



Concordia Institute for Information System Engineering (CIISE)

INSE 6140 Malware Defenses and Application Security

Project Topic:

**Jython script for Ghidra to detect DLL Injection in Gaming**

Submitted to:

**Professor Dr. Makan Pourzandi**

Submitted By:

Student Name	Student ID
Aniket Agarwal	40266485
Kalyani Batle	40243967
Aathira Dineshan	40270695

**Date: 22/04/2024**

# INTRODUCTION

## 1. Context

Dynamic Link Libraries (DLLs) are collections of small programs that larger programs can load when needed to perform specific tasks. These tasks may include communicating with devices, handling non-core functions, or providing additional functionalities like graphics enhancements or networking support. DLLs contain code, data, and resources that can be shared among multiple programs, promoting modularity and code reuse.

DLLs work through dynamic linking, where the code is linked to a program at runtime rather than embedded directly into the executable. This allows for memory efficiency, fewer faults, and a modular architecture where programs can be delivered and updated more easily. However, dynamic linking also introduces challenges such as dependency errors, security vulnerabilities like DLL injection, and potential performance overhead.

DLL injection is a technique used to insert a DLL into the address space of a process running on a computer. The combination of `OpenProcess`, `VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemoteThread` is a common technique used for DLL injection into a remote process in Windows. By combining these functions, a DLL can be injected into a remote process, allowing arbitrary code execution within that process.

## 2. Problem and Related Issues

DLL injection is commonly used for various purposes, including hooking API calls, injecting game cheats, or performing process injection for malware. Some common problems related to DLL injection are:

- It's very hard to detect the malware due to the stealthy nature of DLL injection, i.e., injecting a malicious code to run within a running parent process.
- In the gaming industry it's pretty common for gamers to create mods and cheat codes using DLL injection to gain advantage over other players.
- Users usually trust an unknown website and download a game zip archive having DLL injector and malicious DLL in the game folder that gets executed when the user runs the game.

To tackle these issues, there is a growing need to develop a detection mechanism to detect a malicious DLL injector file using static analysis and prevent execution of DLL injection attack.

## 3. Objectives

- To demonstrate a sample DLL injection attack using a custom DLL injector executable in a open source game Wesnoth v1.14.9
- To analyze the DLL injector executable and Wesnoth game executable using the open source reverse engineering tool Ghidra.
- To implement a jython script for Ghidra to run a static analysis on the DLL injector suspicious executable and detect DLL injection code in it.

## 4. Motivation

- The dynamic analysis is usually not much fruitful due to the stealthy nature of DLL injection within a running process and hence, this project aims to run a static analysis on a suspicious DLL injector executable and detect DLL injection code in it to prevent execution of it.
- Ghidra is an open-source tool used for reverse engineering of binaries and allows integration of scripts and plugins to extend functionalities. This project aims to build a jython script for Ghidra to detect a DLL injection in an executable.

## 5. Test Environment Used

- Oracle VM Virtual Box v6.1.26
- Windows 10 image (for VM)

## **5. Software Tools and Libraries Used**

- Microsoft Visual Studio 2022 (for building malicious DLL injector executable)
- Ghidra v11.0.3 (for static analysis of executable binary)
- jython 3.0 (for developing the DLL detection script)
- Java library in jython 3.0

# SCRIPT ALGORITHM DESIGN

The following functions are commonly found in the DLL Injectors. The proposed jython script considers these target functions to identify the potential DLL Injector.

1. OpenProcess
2. VirtualAllocEX
3. WriteProcessMemory
4. CreateRemoteThread
5. CloseHandle

Below is the pseudocode for the proposed jython script for Ghidra analysis that warns the user about the potential DLL Injection attack in an executable.

