Técnicas Anti-Debugging en Sistemas Operativos Windows

Trabajo final de master.

Máster Interuniversitario en Seguridad de las Tecnologías de la Información y de las Comunicaciones.

Autor: Edwin Roney Argueta Andara.

Seguridad en redes y aplicaciones distribuidas.
Director: Angel Elbaz Sanz.

Universidades: Universitat Oberta de Catalunya, Universitat Autònoma de Barcelona, Universitat Rovira y Virgili y Universitat de les Illes Balears

Junio de 2019.

C) Copyright

Dedicatoria

A Dios el divino creador, por darme la fuerza, los ánimos, la voluntad y la vida para concluirla, y por poner las personas que son de inspiración e imagen de superación: mis padres.

Felipe Argueta Gámez, mi padre, por ser un ejemplo a seguir, maestro y hombre; por siempre inspirarme a estudiar y nunca desistir de mis metas. Por el tiempo y esfuerzo dedicado cuando más lo he necesitado.

A mi madre Adela Andara Rodríguez por ser mi apoyo moral y mi motivación, por ser esa mujer que me llena de orgullo, y no existirá manera de devolverle tanto. Te amo.

Agradecimientos

A la estimada Lic. en lengua y literatura Sandra Elizabeth Ayala Moreno por su valiosa colaboración.

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Técnicas Anti-Debugging en Sistemas Operativos Windows*
Nombre del autor:	*Edwin Roney Argueta Andara*
Nombre del consultor/a:	*Edwin Roney Argueta Andara.*
Nombre del PRA:	*Nombre y dos apellidos*
Fecha de entrega (mm/aaaa):	JUNIO/2019
Titulación::	*Máster interuniversitario en Seguridad de las Tecnologías de la Información y de las Comunicaciones.*
Área del Trabajo Final:	*Seguridad en redes y aplicaciones distribuidas.*
Idioma del trabajo:	*Español*
Palabras clave	*Anti- Debugging, Windows, C++*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados i conclusiones del trabajo.*

En el presente documento se expone una investigación a cerca de las técnicas anti-debugging para el sistema operativo Windows que, se emplean normalmente para esconder rutinas o algoritmos contenidos en malware, y algunos software comerciales, asimismo se explica de una manera breve la forma para poder evadir dichas técnicas, consecuentemente para poder discernir los conceptos acá planteados; es necesario comprender algunos temas que se desarrollan a lo largo del documento.

La investigación, se desarrolla en capítulos donde cada uno de ellos forma parte de un conjunto de conocimientos necesarios para, en primer lugar; comprender los conceptos como el estado de un programa contenido en un archivo binario, su representación en opcodes visto a través de un debugger, y qué representan estos opcodes y segundo, implementar estos conocimientos en un programa escrito en C++ donde se desarrollarán las técnicas anti-debugging protegiendo un fragmento de código para posteriormente analizarlo bajo un debugger con la finalidad de demostrar con ejemplos prácticos como esquivar la técnica.

Abstract (in English, 250 words or less):

The present document exposes an investigation about antidebugging techniques for windows OS, that are usually used to hide routines or algorithms inside malware, and some commercial softwares, likewise it explains briefly the way to evade those techniques, consequently to help discern the concepts raised here; its necessary to comprehend some topics developed throughout this document.

The investigation, develops in chapters where each one of them is part of a set of knowledge needed to, in first place; understand the concepts like, the state of a program contained in a binary archive. Its representation in opcodes seen through a debugger, and what this opcodes represent. And second, implement this knowledge in a program written in C++, where anti-debugging techniques will be developed, protecting a fragment of code to later analyze it under a debugger with the purpose of demonstrating with practical examples how to avoid the technique.

Índice

Lista de figuras

1. Introducción

En el siguiente capítulo se definirá y establecerá la metodología que se empleará para desarrollar el trabajo de final de master.
Que tiene como tema "**Técnicas anti-debugging en sistemas operativos Windows**".
Este informe también pretende abordar la temática que implica el desarrollo de técnicas anti-debugging.
Se estudiará cual es el proceso y herramientas necesarias que incurren en el análisis del software. Y Para comprender de una manera más eficiente, se desarrollará cada una de estas técnicas en un código que se escribirá utilizando el lenguaje de programación de alto nivel C/C++. Se presentará la forma de evadir dichas técnicas y, comprender la finalidad para la que se programó ese software.

El desarrollo de este documento, incluye el estudio de otra áreas necesarias para comprender la temática a tratar, evidenciaremos cómo es que un debugger realiza su función en conjunto con el sistema operativo. Estudiaremos cómo funciona y cuáles son las diferentes formas para debuggear un archivo ejecutable. Finalmente, estudiaremos las diferentes instrucciones en lenguaje ensamblador para comprender y entrar de lleno a la temática del anti-debugging y la forma en que ésta se puede evadir dependiendo de la técnica implementada.

1.1. Justificación del trabajo y problemas a resolver.

Los motivos que nos llevaron a investigar las técnicas anti-debugging en sistemas operativos Windows, se centran en que en el área de la seguridad informática, poco se conoce de las funcionalidades que prevé este sistema para facilitar la inclusión de técnicas, funciones o métodos que hacen que un programa se comporte de manera diferente al momento de analizar su funcionamiento, o simplemente realice acciones diferentes a las programadas, cuando se esta analizando.

1.2. Contexto actual.

Gran parte de la comunidad en internet, como usuarios o instituciones que se sirven de este recurso, no están exentos al ataque de malware, ya sea de amenazas que llevan tiempo circulando por el mundo digital, atacando y destruyendo información, y otras que recién salen al mundo para provocar pérdidas millonarias. Si bien es cierto, contamos con una gran cantidad de alternativas de seguridad para prevenir estas amenazas. Grandes empresas dedicadas a la seguridad informática como ESET, Kasperky Labs, Panda Labs etc. se ven forzadas a lanzar nuevos productos anti-virus para la detección/neutralización de dichas amenazas. Estos productos se lanzan tomando como referencia el comportamiento y la finalidad del malware. Normalmente estas vacunas o neutralizaciones se encapsulan en firmas antivirus, que

posteriormente se instalan en un motor anti-virus que se encarga de detectarlos y limpiarlos del sistema, pero que pasa cuando una amenazada nueva es detectada y no se conoce su verdadera finalidad, es aquí cuando un analista de malware emplea sus conocimientos para determinar qué es lo qué el software malintencionado desea hacerle a un sistema informático, recurriendo al análisis estático y análisis dinámico de software. Estos análisis difieren en la profundidad del estudio realizado al programa en cuestión, mientras el análisis estático "***se centra en hacer una referencia y estudio de una amenaza sin tener que ejecutarla, como si tuviéramos que realizar una autopsia para conocer qué es lo que hace, o cuáles son las consecuencias que generará si llegase a infectar un sistema"***.(Pablo Ramos 2014) [1] .El análisis dinámico va mas allá, y nos "***puede dar ventaja para conocer el comportamiento de las amenazas y sus tendencias de propagación***" (Camilo Amaya, 2014) [2].
Cada unos de estos análisis dependerá de lo avanzado que sea el código que está examinando el experto. Un analista de malware es aquella persona o "***profesional que se encarga de detectar, analizar y combatir la presencia de códigos maliciosos***" (Lucas Paus, 2017) [3].

ESET enumera algunos métodos empleados por los malware para "***arruinar el día a un analista***", y entre ellos menciona técnicas anti-debugging, donde en muchas ocasiones es necesario estudiar profundamente una amenaza, y no basta con un análisis estático y es ahí donde "***los debuggers son la alternativa natural para el estudio del malware"*** (Matías Porolli, 2014). [4]

Un debuggers normalmente, se asocia al estudio de malware o virus informáticos, pero esto no necesariamente debe ser así, ya que también se emplean para corregir errores de programación, o comprender el funcionamiento de un software comercial para el cual no se posee el código fuente, pero si se cuenta con el archivo ejecutable.
Para el sistema operativo Windows disponemos, de una gran cantidad de debuggers, ya sean comerciales como IDA-PRO, y no comercialescon una enorme potencia, como OllyDbg, xDbg, Immunity Debugger, WinDbg (desarrollado por el equipo de Microsoft).
Es importante destacar que las técnicas anti-debugging nacen con el malware, pero también se implementan en software comercial para proteger algunos fragmentos de código o algoritmos que se utilizan para procesar información.

