



**МІНІСТЕРСТВО ОСВІТИ, НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ**

Лабораторна робота 2

Засоби автоматизації аналізу

Варіант №5

Підготував:

студент 4 курсу

групи ФІ-84

Коломієць Андрій Юрійович

Email: *andkol-ipt22@lil.kpi.ua*

Викладач:

Київ – 2021

Лабораторна робота 2

Засоби автоматизації аналізу

Мета роботи

Отримати навички автоматизації методів аналізу програмного коду.

Постановка задачі

Дослідити методи обфускації та поліморфізму ШПЗ, дослідити статичні та динамічні методи деобфускації.

Завдання

Проаналізуйте обфускатор (encoder) з Metasploit.

Реалізуйте статичний деобфускатор.

Реалізуйте динамічний деобфускатор.

Обфускатор	Коментар
x86/opt_sub	Sub (optimised)

Виконання лабораторної роботи

Проаналізуємо обфускатор (encoder) з Metasploit

Інформацію повну та реалізацію, щодо обфускатора можна знайти за посиланням.

https://github.com/rapid7/metasploit-framework/blob/master/modules/encoders/x86/opt_sub.rb

Опис роботи є в самому файлі обфускатора **x86/opt_sub**:

```
def initialize
  super(
    'Name'          => 'Sub Encoder (optimised)',
    'Description'    => %q{
      Encodes a payload using a series of SUB instructions and writing the
      encoded value to ESP. This concept is based on the known SUB encoding
      approach that is widely used to manually encode payloads with very
      restricted allowed character sets. It will not reset EAX to zero unless
      absolutely necessary, which helps reduce the payload by 10 bytes for
      every 4-byte chunk. ADD support hasn't been included as the SUB
      instruction is more likely to avoid bad characters anyway.
      The payload requires a base register to work off which gives the start
      location of the encoder payload in memory. If not specified, it defaults
      to ESP. If the given register doesn't point exactly to the start of the
      payload then an offset value is also required.
      Note: Due to the fact that many payloads use the FSTENV approach to
      get the current location in memory there is an option to protect the
      start of the payload by setting the 'OverwriteProtect' flag to true.
      This adds 3-bytes to the start of the payload to bump ESP by 32 bytes
      so that it's clear of the top of the payload.
    },
    'Author'        => 'OJ Reeves <oj[at]buffered.io>',
    'Arch'          => ARCH_X86,
    'License'       => MSF_LICENSE,
    'Decoder'       => { 'BlockSize' => 4 }
  )
end
```

Крім принципу роботи обфускатора, необхідно вміти декодувати обфусковані файли. Для цього можна проаналізувати його роботу на основі файлу **x86/opt_sub**. У коді є коментарі, що вказують на те що розкодовується сам **payload** (обфускований обфускатором) за допомогою:

```
185
186 #
187 # Determine the bytes, if any, that will result in the given chunk
188 # being decoded using SUB instructions from the previous EAX value
189 #
```

та значення декодовані виводяться в:

слід зауважити **на підготовці кодування корисного навантаження**, закодовані значення поміщуються у стек в зворотньому порядку:

ТОБТО

корисне

навантаження при декодуванні необхідно буде правильно виводити в прямому або зворотньому порядку.

Реалізуємо статичну деобфускацію

Створимо тестовий шеллкод та обфускуємо, без використання шаблону виконуваного файлу:

```
(kali㉿kali)-[~/Documents]
$ echo -en '\xccMalware analize'>sc
```

```
(kali@kali)-[~/Documents]
$ msfvenom -p generic/custom payloadfile=sc -f raw -o payload.bin -e x86/opt_sub
[-] No platform was selected, choosing Msf::Module::Platform from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/opt_sub
x86/opt_sub succeeded with size 125 (iteration=0)
x86/opt_sub chosen with final size 125
Payload size: 125 bytes
Saved as: payload.bin
```

```
(kali㉿kali)-[~/Documents]
$ ll
total 8
-rw-r--r-- 1 kali kali 125 Oct 15 13:07 payload.bin
-rw-r--r-- 1 kali kali 24 Oct 15 13:02 sc
```