1. Define target\_values array with function names: ['OpenProcess', 'VirtualAllocEx', 'WriteProcessMemory', 'CreateRemoteThread', 'CloseHandle'].
2. Define an array found\_array having the same length as target\_values.
3. Initialize found\_array with False values,
4. Get the current function and instruction addresses.
5. Iterate over each instruction in the current function:
  - a. If the instruction is a CALL:
    - i. Extract the function symbol.
    - ii. Check if it matches any entry in target\_values.
    - iii. If matched:
      1. Set the corresponding entry in found\_array to True.
      2. Highlight the CALL instruction in the Listing window.
      3. Print the mnemonic and function name in the Ghidra console.
6. Check if all entries in found\_array are True.
7. If True for all:
  - a. Print "This binary may be vulnerable to DLL Injection attack" to the Ghidra console.
8. End the process.

The Jython script is uploaded and implemented using the Ghidra Script Manager. The entire source code of the proposed script is attached in Appendix B.

# ATTACK IMPLEMENTATION

The game chosen for implementing the DLL injection attack is **Wesnoth v1.14.9**.

An attacker can inject a malicious DLL in many ways and one way is to use a DLL injector executable. The DLL injector when executed by the victim injects the malicious DLL stored in the victim's system (bundled together with game files) into the running game.

This project aims to first demonstrate the making of malicious DLL injector executable and a sample DLL which when injected in Wesnoth game displays an error message followed by it's detection in Ghidra using a jython script developed in this project.

## Part I: Building a Sample DLL

The DLL is written in C++ language using Microsoft Visual Studio 2022. The configuration of the project type is changed to DLL so that it is built as DLL and not as an executable.

Some basic differences between DLL and a normal executable are as follows:

- DLL's have a `DllMain` function instead of a `main` function.
- The `DllMain` function is called when a process loads or unloads DLL.
- DLL's run inside their parent process and the variables declared inside DLL's are stored in their parent memory.

```
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved) {
```

Fig 1. DllMain function syntax

The `fdwReason` parameter in Fig 1. contains the reason that the `DllMain` function was called by it's parent process. When the DLL is loaded by a process, this parameter holds the value of 1. Inside this function, a Message Box is called to display an error message to indicate that the DLL was loaded successfully as shown in Fig 2.

```
    MessageBox(0, (LPCWSTR)L"DLL injected successfully!",  
               (LPCWSTR)L"Alert", MB_ICONWARNING);
```

Fig 2. Message Box to Display an Error

## Part II: Building a Sample DLL Injector Executable

The DLL gets loaded by the windows executables using the `LoadLibraryA` API function. This function takes a single argument which is the full path to the DLL to load. A process handle is needed to interact with any external process. The API `CreateToolhelp32Snapshot` takes a snapshot of all running process in the system. Each process is then examined using `Process32First` and `Process32Next`. Fig 3. shows how to iterate over each running process to find the target process.

```

HANDLE snapshot = 0;
PROCESSENTRY32 pe32 = { 0 };

DWORD exitCode = 0;

pe32.dwSize = sizeof(PROCESSENTRY32);
snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
Process32First(snapshot, &pe32);

do {
    ...
} while (Process32Next(snapshot, &pe32));

```

Fig 3. Process handle

Each process entry has two fields: *szExeFile* and *th32ProcessID*. The former contains the name of the process as shown in the task manager and the latter contains the process identifier of the process. The name of the Wesnoth process as identified from task manager is Wesnoth.exe. This is compared against the *szExeFile* using *strcmp* function as shown in Fig 4.

```

(wcsncmp(pe32.szExeFile, L"wesnoth.exe") == 0)

```

Fig 4. Process name identification

Next, the value in *th32ProcessID* is passed into the *OpenProcess* function which will create a new process to inject DLL into the Wesnoth.exe as shown in Fig 5.

```

HANDLE process = OpenProcess(PROCESS_ALL_ACCESS, true, pe32.th32ProcessID);

