Creating the WhereAmI Cobalt Strike BOF

33 minute read

Overview

This is a walkthrough of creating the Cobalt Strike Beacon Object File (BOF) “Where Am I?”

This idea was inspired by Matt Eidelberg’s DEF CON 29 talk Operation Bypass Catch My Payload If You Can (https://youtu.be/JXKNdWUs77w).

- In this talk, Matt shows how EDR heuristics can detect Cobalt Strike beacons based on their behavior.
- Matt uses an example where after the beacon compromises the endpoint, the first thing it does is run the `whoami.exe` local binary.
- This behavior of the host beacon process spawning a new `whoami.exe` process, triggers the EDR and the beacon is burned!
- I’ve been doing allot of Windows Internals studying, and this video made a lightbulb go off.
- I thought “Why not just get the `whoami.exe` info from the process? It’s already right there in the beacon processes memory!”

So that’s what I did! I created a Beacon Object File that grabs the information we’d want, right there from the beacon process memory!

Since the goal was to make it ninja/OPSEC safe, I figured why not just do it dynamically with Assembly? About halfway through creation, I bit the bullet and burned the extra time to make it into a blog post as well, so here it is!

For the full code to the project see the GitHub repo:

- GitHub - boku7/whereami (https://github.com/boku7/whereami)

I discovered that TrustedSec had already created a BOF for this, and of course they did because they are awesome! If you’d like to view their original work you can find it here: trustedsec/CS-Situational-Awareness-BOF/env (https://github.com/trustedsec/CS-Situational-Awareness-BOF/blob/master/src/SA/env/entry.c)

Our BOF Flow to get the Environment Variables Dynamically in Memory

Below is the high-level flow & WinDBG commands to map our path from the Thread Environment Block (TEB) to the Environment strings we will ultimately display in our Cobalt Strike interactive beacon console.

- WinDBG has an awesome feature that allows you to supply it a structure & a memory address while debugging a process, and it will format the values there into the struct you supply.
- To make our BOF work from anywhere in memory, we will use windows operating system functionality to get the TEB address, from the TEB we will get the Process Environment Block (PEB) address, from the PEB we will get the ProcessParameters struct address, and from the ProcessParameters struct we will get the address of the Environment string block & the size of the Environment string block.

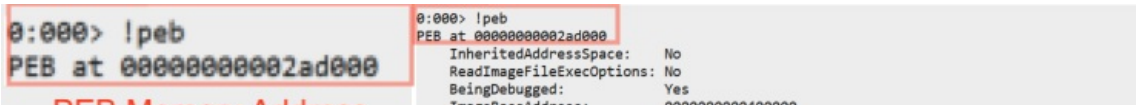
`TEB (GS Register)` -> `PEB` -> `ProcessParameters` -> `Environment Block Address` & `Environment Size`

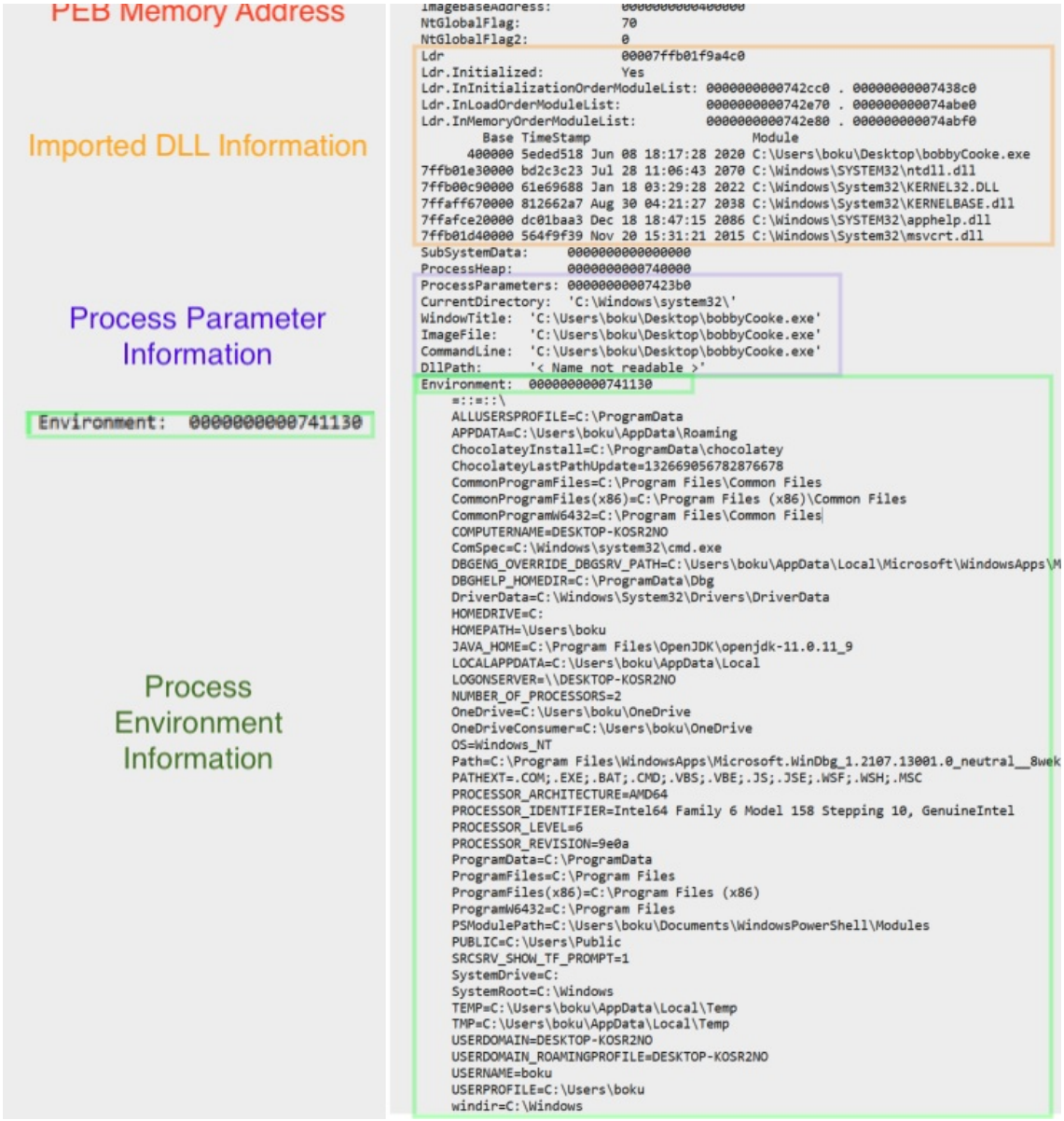
```
# TEB Address
0:000> !teb
TEB at 00000000002ae000
# PEB Address from TEB
0:000> dt !_TEB 2ae000
   +0x060 ProcessEnvironmentBlock : 0x00000000`002ad000 _PEB
# ProcessParamters Address from PEB
0:000> dt !_PEB 2ad000
   +0x020 ProcessParameters : 0x00000000`007423b0 _RTL_USER_PROCESS_PARAMETERS
# Environment Address & Size from ProcessParameters
0:000> dt !_RTL_USER_PROCESS_PARAMETERS 7423b0
   +0x080 Environment      : 0x00000000`00741130 Void
   +0x3f0 EnvironmentSize  : 0x124e
```

- Note that even with ASLR off on your windows device, the TEB & PEB address will change pretty much everytime you create a new process.

