script in Python and a Bitmap to CSV utility written in C so testing data sets can be created from the pixel data of images.

8.1.1 Work Log

- 1/25/2024
  - Created a separate thread in attached processes to periodically send messages to monitor instead of making a WriteFile call within the hooked function to reduce the risk of blocking a process.
- 1/26/2024
  - Wrote function hooks for wsock32 API functions.
- 1/27/2024
  - Started writing function hooks for Advapi32 API functions.
- 1/29/2024
  - Wrote more function hooks.
- 2/1/2024
  - Started hooking User32 API calls.
- 2/11/2024
  - Finished writing function hooks for all API calls monitored by the system.
- 2/14/2024
  - Switched from using a named pipe to shared memory for inter-process communication. Updated constructor for apiCallCounter to work with this change.

- 2/17/2024
  - Created prompt on monitor program to get UUID and API key from user. The credentials are stored on the Windows Registry and the user is not prompted again if they already exist.
- 2/18/2024
  - Created README file for project main page

8.2 Progress report for Milestone 2

8.2.1 Back-end Server

- User Authentication
  - User authentication was initially done by providing a session token on login that would be stored in the database along with an expiration date. Instead of using simple generated tokens, I switched to using JWT. The keys are signed with a private key, so the server can grant access to resources without querying the database as long as the key is valid.
- Page Requests
  - API calls were created that allows clients to request portions of the dataset providing a page number and page size. This is needed for the dataset explorer as once the dataset is large enough, sending the entire dataset all at once would not be practical. Page request API calls were also created for user accounts and registered collection clients for user in the admin dashboard on the frontend.

8.2.2 Data Analysis Job System

The analysis job system was setup such that jobs can be submitted through an API request. An entry for the job is posted to the database containing information likes its status, time started, time finished, analysis type, parameters, and the generated clusters. An abstract class for the analysis method was written so different clustering algorithms could be used in the system. Analysis using the SowAndGrow algorithm being developed by Dr. Che was implemented by creating input files and then executing the compiled program by using the Runtime.exec() Java method. The output could then be saved to the database by parsing the output files generated by the SowAndGrow program.

8.2.3 Front-end

- Main Page
  - The main page provides the project description and a sample of the dataset within a DataGrid component. This page acts as a landing page allowing users to login.
- Dataset Explorer
  - The dataset explorer was implemented using a DataGrid component. DataGrid also supports server-side pagination, so I was able to take advantage of the page request API calls for dataset access that I had previously written.
- Admin Dashboard
  - The link to the admin dashboard is only displayed to users with the correct user role. The page currently only contains a command shell for the server. The implementation for the command shell is mostly taken from some of my previous work on another project with some alterations to work with the user authentication system. The command shell API call exposes Java methods within the ShellCommands class. These methods are able to be invoked when running the server application from the command line, but connecting over SSH and trying to bring the server application into the foreground so that commands can be entered is inconvenient. I plan on changing the frontend command shell component to something more visually appealing and convenient to use, right now it is just a text area element with a text field for entering commands. This page will also include system statistics, collection client info, and a log viewer in the future.

- Data Analysis Page
  - This page contains a DataGrid showing previously submitted jobs with their information. Jobs can also be selected and inspected to show more detailed information including the clusters that were generated by the clustering algorithm used. There is also a form for submitting new jobs to the server.

8.2.4 Project Deployment

The project has been deployed to AWS using EC2 to create a Ubuntu Linux virtual machine to run on and Route 53 for DNS. Both the backend and frontend server software are managed as a Systemd service so that they are ran in the background on server startup. I also purchased the domain name winapimonitoring.com to route traffic to my server.