```

Fig 5. OpenProcess function syntax

Next, the memory is allocated inside Wesnoth game process to store the full path of malicious DLL using function *VirtualAllocEX*. This function returns a void pointer containing the address of the allocated memory. This pointer is stored in a variable and passed as a parameter to the function *WriteProcessMemory* which writes the path of DLL in the allocated memory as shown in Fig 6.

```

void* lpBaseAddress = VirtualAllocEx(process, NULL, strlen(dll_path) + 1, MEM_COMMIT, PAGE_READWRITE);

WriteProcessMemory(process, lpBaseAddress, dll_path, strlen(dll_path) + 1, NULL);

```

Fig 6. VirtualAllocEx and WriteProcessMemory function syntax

The address of *LoadLibraryA* is obtained which is inside *kernel32.dll* using *GetModuleHandle* and *GetProcAddress* function API's and a parallel thread is created using *CreateRemoteThread* API to call *LoadLibraryA* to load the DLL into Wesnoth.exe as shown in Fig7.

```

HMODULE kernel32base = GetModuleHandle(L"kernel32.dll");
HANDLE thread = CreateRemoteThread(process, NULL, 0, (LPTHREAD_START_ROUTINE)GetProcAddress(kernel32base, "LoadLibraryA"), lpBaseAddress, 0, NULL);

```

Fig 7. GetModuleHandle and CreateRemoteThread function syntax

Finally, the allocated memory is freed using the *VirtualFreeEx* function on successful DLL injection and the handle is closed using *CloseHandle* function followed by a break to come out of loop scanning through each running process on the system as shown in Fig 8.

```
VirtualFreeEx(process, lpBaseAddress, 0, MEM_RELEASE);  
CloseHandle(thread);  
CloseHandle(process);  
break;
```

Fig 8. VirtualFreeEx and CloseHandle syntax

Now, the injector and the game are executed and the DLL gets injected into Wesnoth process successfully as shown in Fig 9.

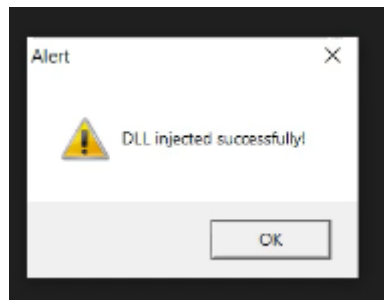


Fig 9. Successful DLL injection

# DETECTION IMPLEMENTATION

A Jython script (refer Appendix B) was designed as per the pseudocode (refer Algorithm Design) in the Visual Studio Code. This was then planted within the folder of the game. The following steps below show how it was integrated into Ghidra and run to detect the target functions that help in identifying a potential DLL attack.