Previewing Our Target Environment Strings with WinDBG

WinDBG has a built in feature `!peb` which will beautifully parse out the PEB structure as it exists in memory for us! By using this command we can neatly see all the Environment strings we will be hunting for when creating this BOF!





- We can see that `!peb` command parses out the PEB structure and displays to us the Loader (Ldr) information, the address & resolved strings of the ProcessParameters struct, as well as the Environment information we are targeting.

Initial Setup

- Boot up a windows box (https://www.microsoft.com/en-us/software-download/windows10ISO)
- Download and Install x64DBG (https://x64dbg.com/#start)
- Download and install WinDBG (https://www.microsoft.com/en-us/p/windbg-preview/9pgjgd53tn86?activetab=pivot:overviewtab)
- Make sure WinDBG symbols are setup
- Open any executable PE file

If you're wanting to conquer malware development and learn how to use x64dbg, then work through these epic courses first:

- Sektor7 (@SEKTOR7net) - RED TEAM Operator: Malware Development Essentials & Intermediate Courses (https://institute.sektor7.net/)

If you are new to WinDBG check out this awesome course:

- Pavel Yosifovich (@zodiacon) - WinDbg Fundamentals: User Mode (https://www.pentesteracademy.com/course?id=52)

If you want to conquer Intel Assembly check out these great courses:

- Pentester Academy - x86_64 Assembly Language and Shellcoding on Linux (https://www.pentesteracademy.com/course?id=7)
- Offensive Security - Windows User Mode Exploit Development (https://www.offensive-security.com/...)

- Offensive Security - Windows User Mode Exploit Development (https://www.offensive-security.com/exp301-osed/)

From TEB to PEB

The address of the Thread Environment Block (TEB) can be discovered from anywhere in memory by referencing the `GS` register for 64 bit, and the `FS` register for 32 bit. The TEB includes within it the address of the Process Environment Block (PEB). Therefor once we get the TEB using the `GS`/`FS` register, we can find the PEB. This walkthrough is for a x64 BOF, so we will be using the `GS` register.

Viewing the TEB in WinDBG

To see the TEB for our current thread in WinDBG, just use the `!teb` command. This displays the TEB for us nicely.

```
0:000> !teb
TEB at 00000000002ae000
    ExceptionList:        0000000000000000
    StackBase:            0000000000650000
    StackLimit:           000000000064d000
    SubSystemTib:         0000000000000000
    FiberData:            0000000000001e00
    ArbitraryUserPointer: 0000000000000000
    Self:                 00000000002ae000
    EnvironmentPointer:   0000000000000000
    ClientId:             00000000000008f0 . 0000000000001f30
    RpcHandle:            0000000000000000
    Tls Storage:          0000000000743340
    PEB Address:          00000000002ad000
```

- We can see that the PEB Address is `0x2ad000` for our process.
- Although we can see the PEB address here, we need to know the offset to the PEB Address pointer within the TEB, so we can do this programmatically in our BOF.

Parsing the TEB Structure in Memory

Using the TEB address we discovered by using the `!teb` command, we will feed that into the `dt` command and parse the memory at the TEB Address `0x2ae000` so we can discover the offset of the PEB Address.

```
0:000> dt !_TEB 2ae000
ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
   +0x038 EnvironmentPointer : (null)
   +0x040 ClientId         : _CLIENT_ID
   +0x050 ActiveRpcHandle  : (null)
   +0x058 ThreadLocalStoragePointer : 0x00000000`00743340 Void
   +0x060 ProcessEnvironmentBlock : 0x00000000`002ad000 _PEB
...
```

- We can see that the PEB Address is at an offset of `+0x060` within the TEB.

Creating TEB to PEB Shellcode

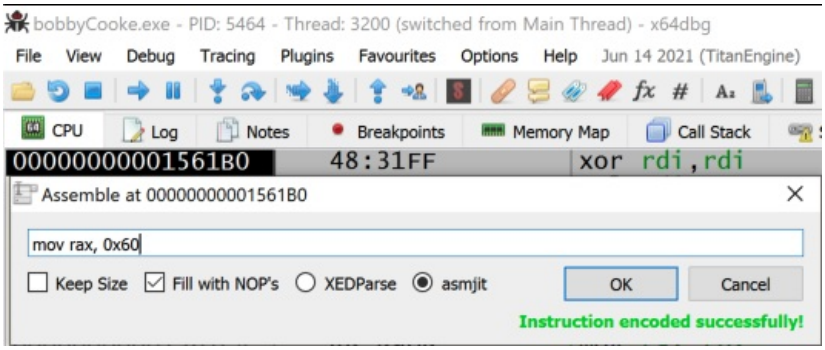
Our goal is to do this in a Cobalt Strike Beacon Object File, so we will need to create the Assembly code to discover the PEB from the TEB programmatically. We will make sure this is Position Independent Code (PIC) by using the `GS` register to discover the TEB.

- To test that this works, we will open our PE file in x64dbg.
- x64dbg has advantages over WinDBG, and WinDBG has advantages over x64dbg. I switch between them allot

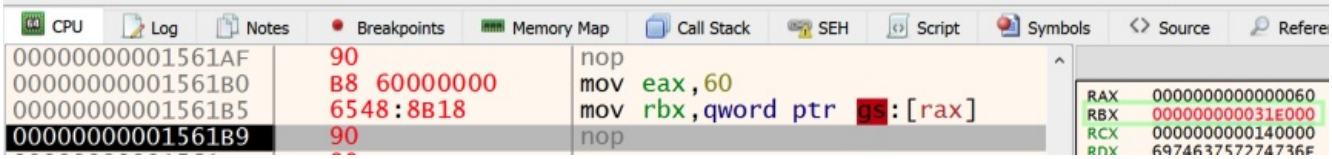
depending on what I’m trying to do.

- Set a break point anywhere. Then select the current line that RIP is on.
- Press the spacebar and edit the assembly.

