

🔒 Userland API Monitoring and Code Injection Detection

■ **Malware** freestylefebruary, windows, malware



dtm Law Abiding Citizen

1 ✎ Feb 21

Userland API Monitoring and Code Injection Detection

About This Paper

The following document is a result of self-research of malicious software (malware) and its interaction with the Windows Application Programming Interface (WinAPI). It details the fundamental concepts behind how malware is able to implant malicious payloads into other processes and how it is possible to detect such functionality by monitoring communication with the Windows operating system. The notion of observing calls to the API will also be illustrated by the procedure of *hooking* certain functions which will be used to achieve the code injection techniques.

Disclaimer: Since this was a relatively accelerated project due to some time constraints, I would like to kindly apologise in advance for any potential misinformation that may be presented and would like to ask that I be notified as soon as possible so that it may be revised. On top of this, the accompanying code may be under-developed for practical purposes and have unforeseen design flaws.

Contents

1. [Introduction](#)
2. [Section I: Fundamental Concepts](#)
 - [Inline Hooking](#)
 - [API Monitoring](#)
 - [Code Injection Primer](#)
 - [DLL Injection](#)
 - [CreateRemoteThread](#)
 - [SetWindowsHookEx](#)
 - [QueueUserAPC](#)

- [Process Hollowing](#)
- [Atom Bombing](#)
- 3. [Section II: UnRunPE](#)
 - [Code Injection Detection](#)
 - [Code Injection Dumping](#)
 - [UnRunPE Demonstration](#)
- 4. [Section III: Dreadnought](#)
 - [Detecting Code Injection Method](#)
 - [Heuristics](#)
 - [Dreadnought Demonstration](#)
 - [Process Injection - Process Hollowing](#)
 - [DLL Injection - SetWindowsHookEx](#)
 - [DLL Injection - QueueUserAPC](#)
 - [Code Injection - Atom Bombing](#)
- 5. [Conclusion](#)
 - [Limitations](#)
- 6. [References](#)

Introduction

In the present day, malware are developed by cyber-criminals with the intent of compromising machines that may be leveraged to perform activities from which they can profit. For many of these activities, the malware must be able survive out in the wild, in the sense that they must operate covertly with all attempts to avert any attention from the victims of the infected and thwart detection by anti-virus software. Thus, the inception of stealth via code injection was the solution to this problem.

Section I: Fundamental Concepts

Inline Hooking

Inline hooking is the act of detouring the flow of code via *hotpatching*. Hotpatching is defined as the modification of code during the runtime of an executable image^[1]. The purpose of inline hooking is to be able to capture the instance of when the program calls a function and then from there, observation and/or manipulation of the call can be accomplished. Here is a visual representation of how normal execution works:

Normal Execution of a Function Call

```
+-----+
| Program | ----- calls function ----->
+-----+
```

versus execution of a hooked function:

Execution of a Hooked Function Call

```
+-----+          +-----+          + ----->
| Program | -- calls function --> | Intermediate | | execution   |
+-----+          |   Function   | | of             |
                  |   .           | | intermediate  | calls
                  |   .           | | function      | normal
                  |   .           | | v             | function
                  +-----+      +-----+
```

This can be separated into three steps. To demonstrate this process, the WinAPI function `MessageBox` ¹⁰ will be used.

1. Hooking the function

To hook the function, we first require the intermediate function which **must** replicate parameters of the targetted function. Microsoft Developer Network (MSDN) defines `MessageBox` as the following:

```
int WINAPI MessageBox(
    _In_opt_ HWND    hWnd,
    _In_opt_ LPCTSTR lpText,
    _In_opt_ LPCTSTR lpCaption,
```

```
    _In_    UINT    uType  
);
```

So the intermediate function may be defined like so:

```
int WINAPI HookedMessageBox(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType)  
    // our code in here  
}
```

Once this exists, execution flow has somewhere for the code to be redirected. To actually *hook* the `MessageBox` function, the first few bytes of the code can be *patched* (keep in mind that the original bytes must be saved so that the function may be restored for when the intermediate function is finished). Here are the original assembly instructions of the function as represented in its corresponding module `user32.dll`:

```
; MessageBox  
8B FF  mov edi, edi  
55      push ebp  
8B EC  mov ebp, esp
```

versus the hooked function:

```
; MessageBox  
68 xx xx xx xx  push <HookedMessageBox> ; our intermediate function  
C3          ret
```

Here I have opted to use the `push-ret` combination instead of an absolute `jmp` due to my past experiences of it not being reliable for reasons to be discovered. `xx xx xx xx` represents the little-endian byte-order address of `HookedMessageBox`.

2. Capturing the function call

When the program calls `MessageBox`, it will execute the `push-ret` and effectively jump into the `HookedMessageBox` function and once there, it has complete control over the parameters and the call itself. To replace the text that will be shown on the message box dialog, the following can be defined in `HookedMessageBox`:

```
int WINAPI HookedMessageBox(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType)
{
    TCHAR szMyText[] = TEXT("This function has been hooked!");
}
```

`szMyText` can be used to replace the `LPCTSTR lpText` parameter of `MessageBox`.

3. Resuming normal execution

To forward this parameter, execution needs to continue to the original `MessageBox` so that the operating system can display the dialog. Since calling `MessageBox` again will just result in an infinite recursion, the original bytes must be restored (as previously mentioned).

```
int WINAPI HookedMessageBox(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType)
{
    TCHAR szMyText[] = TEXT("This function has been hooked!");

    // restore the original bytes of MessageBox
    // ...

    // continue to MessageBox with the replaced parameter and return the return value
    return MessageBox(hWnd, szMyText, lpCaption, uType);
}
```

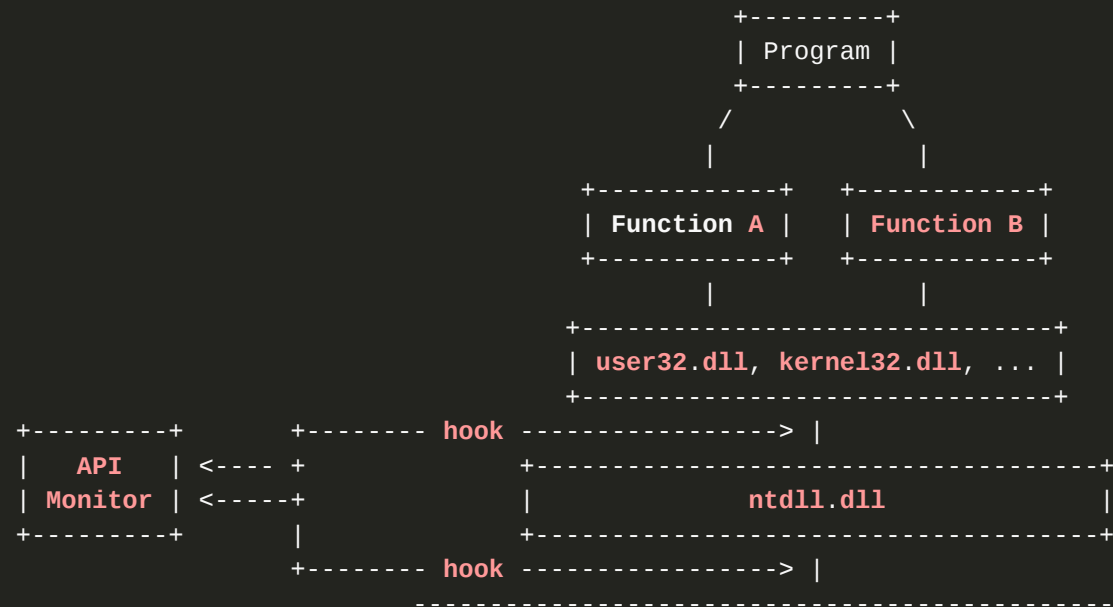
If rejecting the call to `MessageBox` was desired, it is as easy as returning a value, preferably one that is defined in the documentation. For example, to return the “No” option from a “Yes/No” dialog, the intermediate function can be:

```
int WINAPI HookedMessageBox(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType)
{
    return IDNO; // IDNO defined as 7
}
```

