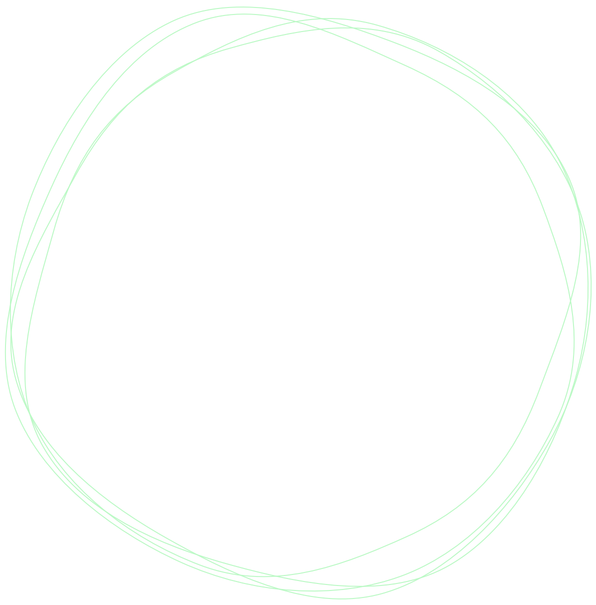
Stairwell

Threat research blog • January 24, 2022

Hunting with weak signals



How to find malware with mutated strings and YARA rules

By Steve Miller, Threat Researcher

Overview

A core piece of threat detection is managing for inevitable subterfuge. Attackers must duck and dodge and evade defenses, so defenders must strive to increase their aperture and improve the perception of detection technologies. Yet it can be tough to increase our visibility and discover stealthy attacks and malware without also bringing extraneous noise into our systems. One approach to improve the signal-to-noise ratio of expanded detection systems is to apply deliberate hunting methodologies that can turn “weak signals” into meaningful data points for defensive purposes.

This blog expands an approach to detection that many researchers refer to as “weak signals,” where instead of just looking for specific malicious things, we also look for threats based on anomalous, odd, rare, or otherwise interesting features. We illustrate the concept of weak signals by looking at mutations of common file strings, and we codify these signals into YARA rules that defenders can use to discover and label or contextualize interesting files that may be malware.

Contents:

Weak signals: A different approach to detection and discovery

It should go without saying that when we defenders do threat detection, we are generally trying to record evidence of malicious activity. We often jam a detection rule into a system and when we match a file or a packet or a process, we can seize the associated data for investigation and analysis. In an incident, we wish to capture as much volatile evidence as possible before it disappears into the ether, before logs roll past the retention window, or before the attacker cleans up their tracks. But does it always have to be that way?

Let's pretend for a moment that files (containers of associated data) represent all (or at least the majority) of the malware on computers. Imagine that instead of capturing malicious activity, you already *have* all the files from every computer from your organization collected, stored, and indexed out of band. You *have* every file that was ever created on disk and mapped to memory, even if it was later deleted. This would effectively mean that you captured any and all malicious activity, right?

When you can record and keep all the data, new doors open for your detection strategy, because instead of trying to capture particular malicious activity, you are trying to merely distinguish it from the rest of the data. You can move beyond the task of finding the known-bad, and into a new realm of detection: finding the unknown bad things by bubbling up and discovering interesting files that are *more likely to be* malicious from a massive data set. And not just with what you know at this moment but what you may learn in the future. You can find things that would otherwise go unnoticed, things that were "missed" by everything else, so long as you can take the wheat field and divide it into haystacks.

At Stairwell, we like to use YARA (amongst other things) to unleash our analytical ideas and build those haystacks. Taking threat actor techniques we observe across intrusions and malware, we can search for files with features or equities that are not necessarily evil, but rare or unique enough to prove fruitful. We scribe conditional logic into YARA rules that when matched serve as "weak signals" that tell us a file merits extra scrutiny.

Conceptually, weak signals represent outcomes of adversary tradecraft, deliberate decisions, and techniques used by malware developers or intrusion operators. Some of these weak signals may be caused by accidents or oversights in the development process, resulting in unique file features, toolmarks, and other trace evidence. In many cases, the attackers or original malware developers do not know that a given feature is rare or "threat dense," and obviously, most features of malware are not meant to be noticed.

