파일포렌식

19반 정민석

목차

1. 환경 구성

먼저 과제는 SPY++, API Monitor, Notepad를 통하여 수행해야했습니다. 이때, SPY++, API Monitor는 x64, x32 등 arm64가 아닌 환경에서 정상 작동하는 응용프로그램입니다. 하지만 MAC m1 UTM 가상환경에서 Window x64를 emulating하여 실행 시, 정상적으로 과제를 수행할 수 없을 정도의 성능저하가 발생하였습니다. 따라서 SPY++, API Monitor를 Frida로 대체하였습니다. SPY++은 메세지를 분석하는 용도로, API Monitor는 API 분석 목적으로 활용되기에, Frida로 해당 기능을 구현하여 과제를 수행하였습니다.

Notepad는 arm64기반으로 동작한다는 것을 확인하였습니다. 하지만 Frida의 경우, Window arm64의 빌드 버전을 출시하지 않았습니다.[1] Window x86_64의 Frida로 arm64 Notepad를 hooking 시, 헬퍼함수가 없다는 오류 메세지가 출력되었습니다. 따라서 amd64기반의 notepad++의 응용프로그램으로 Notepad를 대체하였습니다.

[1] Frida 릴리즈 버전 목록 (https://github.com/frida/frida/releases)

2. 실행파일의 윈도우 메세지 분석

먼저 윈도우 메세지 공식 문서[2]에서 어떤 메세지가 있는지 확인하였습니다. 하지만 notepad++은 내부적으로 Scintilla를 사용하였습니다. Scintilla 공식문서[3]와 각 메세지를 식별하는 값을 찾기 위하여 배포 Scintilla Github[4]를 참고하였습니다. 이를 토대로 아래의 코드를 작성하였습니다.

- message_hook.py

```
import frida
import sys

script_code = open("message_hook.js", encoding="utf-8").read()

session = frida.attach("notepad++.exe")
def on_message(message, data):
    if message["type"] == "send":
        print("[*] Message from script:", message["payload"])
    elif message["type"] == "error":
        print("[!] Error:", message["stack"])

script = session.create_script(script_code)
script.on('message', on_message)
script.load()
session.resume()
sys.stdin.read()
```

- message_hook.js

```
const apiNames = [
  "SendMessageW",
  "PostMessageW",
  "DispatchMessageW",
  "GetMessageW",
```

---

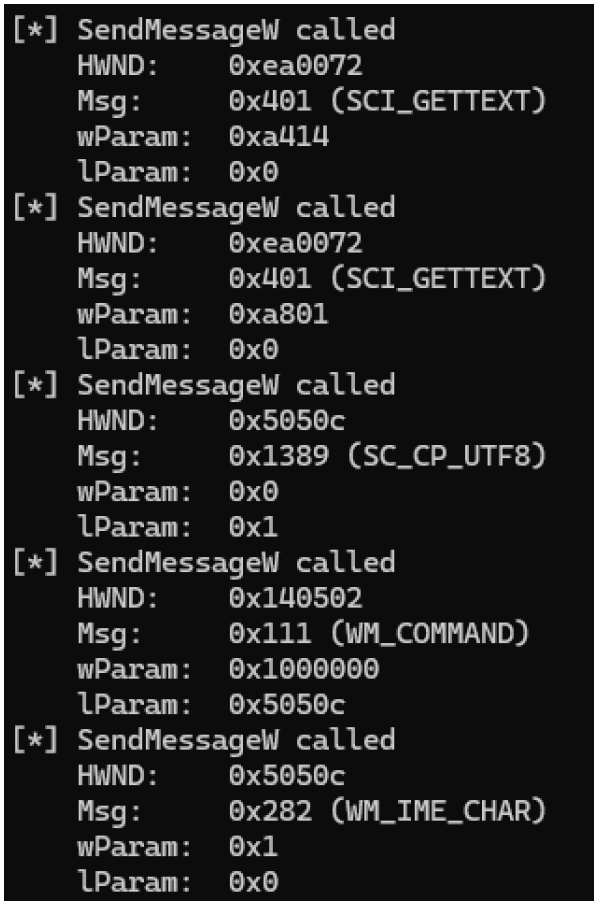
[2] 윈도우 메세지 공식 문서 (https://learn.microsoft.com/en-us/windows/win32/winmsg/about-messages-and-message-queues)

