Your BOFs are gross, put on a mask: How to hide beacon during BOF execution

June 28, 2023

By Joshua Magri

8 min read

Adversary Services Incident Response Intelligence & Analytics Threat Hunting X-Force

In this post, we'll review a simple technique that we've developed to encrypt Cobalt Strike's Beacon in memory while executing BOFs to prevent a memory scan from detecting Beacon.

in memory and calling back to you. The hard part is over, time to do some post-exploitation. You fire up your trusty BOF toolkit and watch the “last” timer tick up indefinitely.

While an initial beacon can go undetected, performing common post-exploitation activities from a Beacon Object File can trigger a memory scan of your process by EDR. This can result in an EDR product finding your Beacon sitting in memory and killing the process.

Cobalt Strike (somewhat) recently introduced the Sleep Mask functionality, which serves to hide Beacon in memory while it’s sleeping. This helps prevent detection by threat hunting tools or memory scanners that look for Beacon signatures or suspicious artifacts like unbacked executable memory. As of Cobalt Strike 4.7, Sleep Mask is implemented as a BOF, which provides the operator with much more control over how Sleep Mask works. This demonstrates that it is possible to have Beacon encrypted and sleeping during BOF execution. However, during normal BOF execution, Beacon is sitting in memory. Let’s look at how to change that.

Beacon object file basics

If you’re unfamiliar with the internals of Beacon Object Files (BOFs), they’re essentially a way to write position independent code where Beacon handles loading and linking any dependencies. This allows operators to quickly develop post-exploitation tooling without the hassle of writing shellcode or reflective DLLs.

When you execute a BOF, it looks something like this:

1. Beacon allocates memory according to your Malleable C2 settings and writes the BOF content
2. Beacon loader handles linking any imported functions and finding the specified entry point of the BOF
3. Execution is passed to the entry point, your BOF content runs, and Beacon resumes executing
4. Allocated BOF memory is cleaned up according to your Malleable C2 settings

This blog is not intended to be a reference on BOFs, so you can find more information about BOFs here:

Finding Beacon's base address

To mask Beacon in memory, we need to know its base address and its size. There are a few ways we could figure out this information, but the method I found to be most reliable uses a bit of assembly and the VirtualQuery API.

When a BOF is executed, and we call a function from our BOF entry point, our stack frame looks like this at the top:

1. Current function
2. BOF entry point
3. Beacon

Below is a snippet of assembly to go back two stack frames. This will get us from our function to our BOF entry point, to the return address of Beacon. This will give us the address that Beacon will resume executing from after our BOF finishes, which is inside Beacon's .text section.

```
mov r8, [rbp]
mov rcx, [r8]
mov rax, [r8+0x8] ; RDX holds Beacon's return address
```

Now that we have an address for Beacon's memory range, we need to find its base address. We can accomplish this with two calls to VirtualQuery. The first one will get us the base address of the region that the previous address is in, and the second will give us the base address and size of the allocation that was made for Beacon. These second two values are what we will need for masking.

Accounting for malleable C2 and UDRLs

One of Beacon's greatest features is how it exposes flexibility to operators at many points. There are two main mechanisms for changing how Beacon is loaded into memory: Malleable C2 settings and User Defined Reflective Loaders. Here are some links diving into both:

- Defining the Cobalt Strike Reflective Loader, Security Intelligence
- PE and Memory Indicators, Cobalt Strike

VirtualQuery on a region of memory, it will return results for all of the following pages in memory that share the same attributes (memory protection and page state). So if Beacon is allocated entirely with RWX permissions then calling VirtualQuery twice works perfectly. But if you properly set the memory protections for each section of Beacon, then calling VirtualQuery on the base address of Beacon will only return the size of the NT Header section, since it will have a different memory protection setting than the .text section.

Thankfully, compensating for this is pretty straightforward. We can query the next region in memory and verify that it is executable and that the return address we got for Beacon earlier falls within that region. If it does, then we've found Beacon's .text section!

Masking Beacon

Now that we have the base address and size of Beacon's .text section, we can change the page protection and then apply our mask.

```
KERNEL32$VirtualProtect(beaconBaseAddress, beaconSize, PAGE_READWRITE, &OldProtect);
DWORD start = 0;
while (start < beaconSize) {
   *(beaconBaseAddress + start) ^= mask[start % MASK_SIZE];
   start++;
}
```

In the snippet above, we call VirtualProtect to change Beacon's page protection to RW, and then apply a simple XOR mask across our Beacon. We generate this random mask once in our setup function for simplicity. To unmask Beacon, we just reverse the order of these two operations by applying the XOR again and changing Beacon's page protection to whatever it was before (RWX or RX).