Editing Opcodes in memory with x64dbg



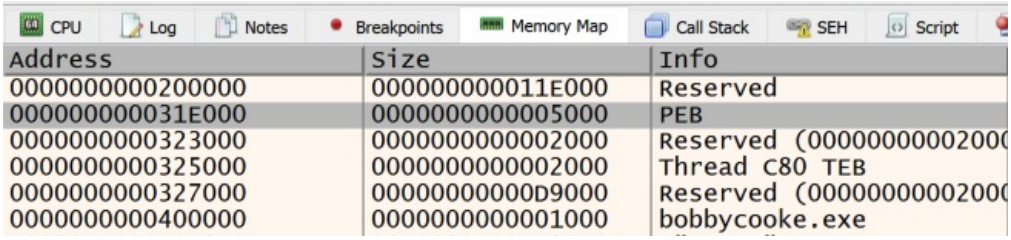
- We will put `0x60` into the `RAX` register because we know that the PEB Address is at `TEB+0x60`.
- For the next instruction put in `mov rbx, gs:[rax]`.
  - We are referencing the TEB address using the `GS` register. This is a Windows internals operating system functionality.
  - We are telling the processor to move the 8-byte value at `TEB+0x60` into the `RBX` register.
  - Our PEB Adress is at `TEB+0x60`.
- Now that we have our 2 instructions in, we press `F7` to step forward and execute our instructions.



- The address of the PEB is in `RBX` and is `0x31E000`.

Confirming PEB Address

To confirm that our assembly code resolves the correct address of the PEB dynamically in memory we can confirm using the Memory Map tab.



Our Assembly Code So Far

```
mov rax, 0x60     // RAX = 0x60 = Offset of PEB Address within the TEB
mov rbx, gs:[rax] // RBX = PEB Address
```

From PEB to ProcessParameters

Get the Address of the PEB Again

Now that we have successfully discovered the path to get from any place in process memory to the PEB, we will work on the next goal. Which is getting from the PEB to the ProcessParameters struct. Saving our above PIC shellcode for later, we’ll close down x64dbg for now, and open a PE file in WinDBG. We’ll use WinDBG to walk the PEB struct for the

ProcessParameters struct address.

Since we are launching a new process, the address of the PEB has changed. We will get this new PEB address to continue our path discovery. This time we will just use the `!peb` command and skip the TEB stuff as we've already figured that out.

- in WinDBG enter the `!peb` command in the console to get the address of the PEB in memory

```
0:000> !peb
PEB at 00000000002ad000
```

Walk the PEB Struct to find ProcessParameters Struct

The Process Environment Block (PEB) contains allot of information. Right now, we are discovering where the `ProcessParameters` struct exists within the PEB. We will note the offset: `+0x020 ProcessParameters`.

```
0:000> dt !_PEB 00000000002ad000
ntdll!_PEB
...
   +0x010 ImageBaseAddress : 0x00000000`00400000 Void
   +0x018 Ldr              : 0x00007ffb`01f9a4c0 _PEB_LDR_DATA
   +0x020 ProcessParameters : 0x00000000`007423b0 _RTL_USER_PROCESS_PARAMETERS
...
```

From ProcessParameters to Environment

Walk the ProcessParameters Struct to find our Environment

From the ProcessParameters Struct we will want to note the pointer to the `Environment` and the `EnvironmentSize`.

```
0:000> dx -r1 ((ntdll!_RTL_USER_PROCESS_PARAMETERS *)0x7423b0)
((ntdll!_RTL_USER_PROCESS_PARAMETERS *)0x7423b0)                 : 0x7423b0 [Type: _RTL_USER_PROCESS_PARAMETERS *]
...
    [+0x080] Environment      : 0x741130 [Type: void *]
...
    [+0x3f0] EnvironmentSize  : 0x124e [Type: unsigned __int64]
```

- Now we know that the `Environment` is at address `0x741130`.
- The size of the Environment is `0x124e` (4686 bytes)

Viewing the Environment Unicode Strings

Now that we know the address and size of the Environment, we can view the memory at that address to confirm

```
0:000> db 0x741130 0x741130+0x124e
00000000`00741130  3d 00 3a 00 3a 00 3d 00-3a 00 3a 00 5c 00 00 00  =.:.:.=.:.:.\...
00000000`00741140  41 00 4c 00 4c 00 55 00-53 00 45 00 52 00 53 00  A.L.L.U.S.E.R.S.
```

```
00000000`00741150  50 00 52 00 4f 00 46 00-49 00 4c 00 45 00 3d 00  P.R.O.F.I.L.E.=.
00000000`00741160  43 00 3a 00 5c 00 50 00-72 00 6f 00 67 00 72 00  C.:.\.P.r.o.g.r.
00000000`00741170  61 00 6d 00 44 00 61 00-74 00 61 00 00 00 41 00  a.m.D.a.t.a...A.
00000000`00741180  50 00 50 00 44 00 41 00-54 00 41 00 3d 00 43 00  P.P.D.A.T.A.=.C.
00000000`00741190  3a 00 5c 00 55 00 73 00-65 00 72 00 73 00 5c 00  :.\.U.s.e.r.s.\.
00000000`007411a0  62 00 6f 00 6b 00 75 00-5c 00 41 00 70 00 70 00  b.o.k.u.\.A.p.p.
00000000`007411b0  44 00 61 00 74 00 61 00-5c 00 52 00 6f 00 61 00  D.a.t.a.\.R.o.a.
00000000`007411c0  6d 00 69 00 6e 00 67 00-00 00 43 00 68 00 6f 00  m.i.n.g...C.h.o.
00000000`007411d0  63 00 6f 00 6c 00 61 00-74 00 65 00 79 00 49 00  c.o.l.a.t.e.y.I.
```

- We see that the strings are there as Unicode. You can tell because of the `00` after everything.
  - Windows Unicode strings are 2 bytes (4 hex characters).
- We can see that the Unicode strings end with a `00 00` where normally its a hex ASCII value followed by a `00`.

Assembly Shellcode to get to Environment from Anywhere in Memory

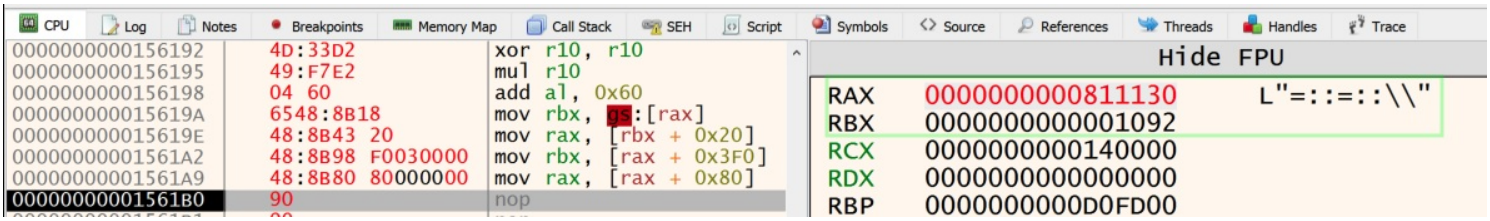
---

TEB (GS Register) –> PEB –> ProcessParameters –> Environment

```
xor r10, r10         // R10 = 0x0 - Null out some registers
mul r10              // RAX&RDX = 0x0
add al, 0x60         // RAX = 0x60 = Offset of PEB Address within the TEB
mov rbx, gs:[rax]    // RBX = PEB Address
mov rax, [rbx+0x20]  // RAX = ProcessParameters Address
mov rbx, [rax+0x3f0] // RBX = Environment Size
mov rax, [rax+0x80]  // RAX = Environment Address
```

Testing That our Code Works

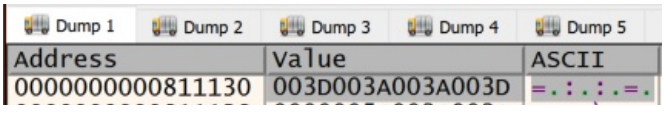
We enter in the above Assembly code into a process using x64dbg to test it out. We step through it and see that it resolves the Environment Address & Environment Size.