1. The current chosen executable file in the Ghidra is the DLL Injector.

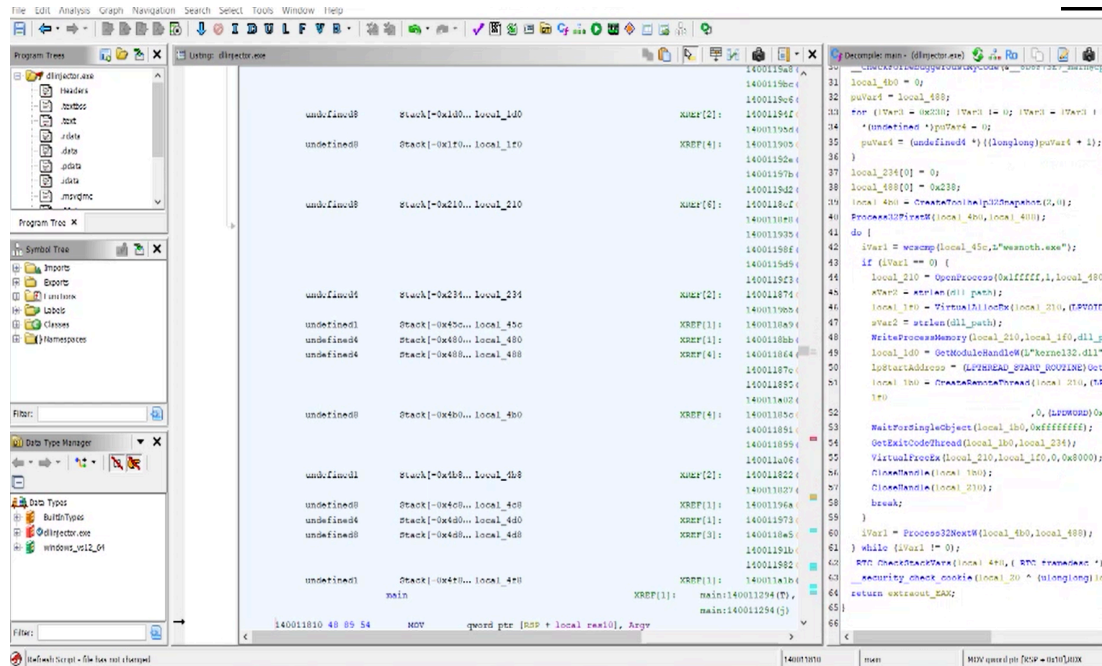


Fig 10. Ghidra with DLL Injector as the current file

2. The Jython script is then loaded into Ghidra's script manager and selected to execute.

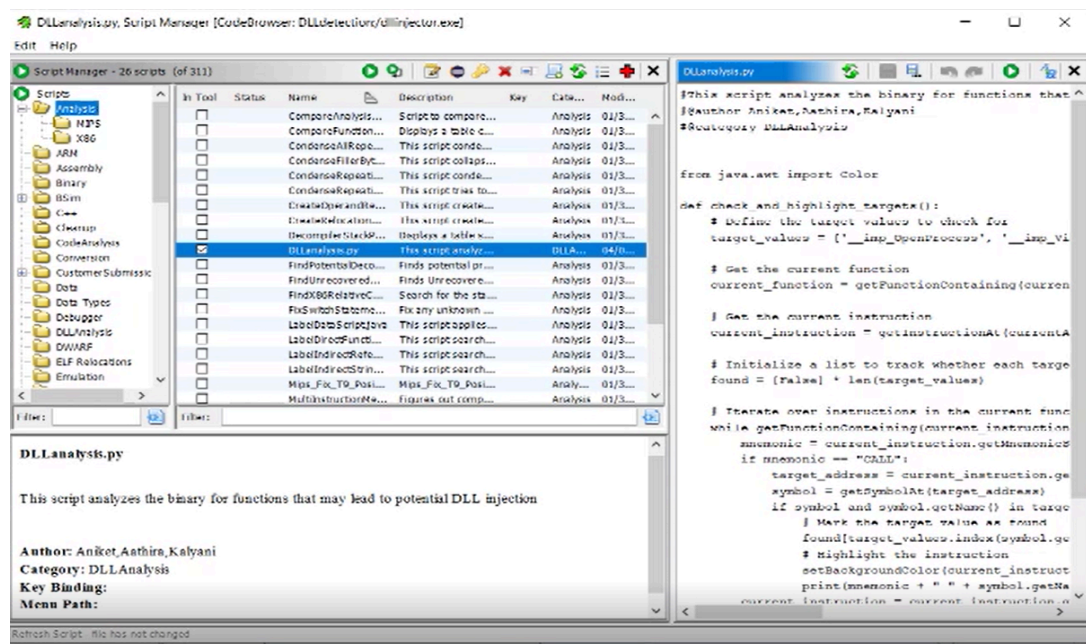


Fig 11. Jython script uploaded and chosen to run



- Once the script runs, if there are target functions, the following are printed on the console.



