

# Adversary emulation: anatomia de un ciberataque

Tu Nombre

6 de noviembre de 2025

# Índice general

<b>1. Introducción</b>	<b>4</b>
1.1. Anatomía de un adversario . . . . .	4
1.2. Objetivos del trabajo . . . . .	4
<b>2. Marco Teórico</b>	<b>5</b>
2.1. MITRE ATT&CK . . . . .	5
<b>3. Threat Intelligence</b>	<b>6</b>
<b>4. Panorama Actual de Amenazas</b>	<b>7</b>
<b>5. Creando nuestro adversario</b>	<b>8</b>
<b>6. Anexo A</b>	<b>9</b>
6.1. Tipos de Malware . . . . .	9
6.2. Listado de EDRs . . . . .	9
<b>7. Anexo B: Windows Internals</b>	<b>10</b>
7.1. Memoria Virtual . . . . .	10
7.1.1. Problemas Memoria Física . . . . .	10
7.1.2. Acceso indirecto a memoria . . . . .	10
7.1.3. Paginación . . . . .	10
7.1.4. Tabla de paginas . . . . .	11
7.1.5. MMU . . . . .	11
7.1.6. Acceso a memoria . . . . .	11
7.2. Process . . . . .	12
7.2.1. EPROCESS . . . . .	12
7.2.2. Threads . . . . .	12
7.2.3. Handles . . . . .	12
7.2.4. Memory . . . . .	12
7.2.5. Modules . . . . .	12
7.3. Process Creation (Kernel) . . . . .	12

7.4.	Portable Executable (PE)	14
7.4.1.	Sections	14
7.4.2.	Imports	14
7.4.3.	Exports	14
7.4.4.	Relocations	14
7.4.5.	AddressOfEntryPoint	14
7.4.6.	Subsystem	14
7.4.7.	Virtual Address	15
7.4.8.	Raw Data	15
7.5.	Secciones de un proceso	16
7.6.	DLL (Dynamic-Link Library)	17
7.6.1.	Carga de DLLs en un proceso	17
7.6.2.	PEB y la lista de módulos	17
7.6.3.	Import Address Table (IAT)	18
7.6.4.	Resumen	18
7.7.	Win32 API	20
7.8.	SO Security	21
7.8.1.	ASLR (Address Space Layout Randomization)	21
7.8.2.	DEP (Data Execution Prevention)	21
7.8.3.	Control Flow Guard (CFG)	21
7.8.4.	Code Integrity Guard (CIG)	21
7.8.5.	Protected Process Light (PPL)	21
7.8.6.	Virtualization-Based Security (VBS)	21
7.8.7.	Kernel-mode code signing (KMCI)	21
7.9.	Genealogía de Procesos en Windows	22
7.10.	Debugging flags	24
7.11.	ABI	25

*Jugez un homme par ses  
questions plutôt que par ses  
réponses.*

---

— Voltaire

# Capítulo 1

## Introducción

*¿Porque alguien debería leer este trabajo?*

### 1.1. Anatomía de un adversario

Que es un adversario

### 1.2. Objetivos del trabajo

El objetivo de este trabajo es desarrollar un escenario didáctico que permita a los analistas de ciberseguridad mejorar sus capacidades de detección e investigación mediante la comprensión del proceso completo de creación de un ciberataque. El resultado final del proyecto será un conjunto de artefactos forenses de los sistemas infectados, que servirá como base para ejercicios de análisis forense (blue team).

Para generar dicho recurso, el núcleo del trabajo se centrará en la concepción, diseño y desarrollo de un ciberataque realista, su ejecución controlada en un sistema víctima y la documentación íntegra del proceso ofensivo llevado a cabo. De este modo, quien reciba los artefactos forenses podrá analizar la infección desde el punto de vista del analista forense, pero también interpretar las acciones del atacante, entendiendo mejor el cómo y el porqué de cada artefacto encontrado.

En definitiva, este trabajo busca aproximar al analista a la mentalidad del atacante, con el fin de reforzar sus habilidades de detección, interpretación y respuesta ante amenazas reales.