8.2.5 Work Log

- 2/25/2024
  - Fixed Windows API Monitor program overwritting credentials retrieved from the Windows Registry. Implemented JWT user authentication and updated other classes to support JWT on the backend server.
- 2/27/2024
  - Created API requests providing pageing for datapoints, user accounts, and registered collection clients.
- 3/1/2024
  - Generated React project for frontend
- 3/2/2024
  - React setup
- 3/3/2024
  - Added frontend pages
- 3/4/2024
  - Frontend Changes:
    - Added navbar, login form, profile menu with logout button, and cookies for storing session token after login
  - Backend Changes:
    - Added API call for getting a sample of the dataset accessible to unauthenticated users
- 3/5/2024
  - Frontend Changes:
    - Created sample dataset viewer. Created pages for dataset explorer and data analysis. Started dataset explorer component.
- 3/12/2024
  - Frontend Changes:
    - Set up server side pagination for dataset explorer. Set fixed page size on dataset explorer
  - Backend Changes:
    - Created shell command to create test data points for testing the dataset explorer

- 3/14/2024
  - Backend Changes:
    * Began work on the data analysis system
- 3/15/2024
  - Backend Changes:
    * More work on data analysis system. Began implementing RunSowAndGrow class.
- 3/16/2024
  - Backend Changes:
    * Finished implementing RunSowAndGrow class.
- 3/17/2024
  - Backend Changes:
    * Created DataAnalysisController class to contain REST controllers for data analysis API calls
- 3/23/2024
  - Backend Changes:
    * Implemented API calls for data analysis job access
- 3/23/2024
  - Backend Changes:
    * Implemented API calls for data analysis job access
- 3/24/2024
  - Frontend Changes:
    * Added admin dashboard with command shell. Added analysis job submit form. Added button for inspecting selected analysis jobs. Created data analysis job table
  - Project Deployment:
    * Purchased domain name from AWS. Created EC2 instance running Ubuntu Linux. Basic system setup. Created bash script for launching server and Systemd service for executing script on startup.

8.3 Progress report for Milestone 3

8.3.1 Back-end Server

- API Rate Limits
  - Implemented a token bucket API rate limiter using the bucket4j library. Should prevent bad actors from brute forcing passwords and API tokens. The limits were set to 15 per minute for login requests and 200 per minute for regular API calls
- DBSCAN with Cosine Similarity
  - Created a python script to be run on a separate Java thread on the backend similar to how the SowAndGrow program was executed. Originally intended on finding a Java library for the clustering algorithm, but most libraries only had DBSCAN using euclidean distance. The sklearn Python library supports DBSCAN with cosine similarity and was simple to implement in under 50 lines of code.
- Logging System
  - Implemented a logging system that stores log entries in memory that can be accessed through API calls. I wrote it to look similar to the default logs that come with the Spring framework with fields for log level, source, and message. This allows information about events on the system to be viewed from the admin dashboard on the frontend.

8.3.2 Front-end Server

- Admin Dashboard
  - information about the host system including hostname, system architecture, and kernel version.
  - Memory usage gauge
  - Tab menu containing tables for registered clients, users, and log entries
  - The styling of the command console was also changed to be more visually appealing. The text input was also changed so that it keeps focus on the element after entering a command without the user having to click on the text box again.

8.3.3 Work Log

- 3/29/2024
  - Backend Changes:
    - Implemented unit tests for authentication methods
- 3/31/2024
  - Frontend Changes:
    - Updated styling of command console. Added ability to submit analysis jobs for DBSCAN with cosine similarity.
  - Backend Changes:
    - Created python script for DBSCAN with cosine similarity. Implemented logging service and API functions for accessing log entries.
- 4/1/2024
  - Frontend Changes:
    - Began work on system monitor component
- 4/2/2024
  - Frontend Changes:
    - Setup memory usage gauge. Implemented CPU usage graph. Added system information to system monitor component.
- 4/3/2024
  - Backend Changes:
    - Fixed parameters used for mpstat command for system info API call.
- 4/6/2024
  - Frontend Changes:
    - Created log viewer table on admin dashboard. Added tab menu to admin dashboard. Created client datagrid to admin tab menu. call.
- 4/10/2024
  - Frontend Changes:
    - Added documentation page with PDF downloads. Added user list to admin tab menu.
- 4/11/2024
  - Backend Changes:

        * Fixed date formats on JSON objects returned by API calls

- 4/14/2024
    - Frontend Changes:
        * Added notifications for login and job submission
    - Backend Changes:
        * Implemented API rate limits
    - API Monitor Changes:
        * Excluded UWP applications from monitoring. UWP (Windows Store) applications use some sort of sandboxing and crash upon DLL injection.

- 4/17/2024
    - Frontend Changes:
        * Changed job viewer to allow for scrolling