API Monitoring

The concept of API monitoring follows on from function hooking. Because gaining control of function calls is possible, observation of all of the parameters is also possible, as previously mentioned hence the name *API monitoring*. However, there is a small issue which is caused by the availability of different high-level API calls

that are unique but operate using the same set of API at a lower level. This is called *function wrapping*, defined as *subroutines whose purpose is to call a secondary subroutine*. Returning to the MessageBox example, there are two defined functions: `MessageBoxA` for parameters that contain ASCII characters and a `MessageBoxW` for parameters that contain wide characters. In reality, to hook `MessageBox`, it is required that both `MessageBoxA` and `MessageBoxW` be patched. The solution to this problem is to hook at the **lowest** possible **common** point of the function call hierarchy.



Here is what the `MessageBox` call hierarchy looks like:

Here is `MessageBoxA`:

```
user32!MessageBoxA -> user32!MessageBoxExA -> user32!MessageBoxTimeoutA -> user32
```

and `MessageBoxW`:

```
user32!MessageBoxW -> user32!MessageBoxExW -> user32!MessageBoxTimeoutW
```

The call hierarchy both funnel into `MessageBoxTimeoutW` which is an appropriate location to hook. For functions that have a deeper hierarchy, hooking any lower could prove to be unnecessarily troublesome due to the possibility of an increasing complexity of the function's parameters. `MessageBoxTimeoutW` is an undocumented WinAPI function and is defined^[2] like so:

```
int WINAPI MessageBoxTimeoutW(
    HWND hWnd,
    LPCWSTR lpText,
    LPCWSTR lpCaption,
    UINT uType,
    WORD wLanguageId,
    DWORD dwMilliseconds
);
```

To log the usage:

```
int WINAPI MessageBoxTimeoutW(HWND hWnd, LPCWSTR lpText, LPCWSTR lpCaption, UINT uType, WORD wLanguageId, DWORD dwMilliseconds) {
    std::wofstream logfile;      // declare wide stream because of wide parameters
    logfile.open(L"log.txt", std::ios::out | std::ios::app);

    logfile << L"Caption: " << lpCaption << L"\n";
    logfile << L"Text: " << lpText << L"\n";
    logfile << L"Type: " << uType << L"\n";

    logfile.close();

    // restore the original bytes
    // ...

    // pass execution to the normal function and save the return value
    int ret = MessageBoxTimeoutW(hWnd, lpText, lpCaption, uType, wLanguageId, dwMilliseconds);

    // rehook the function for next calls
    // ...

    return ret; // return the value of the original function
}
```

Once the hook has been placed into `MessageBoxTimeoutW`, `MessageBoxA` and `MessageBoxW` should both be captured.

Code Injection Primer

For the purposes of this paper, code injection will be defined as the insertion of executable code into an external process. The possibility of injecting code is a natural result of the functionality allowed by the WinAPI. If certain functions are strung together, it is possible to access an existing process, write data to it and then execute it remotely under its context. In this section, the relevant techniques of code injection that was covered in the research will be introduced.

DLL Injection

Code can come from a variety of forms, one of which is a *Dynamic Link Library* (DLL). DLLs are libraries that are designed to offer extended functionality to an executable program which is made available by exporting subroutines. Here is an example DLL that will be used for the remainder of the paper:

```
extern "C" void __declspec(dllexport) Demo() {
    ::MessageBox(nullptr, TEXT("This is a demo!"), TEXT("Demo"), MB_OK);
}

bool APIENTRY DllMain(HINSTANCE hInstDll, DWORD fdwReason, LPVOID lpvReserved) {
    if (fdwReason == DLL_PROCESS_ATTACH)
        ::CreateThread(nullptr, 0, (LPTHREAD_START_ROUTINE)Demo, nullptr, 0, nullptr);
    return true;
}
```

When a DLL is loaded into a process and initialised, the loader will call `DllMain` with `fdwReason` set to `DLL_PROCESS_ATTACH`. For this example, when it is loaded into a process, it will thread the `Demo` subroutine to display a message box with the title `Demo` and the text `This is a demo!`. To correctly finish the initialisation of a DLL, it must return `true` or it will be unloaded.

CreateRemoteThread

DLL injection via the `CreateRemoteThread` ³ function utilises this function to execute a remote thread in the virtual space of another process. As mentioned above, all that is required to execute a DLL is to have it load

into the process by forcing it to execute the `LoadLibrary` function. The following code can be used to accomplish this:

```
void injectDll(const HANDLE hProcess, const std::string dllPath) {
    LPVOID lpBaseAddress = ::VirtualAllocEx(hProcess, nullptr, dllPath.length(),
    MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);

    ::WriteProcessMemory(hProcess, lpBaseAddress, dllPath.c_str(), dllPath.length(),
    MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);

    HMODULE hModule = ::GetModuleHandle(TEXT("kernel32.dll"));

    LPVOID lpStartAddress = ::GetProcAddress(hModule, "LoadLibraryA"); // LoadLibraryA

    ::CreateRemoteThread(hProcess, nullptr, 0, (LPTHREAD_START_ROUTINE)lpStartAddress,
    lpBaseAddress, 0, 0);
}
```

MSDN defines `LoadLibrary` as:

```
HMODULE WINAPI LoadLibrary(
    _In_ LPCTSTR lpFileName
);
```

It takes a single parameter which is the path name to the desired library to load. The `CreateRemoteThread` function allows one parameter to be passed into the thread routine which matches exactly that of `LoadLibrary`'s function definition. The goal is to allocate the string parameter in the virtual address space of the target process and then pass that allocated space's address into the parameter argument of `CreateRemoteThread` so that `LoadLibrary` can be invoked to load the DLL.