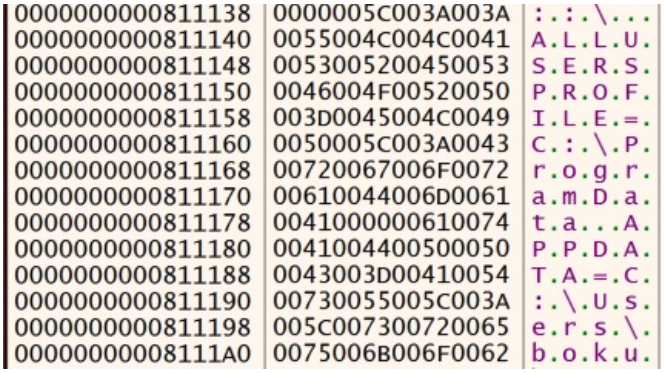


- We see that the Environment Address is in the `RAX` register.
- The Environment Size is in the `RBX` register.

Confirming the Environment Address

Just to make sure, we right-click the `RAX` value in x64dbg and click ‘View in Dump’. We can confirm that our Environment Unicode strings are at that address.





```
0000000000811138 0000005C003A003A :.:.\...
0000000000811140 0055004C004C0041 A.L.L.U.
0000000000811148 0053005200450053 S.E.R.S.
0000000000811150 0046004F00520050 P.R.O.F.
0000000000811158 003D0045004C0049 I.L.E.=.
0000000000811160 0050005C003A0043 C.:.\.P.
0000000000811168 0072006700 6F0072 r.o.g.r.
0000000000811170 0061004400 6D0061 a.m.D.a.
0000000000811178 0041000000610074 t.a...A.
0000000000811180 0041004400500050 P.P.D.A.
0000000000811188 0043003D00410054 T.A.=.C.
0000000000811190 0073005500 5C003A :.\.U.s.
0000000000811198 005C007300720065 e.r.s.\.
00000000008111A0 0075006B006F0062 b.o.k.u.
```

Create a BOF Prototype

Now that we know how to dynamically get to the Unicode Environment strings, we will create a simple Cobalt Strike Beacon Object File (BOF) & an Aggressor CNA script (for UI/UX).

Creating the our BOF Prototype

- From a macOS or Linux x64 intel device, install GCC & Ming

How to install Ming on macOS:

```
# Install brew on macOS if you need it (https://brew.sh/)
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
# Install Ming using Brew
brew install mingw-w64
```

- Make a folder and change directory into it: `mkdir WhereAmI && cd WhereAmI'
- Create a C file named `whereami.x64.c` with these contents:

```
#include <windows.h>
#include "beacon.h"
void go(char * args, int len) {
    BeaconPrintf(CALLBACK_OUTPUT, "[+] Our 'Where am I?' BOF prototype works!");
}
```

Compile the BOF Prototype

```
x86_64-w64-mingw32-gcc -c whereami.x64.c -o whereami.x64.o
```

Executing our BOF from Cobalt Strike

- Now get a Windows VM and boot it up
- Start up your Cobalt Strike Team Server
- Make a beacon in Cobalt Strike and execute it on the windows VM
- Right click your beacon and click 'Interact' to pull up the beacon CLI
- Use `inline-execute` from your Cobalt Strike CLI and supply the path to your `whereami.x64.o` BOF

If you need help setting up a Cobalt Strike Team Server, navigating the UI/setup, and just general knowledge on how to operate Cobalt Strike, then 100% check out these AWESOME Cobalt Strike videos created by Raphael Mudge!

- https://www.cobaltstrike.com/training

```
beacon> inline-execute /Users/bobby.cooke/git/boku7/WhereAmI/whereami.x64.o
[*] Tasked beacon to inline-execute /Users/bobby.cooke/git/boku7/WhereAmI/whereami.x64.o
[*] host called home, sent: 169 bytes
```

```
[+] host called home, sent: 169 bytes
[+] received output:
[+] Our 'Where am I?' BOF prototype works!
```

- We can see that our prototype works and prints the string to the console after running!

Create an Aggressor Script Prototype for UI/UX

In our `/WhereAmI/` directory, create a file named `whereami.cna`. This will be the Aggressor script responsible for adding our `whereami` command to the Cobalt Strike beacon console.

whereami.cna

```
beacon_command_register(
    "whereami",
    "Displays the beacon process environment without any DLL usage.",
    "Synopsis: whereami"
);

alias whereami {
    local('$handle $data');
    $handle = openf(script_resource("whereami.x64.o"));
    $data = readb($handle, -1);
    closef($handle);

    btask($1, "Where Am I? BOF (Bobby Cooke//SpiderLabs|@0xBoku|github.com/boku7)");
    beacon_inline_execute($1, $data, "go");
}
```

Load our Aggressor Script into Cobalt Strike

- Go to ‘Cobalt Strike’ –> ‘Script Manager’ from the menu bar of Cobalt Strike
- Click the ‘Load’ button and select our `whereami.cna` script

Testing our BOF & Aggressor Script

Now the `whereami` command is accessible from the interactive beacon console.

```
beacon> help
...
    whereami    Displays the beacon process environment without any DLL usage.
beacon> whereami
[*] Where Am I? BOF (Bobby Cooke//SpiderLabs|@0xBoku|github.com/boku7)
[+] host called home, sent: 164 bytes
[+] received output:
[+] Our 'Where am I?' BOF prototype works!
```

- Everything works! Now time to make it do the thing.

Resolving Environment Address & Size with our BOF

- We will now adjust our code to resolve the Environment Address and Size with our C BOF code.
- We will use inline assembly code to do this by using the `__asm__()` GCC function.