Now let's explore some examples of weak signals, where we look for uncommon mutations of common strings using YARA rules.

Reasonable expectations and string mutations

Most files have plenty of features that will never tip the scales on a detection verdict. However, when certain features show up in an encoded form, they become immediately more threat dense, and much more likely to be malicious. For example, seeing the plaintext string “https://www” in a file is completely typical, but when that same string is XOR encoded, you should begin to suspect that there is evil afoot.

If we expect that malware developers make malware functionally cooperative with host operating systems and network gear, we may anticipate that malware contains a litany of features associated with common APIs and RFC web standards. If we then expect that malware developers must attempt to evade detection, we can then anticipate that many toolmarks will be hidden through common obfuscations such as Base64 or XOR encoding schemes.

Marrying up these two expectations, we arrive at some ideas on encoded features that we can search for using YARA and the built-in string operators “base64” and “xor”.

To assemble a simple YARA rule using this technique, we may take a feature string such as “kernel32.dll” and search for all PEs that have that string base64 encoded. Kernel32.dll is one of the most common access points for the Windows API, which is why that DLL name shows up as a string in almost all PE files. When we look for that string in encoded form in a YARA rule, matches on this rule tell us that a file may have a base64 encoded PE subfile, which is a weak signal for malware.

Sample Rule 1 – YARA rule looking for PEs with base64 encoded kernel32.dll string

—

```
 rule Methodology_Mutation_Base64_kerneldll_PE {

   meta:

       author = "stvemillertime"

       description = "Searching for PE files with mutations of odd,
rare, or interesting string equities."

   strings:

         $s = "kernel32.dll" base64
```

```
  condition:

          uint16be(0) == 0x4d5a and $s

}
```

—

Similarly, we would expect malware developers to obfuscate more telling toolmarks such as implant configuration details or the names of important functions. One may try creating a variety of rules searching for common PE strings or revealing toolmarks in different encoding formats using the built-in YARA options such as `base64` `base64wide` and `xor(0x01–0xff)`.

There's nothing inherently malicious about Base64 or XOR encoding, so these rules cannot tell you if a file is malicious or not. But they can serve as a weak signal to decorate a file or provide a lead to a file, or serve as some other signifier to automated systems or human analysts. While not exclusive to malware, encoding in malware is categorically an "adversary method" or a piece of "adversary tradecraft" done by a developer. Our codification of these things go into rules we casually refer to as "TTP" or "methodology" rules.

What we cannot do quickly with built-in YARA string operators, we may be able to perform by other means with some scripting and some guesswork. Let us explore a couple of other commonly employed obfuscation methods that result in mutated strings, and we will walk through some approaches to aid in YARA rule creation.

Uncommon mutation: NOBELIUM (APT29/CozyBear) flip-flop strings

In May 2021, Volexity detailed NOBELIUM's loader for Cobalt Strike which they called FLIPFLOP. This malware family used a simple scheme that flip flopped every two bytes of a loaded file. Statically, the embedded file has a file string that appears as "iMrcsofo taBesC yrtpgoarhpciP orived r1v0." and when the file is loaded, it is read in with every two bytes flipped and we arrive at "Microsoft Base Cryptographic Provider v1.0".

Image 1 – NOBELIUM FLIPFLOP sample in hex view showing the flip-flopped strings.

Now we will make some educated guesses based on our experience studying malware developers. First, we think (and maybe hope) that this actor uses this technique again to encode whole subfiles. Second, we think (and hope) that this actor is not alone in using this technique. Third, we think this is going to be relatively rare and we will probably find more interesting files, but we won't really know until we get to measuring it. So let's get to it.

If we're looking for obfuscation of common strings with this method, we need to take plaintext strings and generate the mutations. This is an uncommon string obfuscation method and there's no YARA option for this, but we can write one ourselves in Python.