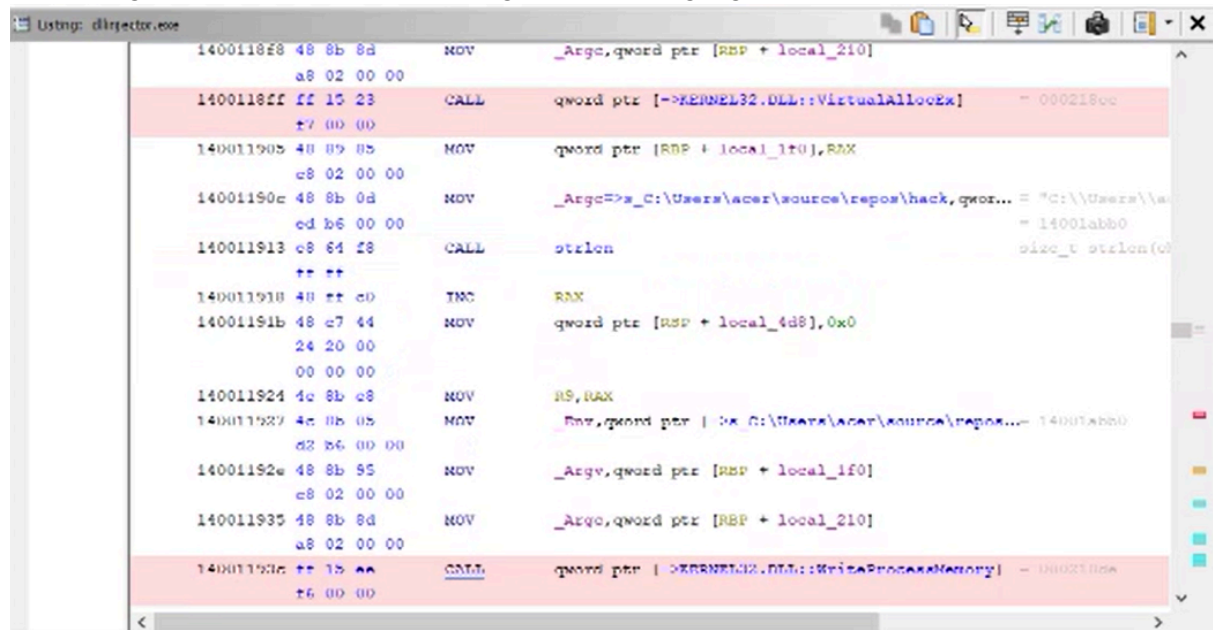
```

Console - Scripting
CALL __imp_CloseHandle
This binary maybe vulnerable to DLL Injection attack
DLLanalysis.py> finished!
DLLanalysis.py> Running...
CALL __imp_OpenProcess
CALL __imp_VirtualAllocEx
CALL __imp_WriteProcessMemory
CALL __imp_CreateRemoteThread
CALL __imp_CloseHandle
CALL __imp_CloseHandle
This binary maybe vulnerable to DLL Injection attack
DLLanalysis.py> finished!

```

Fig 12. Console when target functions are detected

- The Listing Window will also have the target functions highlighted.



```

Listing: dirsect.exe
1400118f8 48 8b 8d MOV     _Argc,qword ptr [RBP + local_210]
a8 02 00 00
1400118ff ff 15 23 CALL    qword ptr [->KERNEL32.DLL::VirtualAllocEx] - 000218cc
+7 00 00
140011905 48 8b 85 MOV     qword ptr [RBP + local_1f0],RAX
a8 02 00 00
14001190c 48 8b 0d MOV     _Argv,qword ptr [C:\Users\acer\source\repos\hack, qwor... = "C:\Users\acer\source\repos\hack, qwor...
ed b6 00 00 = 14001abb0
140011913 c8 64 28 CALL    strlen size_t strlen(0)
+2 +2
140011918 48 ff c0 INC     RAX
14001191b 48 c7 44 MOV     qword ptr [RBP + local_4d8],0x0
24 20 00
00 00 00
140011924 4c 8b c8 MOV     RS,RAX
140011929 4c 8b 05 MOV     _Env,qword ptr [C:\Users\acer\source\repos\hack, qwor... = 14001abb0
d2 b6 00 00
14001192e 48 8b 55 MOV     _Argv,qword ptr [RBP + local_1f0]
c8 02 00 00
140011935 48 8b 8d MOV     _Argc,qword ptr [RBP + local_210]
a8 02 00 00
14001193e ff 15 aa CALL    qword ptr [ >KERNEL32.DLL::WriteProcessMemory] - 000218cc
+6 00 00

```

Fig 13. Listing window with target functions highlighted

# RESULTS

- The script successfully detects 8/10 instances of *Remote Thread Injection attacks* by identifying the presence of *CreateRemoteThread function* calls and analyzing DLL loading behavior and thread execution flow.
- DLL injection detection rate: **80%**
- False Negative rate: **20%** (2/10 instances not detected)

Below are the list of 10 executables used for testing the proposed script and their respective result for DLL injection detection.