- When we compile the code with ming, we will add the `-masm=intel` flag to tell ming that we want to compile with the GCC C inline assembly functionality.

```
#include <windows.h>
#include "beacon.h"
void go(char * args, int len) {
    PVOID envAddr = NULL;
    PVOID envSize = NULL;
    __asm__(
        //"int3 \n"
        "xor r10, r10 \n"         // R10 = 0x0 - Null out some registers
        "mul r10 \n"              // RAX&RDX = 0x0
        "add al, 0x60 \n"         // RAX = 0x60 = Offset of PEB Address within the TEB
        "mov rbx, gs:[rax] \n"    // RBX = PEB Address
        "mov rax, [rbx+0x20] \n"  // RAX = ProcessParameters Address
        "mov rbx, [rax+0x80] \n"  // RAX = Environment Address
        "mov rax, [rax+0x3f0] \n" // RBX = Environment Size
        "mov %[envAddr], rbx \n"
        "mov %[envSize], rax \n"
        :[envAddr] "=r" (envAddr),
  [envSize] "=r" (envSize)
    );
    BeaconPrintf(CALLBACK_OUTPUT, "[+] Evironment Address: %p",envAddr);
    BeaconPrintf(CALLBACK_OUTPUT, "[+] Evironment Size:    %d",envSize);
}
```

Compiling our BOF with Inline Assembly

We add the flag to our compile command, and for ease of use we make it into a bash script.

```
cat compile.cmds
x86_64-w64-mingw32-gcc -c whereami.x64.c -o whereami.x64.o -masm=intel
bash compile.cmds
```

Testing our Inline Assembly BOF

We do not need to reload our `whereami.cna` Agressor script because our script will use the contents of the `whereami.x64.o` object file that we just compiled with our bash script.

```
beacon> whereami
[*] Where Am I? BOF (Bobby Cooke//SpiderLabs|@0xBoku|github.com/boku7)
[+] host called home, sent: 300 bytes
[+] received output:
[+] Evironment Address: 0000000000071130
[+] received output:
[+] Evironment Size:    4242
```

Making our BOF Modular

Since we do not know how much we will want to expand or reuse this code in the future, we'll take some time to clean it up and make it more modular.

```
#include <windows.h>
#include "beacon.h"
PVOID getProcessParamsAddr()
```

```
PVOID getProcessParamsAddr()
{
    PVOID procParamAddr = NULL;
    __asm__(
        "xor r10, r10 \n"         // R10 = 0x0 - Null out some registers
        "mul r10 \n"              // RAX&RDX = 0x0
        "add al, 0x60 \n"         // RAX = 0x60 = Offset of PEB Address within the TEB
        "mov rbx, gs:[rax] \n"    // RBX = PEB Address
        "mov rax, [rbx+0x20] \n"  // RAX = ProcessParameters Address
        "mov %[procParamAddr], rax \n"
        :[procParamAddr] "=r" (procParamAddr)
    );
    return procParamAddr;
}
PVOID getEnvironmentAddr(PVOID procParamAddr)
{
    PVOID environmentAddr = NULL;
    __asm__(
        "mov rax, %[procParamAddr] \n"
        "mov rbx, [rax+0x80] \n"  // RBX = Environment Address
        "mov %[environmentAddr], rbx \n"
        :[environmentAddr] "=r" (environmentAddr)
        :[procParamAddr] "r" (procParamAddr)
    );
    return environmentAddr;
}
PVOID getEnvironmentSize(PVOID procParamAddr)
{
    PVOID environmentSize = NULL;
    __asm__(
        "mov rax, %[procParamAddr] \n"
        "mov rax, [rax+0x3f0] \n" // RAX = Environment Siz
        "mov %[environmentSize], rax \n"
        :[environmentSize] "=r" (environmentSize)
        :[procParamAddr] "r" (procParamAddr)
    );
    return environmentSize;
}
void go(char * args, int len) {
    PVOID procParamAddr = NULL;
    PVOID environmentAddr = NULL;
    PVOID environmentSize = NULL;
    procParamAddr = getProcessParamsAddr();
    environmentAddr = getEnvironmentAddr(procParamAddr);
    environmentSize = getEnvironmentSize(procParamAddr);
    BeaconPrintf(CALLBACK_OUTPUT, "[+] Evironment Address: %p",environmentAddr);
    BeaconPrintf(CALLBACK_OUTPUT, "[+] Evironment Size:    %d",environmentSize);
}
```

Compile & Test our Inline Assembly BOF

```
bobby.cooke$ cat compile.cmds
x86_64-w64-mingw32-gcc -c whereami.x64.c -o whereami.x64.o -masm=intel
bobby.cooke$ bash compile.cmds
```

```
beacon> whereami
[*] Where Am I? BOF (Bobby Cooke//SpiderLabs|@0xBoku|github.com/boku7)
[+] host called home, sent: 460 bytes
```

```
[+] host called home, sent: 400 bytes
[+] received output:
[+] Environment Address: 0000000000071130
[+] received output:
[+] Environment Size:    4242
```

- Looking good! Now we need to figure out how to parse out all those Unicode strings.

Resolving the Unicode Strings in the Enviroment Block

So far our BOF can get the size and address of the Environment block. We also saw earlier that the strings are just all mashed in there together, separated by a 2 byte `0x0000` delimiter. We will want to scan the Environment block, extract the strings, and output them to the Cobalt Strike interactive beacon console.

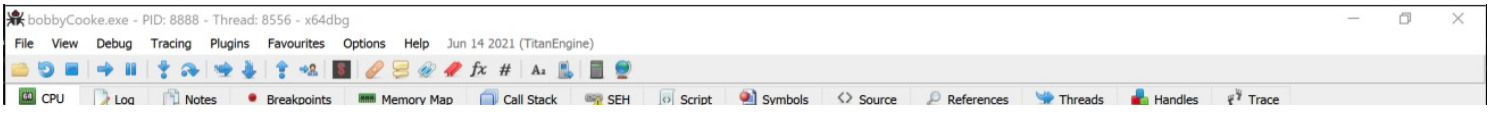
To make our shellcode that grabs the strings, we will fire up another `bobbyCooke.exe` beacon in x64dbg. We'll write and test our code right there in the x64dbg disassembly window.