[3] Scintilla 공식문서 (https://www.scintilla.org/ScintillaDoc.html)

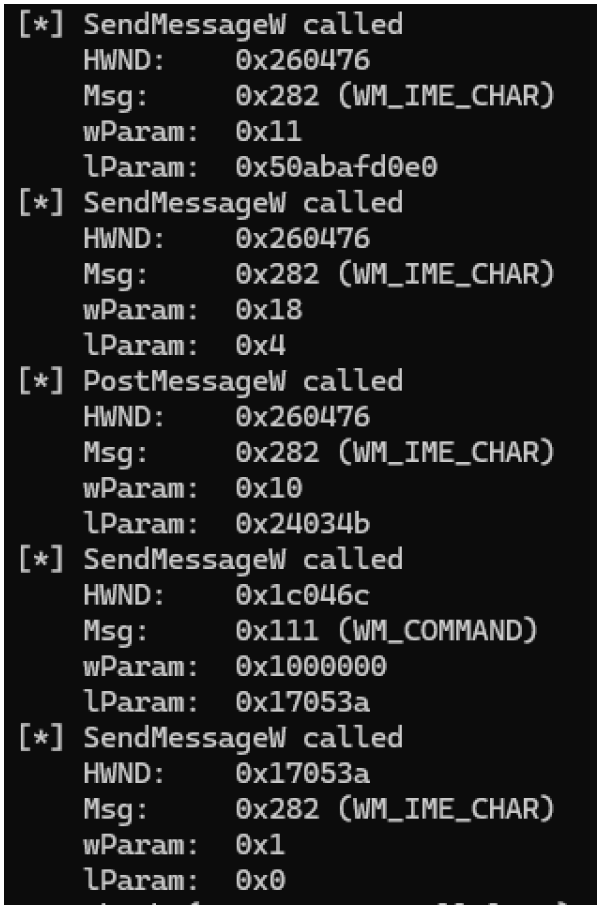
[4] Scintilla 메세지 정보 (https://github.com/LuaDist/scintilla/blob/master/include/Scintilla.h)

```
  "PeekMessageW",
  "SetClipboardData",
  "GetClipboardData",
  "TranslateMessage",
];

function getMessageName(msg) {
  const id = msg.toInt32();
  const messageMap = {
    0x0111: "WM_COMMAND",
    0x0282: "WM_IME_CHAR",
    0x0401: "SCI_GETTEXT",
    0x07d3: "SCI_INSERTTEXT",
    0x0882: "SCI_COPY",
    0x0883: "SCI_PASTE",
    0x0885: "SCI_SETTEXT",
    0x1389: "SC_CP_UTF8",
  };

  return messageMap[id] || null; // "UNKNOWN (0x" + id.toString(16) + ")";
}

apiNames.forEach(function (name) {
  try {
    const addr = Module.getExportByName("user32.dll", name);
    Interceptor.attach(addr, {
      onEnter: function (args) {
        const hwnd = args[0];
        const msg = args[1];
        const msgName = getMessageName(msg);
        if (msgName == null) return;

        const wParam = args[2];
        const lParam = args[3];
        console.log("[*] " + name + " called");
        console.log("    HWND:    " + hwnd);
        console.log("    Msg:     " + msg + " (" + msgName + ")");
        console.log("    wParam:  " + wParam);
        console.log("    lParam:  " + lParam);
      },
    });
  } catch (e) {
    console.log("[-] Could not hook " + name + ": " + e);
  }
});
```

위의 코드를 실행하고 notepad++에 글자를 입력하고, 복사 붙여넣기한 결과는 다음과 같습니다.



