[TrendMicro CTF 2020 Finals] Wildcard-2: Yara exfiltration

ubcctf.github.io/2020/12/tmctf-2020-finals-wildcard2/

22 Dec 2020 by - Arctic Wyvern

Maple Bacon qualified and participated in the TrendMicro CTF finals and placed a solid 5th place. One of the challenges used a popular malware analysis/antivirus tool called `yara` which was quite fun, so here's a writeup.

Note: I almost never touch these sorts of binary problems, but I have some experience with yara so I took a stab at it. Bear with me when I explain obvious things :)

A quick primer on yara

What is yara? Yara is a tool developed by VirusTotal, a malware detection service which combines some custom analysis tools with many many third party antivirus tools.

Yara is a static analysis tool that works sorta like regex, but with the goal of identifying malware in mind. So like regex, yara lets you write rules and will tell you if a file matches the given rule.

Here's an example rule, meant for detecting the `silent_banker` malware family

```
rule silent_banker : banker
{
    meta:
        description = "This is just an example"
        threat_level = 3
        in_the_wild = true

    strings:
        $a = {6A 40 68 00 30 00 00 6A 14 8D 91}
        $b = {8D 4D B0 2B C1 83 C0 27 99 6A 4E 59 F7 F9}
        $c = "UVODFRYSIHLNWPEJXQZAKCBGMT"

    condition:
        $a or $b or $c
}
```

The challenge

The challenge is no longer accessible, so some of the details may be a bit wrong, but here are the important details

- The challenge presents an api where we can submit a rule file. The server then generates a new binary, runs the given rule, and returns the result (hit/no hit)
- There are limitations on the rule files we can send:
    - There can only be one rule in the file
    - It must be a yara text file, not a compiled one
    - The file must be less than 5kb

The binary generation works by filling in the following template with garbage and the gold_dust variable

```
#include <stdio.h>
#include <stdlib.h>

// GARBAGE, Encoded FLAG and KEY
char gold_dust[] = {}; // <-- gets filled before compilation

// <Garbage functions 1>

char *decode(char *enc, int enc_size, char *key, int key_size)
{
    char *dec;

    dec = (char *)malloc(enc_size + 1);
    if (!dec)
    {
        return NULL;
    }

    for (int i = 0; i < enc_size; i++)
    {
        dec[i] = enc[i] ^ key[i % key_size];
    }

    dec[enc_size] = 0;

    return dec;
}

// <Garbage functions 2>

int main(int argc, char **argv)
{
    char *flag;

    // <Garbage code>

    flag = decode(gold_dust + <flag offset>, <flag len>, gold_dust + <key offset>,
<key len>);
    if (!flag)
    {
        return -1;
    }

    printf("FLAG is TMCTF{ %s }\n", flag);
    free(flag);

    return 0;
}

// <Garbage functions 3>
```

where garbage functions are generated with

```
def generate_garbage_function(label, count):
    template = "void garbage_func_%s_%d() { return; }"
    return '\n'.join([template % (label, n) for n in range(count)])
```

and garbage code is generated with

```
def generate_garbage_code(min_size, max_size):
    count = generate_random_num(min_size, max_size)
    return '__asm__("{}");'.format('nop;'*count)
```

The most important part for us is the `gold_dust` buffer, where the flag will be encoded. It's generated by:

1. Generating a random key, with random length
2. Encrypting the flag with repeating XOR
3. Creating a random buffer with a length between 256 and 512 kb
4. Putting encrypted flag and key at random offsets in the buffer

```
def encode_flag(flag, key):
    return bytes([flag[i] ^ key[i%len(key)] for i in range(len(flag))])

def generate_gold_dust(garbage, flag, flag_offset, key, key_offset):
    flag_size = len(flag)
    key_size = len(key)
    gold_dust = bytearray(garbage)
    gold_dust[flag_offset: flag_offset+flag_size] = flag
    gold_dust[key_offset: key_offset+key_size] = key
    return bytes(gold_dust)
```

Now that we have all that set up, you might be wondering ok what else? We can submit yara rules to random binaries, there has to be something else right?

No, there's nothing else. We need to exfiltrate the flag just from what's been mentioned.

- A flag that is only ever encrypted and cryptographically unrecognizable from the buffer that surrounds it and the key.
- Exfiltrating information one bit at a time.
- There's no information kept between two rule runs, with completely new randomly generated binaries for each run.

