

!error OKN	: Display error	***** rax (64 bits)
!address	: Display information about memory	***** eax (32 bits)
~	: List threads	*** ax (16 bits)
bl	: List breakpoints	** ah (8 bits)
bc	: Cancel breakpoints	** al (8 bits)
be	: Enable breakpoints	
bd	: Disable breakpoints	
bp [Addr]	: Set breakpoint at the address	[IDA Pro shortcuts]
bm SymPattern	: Set breakpoint at the symbol	Navigation:
ba [r w e] Addr	: Set breakpoint on Access	Enter : Jump to operand ESC : Jump to previous position
k	: Display call stack	G : Go to address Ctrl+L : Jump by name
r	: Dump all registers	CTRL+P : Jump to function X : xref
u	: Disassemble	CTRL+E : Jump to entry point
dn	: Display where N:	
	a: ascii chars u: Unicode char	Search
	b: byte + ascii w: word	Alt+C : Next code Ctrl+D : Next data
	W: word + ascii d: dword	Alt+I : Immediate value Ctrl+I : Next immediate value
	c: dword + ascii q: qword	Alt+T : Text Ctrl+T : Next text
	b: bin + byte d: bin + dword	Alt+B : Sequence of bytes Ctrl+B : Next sequence of bytes
eN Addr Value	: Edit memory	
.writemem f A S	: Dump memory	Graphing
	f: file name	F12 : Flow chart Ctrl+F12 : Function calls
	A: Address	
	S: Size (Lx)	Subviews
		Shift+F4 : Name Shift+F3 : Functions
		Shift+F12 : Strings Shift+F7 : Segments
dec hex char	dec hex char	dec hex char
0 0x00 NUL	32 0x20 SPACE	64 0x40 @
1 0x01 SOH	33 0x21 !	65 0x41 A
2 0x02 STX	34 0x22 "	66 0x42 B
3 0x03 ETX	35 0x23 #	67 0x43 C
4 0x04 EOT	36 0x24 \$	68 0x44 D
5 0x05 ENQ	37 0x25 %	69 0x45 E
6 0x06 ACK	38 0x26 &	70 0x46 F
7 0x07 BEL	39 0x27 '	71 0x47 G
8 0x08 BS	40 0x28 (72 0x48 H
9 0x09 TAB	41 0x29)	73 0x49 I
10 0x0A LF	42 0x2A *	74 0x4A J
11 0x0B VT	43 0x2B +	75 0x4B K
12 0x0C FF	44 0x2C ,	76 0x4C L
13 0x0D CR	45 0x2D -	77 0x4D M
14 0x0E SO	46 0x2E .	78 0x4E N
15 0x0F SI	47 0x2F /	79 0x4F O
16 0x10 DLE	48 0x30 0	80 0x50 P
17 0x11 DC1	49 0x31 1	81 0x51 Q
18 0x12 DC2	50 0x32 2	82 0x52 R
19 0x13 DC3	51 0x33 3	83 0x53 S
20 0x14 DC4	52 0x34 4	84 0x54 T
21 0x15 NAK	53 0x35 5	85 0x55 U
22 0x16 SYN	54 0x36 6	86 0x56 V
23 0x17 ETB	55 0x37 7	87 0x57 W
24 0x18 CAN	56 0x38 8	88 0x58 X
25 0x19 EM	57 0x39 9	89 0x59 Y
26 0x1A SUB	58 0x3A :	90 0x5A Z
27 0x1B ESC	59 0x3B ;	91 0x5B [
28 0x1C FS	60 0x3C <	92 0x5C \
29 0x1D GS	61 0x3D =	93 0x5D]
30 0x1E RS	62 0x3E >	94 0x5E ^
31 0x1F US	63 0x3F ?	95 0x5F _
		96 0x60 `
		97 0x61 a
		98 0x62 b
		99 0x63 c
		100 0x64 d
		101 0x65 e
		102 0x66 f
		103 0x67 g
		104 0x68 h
		105 0x69 i
		106 0x6A j
		107 0x6B k
		108 0x6C l
		109 0x6D m
		110 0x6E n
		111 0x6F o
		112 0x70 p
		113 0x71 q
		114 0x72 r
		115 0x73 s
		116 0x74 t
		117 0x75 u
		118 0x76 v
		119 0x77 w
		120 0x78 x
		121 0x79 y
		122 0x7A z
		123 0x7B {
		124 0x7C
		125 0x7D }
		126 0x7E ~
		127 0x7F DEL