# Capítulo 2

## Marco Teórico

*¿Como la industria consigue tener un lenguaje comun para poder transmitir informacion sobre adversarios de forma efectiva?*

### 2.1. MITRE ATT&CK

# Capítulo 3

## Threat Intelligence

*¿Como obtener informacion sobre adversarios?*

## Capítulo 4

# Panorama Actual de Amenazas

*¿Cuales son las ttp mas utilizadas en la actualidad?*

Nos centraremos en dos documentos principalemten el m-trends de man-diant y el



# Capítulo 5

## Creando nuestro adversario

nos apoyaremos en el framework cyber kill chain para estructurar el ataque

el framework consiste en dividir un ciberataque en cinco fases:

1. Reconocimiento: investigacion del objetivo 2. Armazenamiento: creacion del payload 3. Entrega: envio del payload al objetivo 4. Explotacion: ejecucion del payload 5. Instalacion: sistema de staging 6. Comando y control: establecimiento del canal de comunicacion 7. Acciones sobre objetivos: acciones finales

stage 1 (stub) (Downloader) stage 2 (Dropper) payload payload final

# Capítulo 6

## Anexo A

### 6.1. Tipos de Malware

RAT ransomware reverse shell

### 6.2. Listado de EDRs

Sylantstrike puede ser utilizado para hacer pruebas de concepto de EDRs

- CrowdStrike Falcon
- Microsoft Defender for Endpoint

# Capítulo 7

## Anexo B: Windows Internals

### 7.1. Memoria Virtual

#### 7.1.1. Problemas Memoria Fisica

- Escasez de memoria
- Fragmentación
- Acceso a memoria (Seguridad)

#### 7.1.2. Acceso indirecto a memoria

Cada proceso tiene su propio espacio de direcciones virtual, Para el proceso es como tener un espacio de memoria fisica contiguo reservado, Pero por detras el SO esta mapeando esas direcciones virtuales a distintas regiones de memoria fisica.

#### 7.1.3. Paginación

Mapear cada byte de la memoria virtual a un byte de la memoria fisica seria muy ineficiente, Se crean paginas de memoria que son bloques de bytes para simplificar este proceso (normalmente de 4KB | 4096 bytes),

page - memoria virtual frame - memoria fisica

Cuando un proceso se carga sus paginas son cargadas en frames disponibles en memoria fisica

#### **7.1.4. Tabla de paginas**

Cada proceso tiene una tabla de paginas que mapea las direcciones virtuales a las fisicas, La tabla de paginas es gestionada por la MMU (Memory Management Unit), Para cada pagina se mantiene, su numero de frame, permisos, etc.

#### **7.1.5. MMU**

La CPU tiene que acceder a la RAM primero para consultar la tabla de paginas y Despues para acceder a la memoria fisica, para cada lectura tenemos que hacer 2 accesos a memoria, Para optimizar esto, la MMU, hardware especifico de la CPU, se encarga de traducir las direcciones virtuales a fisicas, la MMU tiene un caché de la tabla de paginas llamado TLB (Translation Lookaside Buffer), La MMU tambien es la encargada de gestion que un proceso no acceda a memoria que no le pertenece,

#### **7.1.6. Acceso a memoria**

Cuando un proceso quiere acceder a una dirección de memoria, la CPU consulta la MMU, En la MMU la direccion se divide en dos partes, el numero de pagina y el offset dentro de la pagina,

- Page number - los bits mas significativos de la direccion virtual
- Page offset - los bits menos significativos de la direccion virtual

Ejemplo: En una pagina de 4KB (4096 bytes), el offset ocupa los 12 bits menos significativos, El resto de bits son los mas significativos y representan el numero de pagina,

## 7.2. Process

### 7.2.1. EPROCESS

Estructura que mantiene toda la información de un proceso en el kernel.

### 7.2.2. Threads

Representan las hebras de ejecución asociadas al proceso.

### 7.2.3. Handles

Son equivalentes a los *file descriptors* en Linux.

### 7.2.4. Memory

Gestión del espacio de direcciones y memoria asignada al proceso.

### 7.2.5. Modules

Lista de módulos y librerías cargadas en el proceso.

## 7.3. Process Creation (Kernel)

La creación de un proceso en Windows sigue una serie de pasos específicos a nivel del kernel.

