

The good, the bad and the stomped function

 idov31.github.io/2022/01/28/function-stomping.html

January 28, 2022

 STAR

 FORK

 FOLLOW

Introduction

When I first heard about ModuleStomping I was charmed since it wasn't like any other known injection method.

Every other injection method has something in common: They use VirtualAllocEx to allocate a new space within the process, and ModulesStomping does something entirely different: Instead of allocating new space in the process, it stomps an existing module that will load the malicious DLL.

After I saw that I started to think: How can I use that to make an even more evasive change that won't trigger the AV/EDR or won't be found by the injection scanner?

The answer was pretty simple: Stomp a single function! At the time I thought it is a matter of hours to make this work, but I know now that it took me a little while to solve all the problems.

How does a simple injection look like

The general purpose of any injection is to evade anti-viruses and EDRs and be able to deliver or execute malware.

For all of the injection methods, you need to open a process with `PROCESS_ALL_ACCESS` permission (or a combination of permissions that allow you to spawn a thread, write and read the process' memory).

since the injector needs to perform high-privilege operations such as writing to the memory of the process and executing the shellcode remotely. To be able to get `PROCESS_ALL_ACCESS` you either need that the injected process will run under your user's context or need to have a high-privileged user running in a high-privileged context (you can read more about UAC and what is a low privileged and high-privileged admin in [MSDN](#)) or the injected process is a process that you spawned under your process and therefore have all access.

After we obtain a valid handle with the right permissions we need to allocate space to the shellcode within the remote process virtual memory with VirtualAllocEx. That gives us the space and the address we need for the shellcode to be written. After we have a page with the right permissions and enough space, we can use WriteProcessMemory to write the shellcode into the remote process.

Now, all that's left to do is to call CreateRemoteThread with the shellcode's address (that we got from the VirtualAllocEx) to spawn our shellcode in the other process.

To summarize:



Research - How and why FunctionStomping works?

For the sake of the POC, I chose to target User32.dll and MessageBoxW. But, unlike the regular way of using GetModuleHandle, I needed to do it remotely. For that, I used the EnumProcessModules function:

It looks like a very straightforward function and now all I needed to do is to use the good old GetProcAddress. The implementation was pretty simple: Use GetModuleFileName to get the module's name out of the handle and then if it is the module we seek (currently User32.dll). If it is, just use GetProcAddress and get the function's base address.

```

// Getting the module name.
if (GetModuleFileNameEx(procHandle,
currentModule, currentModuleName,
MAX_PATH - sizeof(wchar_t)) == 0) {
    std::cerr << "[-] Failed to get
module name: " << GetLastError() <<
std::endl;
    continue;
}

// Checking if it is the module we seek.
if (StrStrI(currentModuleName,
moduleName) != NULL) {

    functionBase =
(Byte*)GetProcAddress(currentModule,
functionName);
    break;
}

```

```

BOOL EnumProcessModules(
    [in] HANDLE hProcess,
    [out] HMODULE *lphModule,
    [in] DWORD cb,
    [out] LPDWORD lpcbNeeded
);

```

But it didn't work. I sat by the computer for a while, staring at the valid module handle I got and couldn't figure out why I could not get the function pointer. At this point, I went back to MSDN and read again the description, and one thing caught my eye:

The `EnumProcessModules` function does not retrieve handles for modules that were loaded with the `LOAD_LIBRARY_AS_DATAFILE` or similar flags. For more information, see [LoadLibraryEx](#).

Well... That explains some things. I searched more about this permission and found this explanation:

`LOAD_LIBRARY_AS_DATAFILE`
0x00000002

If this value is used, the system maps the file into the calling process's virtual address space as if it were a data file. Nothing is done to execute or prepare to execute the mapped file. Therefore, you cannot call functions like `GetModuleFileName`, `GetModuleHandle` or `GetProcAddress` with this DLL.

That was very helpful to me because at this moment I knew why even when I have a valid handle, I cannot use `GetProcAddress`! I decided to change `User32.dll` and `MessageBoxW` to other modules and functions: `Kernel32.dll` and `CreateFileW`.

If you are wondering why `Kernel32.dll` and not another DLL, the reason is that `Kernel32.dll` is always loaded with any file (you can read more about it in the great Windows Internals books) and therefore a reliable target.