<그림1> 글자 입력



<그림2> 복사 붙여넣기

메모장에 저장된 글자를 읽어오기 위하여 SCI_GETTEXT가 호출되고, 인코딩을 처리하기 위하여 SC_CP_UTF8을 사용하고, "IME가 변환결과의 문자를 가져와 어플리케이션으로 전송"[5]하는 WM_IME_CHAR가 확인됩니다.

[5] WM_IME_CHAR windows 공식문서 (https://learn.microsoft.com/en-us/windows/win32/intl/wm-ime-char)

3. 복사 붙여넣기가 가능한 C언어 코드 구현

복사 붙여넣기의 본질은 클립보드에 데이터를 저장하고, 다시 가져오는 것입니다. 클립보드에 데이터를 저장하고, 다시 가져오는 C코드를 작성하였습니다.

- my_clipboard.c

```c
#include <windows.h>
#include <stdio.h>

void copy_to_clipboard(const char *text) {
    const size_t len = strlen(text) + 1;
    HGLOBAL hMem = GlobalAlloc(GMEM_MOVEABLE, len);
    if (!hMem) return;

    memcpy(GlobalLock(hMem), text, len);
    GlobalUnlock(hMem);

    if (OpenClipboard(NULL)) {
        EmptyClipboard();
        SetClipboardData(CF_TEXT, hMem);
        CloseClipboard();
        printf("Copyed\n");
    }
}

void paste_from_clipboard() {
    if (!OpenClipboard(NULL)) {
        return;
    }

    HANDLE hData = GetClipboardData(CF_TEXT);
    if (hData == NULL) {
        CloseClipboard();
        return;
    }

    char *pszText = (char *)GlobalLock(hData);
    if (pszText != NULL) {
        printf("[+] pasted text: %s\n", pszText);
        GlobalUnlock(hData);
    }
    CloseClipboard();
}

int main() {
    char text[100];
```

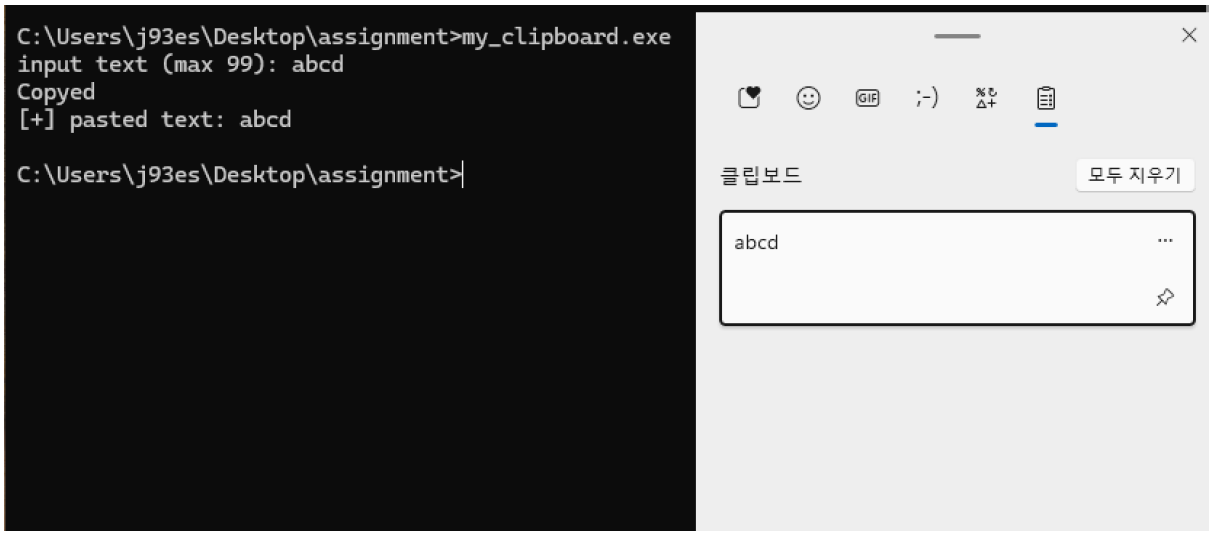
```
    printf("input text (max 99): ");
    scanf("%99s", text);

    copy_to_clipboard(text);

    Sleep(1000);

    paste_from_clipboard();

    return 0;
}
```

위의 코드는 copy_to_clipboard 함수에서 입력된 문자를 클립보드에 저장하고, paste_from_clipboard 함수에서 저장된 클립보드 데이터를 받아와서 출력하는 방식으로 구현하였습니다. 해당 코드는 아래와 같이 컴파일합니다.