Máster en Análisis de Malware, Reversing y Bug Hunting



MÓDULO 3 – Análisis de Código Fuente

<https://campusciberseguridad.com>

1. INTRODUCCIÓN

Esta es la segunda y última evaluación de la asignatura de análisis de código fuente. En ella, nos centraremos en los dos últimos temas, los cuales versan sobre el uso de todas las técnicas y definiciones que hemos estado viendo en los temas anteriores de forma práctica.

- El primer ejercicio está basado en código de bajo nivel (ensamblador x86)
- El segundo ejercicio está basado en código de alto nivel (lenguaje C)

1.1. OBJETIVOS DE APRENDIZAJE

El aprendizaje de esta práctica os mete de lleno en el análisis de código, tanto fuente como procedente de un desensamblado.

Esta práctica no solo se asemeja, sino que es prácticamente parte del proceso que una persona que analiza malware de forma profesional efectúa en su día a día.

2. EVALUACIÓN

Cada uno de los dos ejercicios está dividido en varias tareas con dificultad ascendente. La puntuación es idéntica para cada uno de los ejercicios y para cada una de las tareas dentro de cada ejercicio.

Criterios de evaluación

- **Claridad en las explicaciones.** Incluso una solución que está mal se puntuará de forma parcialmente positiva si se ha explicado e ilustrado de forma correcta y apropiada. Ya en vuestras calificaciones se os indicará que está mal.
- **No se tendrán en cuenta respuestas que no razonen los hallazgos encontrados.** Ejemplo:

“Esta función parece que cifra un archivo”

vs

“La función toma como parámetro un archivo (su descriptor) y lee el contenido el cual va almacenando en un buffer que se recorre en un bucle for. Este bucle va iterando sobre cada byte cifrando el contenido con RC4. La función no retorna nada, ya que su objetivo es modificar lo que le es pasado por parámetro.”

- **No es necesario que os expliquéis de manera formal.** Se os anima a hacerlo, puesto que es la forma de expresión que se os pedirá de manera profesional cuando redactéis un informe.
- **En ningún caso se puntua negativamente,** solo positivo. Si algo está mal se os dirá y corregirá en vuestra evaluación personal.

3. FORMATO DE ENTREGA

Podéis usar el formato en el que os encontréis más cómodos siempre y cuando sea un formato conocido: Word, OpenOffice, PDF, etc.

Idealmente, el formato sería en PDF.

4. AVISO

IMPORTANTE IMPORTANTE IMPORTANTE

NO USEIS VUESTRO SISTEMA

USAD LA MÁQUINA VIRTUAL

**DESCONECTAD LA RED DE LA MÁQUINA
VIRTUAL**

**DESECHAD EL ESTADO DE LA MÁQUINA
VIRTUAL UNA VEZ FINALIZADOS LOS
EJERCICIOS**

IMPORTANTE IMPORTANTE IMPORTANTE

5. EJERCICIO 1

Para este ejercicio os descargaréis un ejecutable (**NO ES MALWARE**) creado específicamente para vosotros.

Se trata de un ejecutable para sistemas Linux (32 bits) basados en arquitectura x86. Para ayudaros, se le han incluido los símbolos de depuración.

Al ejecutar el programa, **este examinará ciertas condiciones** y no se ejecutará de forma real sino se cumplen. Es lo que haría un malware en un ambiente real: comprobar que se dan las circunstancias oportunas para ejecutar la parte maliciosa.

Podéis utilizar cualquier herramienta. Lo fundamental en esta prueba es que aprendáis a desensamblaros con un desensamblador e interpretar el código.