Дизасемблюємо отримані файли:

```
(kali㉿kali)-[~/Documents]
$ ndisasm -b32 sc
00000000 E280          loop 0xffffffff82
00000002 99          cdq
00000003 7863        js 0x68
00000005 634D61      arpl [ebp+0x61],cx
00000008 6C          insb
00000009 7761        ja 0x6c
0000000B 7265        jc 0x72
0000000D 20616E      and [ecx+0x6e],ah
00000010 61          popa
00000011 6C          insb
00000012 69          db 0x69
00000013 7A65        jpe 0x7a
00000015 E280          loop 0xffffffff97
00000017 99          cdq
```

```

(kali@kali)-[~/Documents]
$ ndisasm -b32 payload.bin
00000000 54          push esp
00000001 58          pop eax
00000002 2D6BFFFFFF sub eax,0xffffffff
00000007 2D00000000 sub eax,0x0
0000000C 2D00000000 sub eax,0x0
00000011 50          push eax
00000012 5C          pop esp
00000013 2500000000 and eax,0x0
00000018 2500000000 and eax,0x0
0000001D 2D9B1D7F66 sub eax,0x667f1d9b
00000022 2D00000000 sub eax,0x0
00000027 2D00000000 sub eax,0x0
0000002C 50          push eax
0000002D 2D0476171F sub eax,0x1f177604
00000032 2D00000000 sub eax,0x0
00000037 2D00000000 sub eax,0x0
0000003C 50          push eax
0000003D 2DFC4B080C sub eax,0xc084bfc
00000042 2D00000000 sub eax,0x0
00000047 2D00000000 sub eax,0x0
0000004C 50          push eax
0000004D 2DF9A8FFFB sub eax,0xfbffa8f9
00000052 2D00000000 sub eax,0x0
00000057 2D00000000 sub eax,0x0
0000005C 50          push eax
0000005D 2D09141411 sub eax,0x11141409
00000062 2D00000000 sub eax,0x0
00000067 2D00000000 sub eax,0x0
0000006C 50          push eax
0000006D 2D81E2B3E8 sub eax,0xe8b3e281
00000072 2D00000000 sub eax,0x0
00000077 2D00000000 sub eax,0x0
0000007C 50          push eax

```

Наше корисне навантаження для експерименту файл *sc*, за допомогою обфускатора *x86/opt_sub* був закодований в файл вже *payload.bin*.

Додатковий аналіз можна зробити за допомогою дизасемблера *Cutter*.

В даній програмі можна декомпілювати зразки асемблерного коду на мови програмування *C/C++*:

```

/* jsdec pseudo code output */

/* /home/kali/Documents/payload.bin @ 0x0 */
#include <stdint.h>
int32_t fcn_00000000 (void) {
    rax = rsp;
    eax -= 0xffffffff6b;
    eax = 0;
    eax = 0;
    eax -= 0x667f1d9b;
    eax -= 0x1f177604;
    eax -= 0xc084bfc;
    eax -= 0xfbffa8f9;
    eax -= 0x11141409;
    eax -= 0xe8b3e281;
}

```

```

/* jsdec pseudo code output */
/* /home/kali/Documents/payload.bin @ 0x0 */
#include <stdint.h>

int32_t fcn_00000000 (void) {
    rax = rsp;
    eax -= 0xffffffff6b;
    eax = 0;
    eax = 0;
    eax -= 0x667f1d9b;
    eax -= 0x1f177604;
    eax -= 0xc084bfc;
    eax -= 0xfbffa8f9;
    eax -= 0x11141409;
    eax -= 0xe8b3e281;
}

```

Також можна поглянути на шістнадцятковий дамп пам'яті:

Шестнадцатеричный дамп																																	
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	
0x0000000000000000	54	58	2d	6b	ff	ff	ff	2d	00	00	00	00	2d	00	00	00	50	5c	25	00	00	00	00	25	00	00	00	00	2d	9b	1d	TX-KP%.....%
0x0000000000000020	7f	66	2d	00	00	00	00	00	00	00	00	00	50	2d	04	76	17	1f	2d	00	00	00	00	2d	00	00	00	00	50	2d	fc	4b	..f-.....P-..v.....P-..K
0x0000000000000040	08	0c	2d	00	00	00	00	2d	00	00	00	00	50	2d	f9	a8	ff	fb	2d	00	00	00	00	2d	00	00	00	00	50	2d	09	14	..f-.....P-..v.....P-..
0x0000000000000060	14	11	2d	00	00	00	00	2d	00	00	00	00	50	2d	81	e2	b3	e8	2d	00	00	00	00	2d	00	00	00	00	50	ff	ff	ff	..f-.....P-.....P-..
0x0000000000000080	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff
0x00000000000000a0	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff
0x00000000000000c0	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff

Певні розпізнати ознаки ключа ми не можемо в шістнадцятковому дампі, оскільки ключ в обфускаторі не використовується. Також тут не можна побачити наше корисне навантаження декодоване в символічне представлення, оскільки воно зберігається в регістрах *eax* чи *esp*.