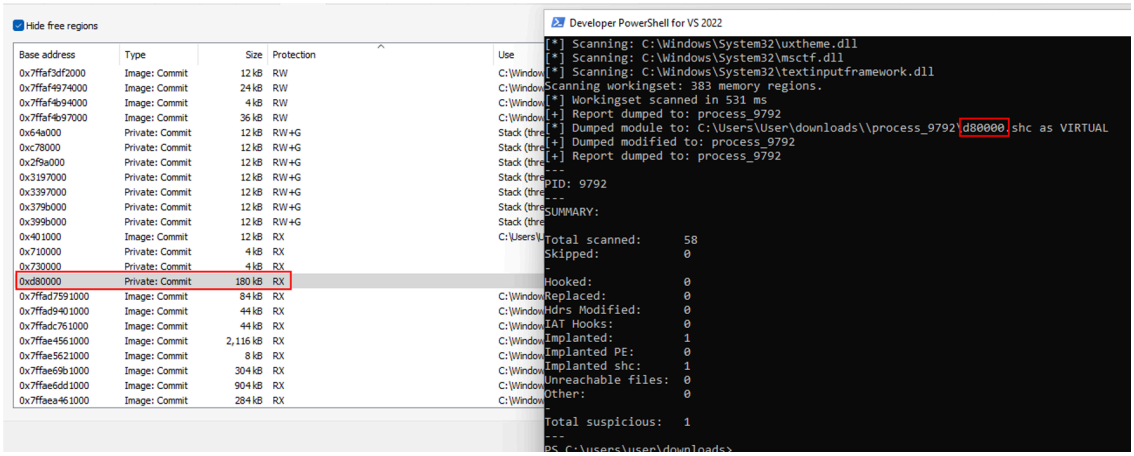
Demonstration

For demonstration purposes, we have a BOF that just calls MessageBoxA to block execution and give us an opportunity to scan the memory of our Beacon process with some common memory analysis tools: Moneta, PESieve, and YARA.

Here is Moneta reporting three unbacked RX regions. This is our Beacon, our Sleep Mask BOF, and the BOF we're currently executing. This may look different for you depending on your Malleable C2 profile.

```
beacon.exe : 9792 : x64 : C:\Users\User\Downloads\beacon.exe
  0x0000000000400000:0x0004f000    | EXE Image           | C:\Users\User\Downloads\beacon.exe | Unsigned module
  0x0000000000710000:0x00002000    | Private
    0x0000000000710000:0x00001000 | RX        | 0x00000000 | Abnormal private executable memory
  0x0000000000730000:0x00002000    | Private
    0x0000000000730000:0x00001000 | RX        | 0x00000000 | Abnormal private executable memory
  0x000000000D80000:0x0007b000     | Private
    0x000000000D80000:0x0002d000  | RX        | 0x00000000 | Abnormal private executable memory
```

Here is PE-Sieve with the /shellc and /data three options dumping out a region that we can see is our Beacon.



Here is a YARA detection from this set of rules published by Elastic to detect Cobalt Strike.

```
PS C:\users\user\downloads> .\yara-4.2.3-2029-win64\yara64.exe C:\users\user\cobalt_strike.yar 9792
warning: rule "Windows_Trojan_CobaltStrike_8519072e" in C:\users\user\cobalt_strike.yar(821): string "$a4" may slow down
 scanning
Windows_Trojan_CobaltStrike_663fc95d 9792
```

With Masking

Now we can apply our masking code to hide Beacon in memory and try again. Here is Moneta reporting on our Sleep Mask BOF and our currently executing BOF, but not our Beacon.


```
beacon.exe : 9792 : x64 : C:\Users\User\Downloads\beacon.exe
  0x0000000000400000:0x0004f000    | EXE Image           | C:\Users\User\Downloads\beacon.exe | Unsigned module
  0x0000000000710000:0x00002000    | Private
    0x0000000000710000:0x00001000 | RX        | 0x00000000 | Abnormal private executable memory
  0x0000000000730000:0x00002000    | Private
    0x0000000000730000:0x00001000 | RX        | 0x00000000 | Abnormal private executable memory
```

PE-Sieve output with zero detections.

```
---
SUMMARY:

Total scanned:      61
Skipped:            0
-
Hooked:             0
Replaced:           0
Hdrs Modified:      0
IAT Hooks:          0
Implanted:          0
Unreachable files:  0
Other:              0
-
Total suspicious:   0
---
```

And no detections from YARA.

```
PS C:\users\user\downloads> .\yara-4.2.3-2029-win64\yara64.exe C:\users\user\cobalt_strike.yar 9792
warning: rule "Windows_Trojan_CobaltStrike_8519072e" in C:\users\user\cobalt_strike.yar(821): string "$a4" may slow down
 scanning
```

As you can see, this technique does yield results and should help prevent memory scanners from detecting our Beacon via signature-based detections or memory artifacts during BOF execution. The presence of our current BOF and our Sleep Mask BOF are not ideal, as they are unbacked RX region IOCs, but EDR may not alert solely on this if no signatures are identified during the memory scan.

