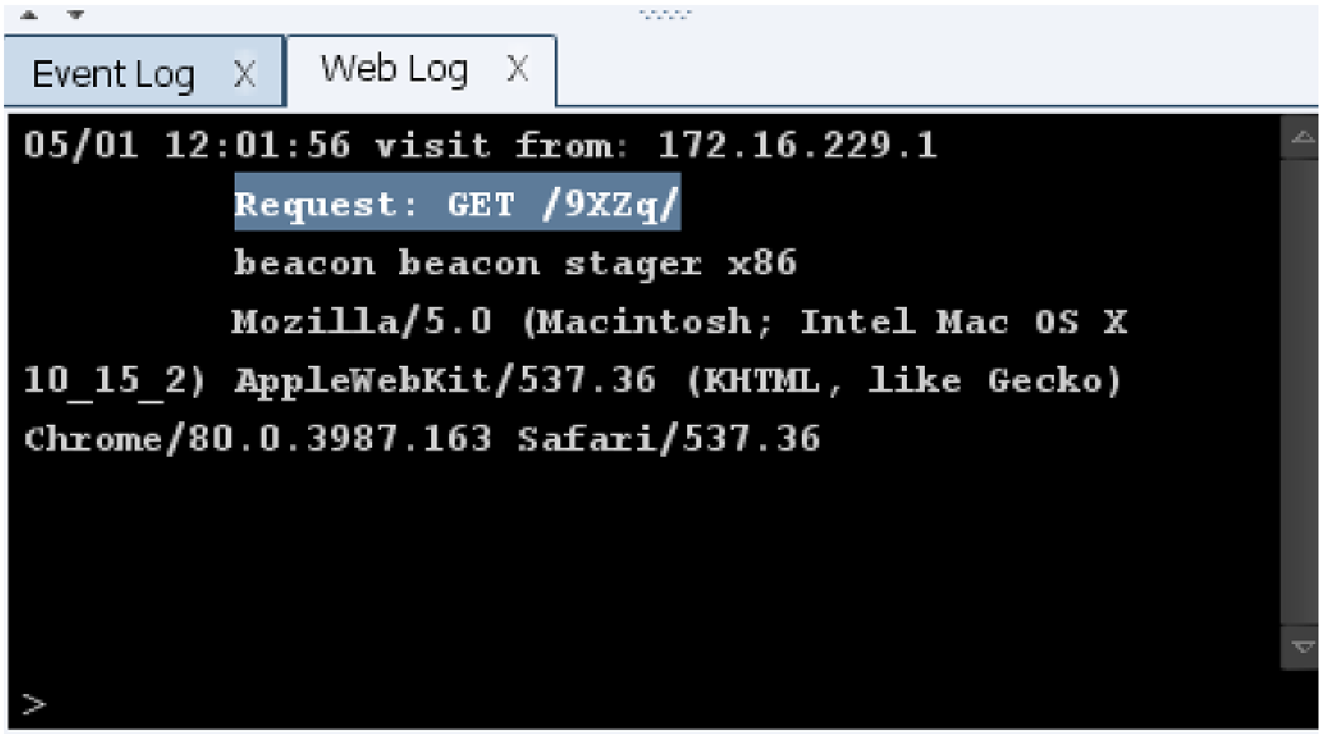
Striking Back at Retired Cobalt Strike: A look at a legacy vulnerability

research.nccgroup.com/2020/06/15/striking-back-at-retired-cobalt-strike-a-look-at-a-legacy-vulnerability/

June 15, 2020



This vulnerability applied to a 5 year old end of life version of CobaltStrike and is being published in the spirit of archaeological interest in the vulnerability.

tl;dr

This blog looks at some of the communication and encryption internals of Cobalt Strike between Beacons and the Team Server in the 3.5 family. We then explore the subsequent exploitation of a vulnerability in Cobalt Strike 3.5 from 2016 to achieve remote unauthenticated code execution on the Team Server.