Дизасембльований код:

```
fcn.00000000 ();
0x00000000 push    rsp
0x00000001 pop     rax
0x00000002 sub     eax, 0xffffffff6b ; 4294967147
0x00000007 sub     eax, 0
0x0000000c sub     eax, 0
0x00000011 push    rax
0x00000012 pop     rsp
0x00000013 and     eax, 0
0x00000018 and     eax, 0
0x0000001d sub     eax, 0x667f1d9b
0x00000022 sub     eax, 0
0x00000027 sub     eax, 0
0x0000002c push    rax
0x0000002d sub     eax, 0x1f177604
0x00000032 sub     eax, 0
0x00000037 sub     eax, 0
0x0000003c push    rax
0x0000003d sub     eax, 0xc084bfc
0x00000042 sub     eax, 0
0x00000047 sub     eax, 0
0x0000004c push    rax
0x0000004d sub     eax, 0xfbf8a8f9
0x00000052 sub     eax, 0
0x00000057 sub     eax, 0
0x0000005c push    rax
0x0000005d sub     eax, 0x11141409
0x00000062 sub     eax, 0
0x00000067 sub     eax, 0
0x0000006c push    rax
0x0000006d sub     eax, 0xe8b3e281
0x00000072 sub     eax, 0
0x00000077 sub     eax, 0
0x0000007c push    rax
```

```
fcn.00000000 ();
0x00000000 push    rsp
0x00000001 pop     rax
0x00000002 sub     eax, 0xffffffff6b ; 4294967147
0x00000007 sub     eax, 0
0x0000000c sub     eax, 0
0x00000011 push    rax
0x00000012 pop     rsp
0x00000013 and     eax, 0
0x00000018 and     eax, 0
0x0000001d sub     eax, 0x667f1d9b
0x00000022 sub     eax, 0
0x00000027 sub     eax, 0
0x0000002c push    rax
0x0000002d sub     eax, 0x1f177604
0x00000032 sub     eax, 0
0x00000037 sub     eax, 0
0x0000003c push    rax
0x0000003d sub     eax, 0xc084bfc
0x00000042 sub     eax, 0
0x00000047 sub     eax, 0
0x0000004c push    rax
0x0000004d sub     eax, 0xfbf8a8f9
0x00000052 sub     eax, 0
0x00000057 sub     eax, 0
0x0000005c push    rax
0x0000005d sub     eax, 0x11141409
0x00000062 sub     eax, 0
0x00000067 sub     eax, 0
0x0000006c push    rax
0x0000006d sub     eax, 0xe8b3e281
0x00000072 sub     eax, 0
0x00000077 sub     eax, 0
0x0000007c push    rax
0x0000007d invalid
```

Не виконуючи програму ми не можемо наперед проаналізувати, які значення в нас в регістрі *eax* та *esp*, але зазначені значення будуть вказувати на декодовані конструкції згідно зауваження, що здійснювали при аналізі.

```
185
186 #
187 # Determine the bytes, if any, that will result in the given chunk
188 # being decoded using SUB instructions from the previous EAX value
189 #
```

Значення в регістрах **eax** можна отримати використовуючи **python shell** або **C++**, але без автоматизації на основі того, як зберігаються від'ємні числа в комп'ютері (**доповнення до двох**):

```
#include <iostream>
#include <stdint.h>

using namespace std;

int32_t main()
{
    int32_t eax = 0;

    cout<<endl<<"In EAX saves decoded payload:"<<endl;

    eax -= 0x667f1d9b;
    cout<<endl<<"\t0x"<<std::hex<<eax<<endl;
    eax -= 0x1f177604;
    cout<<endl<<"\t0x"<<std::hex<<eax<<endl;
    eax -= 0xc084bfc;
    cout<<endl<<"\t0x"<<std::hex<<eax<<endl;
    eax -= 0xfbf8f9;
    cout<<endl<<"\t0x"<<std::hex<<eax<<endl;
    eax -= 0x11141409;
    cout<<endl<<"\t0x"<<std::hex<<eax<<endl;
    eax -= 0xe8b3e281;
    cout<<endl<<"\t0x"<<std::hex<<eax<<endl;

    return 0;
}
```



Зворотні порядок виводу, оскільки алгоритм використовує інструкції `sub` котрі записують результат до стеку. Відповідно треба виводити в зворотньому порядку.