1. Allocating virtual memory in the target process

Using `VirtualAllocEx` allows space to be allocated within a selected process and on success, it will return the starting address of the allocated memory.

Virtual Address Space of Target Process

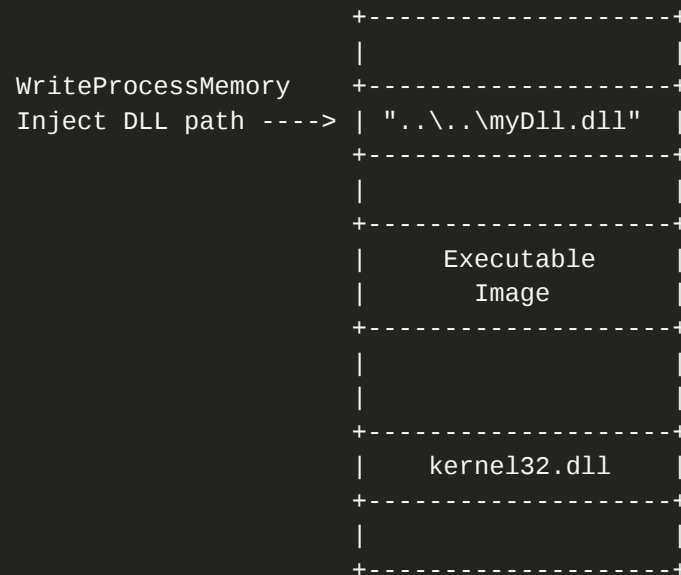
	+-----+ +-----+
VirtualAllocEx	+-----+
Allocated memory --->	Empty space +-----+



2. Writing the DLL path to allocated memory

Once memory has been initialised, the path to the DLL can be injected into the allocated memory returned by `VirtualAllocEx` using `WriteProcessMemory`.

Virtual Address Space of Target Process



3. Get address of `LoadLibrary`

Since all system DLLs are mapped to the same address space across all processes, the address of `LoadLibrary` does not have to be directly retrieved from the target process. Simply calling `GetModuleHandle(TEXT("kernel32.dll"))` and `GetProcAddress(hModule, "LoadLibraryA")` will do the job.

4. Loading the DLL

The address of `LoadLibrary` and the path to the DLL are the two main elements required to load the DLL. Using the `CreateRemoteThread` function, `LoadLibrary` is executed under the context of the target process with the DLL path as a parameter.

Virtual Address Space of Target Process



SetWindowsHookEx

Windows offers developers the ability to monitor certain events with the installation of *hooks* by using the `SetWindowsHookEx` ² function. While this function is very common in the monitoring of keystrokes for

keylogger functionality, it can also be used to inject DLLs. The following code demonstrates DLL injection into itself:

```
int main() {
    HMODULE hMod = ::LoadLibrary(DLL_PATH);
    HOOKPROC lpfn = (HOOKPROC)::GetProcAddress(hMod, "Demo");
    HHOOK hHook = ::SetWindowsHookEx(WH_GETMESSAGE, lpfn, hMod, ::GetCurrentThreadId());
    ::PostThreadMessageW(::GetCurrentThreadId(), WM_RBUTTONDOWN, (LPARAM)0, (LPARAM)0);

    // message queue to capture events
    MSG msg;
    while (::GetMessage(&msg, nullptr, 0, 0) > 0) {
        ::TranslateMessage(&msg);
        ::DispatchMessage(&msg);
    }

    return 0;
}
```

`SetWindowsHookEx` defined by MSDN as:

```
HHOOK WINAPI SetWindowsHookEx(
    _In_ int idHook,
    _In_ HOOKPROC lpfn,
    _In_ HINSTANCE hMod,
    _In_ DWORD dwThreadId
);
```

takes a `HOOKPROC` parameter which is a user-defined callback subroutine that is executed when the specific hook event is triggered. In this case, the event is `WH_GETMESSAGE` which deals with messages in the message queue. The code initially loads the DLL into its own virtual process space and the exported `Demo` function's address is obtained and defined as the callback function in the call to `SetWindowsHookEx`. To force the callback function to execute, `PostThreadMessage` is called with the message `WM_RBUTTONDOWN` which will trigger the `WH_GETMESSAGE` hook and thus the message box will be displayed.

QueueUserAPC

DLL injection with `QueueUserAPC` ³ works similar to that of `CreateRemoteThread`. Both allocate and inject the DLL path into the virtual address space of a target process and then force a call to `LoadLibrary` under its context.

```
int injectDll(const std::string dllPath, const DWORD dwProcessId, const DWORD dwThreadId,
             HANDLE hProcess = ::OpenProcess(PROCESS_ALL_ACCESS, false, dwProcessId);

             HANDLE hThread = ::OpenThread(THREAD_ALL_ACCESS, false, dwThreadId);

             LPVOID lpLoadLibraryParam = ::VirtualAllocEx(hProcess, nullptr, dllPath.length() * sizeof(WCHAR),
                                                         MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
             ::WriteProcessMemory(hProcess, lpLoadLibraryParam, dllPath.data(), dllPath.length() * sizeof(WCHAR),
                                   0);
             ::QueueUserAPC((PAPCFUNC)::GetProcAddress(::GetModuleHandle(TEXT("kernel32.dll")), "LoadLibraryW"),
                           hThread, lpLoadLibraryParam);

             return 0;
}
```

One major difference between this and `CreateRemoteThread` is that `QueueUserAPC` operates on *alertable* states. Asynchronous procedures queued by `QueueUserAPC` are only handled when a thread enters this state.

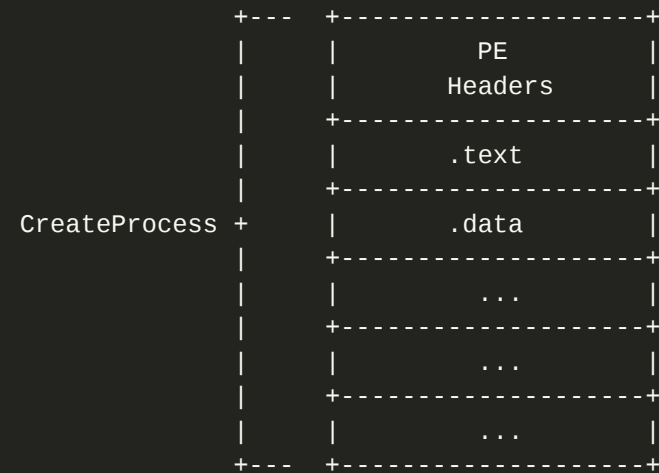
Process Hollowing

Process hollowing, AKA RunPE, is a popular method used to evade anti-virus detection. It allows the injection of entire executable files to be loaded into a target process and executed under its context. Often seen in crypted applications, a file on disk that is compatible with the payload is selected as the host and is created as a process, has its main executable module *hollowed* out and replaced. This procedure can be broken up into four stages.