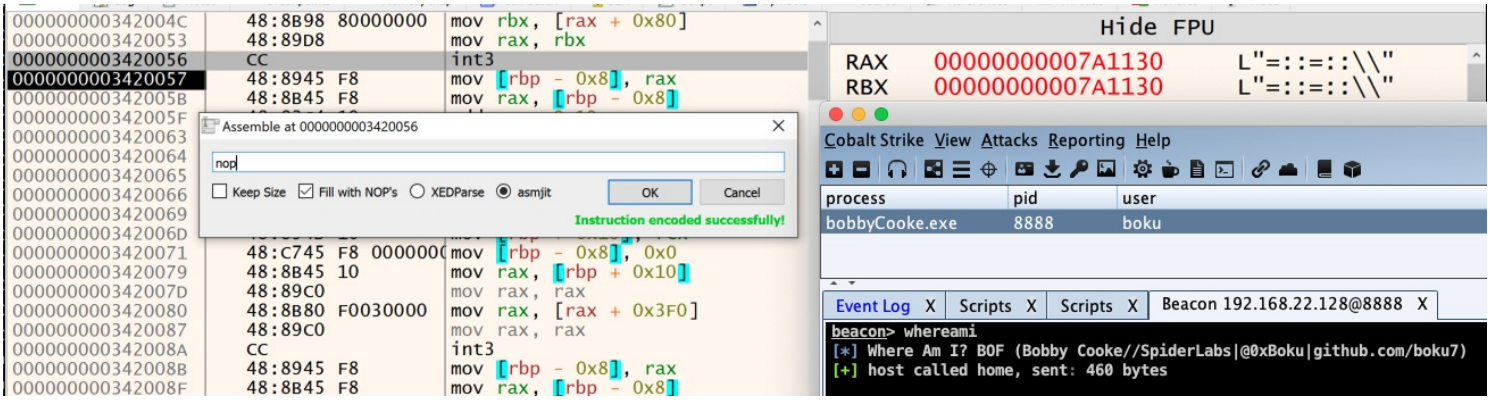
Breaking' on that BOF

Since we don't want to rewrite our entire program into the x64dbg window, we'll recompile our code with a breakpoint in it. After compilation, we'll attach to our beacon process. Then we'll run our BOF again from the interactive beacon console to trigger our breakpoint and work from there.

This is the BOF code with the breakpoints:

```
PVOID getEnvironmentAddr(PVOID procParamAddr)
{
    PVOID environmentAddr = NULL;
    __asm__(
        "mov rax, %[procParamAddr] \n"
        "mov rbx, [rax+0x80] \n"  // RBX = Environment Address
        "mov %[environmentAddr], rbx \n"
        "int3 \n" // <------------ Our BOF Breakpoints for debugging in x64dbg
        :[environmentAddr] "=r" (environmentAddr)
        :[procParamAddr] "r" (procParamAddr)
    );
    return environmentAddr;
}
PVOID getEnvironmentSize(PVOID procParamAddr)
{
    PVOID environmentSize = NULL;
    __asm__(
        "mov rax, %[procParamAddr] \n"
        "mov rax, [rax+0x3f0] \n" // RAX = Environment Siz
        "mov %[environmentSize], rax \n"
        "int3 \n" // <------------ Our BOF Breakpoints for debugging in x64dbg
        :[environmentSize] "=r" (environmentSize)
        :[procParamAddr] "r" (procParamAddr)
    );
    return environmentSize;
}
```

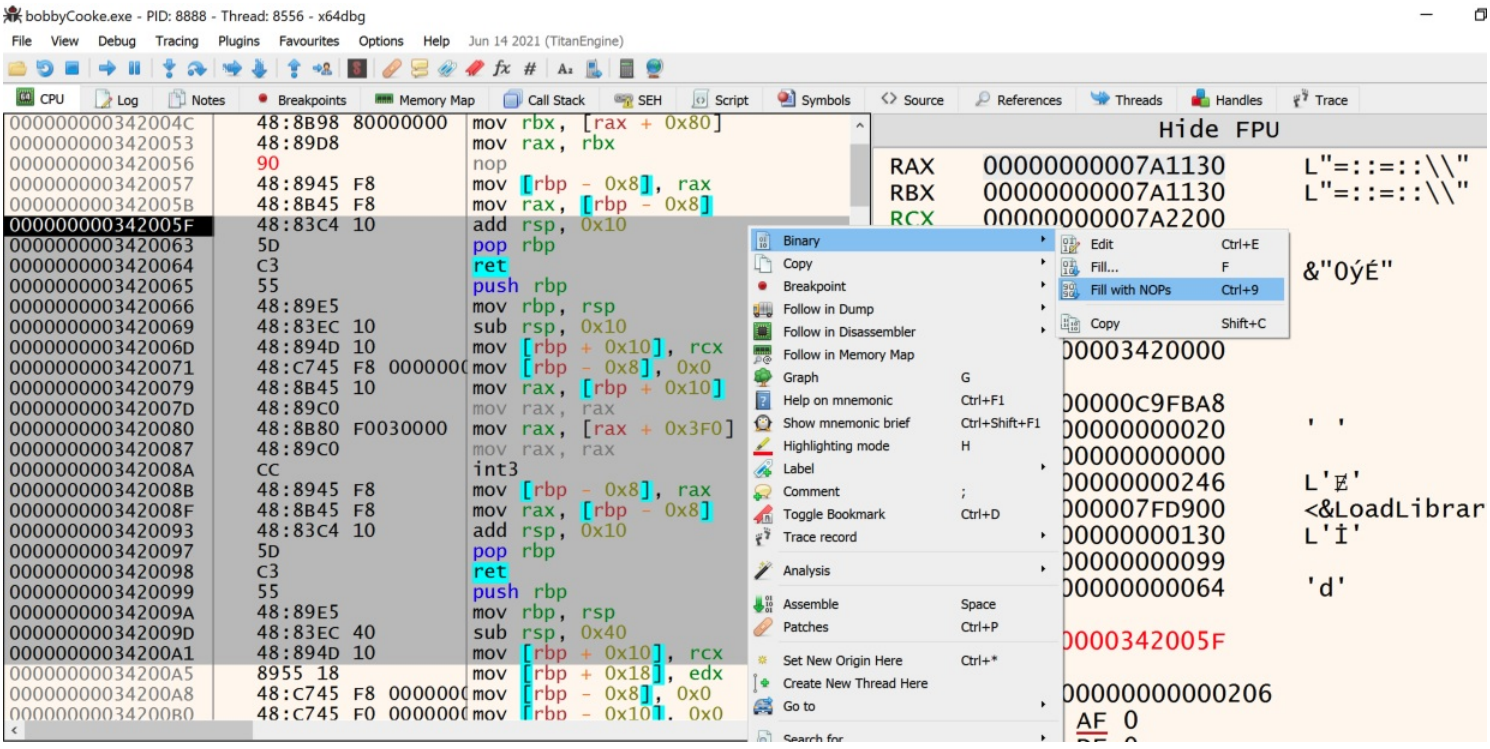




- We trigger the breakpoint by using our `whereami` command from the Cobalt Strike beacon console.
- We catch the breakpoint because we are debugging our beacon process with x64dbg. If you are not debugging, then this will likely kill your beacon.
- First thing we'll need to do after hitting our BOF breakpoint is `nop` out the `int3` instruction. This will allow us to step forward in our code.
- We see that the `RAX` register has the address of our Environment because of that first Unicode string displayed by the `RAX` register.
- We also see that our `PVOID environmentAddr` variable exists on the stack at the location `[rbp-0x8]`.

Creating a Workspace

We'll want some room to work, and less confusing is better. Since we see that the `environmentAddr` is going to be saved on the stack at `[rbp-0x8]`, and the next instruction loads that in `rax`, we will work from there. We select a big amount of memory in the disassembler after the `mov rax,[rsp-0x8]` instruction, and right click to NOP it out.