In EAX saves decoded payload:

0x9980e265

0x7a696c61

0x6e612065

0x7261776c

0x614d6363

0x789980e2

Hex to ASCII Text Converter

Enter hex bytes with any prefix / postfix / delimiter and press the Convert button
(e.g. 45 78 61 6d 70 6C 65 21):

Open File

Paste hex numbers or drop file

9980e265
7a696c61
6e612065
7261776c
614d6363
789980e2

Character encoding

ASCII

Convert

Reset

Swap

âezilana erawlaMccxâ

Copy

Save

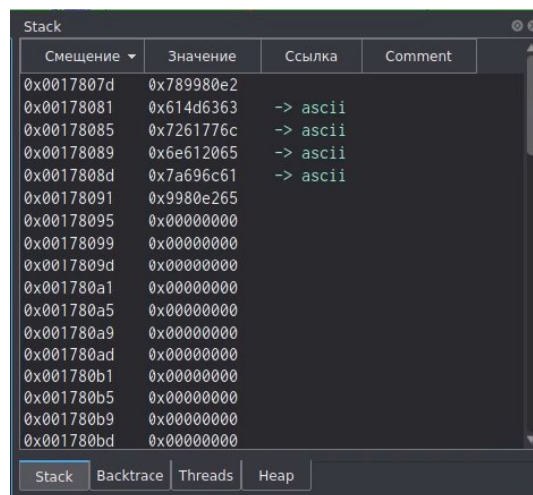
Реалізуємо динамічну деобфускацію

Динамічну деобфускацію можна зрозуміти на емуляції програми в дизасемблері **Cutter**, але необхідно виконуваний файл для цього з розширенням **.exe** в даному випадку буде **C/C++** взята програма з виводом **“Hello world!”**.

```
(kali@kali)-[~/Documents]
$ msfvenom -p generic/custom payloadfile=sc -f exe -x Hello_world.exe -o Hello_world_modify.exe -e x86/opt_sub
[-] No platform was selected, choosing Msf::Module::Platform from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/opt_sub
x86/opt_sub succeeded with size 125 (iteration=0)
x86/opt_sub chosen with final size 125
Payload size: 125 bytes
Final size of exe file: 49152 bytes
Saved as: Hello_world_modify.exe
```

Оскільки значення декодованого корисного навантаження виводиться в стек, на основі цього можна вручну дізнатися вміст нашого корисного навантаження, в іншому випадку слід виконувати **трасування стеку**, або **виводити на етапі відлагодження кожен раз значення з зазначених регістрів**.

Виконаємо **емуляцію** фрагменту коду поданого на диасемблюванні попередніх файлів (**див. Cutter emulation каталог**).



Смещение	Значение	Ссылка	Comment
0x0017807d	0x789980e2		
0x00178081	0x614d6363	-> ascii	
0x00178085	0x7261776c	-> ascii	
0x00178089	0x6e612065	-> ascii	
0x0017808d	0x7a696c61	-> ascii	
0x00178091	0x9980e265		
0x00178095	0x00000000		
0x00178099	0x00000000		
0x0017809d	0x00000000		
0x001780a1	0x00000000		
0x001780a5	0x00000000		
0x001780a9	0x00000000		
0x001780ad	0x00000000		
0x001780b1	0x00000000		
0x001780b5	0x00000000		
0x001780b9	0x00000000		
0x001780bd	0x00000000		

Під час виконання програми, в стек було поміщено наступні значення, що показані на картинці. Ці значення можна спостерігати також в регістрі **eax (rsp)**.