At this point it was 2am, so I went to bed thinking this challenge had to be impossible.

Here's one of the generated binaries (though I messed with the flag encoding a bit so don't try to actually run the solution on it) tortoise.c

The solution

My first instinct was to try and figure out a way of guessing the flag and then generating a rule that pattern matches against the gold_dust, but the massive size, randomized data, and the fact that the flag is encrypted made that approach completely impossible.

Here's a better way of exfiltrating the flag, one byte at a time

1. Figure out how to read the ith byte of the plaintext flag
2. Generate 255 rules where it only matches if the ith byte matches chr(b)
3. Submit them all, and see which one hits

So steps 2 and 3 are easy, step 1 is the hard part.

How can we read the unencrypted flag? To read the first byte of the flag we need the following

- The address of the flag in the gold_dust
- The address of the key in the gold_dust

How can we get these two addresses?

In the main function we see that the generated binaries have a call to `decode`, which has the two arguments that we want. How can we find where the call to decode is?

Well we have a very helpful set of "garbage code" right before the call to decode, so we know that the assembly in that area must be

```
90 - nop
90 - nop
90 - nop
... <hundreds of nops>
90 - nop
90 - nop
C7 44 24 0C <key len>    - loading the key_size arg
C7 44 24 08 <key_addr>   - loading the key arg
C7 44 24 04 <flag len>   - loading the flag_size arg
C7 44 24 <flag_addr>     - loading the flag arg
E8 <relative addr>       - call to decode
```

So we can make a yara string `$nops = {90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 C7 44 24 0c}`, which will match this pattern

With the yara `@` and `[1]` operators we can get the address where the rule matched for the first time. Since this pattern is so distinct we can basically guarantee that this rule will only hit at the desired location. There are 16 `90`'s in the rule, so `@nops[1] + 16` will be the start of the argument loading. Using the `uint32` yara function we can read a 4 byte address from the binary at a given address. So to read the addresses we can do

```
uint32(@nops[1] + 16 + 8 + 8 + 8 + 3)
```

for the flag address and

```
uint32(@nops[1] + 16 + 8 + 4)
```

for the key address

Now you would think we could then call `uint8(uint32(@nops[1] + 16 + 8 + 8 + 3))` to read the first encrypted byte, but that doesn't work because the address in the argument is a virtual address. It's the address where `gold_dust` variable in `.data` will be loaded, not the raw address where the actual data is in the binary.

The binary will load `.data` to the executable's `image_base` + the relative virtual address for `.data`, so what we can do is take our virtual address for `flag` and `key`, subtract the image base and subtract the RVA, leaving just the `flag_offset` and `key_offset`. Then we can just add on the raw offset to `.data` in the binary and we get an address which we can read flag/key data from. This can be easily done with the "pe" module from module, which pulls out the `image_base`, RVA of `.data`, and the raw data offset.

So with that we can read a single byte at those two locations, xor them, and then generate rules that check if that result is a given byte. Then all we need to do is generate all those rules for each index of the flag.

We can exfiltrate the flag length by sending rules to check the value of `uint32(@nops[1] + 16 + 8 + 8 + 4)`, which is the `flag_size` argument for `decode`. Remember we also need to modulo cycle the key using the `key_size` variable, which we can read using `uint32(@nops[1] + 16 + 4)`.

Here is the final rule template

```
template = """import "pe"
rule nopss {
    strings:
        $nops = {90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 C7 44 24 0c}
    condition:
        %s ==
            uint8(
                uint32(@nops[1] + 16 + 8 + 8 + 8 + 3)
                - pe.sections[1].virtual_address
                - pe.image_base + pe.sections[1].raw_data_offset
                + %d
            ) ^
            uint8(
                uint32(@nops[1] + 16 + 8 + 4)
                - pe.sections[1].virtual_address
                - pe.image_base + pe.sections[1].raw_data_offset
                + (%d %% uint32(@nops[1] + 16 + 4))
            )
}"""

flag_size = 10
for i in range(flag_size):
    for guess in range(32,127):
        ch = chr(guess)
        rule = template % (str(hex(guess)), i, i)
```

And boom, flag! `TMCTF{16d1eb767f}`

Yara ends up being used as a simple string/hex search tool in the rare CTF challenge where it appears, but this one was definitely a lot more complicated and fun to figure out. A cool challenge for sure and one that shows off more of what yara can do.