We hope that this post will help Blue Teams with detection engineering and provide a good understanding of the encryption fundamentals that underpin Cobalt Strike.

For the Red Team, we provide an example of why it is important to harden your Command and Control infrastructure.

Back Story

In Cobalt Strike there was a vulnerability fixed that existed in a number of versions:

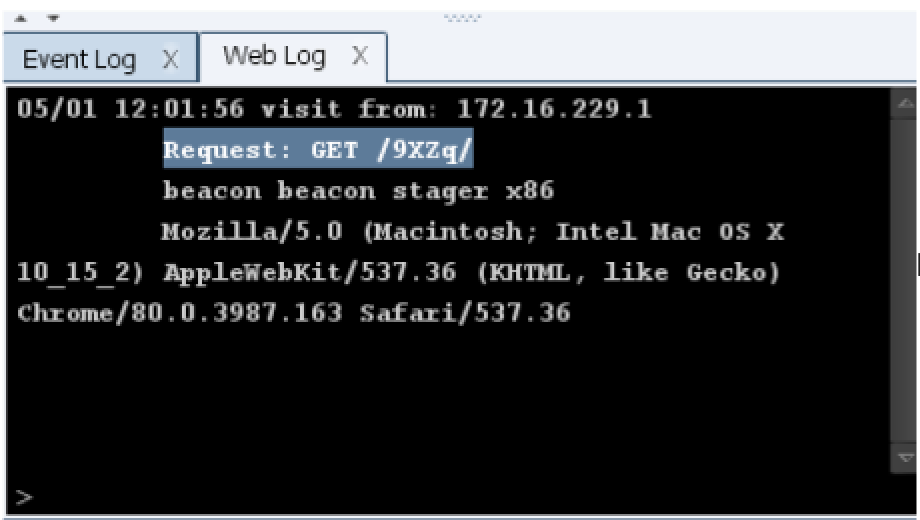
- Cobalt Strike <= 3.5
- Cobalt Strike 3.5-hf1 (hot-fix addressing in-the-wild exploit chain)
- Cobalt Strike 3.5-hf2 (further hardening)

The vulnerability was disclosed by the team at Cobalt Strike in 2016 as being actively exploited in September. A patch was promptly released in the guise of 3.5.1.

Beacon Staging Primer

Beacon staging is the process of downloading a beacon (DLL) shellcode blob, which will be executed via a smaller shellcode stager – typically as a result of an exploit or dropper document. The aim here being to work around size-constrained vulnerability exploitation, for example where you only have a certain amount of space to hold your shellcode as the result of a buffer overflow or similar. That said, from a Red Team Operational perspective, fully staged (a.k.a Stageless) payloads are always preferred where possible.

By default, Cobalt Strike supports the Meterpreter staging protocol and exposes its stager URL via the checksum8 format .



Retrieving stager via Checksum8

Since Cobalt Strike 3.5.1, you can now also disable staging entirely using the “*host_stage = false*” setting. This was added as a feature following the official fix for the vulnerability discussed in this post.

After the stager shellcode is downloaded, a custom XOR encoder is used to decode the rest of the shellcode, before execution is passed to the decoded beacon DLL. The XOR encoder used will not be discussed in the post as this is a feature of the licensed version of Cobalt Strike.

After the DLL is extracted from the stager blob, the beacon settings can be extracted, along with the public key, using a fixed XOR key of 0x69. This was recently published by the SentinelOne team, who released the CobaltStrikeParser tool.

The Internals of Cobalt Strike Beacon Comms

Once decoded and executed, the beacon then needs to communicate with the Team Server. This involves various Cobalt Strike communications and encryption internals we need to understand prior to being able to build an exploit payload.

Beacon Checking In – 1..2..3 – Over: Keys Keys Keys..

Whenever beacon checks in, it sends an encrypted metadata blob. This is encrypted using the RSA public key, extracted from the stager. To aid in debugging you may also wish to dump the RSA private key from the Team Server.

This can be achieved using the following Java code, running on the Team Server. Private keys are serialized in a file named ".cobaltstrike.beacon_keys", in the same folder as the Team Server files.