1.3. Objetivos del Trabajo

1) Comprender cómo y por qué se utilizan algunas técnicas para ofuscar un código ya sea un software comercial o un código malicioso.
2) Proporcionar a través de los ejemplos prácticos los conocimientos necesarios, para la utilización de un debugger.
3) Analizar las instrucciones y estructuras en lenguaje ensamblador.
4) Demostrar el manejo de las técnicas anti-debugging, empleando el lenguaje C/C++.
5) Demostrar adecuadamente el funcionamiento del código, evadiendo las técnicas.
6) Incentivar a la comunidad para el desarrollo de nuevas investigaciones.

1.4. Enfoque y metodología seguida.

El desarrollo de esta investigación se centra en demostrar el funcionamiento de técnicas Anti-debugging que algunos software y malware implementan en su código cuando se exponen a la lupa de un debugger. En otras palabras cuando un programa varía su comportamiento de diferentes formas al detectar que está siendo analizado por algún debugger con el objetivo de ocultar su verdadera finalidad.

El estudio se dividirá en las siguientes tareas:

1- Realizaremos una recopilación, análisis, síntesis de información sobre el proceso que implica el estudio de técnicas anti-debugging en sistemas operativos Windows.

2- Llevaremos a cabo una introducción del por qué y para qué se y dónde se emplean dichas técnicas anti-debugging.

3- Conoceremos cómo funciona un debugger de código, los registro que intervienen en el debuggeo de un programa y, las instrucciones en lenguaje ensamblador que genera un compilador.

4- Estudiaremos cómo es la interfaz que Windows ofrece a los debugger para manejar cada uno de los eventos que se generan al ejecutar una a una las instrucciones de un programa.

5- Programaremos un código de ejemplo y lo debuggearemos para comprender el funcionamiento del proceso de debuggeo.

6- Finalmente nos adentraremos en cada una de las técnicas que un malware y algunos software comerciales incluyen en su código para evitar que se pueda estudiar su comportamiento. Para nuestro ejemplo programaremos un sencillo código al que incluiremos cada una de las técnicas Anti-debugging y, veremos cómo éste se comporta cuando está siendo analizado en un debugger de código y cómo se puede evadir dicha técnica.

1.5. Planificación del Trabajo.

Fase	Tarea	Descripción
1	1.1	Realización de tareas de investigación para determinar cuales son las técnicas anti-debugging que se emplean en software destinado a la plataforma Windows.
1	1.2	Desarrollo de una investigación para determinar cual es el proceso que incurre el estudio de técnicas anti-debugging.

2	2.1	Desarrollo del marco teórico y resumir una historia acerca del malware para determinar el objetivo de aplicar técnicas anti-debugging a dichos malware.
2	2.2	Una vez que hayamos entendido el por qué y para qué de las técnicas, enumeraremos los tipos de análisis que se emplean para comprender el funcionamiento de un software en general.
3	3.1	Definiremos qué es un debugger y para qué se usa.
3	3.2	Detallaremos la arquitectura general de un ordenador y la estructura de un programa cargado en memoria. Identificaremos los registros en la ejecución del mismo.
4	4.1	Estudiaremos cómo es en general, la comunicación que existe entre el sistema operativo y un debugger.
4	4.2	Estudiaremos cómo es la interfaz, estructuras y las funciones que Windows ofrece a los debugger para manejar los eventos en el proceso de debuggeo.
5	5.1	Explicar con un ejemplo práctico, cómo funciona un debugger, cómo se realiza el proceso de debuggeo a un archivo ejecutable o a un proceso en ejecución.
6	6.1	Estudiaremos y definiremos las técnicas anti-debuggin más comunes. Para cada una de ellas daremos un ejemplo práctico mediante código que escribiremos en el lenguaje C/C++, ejemplificaremos cómo funciona y cual es la forma para evadirla.
7	7.1	Entrega de memoria final del TFM.

Tarea	Inicio	Duración(días)	Fin	PEC
1.1	7/3/2019	5	12/3/2019	
1.2	12/3/2019	5	17/3/2019	
2.1	17/3/2019	8	25/3/2019	
2.2	25/3/2019	4	29/3/2019	Entrega PEC 1
3.1	29/3/2019	5	4/4/2019	
3.2	4/4/2019	4	8/4/2019	
4.1	8/4/2019	4	12/4/2019	
4.2	12/4/2019	3	15/4/2019	Entrega PEC2
5.1	15/4/2019	10	25/4/2019	
6.1	25/4/2019	31	24/5/2019	Entrega PEC 3
7.1	25/5/2019	9	3/06/2019	Entrega memoria Final.

1.6. Breve sumario de productos obtenidos.

1.6.1. Aplicación en la cual se implementan todas la técnicas anti-debuggin que estudiaremos en este material.

1.6.2. Análisis de las técnicas y la forma de evadirlas.

1.6.3. Conclusiones.

1.7. Breve descripción de los otros capítulos de la memoria.

En el capítulo 2, se describen los programas que más explotan la característica anti-debugging para evitar o dificultar su forma de propagación o su funcionamiento real.

En el capítulo 3, se describen las herramientas que se utilizan para debuggear un software en la plataforma Windows y la relación intrínseca que este software de debuggeo tiene con el sistema operativo, los registros más importantes y los elementos que intervienen a la hora de debuggear un software.

Ya en el capítulo 4, abordaremos la temática más importante, que es enumerar las técnicas anti-debugging, implementarlas en nuestro código de ejemplo para comprender su funcionamiento.

1.8. Recursos.

Para desarrollar el laboratorio, fueron necesarios los siguientes recursos:

- Una computadora con un sistema operativo Windows 7 o superior.
- un compilador MinGW para desarrollo de aplicaciones para la plataforma Windows.
- Netbeans 8.2 como Entorno Integrado de Desarrollo (IDE).
- OllyDbg para debuggear nuestra aplicaciones.
- Para poder implementar algunas técnicas fue necesario el Visual Estudio comunity en su versión 2017 con el entorno de desarrollo para C++.

2. Malware y tipos de análisis de software.

Si bien es cierto, las técnicas anti-debuggin que veremos a lo largo de éste material son implementadas en diferentes software comerciales, pero fueron los malware que impulsaron estas técnicas para proteger su comportamiento y no delatar su verdadera finalidad en el sistema por lo que es conveniente hablar un poco acerca de estas amenazas informáticas.

Podemos definir malware (o programa malicioso) como un software creado con la finalidad de irrumpir y comprometer un sistema informático.
Uno de los primeros software que se registro a través de la historia como software malicioso fue “CREEPER” en 1979 considerado como el primer malware. Este

comenzó como un juego que consistía en controlar la mayor cantidad de memoria RAM del sistema, y privar a los demás competidores del uso de ella. A partir de este ingenioso juego comenzó a desarrollarse software con técnicas de auto propagación y con objetivos mas oscuros.
En la actualidad existen un sin numeró de malware con funcionalidades y objetivos diferentes, lo que hace posible clasificarlos por familia.