Sample Script 1 – Python function to flip flop strings

—

```
def make_flipflop_strings(thing):

       if isinstance(thing,str):

           str_len = len(thing)

           s = thing
           t = ""

           thing_flip = t.join([ s[x:x+2][::-1] for x in range(0,
len(s), 2) ])

           return(thing_flip)
```

—

This simple function takes an input string and performs the flip-flopping of every two bytes and will spit out a string that we would expect to see in the FLIPFLOP malware. Inputting the string "kernel32.dll" results in "eknrle23d.ll". Okay, cool, now we can do this in a larger Python script.

We compiled a short list of common Windows API DLL names used as an input text file, then we can dump out YARA-friendly string terms that have been flip-flopped and formatted for quick pasting into a rule.

—

```
script.py -f common_wins_dlls.txt -m flipflop

Here are the mutations:

        $kernel32dll_flipflop = "eknrle23d.ll" nocase

        $ws2_32dll_flipflop = "sw_223d.ll" nocase

        $msvcrtdll_flipflop = "smcvtrd.ll" nocase

        $kernelbasedll_flipflop = "eknrleabesd.ll" nocase

        $advapi32dll_flipflop = "daavip23d.ll" nocase

        $advapires32dll_flipflop = "daaviper3s.2ldl" nocase

        $gdi32dll_flipflop = "dg3i.2ldl" nocase
        $gdiplusdll_flipflop = "dgpiul.sldl" nocase

        $win32ksys_flipflop = "iw3nk2s.sy" nocase

        $user32dll_flipflop = "sure23d.ll" nocase

        $comctl32dll_flipflop = "occmlt23d.ll" nocase

        $commdlgdll_flipflop = "ocmmld.gldl" nocase

        $comdlg32dll_flipflop = "ocdmgl23d.ll" nocase

        $commctrldll_flipflop = "ocmmtclrd.ll" nocase

        $shelldll_flipflop = "hsle.lldl" nocase

…<truncated>
```

—

We created a few common string lists and fabricated a handful of rules describing this methodology and string mutation type. Because these are strings, we can use YARA's built-in "nocase" modifier; this way we do not have to worry about the exact casing of the input.

Image 2 – Truncated view of a YARA rule for flip-flopped function names

Now that we've developed a handful of rules, when we view this sample of FLIPFLOP in our tooling, the YARA matches serve as labels for this file's unique features, and signal that this file is worth investigating further.

Image 3 – Methodology rules decorating a sample of FLIPFLOP

Expanding beyond just this one file, how prevalent is this obfuscation methodology? After deploying a handful of these rules to a corpus, we can see at least a couple thousand files, and at first glance, almost all of it is malware of some kind. Can you believe you could've red-flagged the NOBELIUM CobaltStrike loader with a simple flip-flopped Windows DLL name? We can find malware with odd representations of fundamentally non-malicious things. That's kinda neat, right?

Uncommon mutation: Meterpreter style stack strings

When malware authors want to take obfuscation to the next level, they can hide plaintext strings by decoding them at runtime or pushing them to the stack or memory in chunks of bytes. Generically, we refer to these as "stack strings." It can be difficult for analysts to recover stack strings without debugging or disassembly or specialized tooling (such as the incredible FLOSS). But we also know that "stack strings" come in many forms and some may be easier to detect in static bytes.

When Meterpreter shellcode loads wininet, it pushes the string "wininet" to the stack in four byte chunks in reverse byte order, then passes that to the Ruby hash of kernel32 and LoadLibraryA. We can see exactly how this happens in the Meterpreter source code:

—

```
; Input: EBP must be the address of 'api_call'.

; Output: EDI will be the socket for the connection to the server

; Clobbers: EAX, ESI, EDI, ESP will also be modified (-0x1A0)

load_wininet:

  push 0x0074656e       ; Push the bytes 'wininet',0 onto the stack.

  push 0x696e6977       ; ...

  push esp              ; Push a pointer to the "wininet" string on
the stack.

  push 0x0726774C       ; hash( "kernel32.dll", "LoadLibraryA" )

  call ebp              ; LoadLibraryA( "wininet" )
```

—

The funny thing about this is that the x86 opcode for the push instruction is 0x68, which shows up in plaintext form as the ASCII letter "h", evoking a word soup type of look when you read it amidst the strings. In a sample of Meterpreter shellcode we see the static bytes `68 6E 65 74 00 68 77 69 6E 69` which may look in human-readable form something like "hnet.hwini" where the . is actually a null. So while an interesting way to mask the real string (and load the DLL), this presents us with some analytical surface area for us to grab ahold of.
Just as with the flip flop technique, we assume that this method will be used in other malware families to push strings to the stack, so we can use the similar approach to creating mutated strings. First, we create a python function to generate the mutation.