SR NO	TESTED DLL INJECTOR EXECUTABLES	RESULT
1.	<a href="https://github.com/GameHackingAcademy/DLL_Injector">https://github.com/GameHackingAcademy/DLL_Injector</a>	<b>DETECTED</b>
2.	<a href="https://github.com/houjingyi233/dll-injection-by-CreateRemoteThread/blob/master/Source.cpp">https://github.com/houjingyi233/dll-injection-by-CreateRemoteThread/blob/master/Source.cpp</a>	<b>DETECTED</b>
3.	<a href="https://github.com/3gstudent/Inject-dll-by-APC/blob/master/CreateRemoteThread.cpp">https://github.com/3gstudent/Inject-dll-by-APC/blob/master/CreateRemoteThread.cpp</a>	<b>DETECTED</b>
4.	<a href="https://github.com/lem0nSec/CreateRemoteThreadPlus/blob/master/src/CreateRemoteThreadPlus.c">https://github.com/lem0nSec/CreateRemoteThreadPlus/blob/master/src/CreateRemoteThreadPlus.c</a>	<b>DETECTED</b>
5.	<a href="https://github.com/freakanonymus/DLL_ROOTKIT_loader/blob/main/dll_rootkit_loader.cpp">https://github.com/freakanonymus/DLL_ROOTKIT_loader/blob/main/dll_rootkit_loader.cpp</a>	<b>DETECTED</b>
6.	<a href="https://github.com/S3cur3Th1sSh1t/Creds/blob/master/Csharp/CreateRemoteThread.cs">https://github.com/S3cur3Th1sSh1t/Creds/blob/master/Csharp/CreateRemoteThread.cs</a>	<b>DETECTED</b>
7.	<a href="https://github.com/pwndizzle/c-sharp-memory-injection/blob/master/process-dll-injection.cs">https://github.com/pwndizzle/c-sharp-memory-injection/blob/master/process-dll-injection.cs</a>	<b>DETECTED</b>
8.	<a href="https://github.com/mlgualtieri/PurpleTeamSummit/blob/main/Summit-May2021/DllHijack/dllmain.cpp">https://github.com/mlgualtieri/PurpleTeamSummit/blob/main/Summit-May2021/DllHijack/dllmain.cpp</a>	<b>DETECTED</b>
9.	<a href="https://github.com/glitteru/CodMWKernelInjector/blob/main/Injection/injector.h">https://github.com/glitteru/CodMWKernelInjector/blob/main/Injection/injector.h</a>	<b>UNDETECTED</b>
10.	<a href="https://github.com/wolk-1024/CrossInject/blob/master/Inject.cpp">https://github.com/wolk-1024/CrossInject/blob/master/Inject.cpp</a>	<b>UNDETECTED</b>

# CONCLUSION

In conclusion, the development of the innovative detection script for Ghidra marks a significant milestone in the ongoing effort to combat DLL-based threats with accuracy and efficacy. The script's ability to recognise between benign and malicious DLLs, coupled with its low false positive rate fortifies the security in gaming industry operating in dynamic environment.

Future iterations of the script will undoubtedly benefit from continued research and development by leveraging advancements and refinement in detection algorithms , threat intelligence integration to provide real-time updates on emerging threats, expansion in incident response capabilities to enable effective mitigation of security incidents, and raising awareness in gamers through education programs to recognize and report suspicious activity, are some of the potential areas for further exploration. The script iteration will strengthen the security posture of the gaming industry and mitigate the risks associated with DLL-based threats.