And now all that's left is to write the POC.

POC Development - Final stages

The final step is similar to any other injection method but with one significant change: We need to use `VirtualProtectEx` with the base address of our function. Usually, in injections, we give set the address parameter to `NULL` and get back the address that is mapped for us, but since we want to overwrite an existing function we need to give the base address. After `WriteProcessMemory` is executed, the function is successfully stomped!

```
// Changing the protection to PAGE_READWRITE for the shellcode.
if (!VirtualProtectEx(procHandle, functionBase, sizeToWrite, PAGE_READWRITE,
&oldPermissions)) {
    std::cerr << "[-] Failed to change protection: " << GetLastError() << std::endl;
    CloseHandle(procHandle);
    return -1;
}

SIZE_T written;

// Writing the shellcode to the remote address.
if (!WriteProcessMemory(procHandle, functionBase, shellcode, sizeof(shellcode),
&written)) {
    std::cerr << "[-] Failed to overwrite function: " << GetLastError() << std::endl;
    VirtualProtectEx(procHandle, functionBase, sizeToWrite, oldPermissions,
&oldPermissions);
    CloseHandle(procHandle);
    return -1;
}
```

At first, I used `PAGE_EXECUTE_READWRITE` permission to execute the shellcode but it is problematic (Although even with the `PAGE_EXECUTE_READWRITE` flag anti-viruses and hollows-hunter still failed to detect it - I wanted to use something else).

Because of that, I checked if there is any other permission that can help with what I wanted: To be able to execute the shellcode and still be undetected. You may ask yourself: “Why not just use `PAGE_EXECUTE_READ`?”

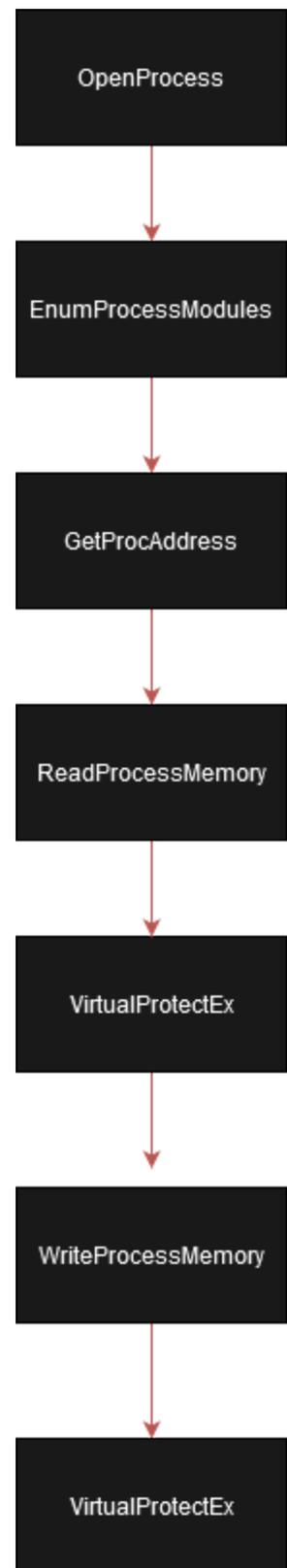
I wanted to create a single POC and be able to execute any kind of shellcode: Whether it writes to itself or not, and I’m proud to say that I found a solution for that.

I researched further about the available page permissions and one caught my eye: `PAGE_EXECUTE_WRITECOPY`.

<code>PAGE_EXECUTE_WRITECOPY</code> 0x80	Enables execute, read-only, or copy-on-write access to a mapped view of a file mapping object. An attempt to write to a committed copy-on-write page results in a private copy of the page being
---	--

It looks like it gives read, write and execute permissions without actually using `PAGE_EXECUTE_READWRITE`. I wanted to dig a little deeper into this and found an article by CyberArk that explains more about this permission and it looked like the fitting solution.

To conclude, the UML of this method looks like that:



Detection

Because many antiviruses failed to identify this shellcode injection technique as malicious I added a YARA signature I wrote so you can import that to your defense tools.

Acknowledgments

- [ModuleStomping](#) for inspiration for this POC.
- [CyberArk's article](#) for validation of further information about WCX permission.