To compile/run this code, you will need set set your classpath to the cobaltstrike.jar file (e.g. -cp cobaltstrike.jar)
When run, the output will look like this:

```
root@cobaltstrike:/opt/cobaltstrike-3.5# ls -l .cobaltstrike.beacon_keys
-rw-r--r-- 1 root root 1447 Apr 21 15:35 .cobaltstrike.beacon_keys
root@cobaltstrike:/opt/cobaltstrike-3.5# java -cp "cobaltstrike.jar" DumpKeys.java
Private Key: MIICdwIBADANBgkqhkiG9w0BAQEFAASCAmEwggJdAgEAAoGBAJVY4v8Qezfp4pSArCfw0ACTqi3pCFGAuWYFmhTV/iENgcnp
p2vWMkhI5nU/tW9uD2JhNerbCHWrsRCC70T9C2AnE4gh9wXAgMBAAECgYBcrEJ3WefMA2LpGYs6YZEAuqCgSnkx8fmZmCJLiZpfMj12aCXRws
I9DH15Zux4zBiwOM5p9nF7pOdSs4XJZpQYkhthZlL0QJBANXPjrJrV3rOMS2zuqze3xKNoUjtPwv7hwj0p7NmCtETGwiWNpgv/egicw7lOcKg
mKnLseH/Oy1x2TWoK6sKsyTGygtIGecZTxfAkEAxnApt0xK165xFEKf+furu9N5Im8Wua9Lp7Mxxh3p4hvCVljb+KlqFT2L3gI/dnQw5S2OYj
4Tni2pPbGndb8w5CGIwTc28J7OwJBAKcvu9cKP+wvwk3fTFSctNhxErTzSCgBOzSKm3hNxau5ZpQ7R782pxw0/os5PNkBaJZvSqEZ0oER3Yiq

Public Key: MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCVWOL/EHs36eKUgKwn8NAAk6ot6QhRgLlmBZoU1f4hDYHJ6SASli25RKOgn
7Vvbg9iYTXq2wh1q7EQgu9E/QtgJxOIIfcFwIDAQAB
```

Dumping Keys

It should be noted that this is strictly only to aid in debugging whilst writing an exploit. In a real-world scenario it is not possible to decrypt *existing* Beacon communications as the keys are negotiated securely over RSA, with the beacon only having the public key. However, if you are in possession of the public key (which can be retrieved via the checksum8 staging URL), then it is possible to encrypt and decrypt taskings via a fake session.

Beacon Communication Encryption and Metadata

Encryption, Decryption and Structure

Metadata from the beacon is sent according to the settings in the malleable C2 profile. This allows the operator to customise various properties of the traffic, such as where the metadata blob is sent (.e.g in a header, or a cookie), and how it is encoded. The following is from the Cobalt Strike blog example.

In this example, the metadata will be sent Base64 encoded as a Cookie named “user”.

```
Malleable C2 Config
http-get {
   set uri "/foobar";
    client {
       metadata {
          base64;
          prepend "user=";
          header "Cookie";
    }
}
```

The following HTTP request capture shows a metadata blob being sent Base64 encoded in the Cookie header, which is the default setting:

```
GET /pixel.gif HTTP/1.1
Accept: */*
Cookie:
MQogPkctf0l36ccsF6jc5pyed8TxVeGUuJ8hotpp9ZNf2qSOr74qhnzcPwJ6iHkpIqWSWOlgxe
R7l20fZdDO0h60Ut0ojHpXcP2u88MdwbQ/VOP5ET2tCU9ZiqEYu0UOeNz/egd94Syr6NZNdKzn
cyvtUqAY9wPgw8ovs2YUDdo=
User-Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64;
Trident/5.0; MALCJS)
Host: 192.168.200.129
Pragma: no-cache
Connection: close
```