## REFERENCES

1. Process Injection: Dynamic-link Library Injection, Sub-technique T1055.001 - Enterprise | MITRE ATT&CK®. (n.d.). Attack.mitre.org. <https://attack.mitre.org/techniques/T1055/001/>
2. DLL Injector · Game Hacking Academy. (n.d.). Gamehacking.academy. Retrieved April 22, 2024, from <https://gamehacking.academy/pages/7/01/>
3. DLL Memory Hack · Game Hacking Academy. (n.d.). Gamehacking.academy. Retrieved April 22, 2024, from <https://gamehacking.academy/pages/3/03/>
4. tisker.dll injection with Ghidra. (n.d.). Www.youtube.com. Retrieved April 22, 2024, from <https://youtu.be/DlbF6J5iuzg?si=UGmq7XpL6Fjt2VtO>
5. Retro Gaming Vulnerability Research: Warcraft 2. (2023, December 19). NCC Group Research Blog. <https://research.nccgroup.com/2023/12/19/retro-gaming-vulnerability-research-warcraft-2/>
6. guided-hacking/GuidedHacking-Injector. (2024, April 22). GitHub. <https://github.com/guided-hacking/GuidedHacking-Injector>
7. isaratech. (2021, November 18). CPP - DLL Injection using CreateRemoteThread on Windows. Isara Tech. <https://isaratech.com/cpp-dll-injection-using-createremotethread-on-windows/>
8. CyberWire, T. (2019, December 3). Mike Bell: Extending Ghidra: from Script to Plugins and Beyond. Vimeo. <https://vimeo.com/377180466/description>
9. GitHub & BitBucket HTML Preview. (n.d.). Htmlpreview.github.io. <https://htmlpreview.github.io/?https://github.com/NationalSecurityAgency/ghidra/blob/stable/GhidraDocs/InstallationGuide.html>

# APPENDIX A

## Team Members Individual Contributions :

Team Members	Student ID	Contribution
Aniket Agarwal	40266485	Coding DLL Injector for Wesnoth game, Integration of Jython script with Ghidra tool, Ghidra script testing  Report: Final editing, Attack Implementation
Kalyani Batle	40243967	Research and analysis, DLL Injection execution on Wesnoth game, Jython script testing in Ghidra  Report: Introduction, Result, Conclusion
Aathira Dineshan	40270695	Research and analysis, Detection Implementation with Jython script  Report: Script Algorithm Design, Detection Implementation, References

## APPENDIX B

Source code for the proposed Ghidra Jython script to detect DLL detection in an executable:

```
from java.awt import Color

def check_and_highlight_targets():
    # Define the target values to check for
    target_values = ['OpenProcess', 'VirtualAllocEx',
'WriteProcessMemory', 'CreateRemoteThread', 'CloseHandle']

    # Get the current function
    current_function = getFunctionContaining(currentAddress)

    # Get the current instruction
    current_instruction = getInstructionAt(currentAddress)

    # Initialize a list to track whether each target value is found
    found = [False] * len(target_values)

    # Iterate over instructions in the current function
    while getFunctionContaining(current_instruction.getAddress()) ==
current_function:
        mnemonic = current_instruction.getMnemonicString()

        if mnemonic == "CALL":
            target_address = current_instruction.getOpObjects(0)[0]
            symbol = getSymbolAt(target_address)

            if symbol:
                # Check if any of the target values are substrings of
the symbol name
                for i, target_value in enumerate(target_values):
                    if target_value in symbol.getName():
                        # Mark the target value as found
                        found[i] = True

                        # Highlight the instruction

setBackgroundColor(current_instruction.getAddress(),
    Color(255, 220, 220))
```

```
        print(mnemonic + " " + symbol.getName())

    current_instruction = current_instruction.getNext()

# Check if all target values are found
if all(found):
    print("This binary maybe vulnerable to DLL Injection attack")

# Call the function to check and highlight target values
check_and_highlight_targets()
```