### (1) Inicialización del espacio de direcciones

- **Modelo en Linux vs. Windows:** En Linux, los nuevos procesos se crean mediante la llamada `fork`, mientras que en Windows los procesos se inician completamente desde cero.
- **Mapeo de KUSER\_SHARED\_DATA:** Al crear un nuevo proceso, el kernel debe mapear memoria. Una de las primeras acciones es mapear la página de 4KB denominada `KUSER_SHARED_DATA`, la cual permite intercambiar información entre *user mode* y *kernel mode* sin necesidad de *syscalls*. En esta página se encuentran datos como el reloj del sistema y las rutas a directorios del sistema.

- **Mapeo del ejecutable:** Posteriormente, el kernel carga el ejecutable dentro del proceso. Concretamente, carga el código PE en la sección `.text`.
- **Mapeo de `ntdll.dll`:** El kernel carga el módulo `ntdll.dll`, el cual actúa como intermediario entre el *user-mode* y el kernel. Este módulo contiene las funciones necesarias para la comunicación con el kernel y cumple un rol similar al `ld.so` en Linux.
- **Asignación del PEB (Process Environment Block):** El PEB es una estructura de datos en *user mode* (aproximadamente 1–2 páginas de memoria) que contiene:
  - Variables de entorno.
  - Línea de comandos completa con la que se inició el proceso.
  - Directorio de trabajo.
  - Lista de módulos cargados (`Ldr`).
  - Punteros al `stack` y `heap`.
  - Dirección base de la imagen.

## (2) Creación del hilo inicial

- **Mapeo del `stack`:** Se asigna el espacio de pila necesario para el hilo principal.
- **Mapeo del TEB (Thread Environment Block):** El TEB es una estructura pequeña (2 páginas aprox.) que almacena información específica de cada hilo. Su función es equivalente al PEB, pero a nivel de hilo.
- **Inicialización del puntero de ejecución:** Finalmente, el puntero de ejecución se coloca en la función `ntdll.LdrInitializeThunk`, encargada de completar la carga dinámica del proceso.

## 7.4. Portable Executable (PE)

El formato *Portable Executable* (PE) es el equivalente en Windows al formato ELF en Linux. Se utiliza para representar ejecutables, bibliotecas dinámicas (DLL) y otros ficheros ejecutables en dicho sistema operativo.

### 7.4.1. Sections

Son el equivalente a los segmentos en un ELF. Las secciones indican cómo debe ubicarse la memoria dentro del proceso, definiendo distintas regiones con protecciones específicas (lectura, escritura, ejecución).

### 7.4.2. Imports

Contiene la lista de DLLs que son dependencias del ejecutable.

### 7.4.3. Exports

Si el archivo PE corresponde a una biblioteca compartida y exporta alguna funcionalidad, esta información se registra en la sección de exportaciones. A diferencia de Linux, donde las funciones suelen exportarse por defecto, en Windows deben exportarse explícitamente.

### 7.4.4. Relocations

Al habilitar ASLR (Address Space Layout Randomization), el ejecutable puede necesitar ser reubicado en memoria. Las direcciones a modificar en dicho proceso quedan registradas en esta sección.

### 7.4.5. AddressOfEntryPoint

Campo que indica la dirección de inicio de la ejecución del programa.

### 7.4.6. Subsystem

Especifica en qué entorno debe ejecutarse el binario: si será una aplicación de consola, una GUI, etc.

#### **7.4.7. Virtual Address**

Dirección relativa que tendrá la sección al cargarse en memoria. Siempre se calcula como:

$$\text{Dirección en memoria} = \text{ImageBase} + \text{Virtual Address} \quad (7.1)$$

#### **7.4.8. Raw Data**

Indica la dirección dentro del archivo PE donde se encuentra físicamente la sección en disco.



## 7.5. Secciones de un proceso

El espacio de direcciones de un proceso en memoria se organiza en diferentes secciones. A continuación, se enumeran las mismas desde las direcciones más altas hasta las más bajas:

### **Páginas del Kernel**

Ocupan la misma dirección en todos los procesos del sistema y corresponden al kernel. Como todas las páginas de kernel apuntan a los mismos *frames* de memoria física, no se desperdicia espacio cargando el kernel en cada proceso. El acceso se realiza únicamente mediante llamadas al sistema.

### **Stack (lectura y escritura)**

Estructura de almacenamiento *LIFO*, que crece hacia abajo (desde el final del espacio de direcciones del proceso hacia la sección BSS). Cada función tiene su propio *stack frame*, con variables locales y argumentos. El tamaño suele estar limitado entre 1 y 8 MB, según la arquitectura y el sistema operativo. Su gestión mediante el puntero de pila es muy rápida.