Beacon

Metadata Request

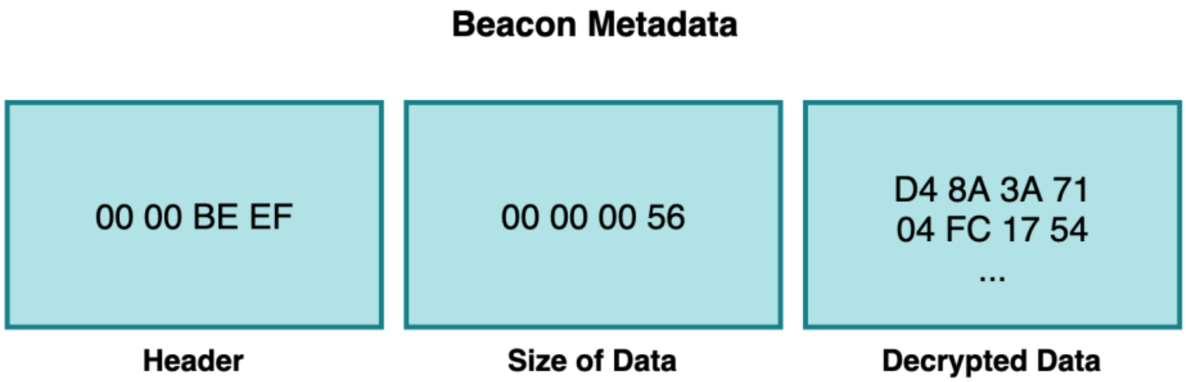
Beacon metadata encryption uses RSA with PKCS1 padding, the following is an example in Python of encrypting beacon metadata using the stager public key:

When decrypted (using the private key we extracted from our test Team Server) the metadata looks like the below:

```
00000000: 0000 BEEF 0000 0048 420C FB76 DDD1 1A26  .......HB..v...&
00000010: 59C5 D420 1C3B 55EB 3334 3830 3009 3639  Y.. .;U.34800.69
00000020: 3234 0936 2E32 0931 3932 2E31 3638 2E32  24.6.2.192.168.2
00000030: 3030 2E31 3031 0944 4553 4B54 4F50 2D33  00.101.DESKTOP-3
00000040: 3732 5251 544D 0961 646D 696E 0931 0930  72RQTM.admin.1.0
```

Decrypted Metadata Blob

All decrypted metadata blobs are prepended with 8 bytes, which must always be present. These 8 bytes are the magic number 48879 (0xBEEF), followed by the data size:



Beacon Metadata Structure

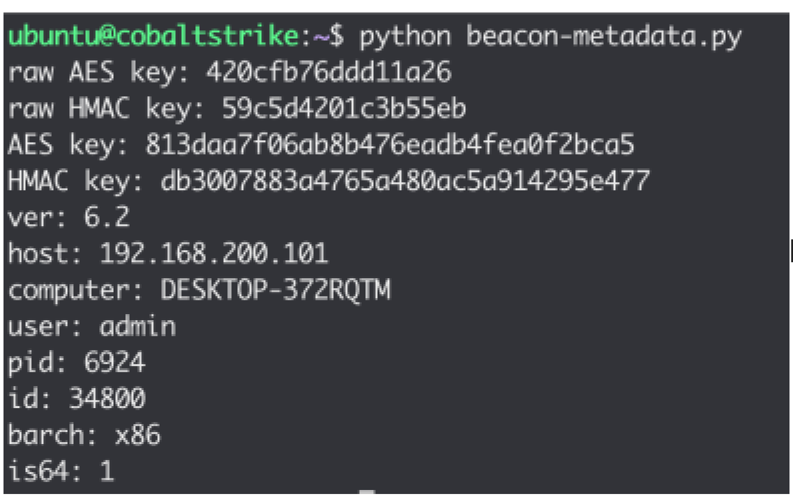
So we can now encrypt / decrypt the metadata. Now onto the parsing..