If these unbacked memory regions are a concern, it is relatively simple to identify and mask the Sleep Mask BOF in a similar fashion. As for masking the current BOF, it should be possible to mask with some existing ROP techniques, but it gets very difficult for more complex BOFs.

The most important thing to note with this technique is that you CANNOT call Beacon APIs from your BOF while Beacon is encrypted — this means any of the internal Beacon APIs like BeaconPrintf, BeaconSpawnInject, etc. Since these functions are located in the .text section of Beacon you will be passing execution to non-executable garbage code and your Beacon will die. If you have output that you need to get from your BOF then you can either send it

Integration with existing tooling

To allow red teams to simulate more advanced threat actors and allow blue teams to be more familiar with memory scanning evasion techniques, we wanted to implement this technique so that integrating it with your BOF arsenal is as easy as possible. To that end, we've published this project as a single C header file that you can include in your existing BOF. You just need to call the GetBeaconBaseAddress function before calling MaskBeacon for the first time, and then you can call the MaskBeacon/UnmaskBeacon functions to toggle the mask. An example BOF entrypoint would look like this.

```
#include "sleepbof.h"
void go(char* args, int len){
    //YOUR CODE HERE, YOU CAN CALL BEACON APIS
    GetBeaconBaseAddress();
    //SERIOUSLY DO ANY ARG PARSING BEFORE THIS!
    MaskBeacon();

    //YOUR CODE HERE, DONT CALL ANY BEACON APIS

    UnmaskBeacon();
    //YOUR CODE HERE, YOU CAN CALL BEACON APIS
}
```

If you want to call Beacon APIs inside of your code, you can just toggle the mask like this.


```
//suspicious code
UnmaskBeacon();
BeaconPrintf(CALLBACK_OUTPUT, "Output from suspicious code: %s", output);
MaskBeacon();
//resume being sneaky
```

Stability is always a concern when operating over a C2 channel, and errors in a BOF are a great way to kill your hard-earned Beacon. We have tried to make this code as reliable and stable as possible, but there is always the

during execution as a result of the masking. The same goes for using this code with any complicated loaders — you should ensure that the .text section of Beacon is properly located before executing. Some debugging directives have been included to help with troubleshooting.

Detections

As with any C2 related subject matter, detection is a chief concern. However, detecting BOF-specific execution is not a particularly useful area to focus on. BOFs are ultimately just position independent code being loaded with a handful of benign API calls. Detection efforts are much better spent on detecting Beacon execution and post-exploitation activity. For a BOF to be useful it must generate some activity on the host or the network, and hunting these behaviors is far more fruitful. Some example BOF activities could include enumerating the local host or Active Directory, credential dumping activities, or injecting into another process.

All of that said, this technique does leave the executing BOF and a Sleep Mask BOF in memory as unbacked RX (or RWX) regions. These are generally good indicators of malicious activity for threat hunters and memory scanners alike. However, as mentioned above, there are ways that these artifacts can be hidden by a skilled operator.

Below is a non-comprehensive list of resources that you can use for detecting Cobalt Strike and performing memory analysis.

- Cobalt Strike YARA rules, Elastic
- PE-Sieve, Hasherezade
- Moneta, Forrest Orr
- Hunt Sleeping Beacons, @thefLinkk
- BeaconEye, Cern Coburn
- Cobalt Strike, A Defender's Guide, The DFIR Report

Conclusion

In this post, we've shown how we can apply the same principle as Beacon's Sleep Mask kit to provide some extra OPSEC to Beacon during BOF execution. We believe that this is a relatively simple technique that can

You can find the published header file here.

Learn about adversary simulation services from IBM X-Force here.