### **Heap (lectura y escritura)**

Espacio para memoria dinámica gestionada por llamadas como `malloc`. Crece hacia arriba, desde el final de la BSS hasta el final del espacio de direcciones. Su administración mediante el puntero de *heap* es más lenta que la del *stack*.

### **BSS y Data (lectura y escritura)**

Almacenan las variables globales y estáticas del programa. La sección **BSS** contiene las variables no inicializadas, mientras que la sección **Data** guarda las variables inicializadas.

### **Text (lectura y ejecución)**

Contiene el código ejecutable del programa, cargado desde el binario en disco. Generalmente está marcada como de solo lectura y ejecución.

## 7.6. DLL (Dynamic-Link Library)

En Windows, un proceso puede depender de múltiples librerías dinámicas (DLLs) para ejecutar ciertas funcionalidades. Estas DLLs se cargan en el espacio de direcciones del proceso y su interacción se realiza a través de estructuras específicas que Windows mantiene en memoria.

### 7.6.1. Carga de DLLs en un proceso

Cuando un proceso arranca, indica qué DLLs necesita y qué funciones de cada DLL va a utilizar. Windows realiza los siguientes pasos:

1. Verifica si la DLL ya está cargada en memoria; si no lo está, la carga en el espacio de direcciones del proceso.
2. Si ya existe en memoria, se reutiliza la misma copia en memoria física, pero cada proceso mantiene su propio mapeo en su espacio de direcciones virtuales.
3. Localiza dentro de la DLL la función solicitada por el proceso.
4. Proporciona al proceso la dirección en memoria de la función, de manera que cada llamada a esa función realmente salta a la dirección correspondiente dentro de la DLL cargada.

Cada DLL se carga como un módulo completo, incluyendo sus secciones `.text`, `.rdata`, `.data`, etc., mapeadas en memoria contigua a partir de la *base address* del módulo.

### 7.6.2. PEB y la lista de módulos

El **Process Environment Block (PEB)** es una estructura interna de Windows que contiene información sobre el proceso, incluyendo los módulos (DLLs) cargados.

Dentro del PEB, el campo:

`PEB->Ldr->InMemoryOrderModuleList`

es una lista enlazada que mantiene todos los módulos cargados en memoria, incluyendo la DLL principal del proceso y todas las DLLs dependientes. Cada entrada de esta lista contiene:

- La `base address` del módulo en memoria.

- El nombre del archivo de la DLL.
- Información sobre sus secciones.

Recorrer esta lista permite obtener todas las DLLs cargadas y sus direcciones en memoria.

### 7.6.3. Import Address Table (IAT)

Para que un proceso llame a funciones de una DLL, Windows utiliza la **Import Address Table (IAT)**. La IAT contiene punteros a las funciones importadas por el proceso. Durante la carga:

1. Windows localiza cada función solicitada dentro de la DLL correspondiente.
2. Escribe en la IAT la dirección en memoria de la función.

Cuando el proceso ejecuta una llamada a una función importada, en realidad está saltando a la dirección contenida en la IAT, que apunta a la DLL cargada en memoria.

### 7.6.4. Resumen

En conjunto, el PEB y la IAT permiten a un proceso interactuar de manera eficiente con sus DLLs:

- El PEB mantiene un registro de todos los módulos cargados y sus direcciones.
- La IAT traduce las llamadas a funciones de la DLL a direcciones concretas en memoria.

Esto garantiza que el proceso pueda utilizar código externo (DLLs) sin necesidad de incorporarlo estáticamente en su binario, manteniendo modularidad y eficiencia en el uso de memoria.