Sample Script 2 – Python function to convert a string to the four-byte stack push format

—

```
def make_stackpush_string(thing):

       if isinstance(thing,str):

           n = 4

           out = [(thing[i:i+n]) for i in range(0, len(thing), n)]

           out.reverse()

           thing_stackpush=str('h'+'h'.join(map(str,out)))

           return(thing_stackpush)
```

—

Then, using common Windows DLL names we will use this function to generate a list of YARA-friendly string terms.

—

```
python script.py -f common_win_dlls.txt -m
stackpush

Here are the mutations:

        $kernel32dll_stackpush = "h.dllhel32hkern" nocase

        $ws2_32dll_stackpush = "hllh32.dhws2_" nocase

        $msvcrtdll_stackpush = "hllhrt.dhmsvc" nocase

        $kernelbasedll_stackpush = "hllhse.dhelbahkern" nocase

        $advapi32dll_stackpush = "h.dllhpi32hadva" nocase

        $advapires32dll_stackpush = "hdllhs32.hpirehadva" nocase
        $gdi32dll_stackpush = "hlh2.dlhgdi3" nocase

        $gdiplusdll_stackpush = "hdllhlus.hgdip" nocase

        $win32ksys_stackpush = "hysh2k.shwin3" nocase

        $user32dll_stackpush = "hllh32.dhuser" nocase

        $comctl32dll_stackpush = "h.dllhtl32hcomc" nocase

        $commdlgdll_stackpush = "hdllhdlg.hcomm" nocase

        $comdlg32dll_stackpush = "h.dllhlg32hcomd" nocase

        $commctrldll_stackpush = "h.dllhctrlhcomm" nocase

<truncated>
```

—

You'll notice here that this is not *quite* what Meterpreter was doing, but that's beside the point. Go do a quick search for h.dllhel32hkern and you'll see what we mean. But if we want to create mutations closer to what Meterpreter was doing, we could strip off the .dll from our input list, and output YARA terms that are null terminated.

—

```
python cerebro.py -f common_win_dlls.txt -m stackpushnull

Here are the mutations:

        $kernel32_stackpushnull = {68656c333200686b65726e}

        $ws2_32_stackpushnull = {68333200687773325f}

        $msvcrt_stackpushnull = {68727400686d737663}

        $kernelbase_stackpushnull = {6873650068656c6261686b65726e}
        $advapi32_stackpushnull = {687069333200686164 7661}

        $advapires32_stackpushnull = {687333320068706972656861647661}

        $gdi32_stackpushnull = {6832006867646933}

        $gdiplus_stackpushnull = {686c7573006867646970}

        $win32ksys_stackpushnull = {6879730068326b2e736877696e33}

        $user32_stackpushnull = {683332006875736572}

        $comctl32_stackpushnull = {68746c33320068636f6d63}

        $commdlg_stackpushnull = {68646c670068636f6d6d}

        $comdlg32_stackpushnull = {686c6733320068636f6d64}

        $commctrl_stackpushnull = {686374726c0068636f6d6d}

<truncated>
```

—

These generic techniques to push bytes to the stack may happen for different reasons and with slightly different implementations, so you'll need to hedge your bets. Try different input lists and strings, try common API DLL names, or maybe try functions from specific areas of the API such as memory management. Maybe try the middle 10 characters of longer functions in case the start or end are trimmed off. Maybe the string is null terminated, and maybe it's double nulled! Should we try triple, too?

—

```
$a = {68 6C 6C 00 00 68 33 32 2E 64 68 77 73 32 5F} //ws2_32.dll
double null term
```

—

You could experiment with broader, more generic implementations of this stack push technique using regular expression matches on the byte pattern, but this has its own set of pros and cons for application to files in the wild as regexes may be memory intensive and prone to incidental matches on large byte streams.