1. Creating a host process

In order for the payload to be injected, the bootstrap must first locate a suitable host. If the payload is a .NET application, the host must also be a .NET application. If the payload is a native executable defined to use the console subsystem, the host must also reflect the same attributes. The same is applied to x86 and x64 programs. Once the host has been chosen, it is created as a suspended process using `CreateProcess(PATH_TO_HOST_EXE, ..., CREATE_SUSPENDED, ...)`.

Executable Image of Host Process

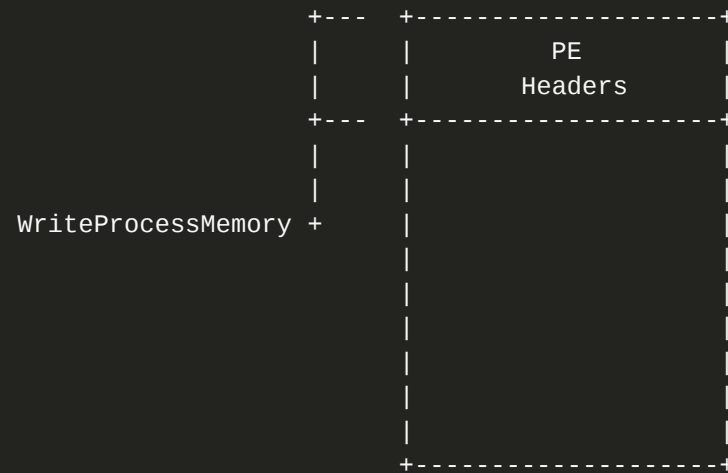


2. Hollowing the host process

For the payload to work correctly after injection, it must be mapped to a virtual address space that matches its `ImageBase` value found in the [optional header](#) of the payload's PE headers.

```
typedef struct _IMAGE_OPTIONAL_HEADER {  
    WORD        Magic;  
    BYTE        MajorLinkerVersion;  
    BYTE        MinorLinkerVersion;  
    DWORD       SizeOfCode;  
    DWORD       SizeOfInitializedData;  
    DWORD       SizeOfUninitializedData;  
    DWORD       AddressOfEntryPoint;    // <---- this is required  
    DWORD       BaseOfCode;  
    DWORD       BaseOfData;  
    DWORD       ImageBase;              // <----  
    DWORD       SectionAlignment;  
    DWORD       FileAlignment;  
    WORD        MajorOperatingSystemVersion;  
    WORD        MinorOperatingSystemVersion;  
    WORD        MajorImageVersion;  
    WORD        MinorImageVersion;  
    WORD        MajorSubsystemVersion;
```


Executable Image of Host Process

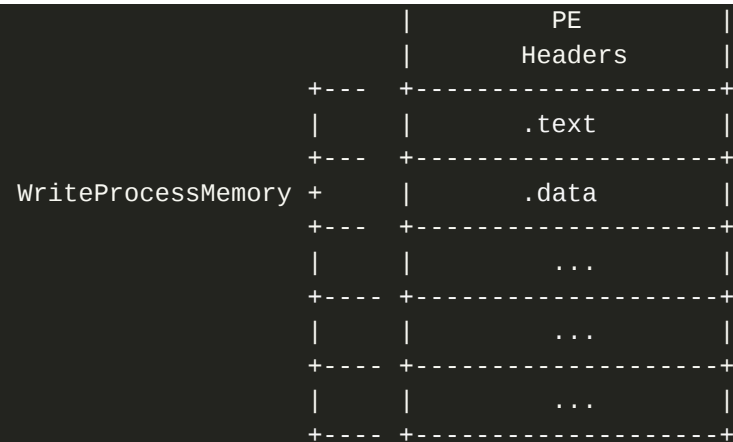


To convert the PE file to an image, all of the sections must be individually read from their file offsets and then placed correctly into their correct virtual offsets using `WriteProcessMemory`. This is described in each of the sections' own [section header](#).

```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        DWORD PhysicalAddress;
        DWORD VirtualSize;
    } Misc;
    DWORD VirtualAddress;           // <---- virtual offset
    DWORD SizeOfRawData;
    DWORD PointerToRawData;        // <---- file offset
    DWORD PointerToRelocations;
    DWORD PointerToLinenumbers;
    WORD  NumberOfRelocations;
    WORD  NumberOfLinenumbers;
    DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

Executable Image of Host Process

+-----+



4. Execution of payload

The final step is to point the starting address of execution to the payload's aforementioned `AddressOfEntryPoint`. Since the process's main thread is suspended, using `GetThreadContext` to retrieve the relevant information. The context structure is defined as:

```
typedef struct _CONTEXT
{
    ULONG ContextFlags;
    ULONG Dr0;
    ULONG Dr1;
    ULONG Dr2;
    ULONG Dr3;
    ULONG Dr6;
    ULONG Dr7;
    FLOATING_SAVE_AREA FloatSave;
    ULONG SegGs;
    ULONG SegFs;
    ULONG SegEs;
    ULONG SegDs;
    ULONG Edi;
    ULONG Esi;
    ULONG Ebx;
    ULONG Edx;
    ULONG Ecx;
```

```

    ULONG Eax;           // <----
    ULONG Ebp;
    ULONG Eip;
    ULONG SegCs;
    ULONG EFlags;
    ULONG Esp;
    ULONG SegSs;

```

To modify the starting address, the Eax member must be changed to the *virtual address* of the payload's `AddressOfEntryPoint`. Simply, `context.Eax = ImageBase + AddressOfEntryPoint`. To apply the changes to the process's thread, calling `SetThreadContext` and passing in the modified `CONTEXT` struct is sufficient. All that is required now is to call `ResumeThread` and payload should start execution.

Atom Bombing

The Atom Bombing is a code injection technique that takes advantage of global data storage via Windows's *global atom table*. The global atom table's data is accessible across all processes which is what makes it a viable approach. The data stored in the table is a null-terminated C-string type and is represented with a 16-bit integer key called the *atom*, similar to that of a map data structure. To add data, MSDN provides a [GlobalAddAtom](#) ² function and is defined as:

```

ATOM WINAPI GlobalAddAtom(
    _In_ LPCTSTR lpString
);

```

where `lpString` is the data to be stored. The 16-bit integer atom is returned on a successful call. To retrieve the data stored in the global atom table, MSDN provides a [GlobalGetAtomName](#) ² defined as:

```

UINT WINAPI GlobalGetAtomName(
    _In_ ATOM nAtom,
    _Out_ LPTSTR lpBuffer,
    _In_ int nSize
);

```

Passing in the identifying atom returned from `GlobalAddAtom` will place the data into `lpBuffer` and return the length of the string *excluding* the null-terminator.

Atom bombing works by forcing the target process to load and execute code placed within the global atom table and this relies on one other crucial function, `NtQueueApcThread`, which is lowest level userland call for `QueueUserAPC`. The reason why `NtQueueApcThread` is used over `QueueUserAPC` is because, as seen before, `QueueUserAPC`'s `APCProc` ¹ only receives one parameter which is a parameter mismatch compared to `GlobalGetAtomName`^[3].