2.1. Familia de malware

Anteriormente vimos como de un juego se captaron ideas y conceptos que más tarde se convertirían en software malicioso con la intención de acceder los sistemas, y obtener información o simplemente destruirla. Diferentes firmas antivirus y equipos dedicados a la seguridad de la información, han optado por categorizar estos programas maliciosos con el fin de neutralizaros; basándose en su comportamiento, para hacer vacunas o parches de seguridad. En este capitulo expondremos una clasificación comúnmente usada, y de manera general puesto que los programas maliciosos o malware pueden ser clasificados según la forma de infección, el objetivo, métodos de propagación y comportamiento, o combinación de ellas, por lo que no existe una clasificación definitiva.

a) *Worms :*

La característica fundamental que diferencia un worm (gusano por su traducción al español) de los demás virus informáticos es su forma de propagación. Muchos de estos ficheros maliciosos realizan copias de si mismos en otras carpetas y sitios de red, los cuales a su vez se auto-replican. Comúnmente utilizan como transporte paquetes de red, protocolos como FTP y mensajes privados como ICQ,IRC. Dentro de esta categoría se podrían incluir gusanos de la clase virus; estos no se centran en replicas por la red, si no en la maquina local como principal objetivo.

b) Troyanos:

Se denomina troyano a los malware que se centran al bloquear, modificar y copiar datos de sus victimas como contraseñas. Su forma de infección se caracteriza por que se esconden en archivos ejecutables y no ejecutables ya que estos no poseen la capacidad de auto-replicarse.

c) Adware:

Esta categoría abarca un sin número de programas que generalmente no destruyen el ordenador, pero si son molestos, ya que su función y característica principal es descargar y mostrar anuncios publicitarios.

d) Ransomware:

Panda Security en su blog https://www.pandasecurity.com define esta amenaza como: “un software malicioso que al infectar nuestro equipo le da al ciberdelincuente la capacidad de bloquear un dispositivo desde una ubicación remota y encriptar nuestros archivos quitándonos el control de toda la información y datos almacenados. El virus lanza una ventana emergente en la que nos pide el pago de un rescate, dicho pago se hace generalmente en moneda virtual”.

e) Rootkit:

Se podría considerar una subcategoría dentro de los troyanos, pero para comprender mejor su objetivo y distinguirlos, los definiremos como una clase de rutina que oculta procesos o servicios que corren dentro del sistema. Por si solos los rootkit no representan una amenaza por eso siempre viene acompañados por algún tipo de worm.

f) Backdoor:

Son un tipo de malware que permite a un atacante obtener el control de un sistema infectado, normalmente estos malware abren un puerto por el intruso se conecta para controlar el ordenador a distancia.

2.2. Análisis de malware.

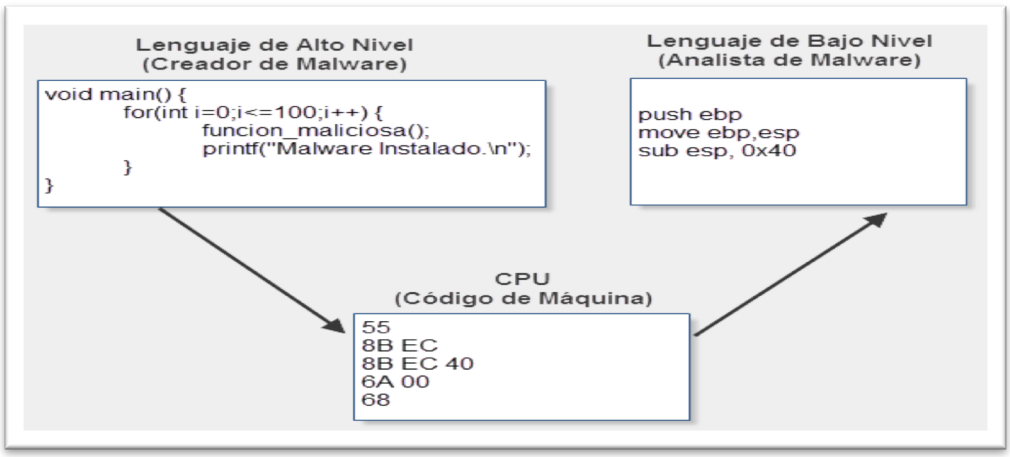
Si bien es cierto, estas categorías agrupan el malware por su comportamiento, objetivos; etc. pero para poder clasificarlos es necesario antes haberlos analizados. Este proceso consta de dos tipos de análisis: El análisis dinámico y análisis estático de malware.

2.2.1. Análisis estático.

(Pablo Ramos, 2014) [1] "define el análisis estático de malware como el estudio de una amenaza sin necesidad de ejecutarla, es realizar una autopsia para conocer que es lo que hace o cuales son las consecuencias que se generaran si se llegara a ejecutar en un sistema".

Este análisis consiste en estudiar mediante el desensamblado del malware o código binario con procesos de ingeniería inversa. Cabe destacar que un archivo binario a estudiar posee 3 niveles de abstracción como lo describe (Pablo Ramos, 2014) [1], Cuando hablamos de abstracción nos referimos al nivel al que se puede revertir un archivo ejecutable, para poder ver su código fuente. Por ejemplo el mas bajo nivel es lenguaje maquina, un nivel intermedio seria lenguaje ensamblador y un lenguaje de alto nivel serian los lenguajes de programación con palabras propias de nuestra vocabulario.

Ilustración 1:niveles de abstracción de un archivo ejecutable. Fuente: weblivesecurity.



2.2.2. Análisis dinámico.

A diferencia del análisis estático, el análisis dinámico permite conocer de una manera más rápida y efectiva qué acciones realiza un malware. Se trabaja en un sistema aislado o en un entorno virtual, con herramientas para realizar debugging de código, ejecutando las instrucciones paso a paso para comprender su comportamiento y finalidad. Algunos malware son tan sofisticados que implementan técnicas anti-debuggin y anti virtualización, para no comprometer el verdadero propósito de su creación. Mas adelante conoceremos y describiremos algunas de estas técnicas que hacen que un malware pase desapercibido o dificulte sus análisis; pero antes es necesario conocer una herramienta muy utilizada y necesaria para llevar a cabo el proceso de investigación.

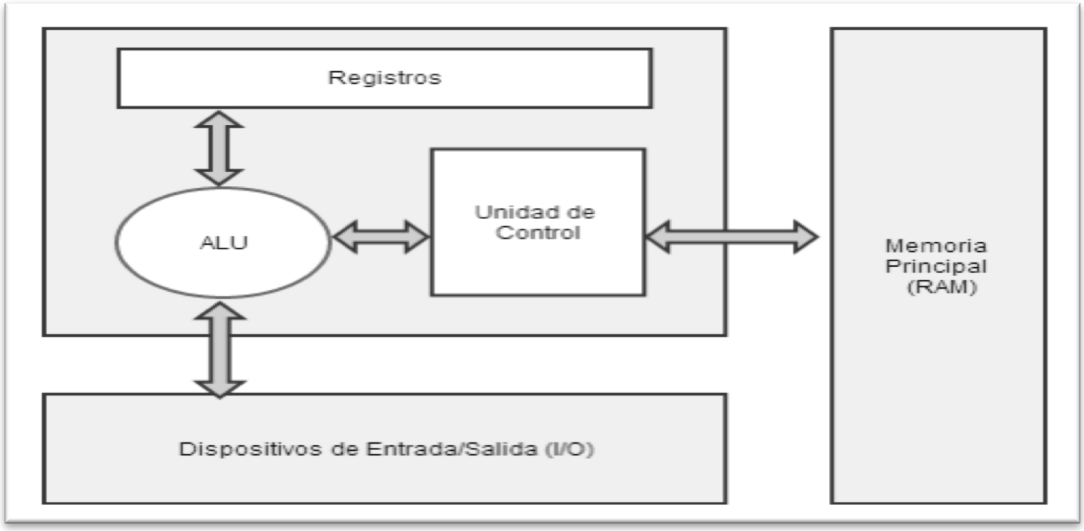
3. Debugger

Un debugger es una herramienta de software que se utiliza normalmente para identificar y corregir errores en programas informáticos.
Imaginemos que tenemos un programa ejecutable para Windows, pero éste tiene un error al ejecutar alguna de sus funciones, y no tenemos el código fuente para corregirlo; aquí es donde un debugger de código nos da la facilidad de poder entender en lenguaje ensamblador el funcionamiento del programa línea a línea a través de mensajes que se generan por eventos que el sistema operativo envía al debugger.
Para entender el funcionamiento interno de un debugger, primero es necesario conocer el funcionamiento del procesador y la memoria.