Beacon Metadata Parsing

The following Python code shows how the metadata from a Cobalt Strike beacon is parsed. On Cobalt Strike < 4.0, the metadata fields (aside from the first 16-bytes) are made up of a tab-delimited string. This results in the IP address being treated as a (non sanity-checked) string, which in version 3.5 leads to the directory traversal issue. However, on later versions the IP address field is validated to ensure it is indeed a valid IP address using a regex.

Note that this changed in Cobalt Strike 4.0, which added a number of new fields. The code below covers both 3.5 and 4.0 versions.

When the parser is run on our decrypted metadata blob, it will result in the following output:



Metadata Parsing

We now have enough information to generate and encrypt our own metadata.

Symmetric Encryption

Cobalt Strike uses AES-256 in CBC mode with HMAC-SHA-256 for task encryption. For the version of Cobalt Strike that the vulnerability existed in, this was included in the trial version, however from version 3.6 this is no longer enabled in non-licensed versions of Cobalt Strike. This means that for some cracked or trial versions of Cobalt Strike used by adversaries, network communications will be sent in cleartext. However, as we are looking at a version prior to 3.6, task encryption is always enabled.

Once the metadata is parsed, the Team Server will do a check to see whether this beacon is a new beacon by checking whether the AES keys specified in the metadata are already registered for the beacon ID value (also parsed from the metadata).

If no AES keys were previously registered for the beacon ID, then it goes ahead and sets the AES key for the beacon session**.** This is achieved by taking the first 16 bytes of the decrypted beacon metadata. The first half (8 bytes) of which are used to derive the AES key, by calculating the SHA256 sum to create a 256 bit key. The same is done with the second half, which is used as the HMAC key. You may have noticed these parsed in the output above. These keys can be used for task encryption and decryption.
The following Python script shows how the AES encryption/decryption works.

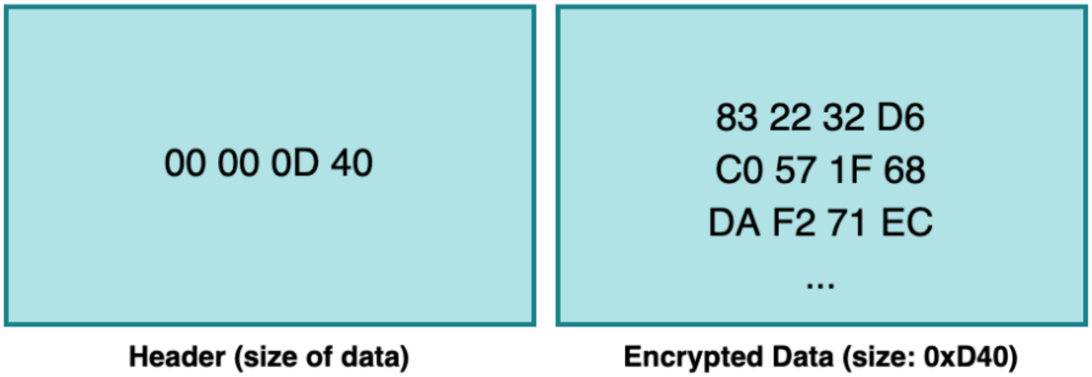
Beacon Tasking

So far we have covered staging, metadata, checkins, asymmetric (RSA) and symmetric (AES) encryption. We can now stage fake beacons and decrypt taskings sent from the Team Server to the beacon. Next we will cover how to decrypt/encrypt beacon output back to the Team Server.

After the beacon has checked in (by including the encrypted metadata we previously covered, within the request), if the Team Server has a task for the beacon it will send this as an encrypted response. As shown earlier, this is decrypted using the negotiated AES session keys.

What does the response to a tasking look like? In short, this response is also encrypted with AES in the same way that a tasking from the server is sent, however the beacon response data is prepended with a length field.