```
VOID CALLBACK APCProc(
    _In_ ULONG_PTR dwParam    ->
    _In_ int nSize
);
UINT WINAPI GlobalGetAtomName(
    _In_ ATOM nAtom,
    _Out_ LPTSTR lpBuffer,
    _In_ int nSize
);
```

However, the underlying implementation of `NtQueueApcThread` allows for three potential parameters:

```
NTSTATUS NTAPI NtQueueApcThread(
    _In_ HANDLE ThreadHandle,
    _In_ PIO_APC_ROUTINE ApcRoutine,
    _In_opt_ PVOID ApcRoutineContext,
    _In_opt_ PIO_STATUS_BLOCK ApcStatusBlock,
    _In_opt_ ULONG ApcReserved
);
UINT WINAPI GlobalGetAtomName(
    _In_ ATOM nAtom,
    _Out_ LPTSTR lpBuffer,
    _In_ int nSize
);
```

Here is a visual representation of the code injection procedure:

Atom bombing code injection

<pre>+-----+ Atom Bombing Process +-----+ </pre>	<pre>+-----+ lpBuffer <--+ +-----+ Executable Image +-----+ </pre>	<pre>Calls GlobalGet specifyin arbitrary address : and load</pre>
---	---	---

```

|
|      NtQueueApcThread      |
+----- GlobalGetAtomName ----> |      ntdll.dll      | --+
|                               |
|                               |
+-----+

```

This is a very simplified overview of atom bombing but should be adequate for the remainder of the paper. For more information on atom bombing, please refer to enSilo's [AtomBombing: Brand New Code Injection for Windows](#) ¹⁴.

Section II: UnRunPE

UnRunPE is a proof-of-concept (PoC) tool that was created for the purposes of applying API monitoring theory to practice. It aims to create a chosen executable file as a suspended process into which a DLL will be injected to hook specific functions utilised by the process hollowing technique.

Code Injection Detection

From the code injection primer, the process hollowing method was described with the following WinAPI call chain:

1. `CreateProcess`
2. `NtUnmapViewOfSection`
3. `VirtualAllocEx`
4. `WriteProcessMemory`
5. `GetThreadContext`
6. `SetThreadContext`
7. `ResumeThread`

A few of these calls do not have to be in this specific order, for example, `GetThreadContext` can be called before `VirtualAllocEx`. However, the general arrangement cannot deviate much because of the reliance on former API calls, for example, `SetThreadContext` *must* be called before `GetThreadContext` or `CreateProcess` *must* be called first otherwise there will be no target process to inject the payload. The tool assumes this as a basis on which it will operate in an attempt to detect a potentially active process hollowing.

Following the theory of API monitoring, it is best to hook the lowest, **common** point but when it comes it malware, it should *ideally* be the **lowest** possible that is accessible. Assuming a worst case scenario, the author may attempt to skip the higher-level WinAPI functions and directly call the lowest function in the call hierarchy, usually found in the `ntdll.dll` module. The following WinAPI functions are the lowest in the call hierarchy for process hollowing:

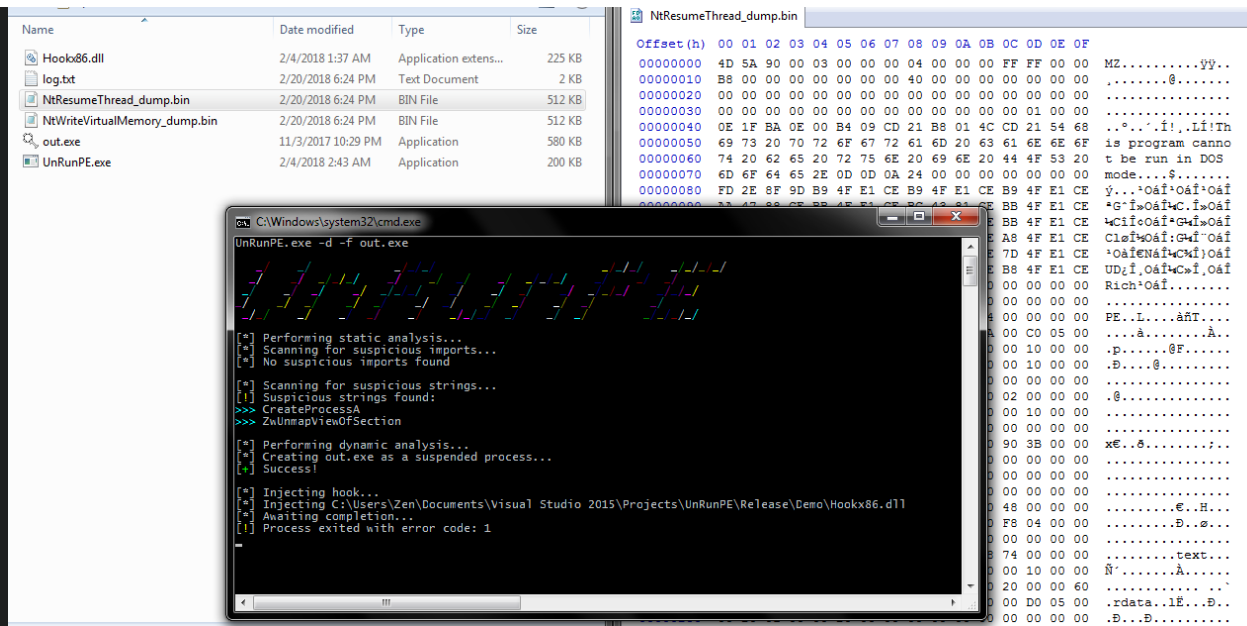
1. `NtCreateUserProcess`
2. `NtUnmapViewOfSection`
3. `NtAllocateVirtualMemory`
4. `NtWriteVirtualMemory`
5. `NtGetContextThread`
6. `NtSetContextThread`
7. `NtResumeThread`

Code Injection Dumping

Once the necessary functions are hooked, the target process is executed and each of the hooked functions' parameters are logged to keep track of the current progress of the process hollowing and the host process. The most significant hooks are `NtWriteVirtualMemory` and `NtResumeThread` because the former applies the injection of the code and the latter executes it. Along with logging the parameters, `UnRunPE` will also attempt to dump the bytes written using `NtWriteVirtualMemory` and then when `NtResumeThread` is reached, it will attempt to dump the entire payload that has been injected into the host process. To achieve this, it uses the process and thread handle parameters logged in `NtCreateUserProcess` and the base address and size logged from `NtUnmapViewOfSection`. Using the parameters provided by `NtAllocateVirtualMemory` may be more appropriate however, due to some unknown reasons, hooking that function results in some runtime errors. When the payload has been dumped from `NtResumeThread`, it will terminate the target process and its host process to prevent execution of the injected code.