CobaltStike | EDR | endpoint detection and response (EDR) | red team | threat hunting | X-Force

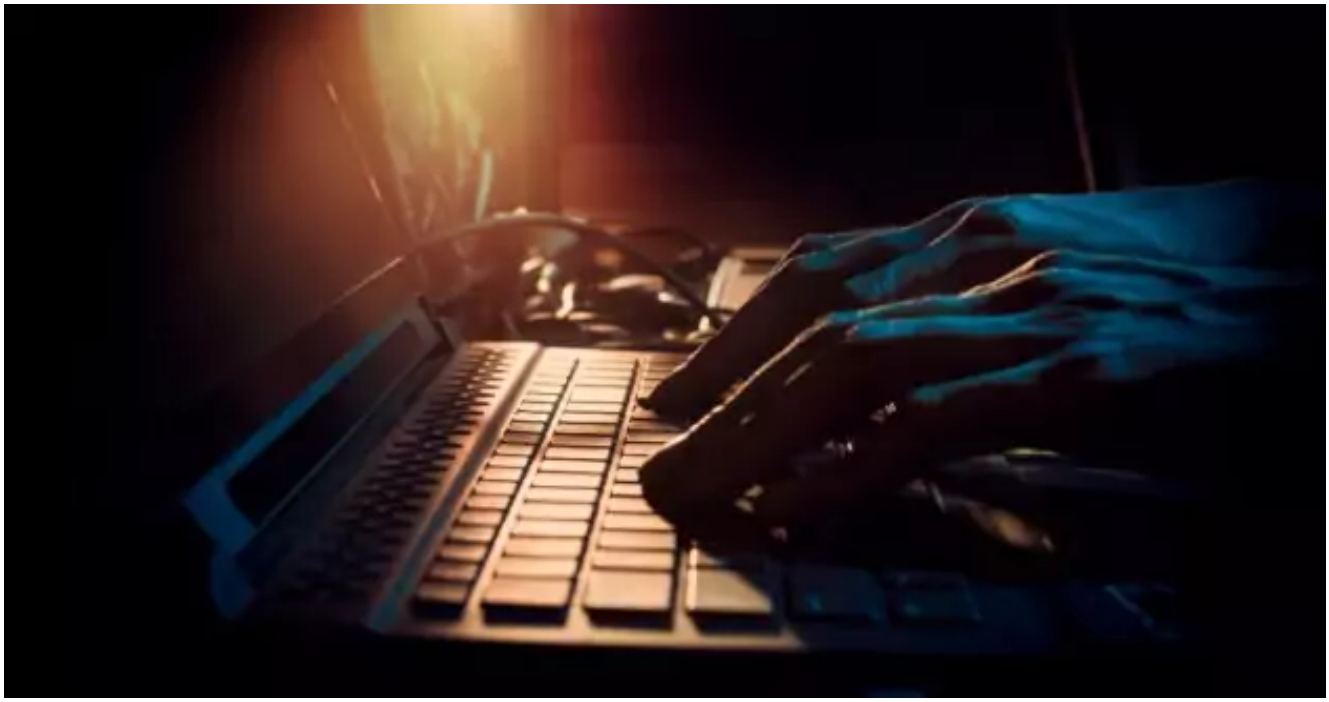
---

Joshua Magri

Senior Managing Security Consultant, Adversary Services, IBM X-Force Red

POPULAR

4 min read - Since its launch in August 2013, Telegram has become the go-to messaging app for privacy-focused users. To start using the app, users can sign up using either their real phone number or an anonymous number purchased from the Fragment blockchain...



DATA PROTECTION | November 8, 2024

SpyAgent malware targets crypto wallets by stealing screenshots

4 min read - A new Android malware strain known as SpyAgent is making the rounds — and stealing screenshots as it goes. Using optical character recognition (OCR) technology, the malware is after cryptocurrency recovery phrases often stored in screenshots on user devices. Here's...

12 min read - As of November 2024, IBM X-Force has tracked ongoing Hive0145 campaigns delivering Strela Stealer malware to victims throughout Europe – primarily Spain, Germany and Ukraine. The phishing emails used in these campaigns are real invoice notifications, which have been stolen...

MORE FROM ADVERSARY SERVICES

September 4, 2024

Getting “in tune” with an enterprise: Detecting Intune lateral movement

13 min read - Organizations continue to implement cloud-based services, a shift that has led to the wider adoption of hybrid identity environments that connect on-premises Active Directory with Microsoft Entra ID (formerly Azure AD). To manage devices in these hybrid identity environments, Microsoft Intune (Intune) has emerged as one of the most popular device management solutions. Since this trusted enterprise platform can...

June 13, 2024

Q&A with Valentina Palmiotti, aka chompie

4 min read - The Pwn2Own computer hacking contest has been around since 2007, and during that time, there has never been a female to score a full win — until now.This milestone was reached at Pwn2Own 2024 in Vancouver, where two women, Valentina Palmiotti and Emma Kirkpatrick, each secured full wins by exploiting kernel vulnerabilities in Microsoft Windows 11. Prior to this year, only Amy Burnett and Alisa Esage had compet...

By Industry

X-Force

Events

About Us

Exclusive Series

Podcast

Contact

Follow us on social