3.1. Arquitectura general de un ordenador.

En la figura 1 podemos ver la estructura general de un ordenador y los componentes del CPU que intervienen al momento de la ejecución de una instrucción. Cada uno de éstos componentes tiene como objetivo realizar tareas especificas.

Ilustración 2:estructura general de un ordenador; fuente: welivesecurity.



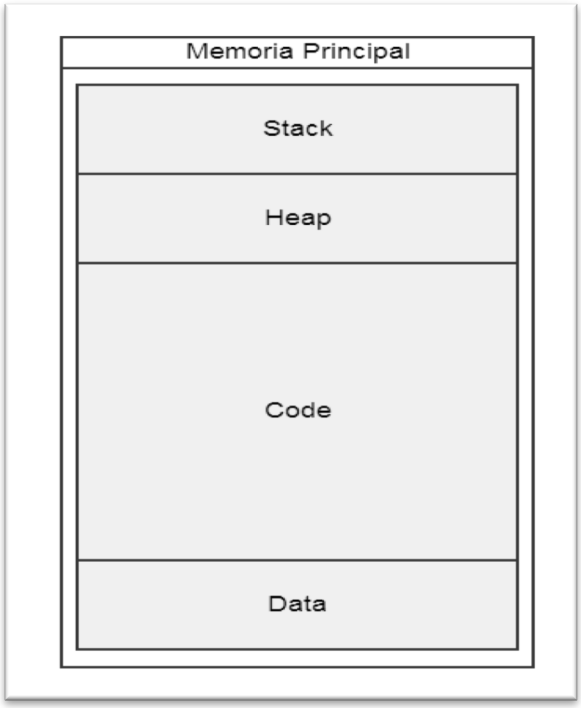
La unidad de control recibe desde la memoria RAM las instrucciones que debe ejecutar a través de un registro en particular. El “Instruction Pointer (IP)”, apunta a la siguiente

instrucción a ejecutar y el ALU (del inglés, Aritmetic Logic Unit) se encargada de ejecutar y almacenar los resultados en la memoria RAM o en los registros que definiremos mas adelante.

3.2. Regiones de memoria de un programa cargado en la memoria RAM.

En la ilustración 2 vemos como se puede dividir las secciones de un programa una vez que se ha cargado en la memoria del ordenador.

Ilustración 3:secciones de memoria de un programa; fuente: welivesecurity.



a) Data: Llamada sección de datos de un programa. Hace referencia a una dirección de memoria especifica, donde se contienen todas las variables estáticas, o que no cambian con la ejecución del programa. En esta sección también se encuentran las variables globales que están disponibles desde cualquier parte del programa.

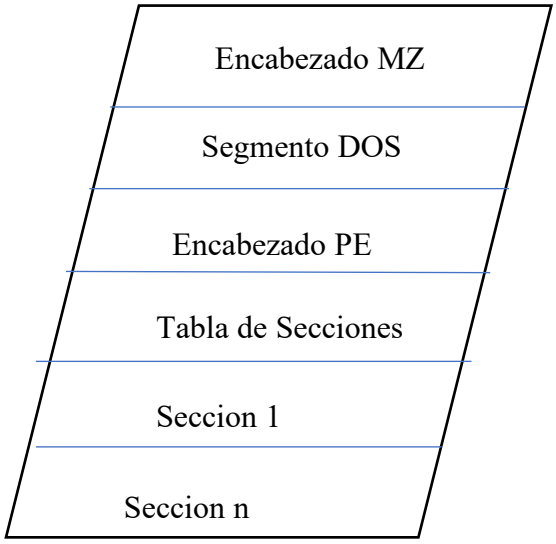
b) Code: Aloja todas las instrucciones del programa que se van a ejecutar.

c) Heap: Es una región de la memoria que guardar nuevos valores durante la ejecución del programa, y de igual forma los elimina cuando se dejen de utilizar. Básicamente es una memoria dinámica la cual varía durante la ejecución del programa.

d) Stack o pila: Esta se utiliza para alojar la variables locales, parámetros y retornos de una función; también almacena direcciones de retorno de una llamada a una función. Básicamente controla el flujo de ejecución del programa.

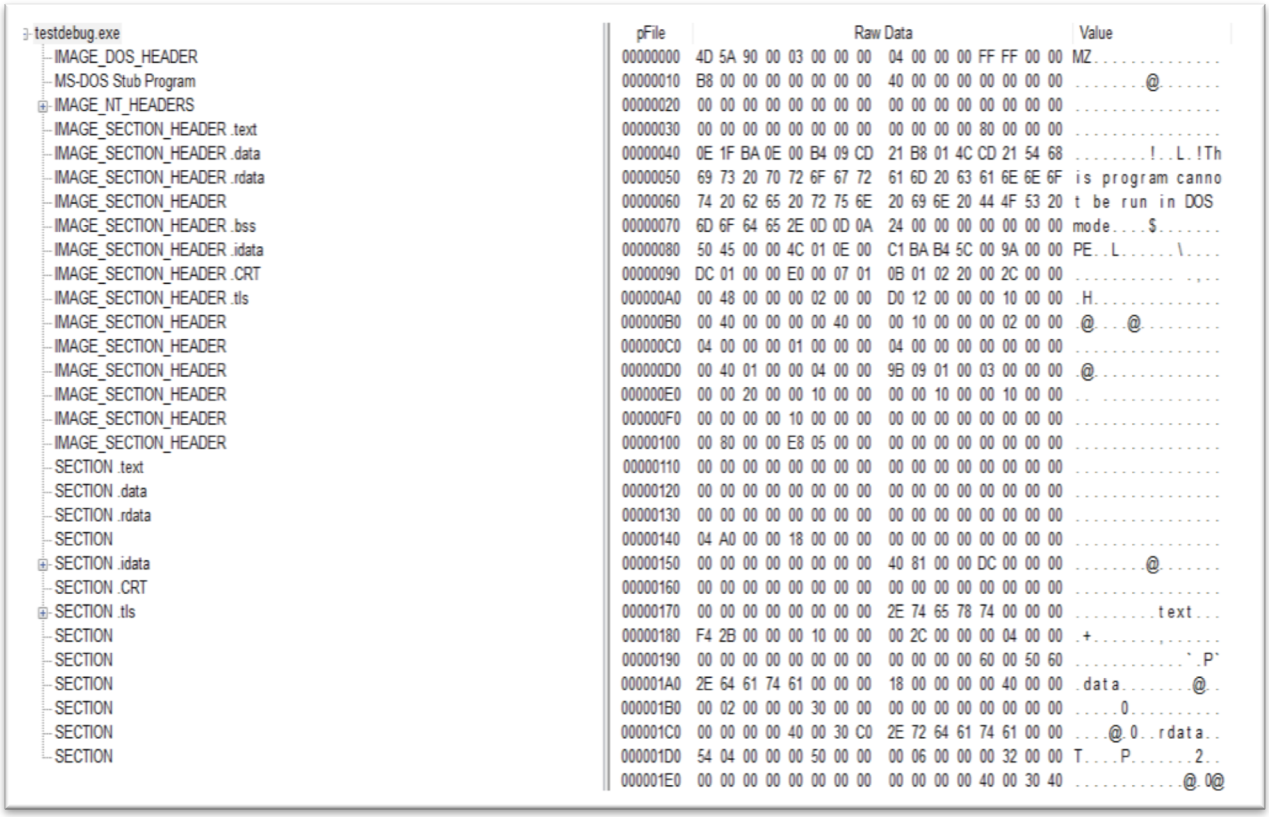
Es importante destacar que, la estructura de un programa en disco duro difiere mucho de la estructura en la memoria.
Cuando el archivo binario esta en el disco duro se le denomina PE (portable ejecutable) y su estructura se compone como vemos en la siguiente imagen tal como la describen en el blog de (TR31NORD,2012) [5].