Розглянемо динамічний аналіз виконуваного коду на прикладі емуляції шелл-коду для **x86_64** за допомогою **Unicorn Engine**:

dinamics_obfuscator.py

```
#!/usr/bin/env ipython

#-----#

from __future__ import print_function

from unicorn import *
from unicorn.x86_const import *

from pwn import *

from capstone import *

#-----#
result=[]

CODE=read('payload.bin')

md = Cs(CS_ARCH_X86, CS_MODE_64)

print("\nDisassembler code:\n")

for i in md.disasm(CODE, 0x1000):
    print("0x%x:\t%s\t%s" %
          (i.address, i.mnemonic, i.op_str))

#-----#

CODE = open("payload.bin","rb").read()

address = 0x1000

print("\nIn RAX saves decoded payload:\n")

def hook_code(uc, address, size, user_data):
    rax = mu.reg_read(UC_X86_REG_RAX)
    result.append(hex(rax))
    print(">>> RAX = 0x%x" %rax)
```

Disassembler code:

```
0x1000: push    rsp
0x1001: pop     rax
0x1002: sub     eax, 0xffffffff6b
0x1007: sub     eax, 0
0x100c: sub     eax, 0
0x1011: push    rax
0x1012: pop     rsp
0x1013: and     eax, 0
0x1018: and     eax, 0
0x101d: sub     eax, 0x667fd9b
0x1022: sub     eax, 0
0x1027: sub     eax, 0
0x102c: push    rax
0x102d: sub     eax, 0x1f177604
0x1032: sub     eax, 0
0x1037: sub     eax, 0
0x103c: push    rax
0x103d: sub     eax, 0xc084bfc
0x1042: sub     eax, 0
0x1047: sub     eax, 0
0x104c: push    rax
0x104d: sub     eax, 0xfbffa8f9
0x1052: sub     eax, 0
0x1057: sub     eax, 0
0x105c: push    rax
0x105d: sub     eax, 0x11141409
0x1062: sub     eax, 0
0x1067: sub     eax, 0
0x106c: push    rax
0x106d: sub     eax, 0xe8b3e281
0x1072: sub     eax, 0
0x1077: sub     eax, 0
0x107c: push    rax
```

```

mu=Uc(UC_ARCH_X86,UC_MODE_64)
mu.mem_map(address,address+0x2000)
mu.mem_write(address, CODE)
mu.reg_write(UC_X86_REG_ESP, address+0x1000)
mu.hook_add(UC_HOOK_CODE, hook_code, begin=0x1001, end=0x107c)
mu.emu_start(address, address+len(CODE))

final = list(dict.fromkeys(result))

final.remove('0x0')

print("\nResult string in hex unordered: ", final)

str = "".join(final)

str = str.replace("0x", "")

print("\nResult string: ", binascii.unhexlify(str)[::-1] )
#-----#

```

In RAX saves decoded payload:

```

>>> RAX = 0x0
>>> RAX = 0x2000
>>> RAX = 0x2095
>>> RAX = 0x2095
>>> RAX = 0x2095
>>> RAX = 0x2095
>>> RAX = 0x2095
>>> RAX = 0x0
>>> RAX = 0x0
>>> RAX = 0x9980e265
>>> RAX = 0x9980e265
>>> RAX = 0x9980e265
>>> RAX = 0x9980e265
>>> RAX = 0x7a696c61
>>> RAX = 0x7a696c61
>>> RAX = 0x7a696c61
>>> RAX = 0x7a696c61
>>> RAX = 0x6e612065
>>> RAX = 0x6e612065
>>> RAX = 0x6e612065
>>> RAX = 0x6e612065
>>> RAX = 0x7261776c
>>> RAX = 0x7261776c
>>> RAX = 0x7261776c
>>> RAX = 0x7261776c
>>> RAX = 0x614d6363
>>> RAX = 0x614d6363
>>> RAX = 0x614d6363
>>> RAX = 0x614d6363
>>> RAX = 0x789980e2
>>> RAX = 0x789980e2
>>> RAX = 0x789980e2

```

Результати роботи *автоматичного динамічного деобфускатора* можна розглянути на відповідних скріншотах:

```

Result string in hex unordered: ['0x2000', '0x2095', '0x9980e265', '0x7a696c61', '0x6e612065', '0x7261776c', '0x614d6363', '0x789980e2']
Result string:  b'\xe2\x80\x99\xccMalware analyze\xe2\x80\x99\x95 \x00 '

```

На кожному кроці емуляції, виводиться значення регістру *rax*, як видно вони співпадають з тими, що ми досліджували в *Cutter* .