NOTA: El archivo de encuentra en el área de descargas de la asignatura. Se trata de un archivo ZIP cuya clave es “**infected**”. Esta clave es un convenio (o estándar de facto) en la industria. Los analistas se suelen intercambiar muestras de esta manera y por convención la clave siempre suele ser esa.

De nuevo, no se trata de un programa malicioso.

5.1. TAREA 1

Descubrid a través de la lectura del código de la función ‘main’ desensamblada como ejecutar el programa.

Es decir, **NO ES NECESARIO MANIPULAR EL PROGRAMA**. Simplemente entendiendo como funciona ‘main’ es posible adivinar como evadir las protecciones y ejecutar el programa.

Se pide:

- Detallad y explicad mostrando el código ensamblador cuales son las condiciones para que el programa se ejecute.
- Demostrad (mediante capturas de pantalla) la ejecución del programa.

5.2. TAREA 2

Detallad que hace el programa realmente.

Se pide:

- El programa, además de “main”, solo tiene una única función. Encontradla y explicad **que hace y como lo hace** esta función mostrando el código ensamblador correspondiente.

5.3. PISTAS

- El ensamblador es x86 32 bits
- Comenzar desensamblando el código (ajustad la sintáxis de x86 a vuestro gusto: Intel o AT&T)
- Comenzad por 'main', es el punto de entrada de todos los programas, es fácil localizarlo
- ¿Qué va haciendo 'main'? ¿Se están cumpliendo todas las condiciones que inspecciona el programa?
- Además de 'main' el programa contiene otra función, el resto son llamadas a funciones externas.
- Podéis usar todas las herramientas que queráis, pero de nuevo, se resuelve con un buen desensamblador y estudiando el código parte a parte. Es decir, analizando.

6. EJERCICIO 2

Para este ejercicio emplearemos el código fuente del ransomware Babuk, el cual **NO DEBEIS COMPILAR NI MUCHO MENOS EJECUTAR**.

Babuk es un ransomware que tuvo un significativo auge a comienzos de 2021 y cuyo código fue liberado de forma súbita por sus creadores. Este malware es multiplataforma, existiendo versiones para Linux, Windows, routers y varios dispositivos IoT. Su código está escrito en los lenguajes Go y C.

El código fuente está disponible en el siguiente repositorio:

<https://github.com/Hildaboo/BabukRansomwareSourceCode>

6.1. TAREA 1

Una de las características del malware es que ejecuta una rutina o función que detiene ciertos procesos del sistema para evitar su detección por parte de estos. En otras ocasiones suele detectar la presencia de herramientas de análisis y detiene su propia ejecución o la camufla para evitar ser analizada.

Esta tarea consiste en la localización de la función (ojo, no donde esta es llamada) que detiene los procesos en **la versión para sistemas Windows** del malware Babuk.

Es decir:

- Localizad el código de la función (de nuevo, NO DONDE ES LLAMADA).
- Realizad un análisis de código de la función.

- Enumerar los procesos que tiene en cuenta esta función para que sean detenidos.

Documentad profusamente el proceso que habéis realizado (capturas de pantalla, evidencias).

Cuanto más detallada (en calidad) esté hecho el análisis mayor será la puntuación.

6.2. TAREA 2

[Aquí](#), se encuentra la función “main” del código del cifrador escrito en Go **de la versión para dispositivos NAS** de Babuk.

Esa función representa el punto de entrada del ejecutable. Dicho ejecutable es el que utilizan los cibercriminales para cifrar un sistema. Es decir, explotan un sistema, filtran el ejecutable y este se encarga de ir cifrando todos los archivos de la víctima en ese sistema.

Comentad el funcionamiento de la función referenciada con todo el detalle posible.

Si no entendéis algunas partes saltadlas y seguid o al menos comentad lo que creéis que es.

Código:

<https://github.com/Hildaboo/BabukRansomwareSourceCode/blob/main/nas/enc/main.go#L810>