Ilustración 4:Estructura de un programa en disco duro o Portable Ejecutable(PE)



a) Encabezado MZ: Cualquier archivo ejecutable incluidos también los DLL, comienza con estos dos bytes 0x4D5A, que le indican al *loader* de Windows identificarlo como archivo ejecutable.
b) Segmento DOS: Indica si es una aplicación valida que se puede ejecutar en modo DOS, siempre se encuentra acompañada por la leyenda "This program cannot be run in DOS mode".
c) Encabezado PE: Esta cabecera dos brinda información acerca del tipo de maquina donde debe ser ejecutada la aplicación.
d) Tabla de secciones: Es una tabla que contiene varias estructuras IMAGE_SECTION_HEADER que a su vez contiene información sobre las secciones del binario.
e) Sección 1: Es una sección opcional, siempre comienza en 1 hasta n y guardan información propia del ejecutable.

Ilustración 5: estructura PE de un programa, desensamblado con PEView.



3.3. Registros

Un registro (según Miquel Orenga y Albert Manonellas [6] en un documento de estudio para la clase de programación en ensamblador para UOC), es un espacio pequeño de almacenamiento para el CPU. Una de sus principales características es que puede ser accedido más rápidamente que cualquier otro dispositivo de almacenamiento de la computadora.

Los procesadores con arquitectura x86 (32 bits) cuentan con una serie de registros que pueden ser utilizados como almacenamiento temporal para variables, valores y demás información que se requiera durante la ejecución de instrucciones. Los registros se pueden dividir en 4 categorías diferentes: Registros generales, Segmentos de registro, Flag o bandera de estado y finalmente Instruction Pointer (IP) o puntero a la próxima instrucción a ejecutar.

Tabla 1. Clasificación de los registros para procesadores con arquitectura x86.

Registros Generales	Segmentos de Registros	Registros de Estado	IP
EAX (AX,AH,AL)	CS	EFLAGS	EIP
EBX (BX,BH,BL)	SS		
ECX (CX,CH,CL)	DS		
EDX (DX,DH,DL)	ES		
EBP (BP)	FS		
ESP (SP)	GS		
ESI (SI)			

Los registros generales almacenan datos o direcciones de memoria y son utilizados de manera intercambiable para lograr que se ejecuten las instrucciones del programa. Algunos de estos registros generales son utilizados para funciones específicas. Por ejemplo, para realizar multiplicaciones o divisiones se utilizan los registros EAX y EBX. Mas adelante describiremos cada uno de ellos.
Los Flag son registros de estado. Sus valores pueden ser 0 y 1, y sirven para controlar las operaciones del CPU. Entre los registros de estados mas importantes podemos encontrar:

a) ZF (Zero Flag). Se activa cuando el resultado de una operación es igual a 0.
b) CF (Carry flag). Se activa cuando el resultado de una operación es muy grande o pequeño para el operador destino.
c) SF (Sign Flag). Si el resultado de una operación es positivo es 0, de lo contrario es 1.
d) TF (Trap Flag). Este Flag se utiliza para realizar debugging de instrucciones, en caso de que este activo el procesador ejecutará una instrucción a la vez.

3.3.1. Registros generales

Nos centraremos en los registros para procesadores con arquitectura de 32 bits o x86. Oscar campos [7] en su blog los define de la siguiente manera.

1- EAX: También llamado acumulador. Su propósito es el de realizar cálculos; también se utiliza para almacenar los valores devueltos por llamadas a funciones.
2- EBX: Este registro no se ha diseñado para ningún fin en particular, y puede ser usado como almacenamiento extra.
3- EDX: Llamado registro de datos. Es básicamente una extensión del registro anterior EAX, este registro sirve de ayuda a EAX para almacenar información extra o procesar cálculos mas complejos.
4- ECX: Es conocido como registro de conteo; en otras palabras es un contador. Este registro es el que se usa en operaciones de bucle. Cabe destacar que el conteo de ciclos se hace de arriba hacia abajo.

5- ESI y EDI: Los bucles que procesan datos se apoyan en estos dos registros para una manipulación eficientes de los mismo. ESI es conocido como índice de origen, para operaciones de datos, y almacena la dirección inicial del flujo de datos de entrada. El registro EDI apunta a la dirección donde se almacena el resultado de una operación de datos o al índice de destino. Una forma sencilla de recordar esto es que ESI se usa para leer y EDI para escribir.
6- ESP y EBP: Puntero a pila y puntero base respectivamente. Son utilizados principalmente para administrar llamadas a funciones y operaciones de pila. Cuando una función es invocada los argumentos de la misma se almacenan en la pila seguidos por una dirección de retorno.
   El registro ESP apunta a la parte superior de la pila y por lo tanto a la dirección de retorno en cambio el EBP apunta a la parte inferior de la pila.
7- EIP: Es un registro especial llamado punto de instrucción, y apunta a la dirección de memoria de la instrucción que esta siendo ejecutada por el procesador.

Cada uno de los registros anteriores son accedidos y modificados por el debugger, para realizar la función de debugging según el propósito de su concepción. Pero no termina acá, el debugger por si mismo no es capaz de llevar a cabo todas estas funciones sin ayuda de una interfaz por parte del sistema operativo, y que interactúa directamente con el procesador. A continuación veremos a profundidad:

a) Cómo funciona un debugger y los modos en que este se puede ejecutar.

b) Cómo se ejecuta una instrucción y los elementos que intervienen,.

3.4. Microsoft Windows, interfaz de debugging.

Como hablamos anteriormente, cada sistema operativo tienen una interfaz que se encarga de proporcionar un manejador de eventos de debuggeo. Para efectos de este material nos centraremos en el sistema operativo Windows.

3.4.1. Eventos en un ambiente de Debuggeo.

Podríamos considerar un evento, como un mensaje o una señal, que se envía desde el sistema operativo al debugger para preguntar, como se debe proceder durante la ejecución de un programa o que acción debe seguirse.
A continuación nombraremos algunos de los eventos mas importantes que Windows pone a disposición de debugger.

- CREATE_PROCESS_DEBUG_EVENT. Ocurre antes de que se inicialice un nuevo proceso al que se está por debuggear, o en el momento en que un debugger se conecta a un proceso activo.
- EXIT_PROCESS_DEBUG_EVENT. Ocurre cuando el proceso que se está debuggeando sale o es finalizado.
- CREATE_THREAD_DEBUG_EVENT. Ocurre cuando el proceso que se está debuggeando crea un nuevo hilo.

- EXIT_THREAD_DEBUG_EVENT. Ocurre cuando un hilo en el proceso que se está debuggeando sale.
- LOAD_DLL_DEBUG_EVENT. Ocurre cuando el proceso que se está debuggeando carga una DLL (la carga puede ser explícita o implícitamente).
- UNLOAD_DLL_DEBUG_EVENT. ocurre cuando el proceso que se está debuggeando libera un DLL.
- EXCEPTION_DEBUG_EVENT. Ocurre cuando salta una excepción en el proceso que se está debuggeando.
- OUTPUT_DEBUG_STRING_DEBUG_EVENT. Ocurre cuando el proceso que se está debuggeando realiza una llamada a la función OutputDebugString .

Ilustración 6:estructura para manejar eventos de debuggeo en Windows.

```
typedef struct _DEBUG_EVENT {   /* de */
    DWORD dwDebugEventCode;
    DWORD dwProcessId;
    DWORD dwThreadId;
    union {
        EXCEPTION_DEBUG_INFO Exception;
        CREATE_THREAD_DEBUG_INFO CreateThread;
        CREATE_PROCESS_DEBUG_INFO CreateProcess;
        EXIT_THREAD_DEBUG_INFO ExitThread;
        EXIT_PROCESS_DEBUG_INFO ExitProcess;
        LOAD_DLL_DEBUG_INFO LoadDll;
        UNLOAD_DLL_DEBUG_INFO UnloadDll;
        OUTPUT_DEBUG_STRING_INFO DebugString;
    } u;
} DEBUG_EVENT, *LPDEBUG_EVENT;
```

Windows prevé de una interfaz muy completa, y una estructura bien definida para controlar los eventos debuggeo. Cuando hablamos de estructura nos referimos a tipos de datos bien definido empleados por los debugger e invocados a través de funciones como "**WaitForDebugEvent**" y "**ContinueDebugEvent**". Estas funciones esperan y suspenden respectivamente el proceso de debuggeo, pero para invocarlas es necesario crear un proceso.