—

```
$stackpush = /\x68[a-zA-Z0-9\._\-]{1,4}\x00\x00\x68[a-zA-Z0-9\._\-]
{4}/
```

—

This is a terrible regex to illustrate a point…if you're not careful, structure matching will get messy and laborious to validate, which is one of the reasons we love to generate mutations of common Windows API and function strings. Long, unique strings are not only easier to validate but are also less likely than regexes to accidentally match on random data.

Looking back at this obfuscation technique and others, some of us recall seeing the toolmark `h.dllhel32hkern` years ago and we recognized that this was analytically interesting long before we knew what was happening behind the scenes (which more or less happened for us this week, while writing this blog). Sometimes you don't need to fully understand features of malware for those features to be "useful" at scale. Once you notice something curious, smash what you see into a YARA rule and start to measure where and when it is happening, and maybe over time, you will come to understand why and how.

Final thoughts: Operating on the broad end of the detection spectrum

Detection is about much more than simple signatures or atomic indicators. If we *have* all data from all computers in organizational purview, we can begin to search it creatively and we can then explore a strategy for surfacing unknown threats using weak signals.

Weak signals are not necessarily about looking for *malicious* things, which is why concepts like "True Positive" or "False Positive" do not always apply. Instead, you can think of weak signals as decorative, contextual or informational labels for features we define as analytically important. The features we think are important are those that typically signify adversary tradecraft, especially that which transcends malware families, threat actors and intrusion operations.

When codified into rule logic, weak signals build rulematch haystacks of different sizes, for different purposes, and different consumers. Some of those haystacks may be small and rare enough to be fed into workflows as “alerts,” whereas others may serve as bigger stacks for investigation and hunting, and still others may serve as large scale data sets for stacking and processing in conjunction with other features such as local and global prevalence.

Weak signals may be expressed, stored, shared and applied to data in a variety of logic formats and technologies such as YARA, Sigma, Snort, Suricata, KQL, Zeek, ClamAV, OpenIOC and more. No matter what technology you are using, a big part of the art of detection is using your imagination to generate possibilities of what a malware operator or developer may decide to do and how that decision may show up in a file, network stream, process, or other data set. Then you must channel your creativity into enumerating edge cases and inventing alternative approaches across a spectrum of precision so that you will have some rules that work where others fail to be effective.

In this blog, we covered just a few small examples of weak signals, looking at uncommon mutations of common strings. There are probably more clever ways to codify these adversary methodologies beyond brute-forcing expected strings into mutations for YARA rules, but we believe this is a fast and easy approach to help defenders come up with rules for testing and tweaking and distinguishing curious file features.

To put some of these ideas to the test, check out the sample scripts, list and rules below. If you’re feeling especially hungry for more, keep an eye on our blog to get blasted with my latest YARA rants.

Appendix: Sample script, string lists and YARA rules

We’ve provided a simple script, sample string lists and a handful of sample rules to help illustrate our approach and to serve as starting points for other detection ideas. Remember that this is about building a variety of “weak signals” or “methodology rules” and not production “signatures” per se. They will build haystacks of different sizes and you may need to tune the rules to suit your appetite. Once you create a YARA rule, you will need to refine it, tune it, open it up and close it down, and iterate again and again until you arrive at a rule (or more likely a set of rules) that suits your various purposes and consumers. When implemented carefully, these rules can apply labels on data that allow analysts to find the needles in the haystacks, or separate the wheat from the chaff. Or something like that.

The sample script “Cerebro” takes a list of strings and transforms them into selected mutations, and outputs the mutated values in YARA-friendly format. We took subsets of the Windows API and used this to generate text lists of common DLLs and functions. You can use these lists, along with your own uncommon obfuscation algorithms, as starting points to generate string mutations for YARA rules.