# Flujo de llamadas: User Mode vs Kernel Mode

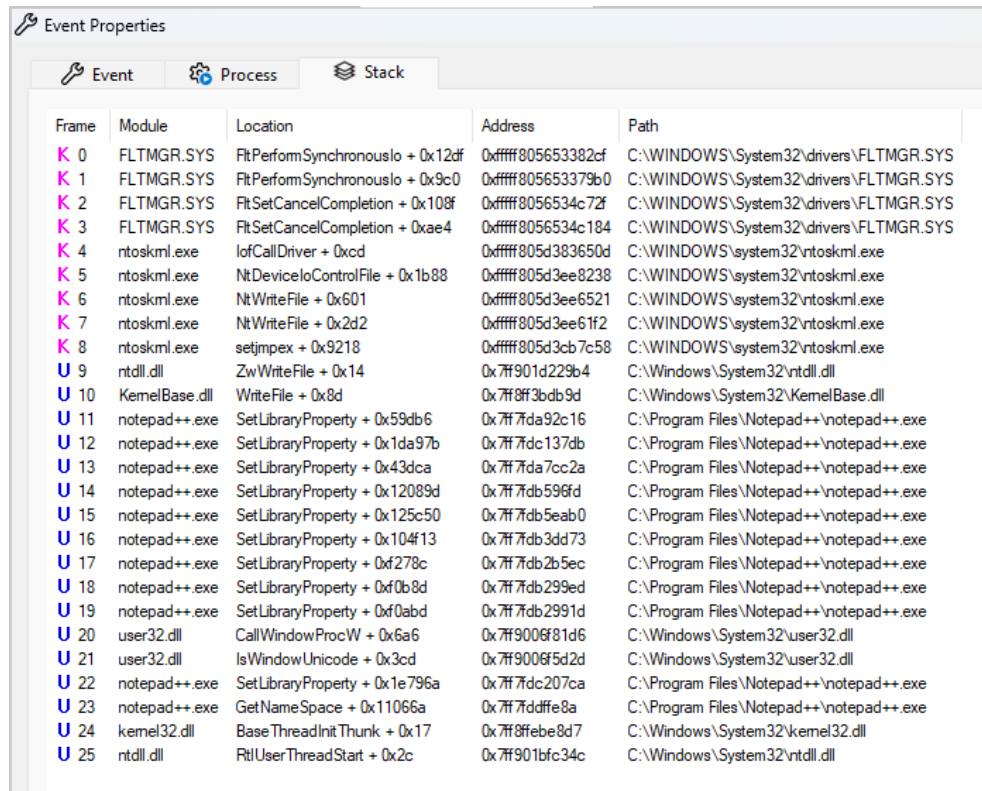
## 1. User Mode (U) - Aplicación y librerías de alto nivel

- notepad++.exe: SetLibraryProperty (operaciones internas de la aplicación)
- user32.dll: CallWindowProcW, IsWindowUnicode (interacción con interfaz Windows)
- KernelBase.dll: WriteFile + 0x8d (función de usuario de alto nivel)
- ntdll.dll: ZwWriteFile (interfaz hacia la syscall)

## 2. Kernel Mode (K) - Operaciones de bajo nivel y hardware

- FLTMRG.SYS: FltPerformSynchronousIo, FltSetCancelCompletion (filtro de sistema de archivos)
- ntoskrnl.exe: IoCallDriver, NtDeviceIoControlFile, NtWriteFile (operaciones sobre drivers y disco)

**Interpretación:** El flujo comienza en User Mode con la aplicación y librerías de alto nivel, y termina en Kernel Mode ejecutando operaciones de escritura en disco con privilegios elevados.