Para crear un proceso a debuggear, el debugger llama en primer lugar a la función **CreateProcess.** Con el parámetro **fdwCreate** establecido en DEBUG_PROCESS o DEBUG_ONLY_THIS_PROCESS.

> DEBUG_PROCESS. Configura la relación padre/hijo; para que el debugger reciba eventos de debuggeo de un proceso que se está debuggeando, y cualquier otro proceso creado por ese proceso. En este caso, los procesos creados por el proceso padre se debuggearan automáticamente por el mismo debugger.
>
> El uso de DEBUG_ONLY_THIS_PROCESS. En cambio restringe el debuggeo al proceso inmediato que se está debuggeando solamente. Los procesos creados por el proceso que se está debuggeando son procesos normales que no tienen ninguna relación. O en otras palabras no forman parte del proceso principal, y no se incluyen en el debugging del hilo principal.

Ilustración 7:definición abreviada de la estructura para crear un proceso para su debugging.

```
BOOL CreateProcess(
   LPCTSTR lpszImageName,        /* address of image file name */
   LPCTSTR lpszCommandLine,      /* address of the command line */
   LPSECURITY_ATTRIBUTES lpsaProcess,  /* optional process attrs */
   LPSECURITY_ATTRIBUTES lpsaThread,   /* optional thread attrs */
   BOOL fInheritHandles,         /* new process inherits handles? */
   DWORD fdwCreate,                           /* creation flags */
   LPVOID lpvEnvironment,    /* address of optional environment */
   LPTSTR lpszCurDir,        /* address of new current directory */
   LPSTARTUPINFO lpsi,                 /* address of STARTUPINFO */
   LPPROCESS_INFORMATION lppi);  /* address of PROCESSINFORMATION */
```

CreateProcess incluye varios parámetros para establecer un entorno para el proceso que se quiere debuggear. veamos a continuación el mas importante:

> LPPROCESS_INFORMATION. Se utiliza para recibir información sobre el proceso que se está iniciando. Específicamente, consiste en el proceso, y los ID de subprocesos del proceso que se está debuggeando.
> Este parámetro ayuda al debugger a identificar los hilos o subprocesos para incluirlos en el debuggeo, si así se establece con el parámetro DEBUG_PROCESS.

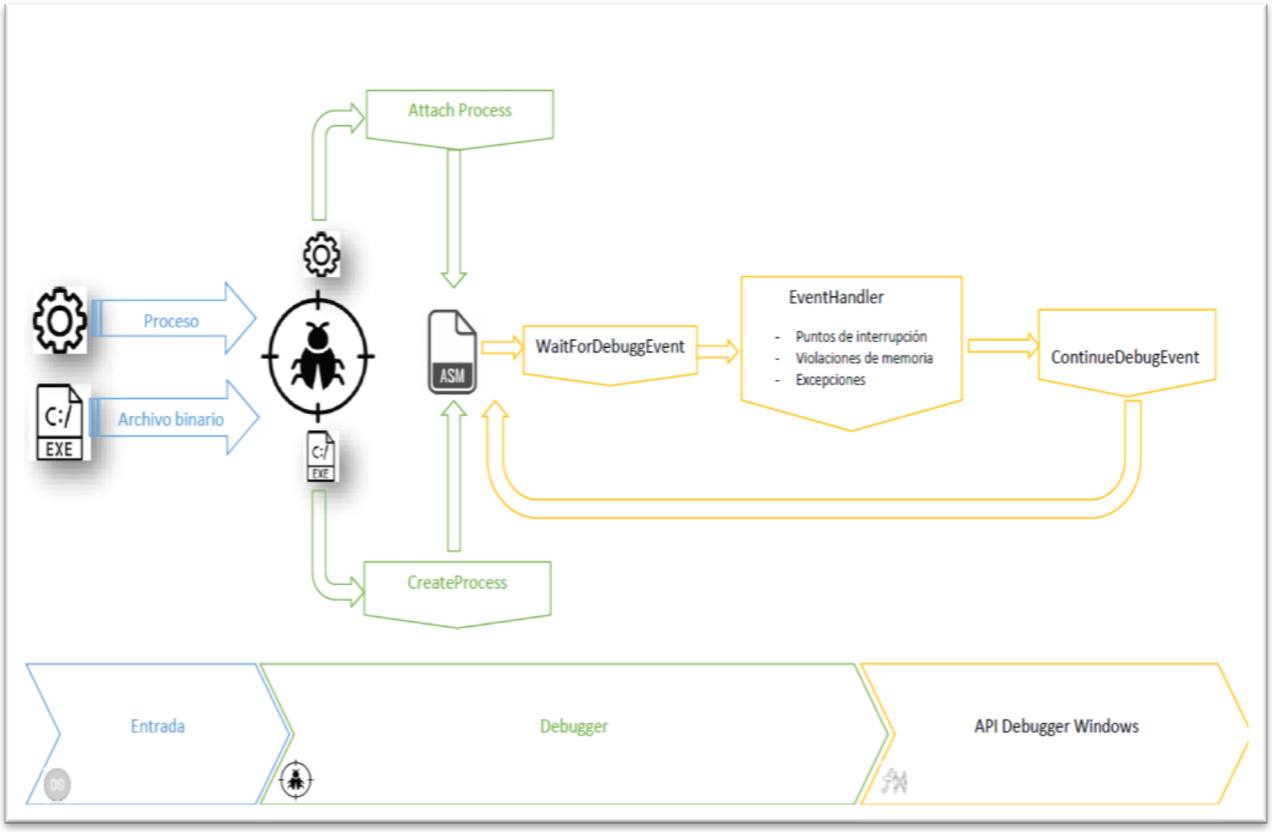
Como mencionábamos al principio las funciones "**WaitForDebugEvent**" y "**ContinueDebugEvent**" permanecen en un estado de espera cuando se coloca un byte al inicio de una instrucción del programa que se esta debuggeando; y a éste evento que detiene el proceso de ejecución se llama "breakpoint".(en español punto de interrupción), existen dos tipos de breakpoint: Breakpoint por software y Breakpoint por hardware:

> Breakpoint por software :son simples y bastante directos, donde el procesador se detiene cuando se intenta ejecutar un hilo o un fragmento de código. Normalmente se pueden establecer cualquier cantidad de breakpoint por software que se deseen al mismo tiempo, una de las característica mas importante de estos breakpoint es que solo se pueden dirigir al código, es decir, no podemos ver un estado de la memoria al realizar un evento de escritura o lectura.
>
> Los breakpoint por software funcionan de tal forma, que los debugger colocan una instrucción int 3 (opcode 0XCC) en el primer byte de la instrucción objetivo, y esto hace que se dispare una interrupción "3" cada vez que la ejecución se transfiera a la dirección en la que se estableció el breakpoint y cuando esto sucede, el debugger se "interrumpe" e intercambia el byte de código de operación 0xCC con el primer byte original de la instrucción, de modo que

puede continuar la ejecución sin llegar al mismo punto de interrupción de inmediato.

Breakpoint por hardware: son mucho más potentes y flexibles a diferencia de los breakpoint por software, los breakpoint por hardware, nos permiten colocar breakpoint de ruptura directamente en la memoria; o en otras palabras podríamos darnos cuenta cuando una instrucción intenta leer, escribir o ejecutar una acción de una dirección especifica. También podríamos poner breakpoint de ruptura en eventos donde se escriben en los dispositivos de E/S (entrada/salida).A diferencia de los breakpoint por software los breakpoint por hardware se limita a 4 simultáneamente.