The following screenshot shows an example of *encrypted* data sent by the beacon in response to a “ps” tasking:



Encrypted Callback Response

Once the data is decrypted, we can see that it is prepended with 12 bytes, which indicate various properties of the output.

```
00 00 00 02 <- Counter (has to be higher than the previous one)
00 00 0D 1B <- Size of the data
00 00 00 11 <- Type of callback (in this case it's 17, which is OUTPUT_PS)
5B 53 79 73 <- Data of size 0xD1B
74 65 6D 20
```

The following python code shows how to decrypt and decode beacon output

Running this code decrypts the output and shows the results of the “ps” command:

```
ubuntu@cobaltstrike:~$ xxd req.bin |head -n 2
00000000: 0000 0d40 8322 32d6 c057 1f68 daf2 71ec  ...@."2..W.h..q.
00000010: f548 b826 ffdb c6a6 731c a720 b037 094a  .H.&....s.. .7.J
ubuntu@cobaltstrike:~$ python aes.py req.bin
Encrypted data should be: 3392
Decrypted length: 3355
Output type: 17
Beacon data: [System Process]   0       0
System  0       4
Registry        4       88
smss.exe        4       340
csrss.exe       408     420
wininit.exe     408     492
csrss.exe       488     512
services.exe    492     572
winlogon.exe    488     600
lsass.exe       492     616
svchost.exe     572     752
fontdrvhost.exe 492     764
fontdrvhost.exe 600     772
svchost.exe     572     864
dwm.exe 600     948
svchost.exe     572     364
svchost.exe     572     376
svchost.exe     572     380
svchost.exe     572     712
svchost.exe     572     1144
svchost.exe     572     1240
```

Decrypting Beacon Output

So at this point we can extract the keys we need, encrypt and decrypt communications so on to the vulnerability and exploitation.

The Vulnerability

The vulnerability itself was a directory traversal vulnerability (as the advisory states) in the reported internal IP address of the beacon which was used to build a file path.

When processing “download” responses, the Team Server would write these to the file-system by re-creating the target system path on the Team Server filesystem, under the “downloads” folder within the working directory. The following screenshot shows an example of what this normally looks like. As shown, the downloaded file is stored within a folder named after the IP address of the beacon. Within this folder is the re-created filesystem structure of the downloaded file.

```
ubuntu@cobaltstrike:/opt/cobaltstrike-3.5$ ls -laR downloads/
downloads/:
total 12
drwxr-xr-x 3 root   root   4096 Jun 15 15:23 .
drwx------ 6 ubuntu ubuntu 4096 Jun 15 13:39 ..
drwxr-xr-x 3 root   root   4096 Jun 15 15:23 192.168.200.101

downloads/192.168.200.101:
total 12
drwxr-xr-x 3 root root 4096 Jun 15 15:23 .
drwxr-xr-x 3 root root 4096 Jun 15 15:23 ..
drwxr-xr-x 3 root root 4096 Jun 15 15:23 C:

'downloads/192.168.200.101/C:':
total 12
drwxr-xr-x 3 root root 4096 Jun 15 15:23 .
drwxr-xr-x 3 root root 4096 Jun 15 15:23 ..
drwxr-xr-x 2 root root 4096 Jun 15 15:23 temp

'downloads/192.168.200.101/C:/temp':
total 12
drwxr-xr-x 2 root root 4096 Jun 15 15:23 .
drwxr-xr-x 3 root root 4096 Jun 15 15:23 ..
-rw-r--r-- 1 root root    8 Jun 15 15:23 test.txt
```

CS 3.5

Downloads Folder

Although traversal checks were carried out on the filename itself, the IP address field was not checked, lading to a directory traversal vulnerability in the IP address field, which as we demonstrated earlier, is set in the Beacon Metadata and controlled by the attacker.

So instead of reporting the beacons IP address as of 10.133.37.10 we report it as our target folder, e.g. ../../../../etc/.

Note: The vulnerable code uses the IP address value to build file paths, in various other places, including writing log files. Although log file poisoning is definitely an exploitable angle, we chose to use the same method as the in-the-wild exploit – download callbacks.