- my_clipboard_build.sh

```
#!/bin/bash
gcc my_clipboard.c -o my_clipboard.exe -mwindows
```

코드를 실행한 결과는 다음과 같습니다.



4. Cntl CV, 작성한 코드 간 API 호출 차이점

먼저 API 분석을 위한 Frida 동작 코드를 작성하였습니다.

- clipboard_monitor.py

```python
import frida
import sys

session = frida.attach("notepad++.exe")
# session = frida.attach(15064)

with open("clipboard_monitor.js", "r", encoding="utf-8") as f:
    script = session.create_script(f.read())

def on_message(message, data):
    print(message)

script.on("message", on_message)
script.load()
print("[*] Frida clipboard monitoring started. Press Ctrl+C to exit.")
sys.stdin.read()
```

- clipboard_monitor.js

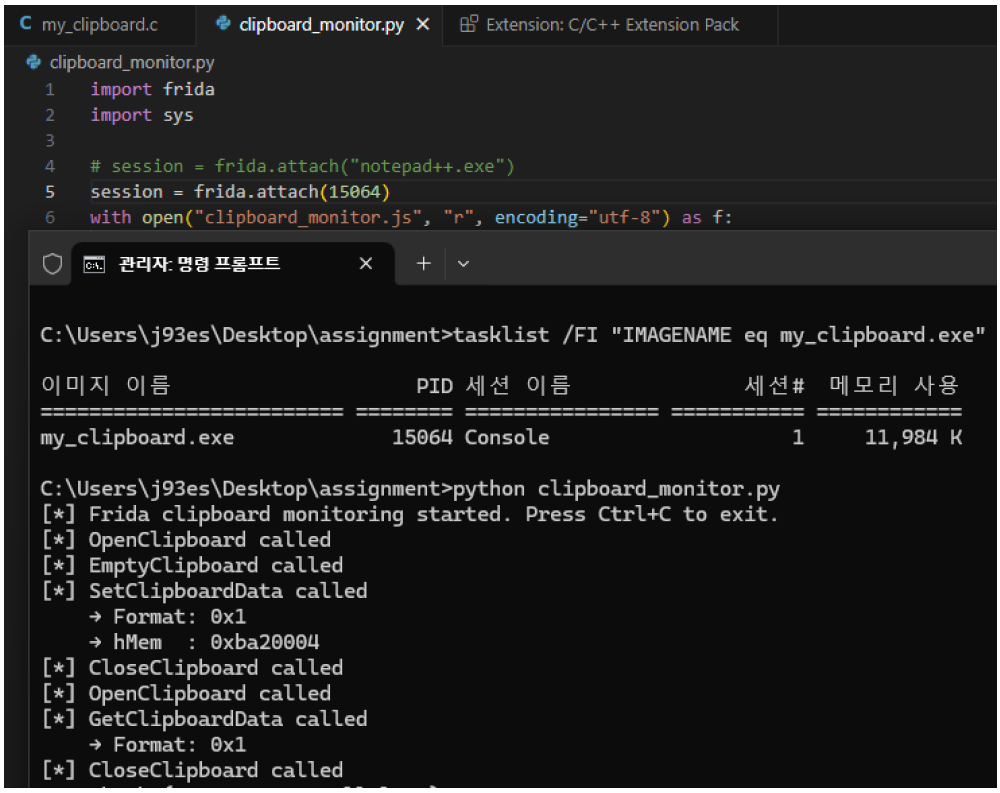
```javascript
const clipboardAPIs = [
  "OpenClipboard",
  "CloseClipboard",
  "EmptyClipboard",
  "SetClipboardData",
  "GetClipboardData",
];

clipboardAPIs.forEach(function (api) {
  try {
    const addr = Module.getExportByName("user32.dll", api);
    Interceptor.attach(addr, {
      onEnter: function (args) {
        console.log("[*] " + api + " called");
        if (api === "SetClipboardData") {
          console.log("    → Format: " + args[0]);
          console.log("    → hMem  : " + args[1]);
        }
        if (api === "GetClipboardData") {
          console.log("    → Format: " + args[0]);
        }
      },
```

```
    });
  } catch (err) {
    console.log("[-] Failed to hook " + api + ": " + err);
  }
});
```