Ilustración 8: ciclo de debuggeo.



3.5. Modos de debugging.

Cuando hablamos sobre modos de debuggin, nos referimos a la profundidad a la que se puede realizar una debuggeo, es decir capturando incluso los eventos propios del sistema operativo.
Los dos modos de debugging que un sistema operativo ofrece a un programador son: debugging en modo kernel y debugging en modo usuario.

Un entorno de debugging en modo kernel generalmente tiene dos equipos: el equipo host, y el equipo de destino. En este entorno el debugger se ejecuta en el

equipo host y el código que se está debuggeando se ejecuta en el equipo destino. El host y el destino están conectados por un cable de debuggeo. Normalmente este modo se emplea para hacer debugging a nivel de controladores (driver en ingles) o módulos propios del sistema operativo.

Debugging en modo usuario: en este modo el debugger se está ejecutando en el mismo sistema que el código que se está debuggeando comprimido en un solo ejecutable, y que está separado de otros ejecutables por el sistema operativo.

una de las principales diferencias que existen entre debugging en modo kernel y debuggin en modo usuario, es que el debugging en modo usuario proporciona acceso a la dirección virtual privada del proceso puesta en una sesión de debuggeo; ésta sesión que está limitada a ese proceso no puede sobrescribir ni alterar el espacio de datos de direcciones virtuales de otros procesos. En cambio el debugging en modo kernel proporciona acceso a otros controladores y procesos del kernel que necesitan acceso completo a múltiples recursos, además del espacio de direcciones del proceso original.

3.6. Instrucciones en ensamblador.

Anteriormente vimos que a partir de un código escrito en un lenguaje de alto nivel, podemos obtener un archivo binario listo para correr o ejecutarse en un sistema operativo especifico, ahora qué pasa cuando no disponemos del código fuente en lenguaje de alto nivel y en vez de ello tenemos un archivo ejecutable en formato binario, este archivo se puede desensamblar y obtener sus instrucciones en lenguaje ensamblador que es una visión del mismo programa a un nivel mas bajo y aun entendible al ser humano, por lo que es necesario conocer cómo se estructuran estas instrucciones y cuáles son sus principales funciones.

Para comenzar, veremos cómo se compone una instrucción en lenguaje ensamblador:

[etiqueta:] instrucción [destino[, fuente]] [;comentario]

donde destino y fuente representan los operandos de la instrucción y los elementos entre [ ] son opcionales. Por lo tanto, el único elemento imprescindible es el nombre de la instrucción. Por ejemplo.

Etiqueta1:
mov rax, 0 ; "Esta instrucción mueve 0 al resgitro rax"

Ahora que sabemos como es la estructura de una instrucción vamos a definir el juego de instrucciones más importantes en el lenguaje ensamblador.

1- MOV, MOVS, MOVSW:
Operaciones de transferencia que son utilizadas para mover el contenido de los operando. Su propósito es la transferencia de datos entre celdas de memoria, registros y acumulador.

2- LODS,LDS,LEA,LES:

Instrucciones de carga. Son instrucciones especificas de los registros y se utilizan para cargar algún byte o cadena.

3- POP,PUSH,POPF,PUSHF:
Instrucciones de pila y control de flujo de pila. En otras palabras sirven para poner u obtener datos de ella.

4- AND,NEG,NOT,OR,TEST,XOR:
Son instrucciones lógicas, y se utilizan para hacer operaciones lógicas entre dos operandos.

5- ADC,ADD,DIVIDIV,MULSBB,SUB:
Son instrucciones aritméticas. Se utilizan para realizar operaciones aritméticas entre dos operandos.

6- JMP:
Instrucción de salto. Se utiliza para transferir el flujo del proceso al operando indicado.

7- DEC,INC:
Instrucciones de conteo. Se utilizan para hacer un decremento o incremento en el contenido de los contadores.

8- CALL:
Llama a la subrutina que se encuentra en la dirección de memoria indicada por la etiqueta. Guarda en la pila la dirección de memoria de la instrucción que sigue en secuencia la instrucción CALL, y permite el retorno desde la subrutina con la instrucción RET; a continuación carga en el RIP (instruction pointer) la dirección de memoria donde está la etiqueta especificada en la instrucción, y transfiere el control a la subrutina.

3.7. debuggeando un programa.

Ahora que ya conocemos cómo es la estructura de un programa visto a través de lenguaje en ensamblador, y conocemos cuales son las diferentes instrucciones que podrían generarse con la compilación de un programa en alto nivel; nos adentraremos un poco mas al tema del debugging de aplicaciones con este sencillo ejemplo, para el cual hemos utilizado las siguientes herramientas que nos ayudará a crear nuestros ejecutables para una arquitectura de 32 bits:

Tabla 2. Herramientas necesarias para crear los ejecutables para el laboratorio.

Herramienta	Descripción	Configuración
Netbeans 8.2	Es un entorno de desarrollo integrado (IDE) que no ayudara a organizar y corregir errores	Dentro del IDE se configuró MinGw para generar ejecutables para la arquitectura de 32 bytes.

	semánticos y sintácticos de nuestro código.	
MinGw ((Minimalist GNU for Windows).)	Para compilar nuestros programas hemos elegido un compilador de la familia GNU-LINUX	Tras descargar e instalar el compilador este realizara la configuración por defecto.
OllyDbg	Es un debugger de código abierto	Para poder realizar el proceso de debuggeo en necesario correr el OllyDbg en modo privilegiado.

Tenemos claro que un debugger es un bucle sin fin que está a la espera de eventos. Cuando un evento es lanzado, el bucle se rompe; y se invoca a un manejador (Handler en inglés) cualificado, para procesarlo. Una vez que esto sucede, el debugger se detiene, y queda a la espera de cómo debe continuar. Entre los eventos que un debugger debe atrapar están:

a) Alcance de un punto de ruptura (breakpoints).
b) Violaciones de acceso a memoria (segmentation fault en ingles).
c) Excepciones generadas por el programa que se esta debuggeando.

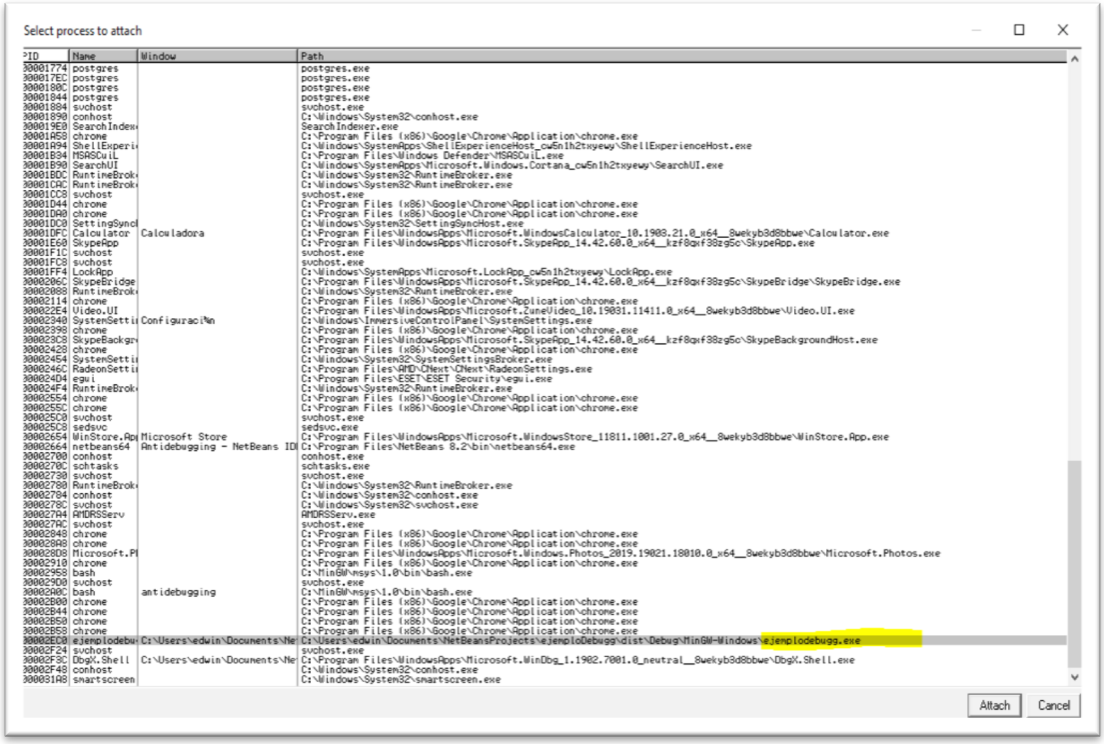
Cada sistema operativo implementa su propia interfaz para enviar estos eventos, algunos incluso pueden atrapar eventos de creación de subprocesos e hilos ; y la carga de librerías dinámicas en tiempo de ejecución.