Frame	Module	Location	Address	Path
K 0	FLTMRG.SYS	FltPerformSynchronousIo + 0x12df	0xffff805653382cf	C:\WINDOWS\System32\drivers\FLTMRG.SYS
K 1	FLTMRG.SYS	FltPerformSynchronousIo + 0x9c0	0xffff805653379b0	C:\WINDOWS\System32\drivers\FLTMRG.SYS
K 2	FLTMRG.SYS	FltSetCancelCompletion + 0x108f	0xffff8056534c72f	C:\WINDOWS\System32\drivers\FLTMRG.SYS
K 3	FLTMRG.SYS	FltSetCancelCompletion + 0xae4	0xffff8056534c184	C:\WINDOWS\System32\drivers\FLTMRG.SYS
K 4	ntoskrnl.exe	IoCallDriver + 0xcd	0xffff805d383650d	C:\WINDOWS\system32\ntoskrnl.exe
K 5	ntoskrnl.exe	NtDeviceIoControlFile + 0x1b88	0xffff805d3ee8238	C:\WINDOWS\system32\ntoskrnl.exe
K 6	ntoskrnl.exe	NtWriteFile + 0x601	0xffff805d3ee6521	C:\WINDOWS\system32\ntoskrnl.exe
K 7	ntoskrnl.exe	NtWriteFile + 0x2d2	0xffff805d3ee61f2	C:\WINDOWS\system32\ntoskrnl.exe
K 8	ntoskrnl.exe	setjmpex + 0x9218	0xffff805d3cb7c58	C:\WINDOWS\system32\ntoskrnl.exe
U 9	ntdll.dll	ZwWriteFile + 0x14	0x7f901d229b4	C:\Windows\System32\ntdll.dll
U 10	KernelBase.dll	WriteFile + 0x8d	0x7f8ff3bdb9d	C:\Windows\System32\KernelBase.dll
U 11	notepad++.exe	SetLibraryProperty + 0x59db6	0x7f7da92c16	C:\Program Files\Notepad++\notepad++.exe
U 12	notepad++.exe	SetLibraryProperty + 0x1da97b	0x7f7dc137db	C:\Program Files\Notepad++\notepad++.exe
U 13	notepad++.exe	SetLibraryProperty + 0x43dca	0x7f7da7cc2a	C:\Program Files\Notepad++\notepad++.exe
U 14	notepad++.exe	SetLibraryProperty + 0x12089d	0x7f7db596fd	C:\Program Files\Notepad++\notepad++.exe
U 15	notepad++.exe	SetLibraryProperty + 0x125c50	0x7f7db5eab0	C:\Program Files\Notepad++\notepad++.exe
U 16	notepad++.exe	SetLibraryProperty + 0x104f13	0x7f7db3dd73	C:\Program Files\Notepad++\notepad++.exe
U 17	notepad++.exe	SetLibraryProperty + 0xf278c	0x7f7db2b5ec	C:\Program Files\Notepad++\notepad++.exe
U 18	notepad++.exe	SetLibraryProperty + 0xf0b8d	0x7f7db299ed	C:\Program Files\Notepad++\notepad++.exe
U 19	notepad++.exe	SetLibraryProperty + 0xf0abd	0x7f7db2991d	C:\Program Files\Notepad++\notepad++.exe
U 20	user32.dll	CallWindowProcW + 0x6a6	0x7f900f81d6	C:\Windows\System32\user32.dll
U 21	user32.dll	IsWindowUnicode + 0x3cd	0x7f900f5d2d	C:\Windows\System32\user32.dll
U 22	notepad++.exe	SetLibraryProperty + 0x1e796a	0x7f7dc207ca	C:\Program Files\Notepad++\notepad++.exe
U 23	notepad++.exe	GetNameSpace + 0x11066a	0x7f7ddffe8a	C:\Program Files\Notepad++\notepad++.exe
U 24	kernel32.dll	BaseThreadInitThunk + 0x17	0x7f8febe8d7	C:\Windows\System32\kernel32.dll
U 25	ntdll.dll	RtlUserThreadStart + 0x2c	0x7f901bfc34c	C:\Windows\System32\ntdll.dll

## 7.7. Win32 API

Nombre de la DLL	Tareas de la DLL
User32.dll	Esta biblioteca contiene funciones para crear ventanas, manejar mensajes y procesar la entrada del usuario.
Kernel32.dll	Esta biblioteca proporciona acceso a una variedad de servicios esenciales del sistema como la gestión de memoria, operaciones de E/S y la creación de procesos e hilos.
Gdi32.dll	Esta biblioteca contiene funciones para dibujar gráficos y mostrar texto.
Comdlg32.dll	Esta biblioteca proporciona diálogos comunes como los diálogos de abrir y guardar.
Advapi32.dll	Esta biblioteca proporciona funciones para trabajar con el registro de Windows y gestionar cuentas de usuario.