위의 코드를 실행시키고, notepad++에서 Cntl CV한 결과는 다음과 같습니다.

```
C:\Users\j93es\Desktop\assignment>python clipboard_monitor.py
[*] Frida clipboard monitoring started. Press Ctrl+C to exit.
[*] OpenClipboard called
[*] EmptyClipboard called
[*] SetClipboardData called
    → Format: 0xd
    → hMem  : 0x161c73e0028
[*] CloseClipboard called
[*] OpenClipboard called
[*] GetClipboardData called
    → Format: 0xc216
[*] GetClipboardData called
    → Format: 0xd
[*] CloseClipboard called
```

직접 작성한 my_clipboard.exe를 실행시킨 결과는 다음과 같습니다.



위의 결과에서 Cntl CV와 my_clipboard.exe의 상이한 점은 단연 Format일 것입니다. SetClipboardData에서 Cntl CV는 0xd의 Format을 활용하는 반면, my_clipboard.exe는 0x1의 Format을 활용합니다. 더하여 GetClipboardData에서 Cntl CV는 0xc216과 0xd의

Format을 불러오는 반면, my_clipboard.exe는 0x1의 Format을 불러옵니다. Clipboard Format 공식문서[6]를 확인한 결과 0x1은 CF_TEXT, 0xd는 CF_UNICODETEXT임을 확인할 수 있었습니다. 0xc216는 표준에 없는 것으로 보아, 표준이 아닌 Format으로 추정됩니다. 이를 통하여 Cntl CV가 my_clipboard.exe보다 넓은 Format을 지원한다는 것을 확인할 수 있습니다.

5. 나오며

지금까지 SPY++, API Monitor, Notepad로 구성되어진 실습을 arm64 아키텍쳐의 환경에서 유사하게 수행하였습니다. 먼저 Frida를 통하여 메세지를 분석하였습니다. Clipboard에 데이터를 저장하고, 불러오는 C언어로 작성된 코드를 소개하였습니다. 이렇게 컴파일된 my_clipboard.exe와 Cntl CV가 API 호출에서 Format과 관련된 차이점이 있음을 확인하였습니다. 그리고 이러한 과정 전반에 공식문서를 통하여 메세지 값과 Format 값을 확인하는 등의 확실함/정확함을 잃지 않으려 노력하였습니다.

[참고문헌]

- Frida 릴리즈 버전 목록 (https://github.com/frida/frida/releases)
- 윈도우 메세지 공식 문서 (https://learn.microsoft.com/en-us/windows/win32/winmsg/about-messages-and-message-queues)
- Scintilla 공식문서 (https://www.scintilla.org/ScintillaDoc.html)
- Scintilla 메세지 정보 (https://github.com/LuaDist/scintilla/blob/master/include/Scintilla.h)
- WM_IME_CHAR windows 공식문서 (https://learn.microsoft.com/en-us/windows/win32/intl/wm-ime-char)
- Clipboard Format 공식문서 (https://learn.microsoft.com/en-us/windows/win32/dataxchg/standard-clipboard-formats)

[6] Clipboard Format 공식문서 (https://learn.microsoft.com/en-us/windows/win32/dataxchg/standard-clipboard-formats)