Resolving Unicode Delimiters via String Size

To list out all the Unicode strings, we first need to find where they end. Once we know where the first-string ends, we can print it out, and then move to the next. We'll continue to do this for all the Unicode strings until we exhaust the size of the environment.

After tinkering around in x64dbg, the getUnicodeStrLen() function has been added to the code. This will return the length of our Unicode string. For our test we will then print the Unicode string using `BeaconPrintf()` with `%ls`.

```
PVOID getUnicodeStrLen(PVOID envStrAddr)
{
    PVOID unicodeStrLen = NULL;
```

```
    PVOID unicodeStrLen = NULL;
    __asm__(
        "mov rax, %[envStrAddr] \n"
        "xor rbx, rbx \n" // RBX is our 0x00 null to compare the string position too
        "xor rcx, rcx \n" // RCX is our string length counter
    "check: \n"
        "inc rcx \n"
        "cmp bl, [rax + rcx] \n"
        "jne check \n"
        "inc rcx \n"
        "cmp bl, [rax + rcx] \n"
        "jne check \n"
        "mov %[unicodeStrLen], rcx \n"
        :[unicodeStrLen] "=r" (unicodeStrLen)
        :[envStrAddr] "r" (envStrAddr)
    );
    return unicodeStrLen;
}
void go(char *args, int len)
{
    PVOID procParamAddr = NULL;
    PVOID environmentAddr = NULL;
    PVOID environmentSize = NULL;
    PVOID unicodeStrSize = NULL;
    procParamAddr = getProcessParamsAddr();
    environmentAddr = getEnvironmentAddr(procParamAddr);
    environmentSize = getEnvironmentSize(procParamAddr);
    unicodeStrSize = getUnicodeStrLen(environmentAddr);
    BeaconPrintf(CALLBACK_OUTPUT, "[+] Environment Address: %p",environmentAddr);
    BeaconPrintf(CALLBACK_OUTPUT, "[+] Environment Size:    %d",environmentSize);
    BeaconPrintf(CALLBACK_OUTPUT, "[+] 1st String Size:    %d",unicodeStrSize);
    BeaconPrintf(CALLBACK_OUTPUT, "[+] 1st String Value:   %ls",environmentAddr);
}
```

We test our BOF again and confirm it is working correctly.

```
beacon> whereami
[*] Where Am I? BOF (Bobby Cooke//SpiderLabs|@0xBoku|github.com/boku7)
[+] host called home, sent: 716 bytes
[+] received output:
[+] Environment Address: 0000000000751130
[+] received output:
[+] Environment Size:    4242
[+] received output:
[+] 1st String Size:    14
[+] received output:
[+] 1st String Value:   =::=::\
```

- We can see that we are successfully printing the first Unicode string from our Environment block into the interactive beacon console.

Looping through all the Unicode Environment Strings

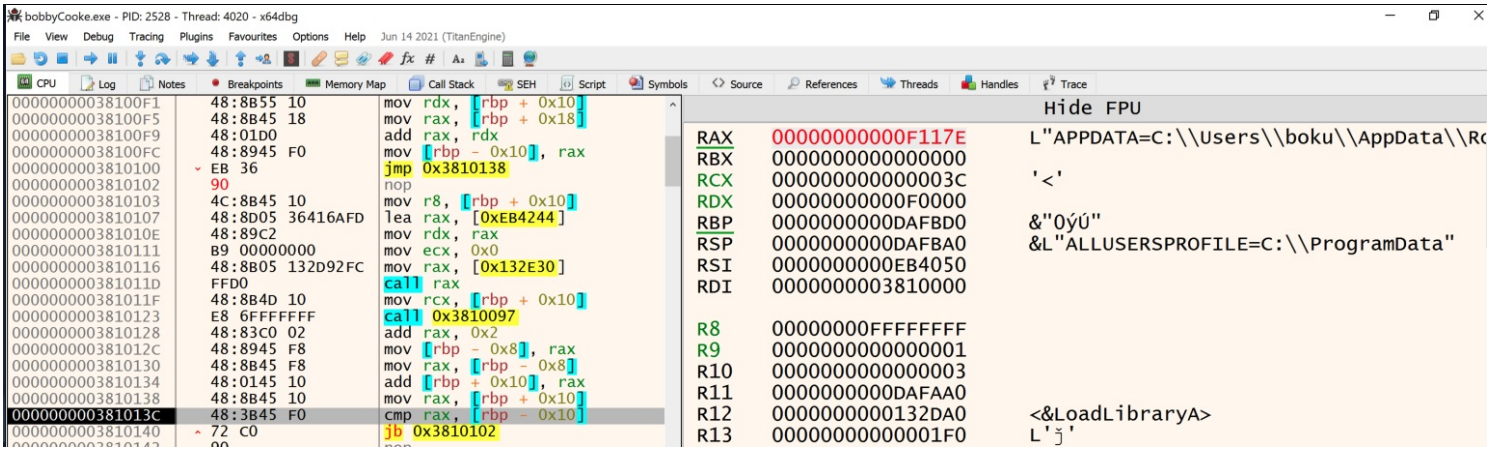
Now we add some code to loop through all the environment Unicode strings and output them to the Cobalt Strike interactive beacon console.

Our Looper Code

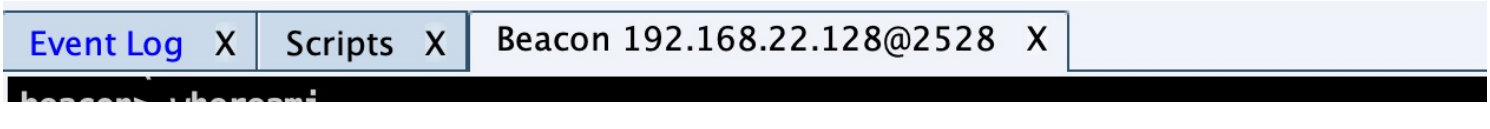
```
void printLoopAllTheStrings(PVOID nextEnvStringAddr, unsigned __int64 environmentSize)
{
    PVOID unicodeStrSize = NULL;
```

```
    PVOID environmentEndAddr = nextEnvStringAddr + environmentSize;
    while (nextEnvStringAddr < environmentEndAddr)
    {
        __asm__(
            "int3 \n"
        );
        BeaconPrintf(CALLBACK_OUTPUT, "%ls",nextEnvStringAddr);
        unicodeStrSize = getUnicodeStrLen(nextEnvStringAddr)+2;
        nextEnvStringAddr += (unsigned __int64)unicodeStrSize;
    }
}
void go(char *args, int len)
{
    PVOID procParamAddr = NULL;
    PVOID environmentAddr = NULL;
    PVOID environmentSize = NULL;
    procParamAddr = getProcessParamsAddr();
    environmentAddr = getEnvironmentAddr(procParamAddr);
    environmentSize = getEnvironmentSize(procParamAddr);
    BeaconPrintf(CALLBACK_OUTPUT, "[+] Environment Address: %p",environmentAddr);
    BeaconPrintf(CALLBACK_OUTPUT, "[+] Environment Size:    %d",environmentSize);
    printLoopAllTheStrings(environmentAddr, (unsigned __int64)environmentSize);
}
```

- This code adds the `printLoopAllTheStrings()` function which loops through all the Unicode strings in the Environment block and then prints them to the beacons console using `BeaconPrintf()`.
- The loop uses the `getUnicodeStrLen()` function we created to find the offset of the next environment string.
- After adding our current environment address with the Unicode string length for our current string, we add 2 bytes to compensate for the `0x0000` delimiter. Now we will be at the start of the next Unicode string.