Exploitation

Having a file system write primitive against typically against a Linux based server gives us various options for exploitation. We replicated the same technique employed by the in-the-wild exploitation, that is:

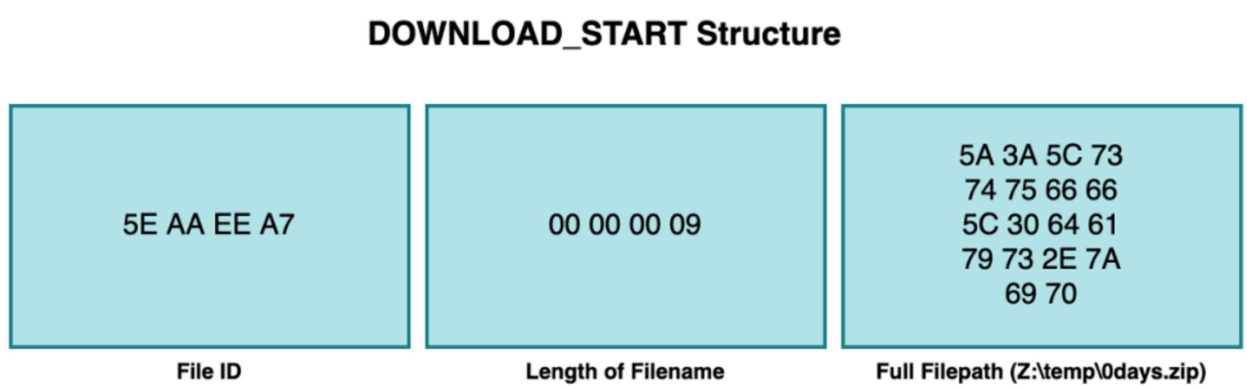
- Check in with a beacon with an internal IP address of ../../../../../[TARGET_FOLDER]/
- Then do a DOWNLOAD_START* callback which causes the file to get created
- Then do a DOWNLOAD_WRITE* callback which causes the contents to be written

*Probably not the official term, but we will use these terms to refer to the task response types here. Whereby, a DOWNLOAD_START is the initial response from a "download" tasking (this causes the file to be *created* on the file-system), and DOWNLOAD_WRITE, is a response containing data to be *written* for the download task.

Before we can do this however, we need to understand the structure of both the DOWNLOAD_START and DOWNLOAD_WRITE callbacks. As previously explained, we know that these are AES encrypted, prepended with an encrypted length, and also a counter and length once decrypted. But what is the structure of the decrypted data? This is explained below.

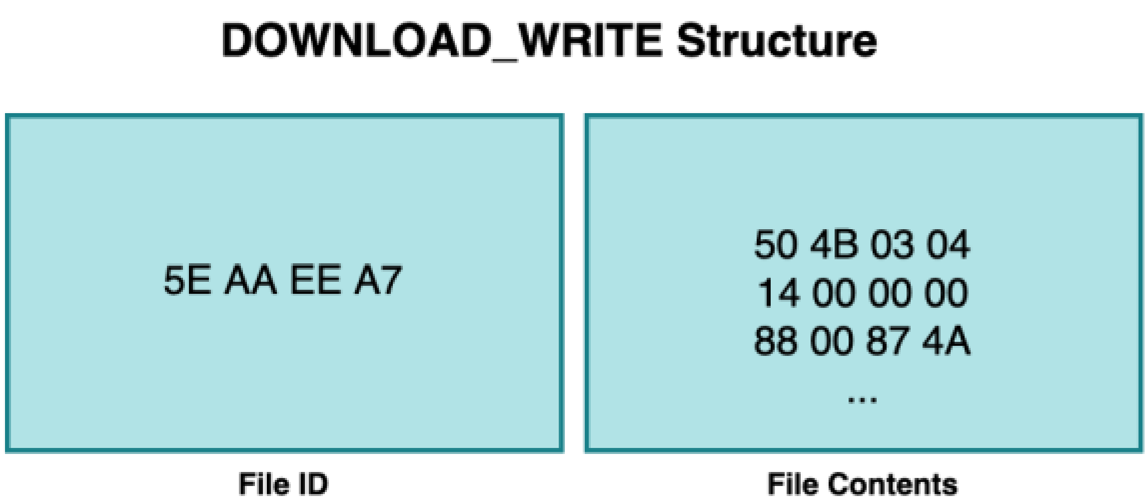
The DOWNLOAD_START callback structure.