> Project: https://github.com/stairwell-inc/threat-research/tree/main/cerebro-string-mutations
>
> Script: https://github.com/stairwell-inc/threat-research/blob/main/cerebro-string-mutations/cerebro-file-basic.py
>
> Sample String List: https://github.com/stairwell-inc/threat-research/blob/main/cerebro-string-mutations/common_windows_dlls.txt
>
> Sample YARA Rules: https://github.com/stairwell-inc/threat-research/blob/main/cerebro-string-mutations/Methodology_Mutation_FLIPFLOP_Sample_Rules.yar

Example API string list – common_windows_dlls.txt

```
kernel32.dll

ws2_32.dll

msvcrt.dll

kernelbase.dll

advapi32.dll

advapires32.dll
```

```
gdi32.dll

gdiplus.dll

win32k.sys

user32.dll

comctl32.dll

commdlg.dll

comdlg32.dll

commctrl.dll

shell.dll

shell32.dll

shlwapi.dll

netapi32.dll

shdocvw.dll

mshtml.dll

urlmon.dll

iphlpapi.dll

httpapi.dll

msvbvm60.dll

shfolder.dll
```

```
ole32.dll

wininet.dll

wsock32.dll
```

Example YARA Rule – Methodology_Mutation_Stackpush_Windows_DLLs

```
rule Methodology_Mutation_Stackpush_Windows_DLLs {

 meta:

   author = "Stairwell & stvemillertime"

   description = "Searching for PE files with mutations of odd, rare, or interesting string equities. Here we look for strings from common Windows strings, DLLs and functions in Meterpreter style stack strings form, where the string pushed onto the stack 4 bytes at a time using PUSH 0x68, appearing in reverse four byte chunk order, where the PUSH which shows up as an ASCII letter h."

 strings:

        $kernel32dll_stackpush = "h.dllhel32hkern" nocase

        $ws2_32dll_stackpush = "hllh32.dhws2_" nocase

        $msvcrtdll_stackpush = "hllhrt.dhmsvc" nocase

        $kernelbasedll_stackpush = "hllhse.dhelbahkern" nocase

        $advapi32dll_stackpush = "h.dllhpi32hadva" nocase

        $advapires32dll_stackpush = "hdllhs32.hpirehadva" nocase

        $gdi32dll_stackpush = "hlh2.dlhgdi3" nocase
```

```
$gdiplusdll_stackpush = "hdllhlus.hgdip" nocase

$win32ksys_stackpush = "hysh2k.shwin3" nocase

$user32dll_stackpush = "hllh32.dhuser" nocase

$comctl32dll_stackpush = "h.dllhtl32hcomc" nocase

$commdlgdll_stackpush = "hdllhdlg.hcomm" nocase

$comdlg32dll_stackpush = "h.dllhlg32hcomd" nocase

$commctrldll_stackpush = "h.dllhctrlhcomm" nocase

$shelldll_stackpush = "hlhl.dlhshel" nocase

$shell32dll_stackpush = "hdllhl32.hshel" nocase

$shlwapidll_stackpush = "hdllhapi.hshlw" nocase

$netapi32dll_stackpush = "h.dllhpi32hneta" nocase

$shdocvwdll_stackpush = "hdllhcvw.hshdo" nocase

$mshtmldll_stackpush = "hllhml.dhmsht" nocase

$urlmondll_stackpush = "hllhon.dhurlm" nocase

$iphlpapidll_stackpush = "h.dllhpapihiphl" nocase

$httpapidll_stackpush = "hdllhapi.hhttp" nocase

$msvbvm60dll_stackpush = "h.dllhvm60hmsvb" nocase

$shfolderdll_stackpush = "h.dllhlderhshfo" nocase

$ole32dll_stackpush = "hlh2.dlhole3" nocase
```

```
        $wininetdll_stackpush = "hdllhnet.hwini" nocase

        $wsock32dll_stackpush = "hdllhk32.hwsoc" nocase

 condition:

    uint16be(0) == 0x4d5a and 1 of them}
```

—

Steve is a Lvl 3 Human Fighter focused on adversary tradecraft, the TTPs, or modus operandi of threat actors. His alignment is chaotic neutral and he loves malware, YARA rules, and making dystopian soundscapes in his synth cave.

Privacy Policy / Terms