Cuadro 7.1: Descripción de las principales DLLs de la Win32 API

## 7.8. SO Security

7.8.1. ASLR (Address Space Layout Randomization)

7.8.2. DEP (Data Execution Prevention)

7.8.3. Control Flow Guard (CFG)

7.8.4. Code Integrity Guard (CIG)

7.8.5. Protected Process Light (PPL)

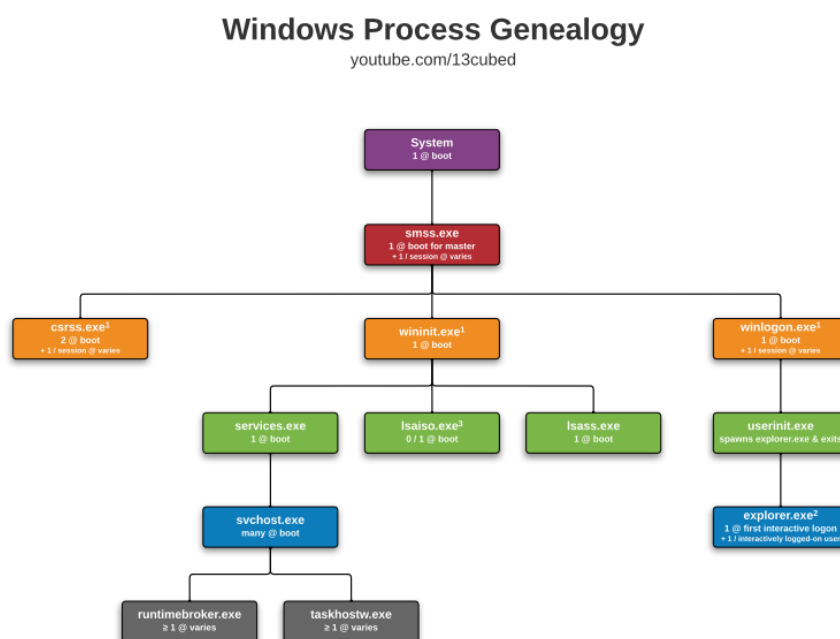
7.8.6. Virtualization-Based Security (VBS)

7.8.7. Kernel-mode code signing (KMCI)

## 7.9. Genealogía de Procesos en Windows

La genealogía de procesos en Windows representa la jerarquía de creación de procesos desde el arranque del sistema hasta el inicio de sesión del usuario. Comprender esta estructura es esencial para el análisis forense, la respuesta a incidentes y la detección de malware, ya que permite identificar comportamientos anómalos en los procesos del sistema.

A continuación se muestra la genealogía de procesos típica en Windows:



Fuente: youtube.com/13cubed

### Nivel 0: Proceso raíz

- **System:** Primer proceso de espacio de usuario iniciado por el kernel. Tiene un PID fijo (normalmente 4) y no tiene padre.

### Nivel 1: Session Manager

- **smss.exe (Session Manager Subsystem):** Primer proceso real del espacio de usuario. Se encarga de iniciar las sesiones del sistema, lanzar procesos críticos como `csrss.exe`, `wininit.exe` y `winlogon.exe`.

## Nivel 2: Procesos críticos del sistema

- **csrss.exe**: Client/Server Runtime Subsystem. Se encarga de funciones esenciales como la gestión de la consola y la creación de procesos. Existe una instancia por sesión.
- **wininit.exe**: Windows Initialization Process. Lanza procesos esenciales como **services.exe**, **lsaiso.exe** y **lsass.exe**.
- **winlogon.exe**: Gestiona la autenticación del usuario y se mantiene activo durante la sesión interactiva.

## Nivel 3: Procesos del sistema

- **services.exe**: Service Control Manager. Se encarga de iniciar y gestionar los servicios del sistema, incluyendo los alojados por **svchost.exe**.
- **lsaiso.exe**: Proceso de seguridad aislado que implementa funciones de cifrado y autenticación en versiones modernas de Windows (opcional).
- **lsass.exe**: Local Security Authority Subsystem Service. Encargado de políticas de seguridad, autenticación y gestión de credenciales.

## Nivel 4: Servicios alojados y utilidades del sistema

- **svchost.exe**: Proceso contenedor que aloja múltiples servicios del sistema. Existen muchas instancias según el grupo de servicios que aloje.
- **runtimebroker.exe** / **taskhostw.exe**: Procesos auxiliares para la ejecución de aplicaciones y tareas programadas.

## Procesos del usuario interactivo

- **userinit.exe**: Iniciado por **winlogon.exe** tras la autenticación. Lanza el shell principal del usuario y termina.
- **explorer.exe**: Shell gráfico de Windows que representa el escritorio, la barra de tareas y el menú de inicio. Es el proceso raíz del entorno del usuario.



## 7.10. Debugging flags

## 7.11. ABI

Calling Convention

- Pasar parámetros Por pila, de derecha a izquierda (el argumento de mas a la izquierda quedara en la direccion mas baja) - Limpieza de pila La función llamada (callee) limpia la pila - Registro de retorno EAX