UnRunPE Demonstration

For the demonstration, I have chosen to use a trojanised binary that I had previously created as an experiment. It consists of the main executable `PEview.exe` and `PuTTY.exe` as the hidden executable.



69

Section III: Dreadnought

Dreadnought is a PoC tool that was built upon UnRunPE to support a wider variety of code injection detection, namely, those listed in [Code Injection Primer](#). To engineer such an application, a few augmentations are required.

Detecting Code Injection Method

Because there are so many methods of code injection, differentiating each technique was a necessity. The first approach to this was to recognise a “trigger” API call, that is, the API call which would perform the remote execution of the payload. Using this would do two things: identify the completion of and, to an extent, the type of the code injection. The *type* can be categorised into four groups:

- Section: Code injected as/into a section
- Process: Code injected into a process
- Code: Generic code injection or shellcode
- DLL: Code injected as DLLs

Process Injection

Get or Create Process

Injected Data

Transfer to Process

Execute

Create

• CreateProcess

Section

• NtUnmapViewOfSection
• NtCreateSection

• SetThreadContext

Process Injection Info Graphic^[4] by [Karsten Hahn](#) 1

Each trigger API is listed underneath *Execute*. When either of these APIs have been reached, Dreadought will perform a code dumping method that matches the assumed injection type in a similar fashion to what occurs with process hollowing in UnRunPE. Reliance on this is not enough because there is still potential for API calls to be mixed around to achieve the same functionality as displayed from the stemming of arrows.

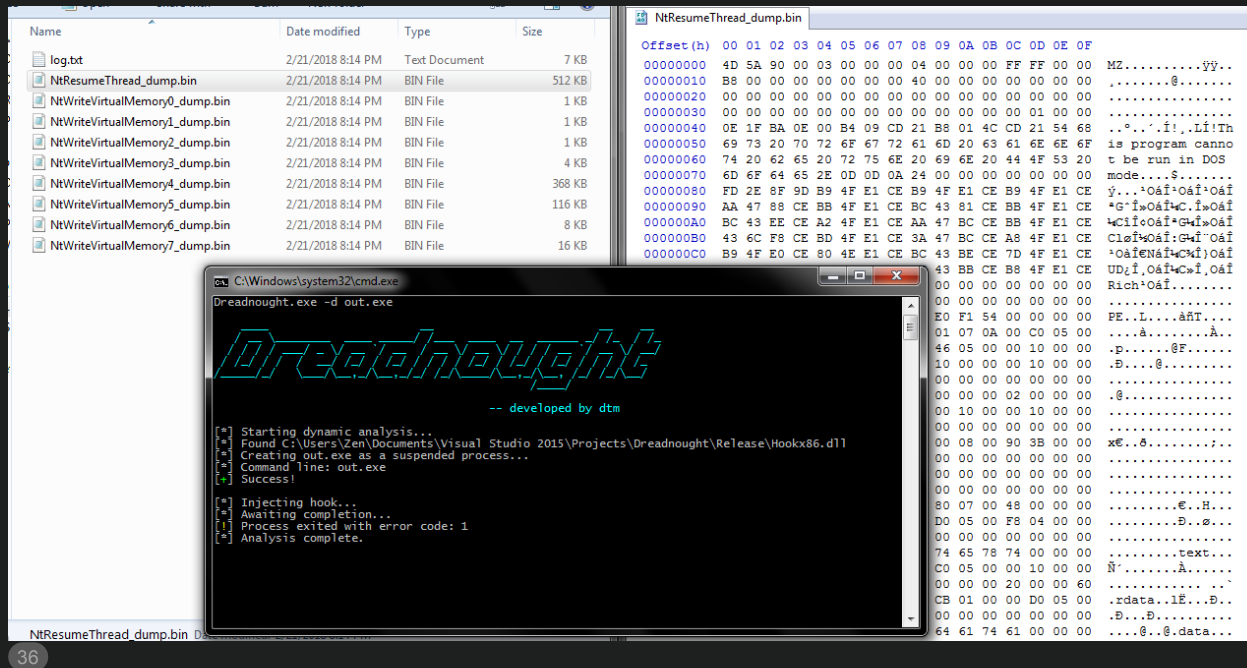
Heuristics

For Dreadought to be able to determine code injection methods more accurately, a heuristic should be involved as an assist. In the development, a very simplistic heuristic was applied. Following the process injection infographic, every time an API was hooked, it would increase the weight of one or more of the associated code injection types stored within a map data structure. As it traces each API call, it will start to

favour a certain type. Once the trigger API has been entered, it will identify and compare the weights of the relevant types and proceed with an appropriate action.

Dreadnought Demonstration

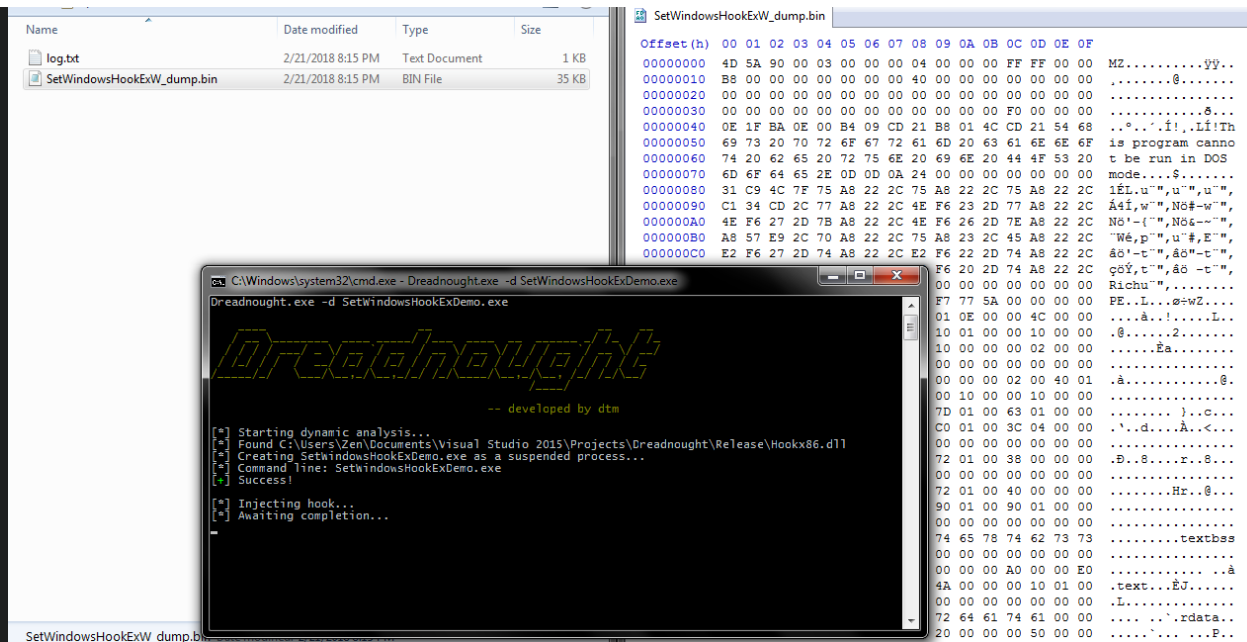
Process Injection - Process Hollowing



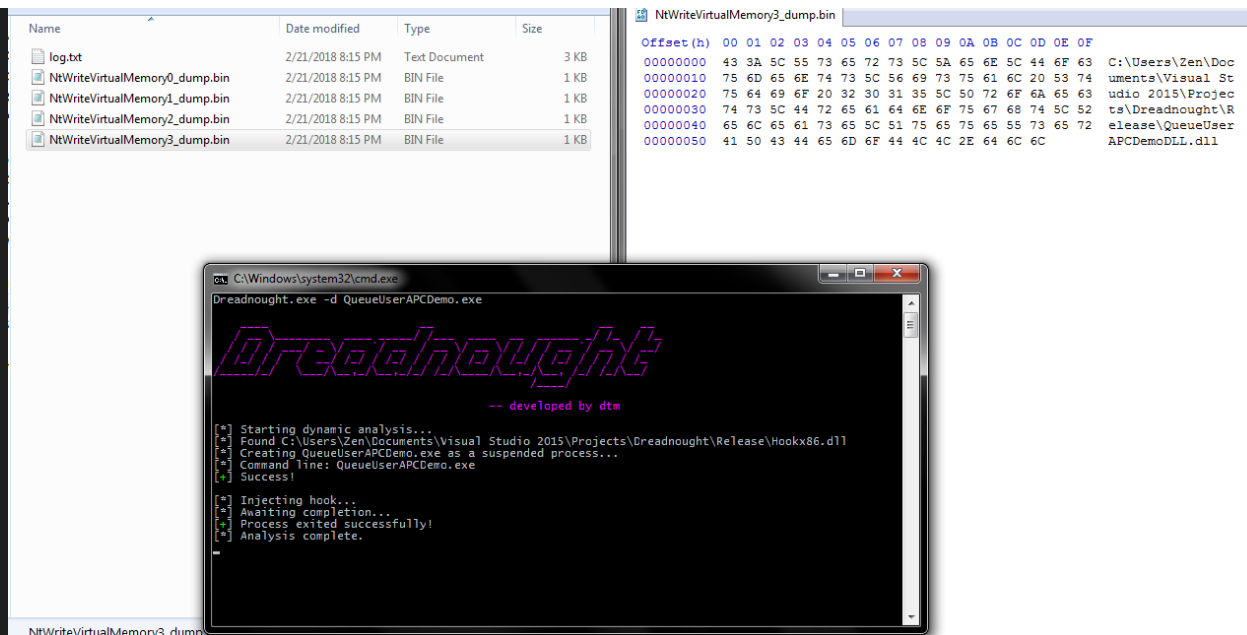
The screenshot shows a Windows file explorer window displaying a directory of files. The files include `log.txt` (Text Document, 7 KB), `NtResumeThread_dump.bin` (BIN File, 512 KB), and several `NtWriteVirtualMemory_dump.bin` files (BIN Files, ranging from 1 KB to 16 KB). A command prompt window is open, showing the execution of `Dreadnought.exe -d out.exe`. The output displays the 'Dreadnought' logo, followed by '-- developed by dtm'. The log shows the following steps:

- [*] Starting dynamic analysis...
- [*] Found C:\Users\Zen\Documents\Visual Studio 2015\Projects\Dreadnought\Release\Hookx86.dll
- [*] Creating out.exe as a suspended process...
- [*] Command line: out.exe
- [*] Success!
- [*] Injecting hook...
- [*] Awaiting completion...
- [*] Process exited with error code: 1