- We set the breakpoint so we could tinker with our code and ensure it works.
- We see that the loop is working and loading the next Unicode string address into `RAX`!



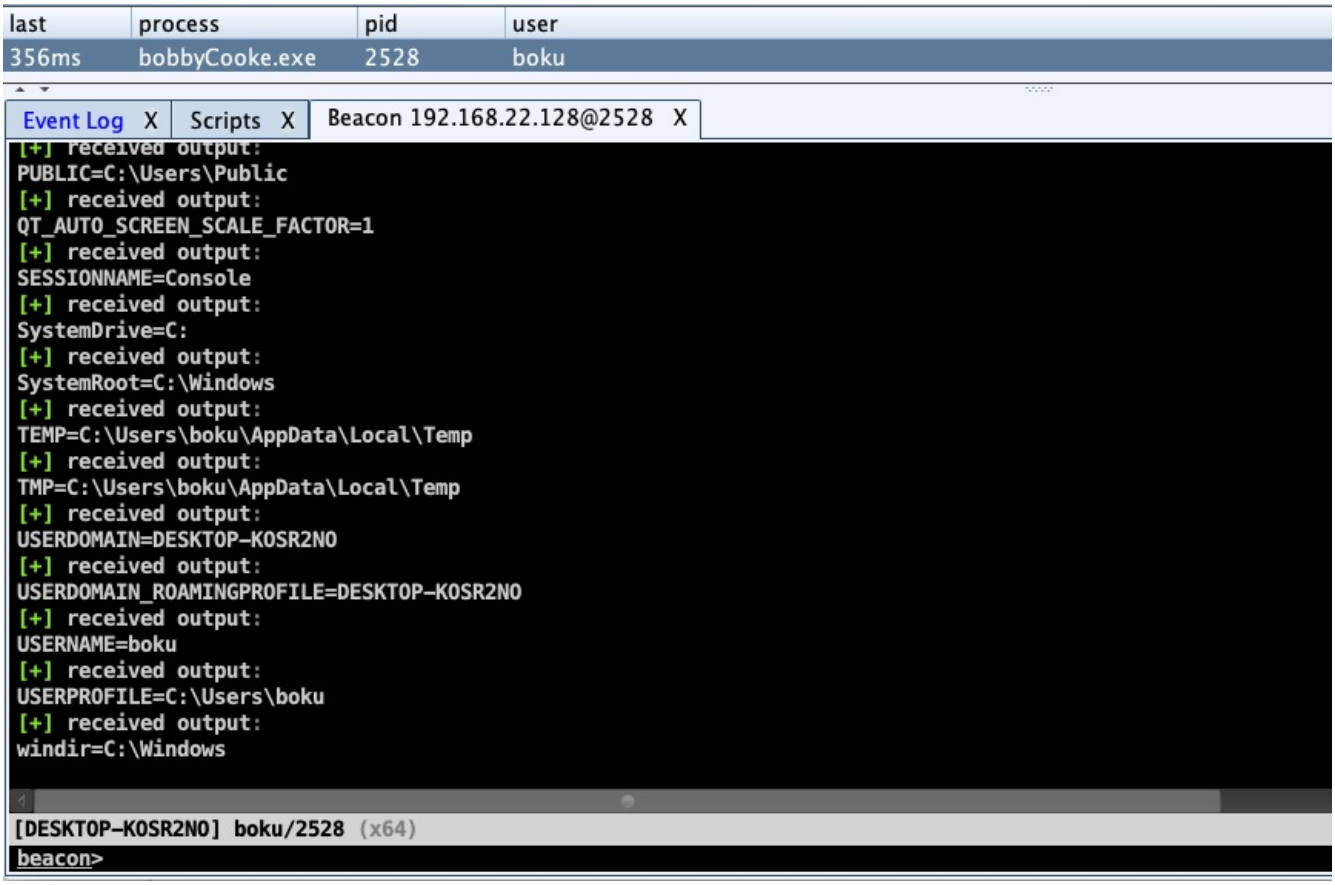
```
beacon> whereami
[*] Where Am I? BOF (Bobby Cooke//SpiderLabs|@0xBoku|github.com/boku7)
[+] host called home, sent: 676 bytes
[+] received output:
[+] Evironment Address: 00000000000F1130
[+] received output:
[+] Evironment Size:    4242
[+] received output:
=::=::\
[+] received output:
ALLUSERSPROFILE=C:\ProgramData

[DESKTOP-KOSR2NO] boku/2528 (x64)
beacon>
```

- As we step through the loops, we can see the environment strings outputting to our beacons console!

Great Success!

Our “Where Am I?” BOF code is working! Also, we can see by resuming the code in the debugger, that we successfully output all the environment strings and do not crash the beacon process!



For the full code to the project see the GitHub repo:

- GitHub - boku7/whereami (https://github.com/boku7/whereami)

References/Resources

- Matt Eidelberg’s DEF CON 29 talk Operation Bypass Catch My Payload If You Can (https://youtu.be/JXKNdWUs77w)

Sektor7 Courses

- https://institute.sektor7.net/

Raphael Mudge - Beacon Object Files - Luser Demo

- https://www.youtube.com/watch?v=gfYswA_Ronw

Cobalt Strike - Beacon Object Files

- https://www.cobaltstrike.com/help-beacon-object-files

Implementing ASM in C Code with GCC

- https://outflank.nl/blog/2020/12/26/direct-syscalls-in-beacon-object-files/
- https://www.cs.uaf.edu/2011/fall/cs301/lecture/10_12_asm_c.html
- http://gcc.gnu.org/onlinedocs/gcc-4.0.2/gcc/Extended-Asm.html#Extended-Asm

BOF Code References

trustedsec/CS-Situational-Awareness-BOF

- https://github.com/trustedsec/CS-Situational-Awareness-BOF

anthemtotheego/InlineExecute-Assembly

- https://github.com/anthemtotheego/InlineExecute-Assembly/blob/main/inlineExecuteAssembly/inlineExecute-Assembly.cna

ajpc500/BOFs

- https://github.com/ajpc500/BOFs/

Updated: August 19, 2021