This callback type for the task is 2. The (decrypted) callback structure is as follows:

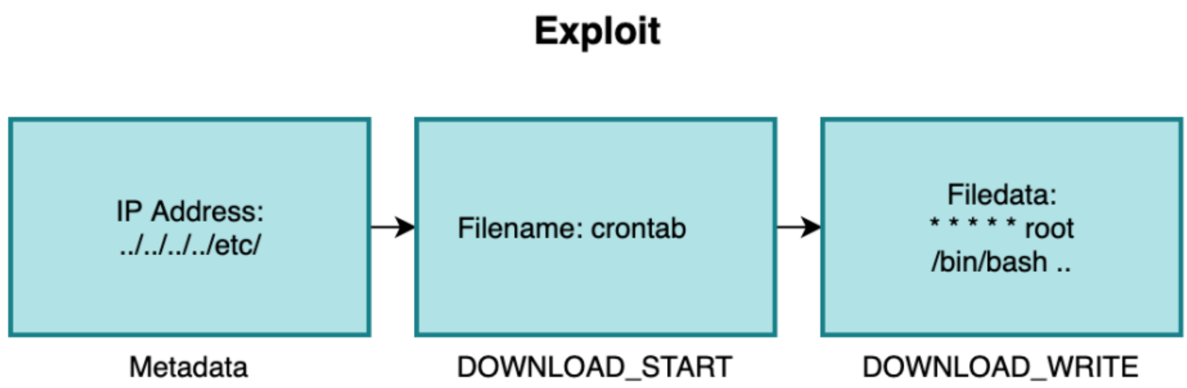


The DOWNLOAD_WRITE callback structure

This callback type for the task is 8. The (decrypted) callback structure is as follows:



To actually achieve code execution we write a cronjob as the in-the-wild attacks did. Typically this would involve sending the following values within the Metadata blob and task callback(s):



Assuming we have written our functions to build the metadata blob (with the IP address traversal string), and our chosen AES keys. We can stage a fake beacon and check in the DOWNLOAD_START and DOWNLOAD_WRITE callbacks with our crafted values. The following example code demonstrates what this would look like:

The following video shows the exploit in action:

The Fix(es)

As described in the follow-up post by Cobalt Strike, the following fixes were added in 3.5.1

- A new SafeFile method was introduced, which takes the path that the file should be written to as the first argument, along with the filename to write as the second. It subsequently ensures that, after canonicalisation, the file does not break out of the canonicalised path passed in the first argument. This new method is used everywhere a file write is carried out, including for writing log files and screenshots.
- The host_stage malleable C2 configuration setting was added. When set to false, this completely disables payload staging, meaning that your Team Server will not host a stager via the checksum8 URL. This should be used whenever you do not require payload staging, however you should note that this may break some post-exploitation workflows that you may be used to working with.
- Downloads are now stored using an ID value on the filesystem. This is mapped to the real file-path in the data-model, which is what you see when you access the downloads tab via the Cobalt Strike GUI.
- The Team Server now checks that the beacon has been tasked at least once before allowing most callback responses from the beacon. This ensures that an attacker can't stage a fake beacon and start spoofing responses without the operator first interacting with the beacon.
- IP address values reported in the Beacon Metadata are sanity checked against a regex to ensure that are actually an IP address.

In summary, the fixes applied in the 3.5.1 update are robust and address the vulnerability from multiple angles. As stated at the top of the post, this vulnerability existed in a *legacy version* of Cobalt Strike and the vulnerability does not exist in the latest versions. Nevertheless, we hope that this post provided some insight into Cobalt Strike internals, and provides opportunities for both Blue and Red teams to improve in their fight against real adversaries.