- [*] Analysis complete.

DLL Injection - SetWindowsHookEx



DLL Injection - QueueUserAPC



Code Injection - Atom Bombing

Log
Notes
Breakpoints
Memory Map
Call Stack
SEH
Script

76E01400 40 inc ecx
76E01401 23 8A 77 FF FF FF and ecx,dword ptr ds:[edx-89]
76E01407 ^ FF 20 jmp dword ptr ds:[ecx]
76E01409 14 E0 adc al,0
76E0140B 76 00 jbe kernelbase.76E0140D
76E0140D 00 00 add byte ptr ds:[ecx],al
76E0140F 00 28 add byte ptr ds:[ecx],ch
76E01411 14 E0 adc al,0
76E01413 76 00 jbe kernelbase.76E01415
76E01415 10 00 adc byte ptr ds:[ecx],al
76E01417 00 40 00 add byte ptr ds:[ecx],al
76E0141A 00 00 add byte ptr ds:[ecx],al
76E0141C 43 inc ebx
76E0141D 01 8C 77 00 00 00 add dword ptr ds:[edi-esi*2],ecx
76E01424 00 15 E0 76 71 01 add byte ptr ds:[!76E0],dl
76E0142A 00 00 add byte ptr ds:[ecx],al
76E0142C 01 00 add dword ptr ds:[ecx],eax
76E0142E 01 00 add dword ptr ds:[ecx],eax
76E01430 00 00 add byte ptr ds:[ecx],al
76E01432 00 00 add byte ptr ds:[ecx],al
76E01434 00 00 add byte ptr ds:[ecx],al
76E01436 00 00 add byte ptr ds:[ecx],al
76E01438 00 00 add byte ptr ds:[ecx],al
76E0143A 00 00 add byte ptr ds:[ecx],al

NtQueueApcThread0_dump.bin
NtQueueApcThread1_dump.bin
NtQueueApcThread2_dump.bin

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 40 23 8A 77 FF FF
0#Šwÿÿ

ph | Log | Notes | Breakpoints | Memory Map | Call Stack | SEH | Script

76E01400	40		inc eax
76E01401	23 8A 77 FF FF FF		and ecx,dword ptr ds:[edx-89]
76E01407	FF 20		jmp dword ptr ds:[eax]
76E01409	14 E0		adc al,E0
76E0140B	76 00		jbe kernelbase.76E0140D
76E0140D	00 00		add byte ptr ds:[eax],al
76E0140F	00 28		add byte ptr ds:[eax],ch
76E01411	14 E0		adc al,E0
76E01413	76 00		jbe kernelbase.76E01415
76E01415	10 00		adc byte ptr ds:[eax],al
76E01417	00 40 00		add byte ptr ds:[eax],al
76E0141A	00 00		add byte ptr ds:[eax],al
76E0141C	43		inc ebx
76E0141D	01 8C 77 00 00 00 00		add dword ptr ds:[edi+esi*2],ecx
76E01424	00 15 E0 76 71 01		add byte ptr ds:[17176E0],dl
76E0142A	00 00		add byte ptr ds:[eax],al
76E0142C	01 00		add dword ptr ds:[eax],eax
76E0142E	01 00		add dword ptr ds:[eax],eax
76E01430	00 00		add byte ptr ds:[eax],al
76E01432	00 00		add byte ptr ds:[eax],al
76E01434	00 00		add byte ptr ds:[eax],al
76E01436	00 00		add byte ptr ds:[eax],al
76E01438	00 00		add byte ptr ds:[eax],al
76E0143A	00 00		add byte ptr ds:[eax],al

NtQueueApcThread0_dump.bin | NtQueueApcThread1_dump.bin | NtQueueApcThread3_dump.bin

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	28	14	E0													(.a

The screenshot shows a debugger window with two panes. The left pane displays assembly code with addresses, hex values, and mnemonics. The right pane shows a hex dump of memory data with corresponding ASCII characters.

Address	Hex	Mnemonic	Comment
76E014FD	12 6A 75	adc ch,byte ptr ds:[edx+75]	
76E01500	8B C7	mov eax,edi	
76E01502	66 05 C4 00	add ax,c4	
76E01506	8B 20	mov esp,dword ptr ds:[eax]	
76E01508	66 81 EC 24 10	sub sp,1024	
76E0150D	90	nop	
76E0150E	90	nop	
76E0150F	90	nop	
76E01510	55	push ebp	
76E01511	8B EC	mov ebp,esp	
76E01513	83 EC 54	sub esp,54	
76E01516	C7 45 FC F8 14 E0 76	mov dword ptr ss:[ebp-4],<kernelbase.&LoadLi	
76E0151D	C6 45 AC 6B	mov byte ptr ss:[ebp-54],6B	
76E01521	C6 45 AD 65	mov byte ptr ss:[ebp-53],65	
76E01525	C6 45 AE 72	mov byte ptr ss:[ebp-52],72	
76E01529	C6 45 AF 6E	mov byte ptr ss:[ebp-51],6E	
76E0152D	C6 45 B0 65	mov byte ptr ss:[ebp-50],65	
76E01531	C6 45 B1 6C	mov byte ptr ss:[ebp-4F],6C	
76E01535	C6 45 B2 33	mov byte ptr ss:[ebp-4E],33	
76E01539	C6 45 B3 32	mov byte ptr ss:[ebp-4D],32	

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
00000000	8B	C7	66	05	C4	00	8B	20	66	81	EC	24	10	90	90	90	<Cf.A.< f.is...
00000010	55	8B	EC	83	EC	54	C7	45	FC	F8	14	E0	76	C6	45	AC	U<lftCEus.&vEE-
00000020	6B	C6	45	AD	65	C6	45	AE	72	C6	45	AF	6E	C6	45	B0	kEE,eEE0:EE"nEE°
00000030	65	C6	45	B1	6C	C6	45	B2	33	C6	45	B3	32	C6	45	B4	eEE+EE'3EE'2EE'
00000040	2E	C6	45	B5	64	C6	45	B6	6C	C6	45	B7	6C	C6	45	B8	.EEuEEqLEE·LEE,
00000050	00	C6	45	C8	6E	C6	45	C9	74	C6	45	CA	64	C6	45	CB	.EEEnEEtEEfAEEÉ
00000060	6C	C6	45	CC	6C	C6	45	CD	2E	C6	45	CE	64	C6	45	CF	lEElLEEi.EEfdEEI
00000070	6C	C6	45	D0	6C	C6	45	D1	00	C6	45	BC	5A	C6	45	BD	lEEBlEEÑ.EE4-2EE4
00000080	77	C6	45	BE	43	C6	45	BF	6F	C6	45	C0	6E	C6	45	C1	wEE%CEE,cEEAnEEÁ
00000090	74	C6	45	C2	69	C6	45	C3	6E	C6	45	C4	75	C6	45	C5	tEEÁLEEAnEEAuEEÁ
000000A0	65	C6	45	C6	00	C6	45	E0	57	C6	45	E1	69	C6	45	E2	eEE.EE&wEE&lEE&

Conclusion

This paper aimed to bring a technical understanding of code injection and its interaction with the WinAPI. Furthermore, the concept of API monitoring in userland was entertained with the malicious use of injection methods utilised by malware to bypass anti-virus detection. The following presents the current status of Dreadnought as of this writing.

Limitations

Dreadnought's current heuristic and detection design is incredibly poor but was sufficient enough for theoretical demonstration purposes. Practical use may not be ideal since there is a high possibility that there will be collateral with respect to the hooked API calls during regular operations with the operating system. Because of the impossibility to discern benign from malicious behaviour, false positives and negatives may arise as a result.

With regards to Dreadnought and its operations within userland, it may not be ideal use when dealing with sophisticated malware, especially those which have access to direct interactions with the kernel and those which have the capabilities to evade hooks in general.





PoC Repositories


- [GitHub - UnRunPE](#) 46
- [GitHub - Dreadnought](#) 60

References

- [1] <https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Sotirov.pdf> 8
- [2] <https://www.codeproject.com/Articles/7914/MessageBoxTimeout-API> 2
- [3] <https://blog.ensilo.com/atombombing-brand-new-code-injection-for-windows> 14
- [4] <http://struppigel.blogspot.com.au/2017/07/process-injection-info-graphic.html> 3
- [ReactOs](#) 2
- [NTAPI Undocumented Functions](#) 8
- [ntcoder](#) 4
- [GitHub - Process Hacker](#) 13
- [YouTube - MalwareAnalysisForHedgehogs](#) 5
- [YouTube - OALabs](#) 5

17  

created	last reply	3	9.6k	2	18	23	 3	
 Feb 21	 Mar 23	replies	views	users	likes	links		





BlackYenii Yenii

Feb 21

Create article @dtm 😊 !!

So rootkits can be detected if we hooked functions used to inject code ?



dtm Law Abiding Citizen

Feb 22

Thank you for reading!

Theoretically, you can hook anything if your monitoring application is at a low enough level but is most ideal when it is within the kernel so that it can oversee all processes. This is generally the case for anti-virus software which lies in the kernel and injects hooks into newly created processes. The main issue may lie with separating benign and malicious behaviour however, if your rules are strict enough and (maybe) works on an assumption that the object you are hooking is suspicious (which is what enables Dreadnought), it *could* potentially detect rootkits.

On paper, it *could* work, but applying it in a practical scenario is an entirely different world.



29 DAYS LATER



CLOSED MAR 23

This topic was automatically closed after 30 days. New replies are no longer allowed.



Reply

Suggested Topics

Topic	Category	Replies	Activity
Nestor10's Malware Analysis 101 - Anatomy of a Trojan Part 1/? reverseengineering	■ Malware	6	23d
Linux.Cephei: a Nim virus	■ Malware	9	Feb 22
Ransomware Development	■ Malware	30	Feb 22
Help: FUD virus	■ Malware	5	10d
Malware Sources resources, malware	■ Malware	1	17d

Want to read more? Browse other topics in ■ Malware or [view latest topics](#).