En este primer ejemplo nuestro código tiene la función principal, la cual llama a otra para mostrar un mensaje en pantalla, usando la API de Windows.

```
/*
 * File:   main.cpp
 * Author: edwin
 *
 * Created on 01 de mayo de 2019, 10:02 AM
 */
#include <cstdlib>
#include <windows.h>
using namespace std;


void mensaje(void);

int main(int argc, char** argv) {
    mensaje();
    return 0;
}

void mensaje() {

    int msj=MessageBox(NULL,"Hola, este es un mensaje desde una
funcion","Informacion",MB_OK);//usamos la API de windows para mostra un mensaje
}
```

Para debuggear nuestro programa utilizaremos OllyDbg. Un debugger que ofrece muchas herramientas a la hora de debuggear programas.
Existen dos formas para poder debuggear un software.
La primera, es abriendo el debugger y seleccionando en el menú File, Open y seleccionar el archivo ejecutable a debuggear.
La segunda, es debuggear nuestro programa a través de “attach”, que se encuentra en el menú File y Attach. Para poder debuggear nuestro programa por este método, debemos primero, ejecutarlo luego ir al OllyDbg y presionar Attach, inmediatamente se nos abrirá una ventana con una lista de procesos o programas en ejecución, de los cuales seleccionaremos el nombre del programa, para nuestro caso sería ejemplodebugg.exe.

Ilustración 9:ventana para seleccionar un proceso a debuggear.



Para nuestro ejemplo lo haremos desde el menú open, al finalizar de cargar nuestro programa, OllyDbg nos mostrara una interfaz como la siguiente.

Ilustración 10:Interfaz de usuario de OllyDbg.

La pantalla de desensamblado es una ventana que contienen todo el código en ensamblador de nuestro programa. La ventana de los registros lista tanto registros como flags que el procesador necesita a la hora de debuggear un programa. La pila o stack contiene los datos y variables que se están usando, en la ventada dump se muestran los elementos contenidos en la memoria del ordenador. En la ventana de desensamblado podemos ver una a una las líneas de código de nuestro programa en lenguaje ensamblador y es aquí justamente donde veremos cada una de las llamadas a las API de Windows, funciones, librerías externas etc.

4. Técnicas anti-debugging.

En este capítulo estudiaremos algunas de las técnicas anti-debuggin usadas en aplicaciones donde se quiere proteger el funcionamiento o el código implementado en un software. Las técnicas basadas en la API que el sistema operativo brinda para identificar que un proceso está bajo la lupa de un debugger, y como hablábamos anteriormente es posible acceder a estas funciones propias del sistema operativo gracias a un conjunto de estructuras, y definiciones contenidas en el API de debugging, que para este caso trabajaremos bajo el sistema operativo Windows. De igual manera conoceremos las técnicas-anti debugging basadas en hardware que son eventos que producen excepciones para los cuales Windows crea una estructura en relación al contexto para la cual se generó la excepción, y se lo pasa al controlador de excepciones para su manejo. La estructura contendrá los valores de los registros generales, flags y registros de control y para el caso de que haya un debugger, se le transferirá la excepción con los puntos de interrupción por hardware.

4.1. Métodos anti-debugging basados en la API de Windows.

Muchas de estas técnicas son las más comunes y/o las más usadas; primero por la sencillez de su implementación ya que pueden ser consultadas en unas pocas instrucciones y segunda por que suelen pasar desapercibidas. Muchas de estas funciones utilizan la estructura PEB (Process Environment Block, por sus siglas en ingles), esta estructura contiene ciertos parámetros de ejecución asociados aun proceso. Entre los parámetros se pueden mencionar las variables de entorno, módulos cargados por el proceso, direcciones en memoria e información sobre debugging.

A continuación presentamos una lista de las técnicas que según Oleg Kulchytsky [8], Peter Ferrie [9] Machael Sikorki [10], Jose Fernández [11] son las más utilizadas bajo el API del sistema operativo. Es importante recalcar que para los ejemplo que estamos a punto de desarrollar se implementaron bajo para arquitectura con instrucciones para 32 bits, ya que el juego de instrucciones x64 bits difieren bastante del de x32 bits y algunas instrucciones no son las mismas.

Ilustración 11: Estructura PEB presentes en la ejecución de un proceso.

```
typedef struct _PEB {
  BYTE                          Reserved1[2];
  BYTE                          BeingDebugged;
  BYTE                          Reserved2[1];
  PVOID                         Reserved3[2];
  PPEB_LDR_DATA                 Ldr;
  PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
  BYTE                          Reserved4[104];
  PVOID                         Reserved5[52];
  PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
  BYTE                          Reserved6[128];
  PVOID                         Reserved7[1];
  ULONG                         SessionId;
} PEB, *PPEB;
```

4.1.1. Función **IsDebuggerPresent**.

Es una función de la librería **kernell32**.***dll***, el uso de esta función consiste en que al ser llamada en un bloque de código, ésta retornará un valor distinto de 0 cuando el proceso está siendo debuggeado, en caso contrario el valor devuelto es 0. Explicada también por Joshua Tully [12]), utilizando ASM en el código fuente.

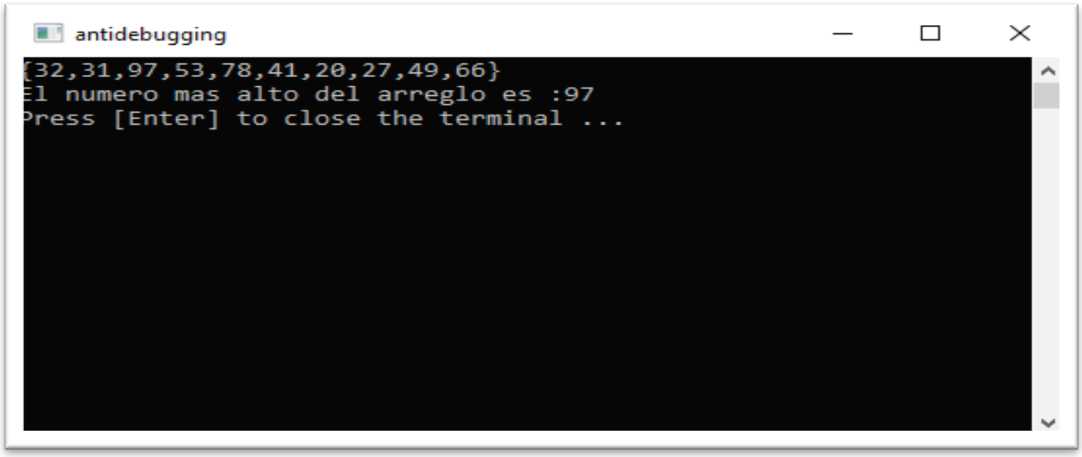
IsDebuggerPresent: lo que realmente hace es consultar el segundo parámetro de la estructura PEB que es un flag llamado ***BeingDebugged*** el cual se activará si el proceso está siendo debuggeado.

4.1.1.1. Un analista de seguridad puede evadir ***IsDebuggerPresent*** de la siguiente manera.

Supongamos que tenemos el siguiente programa, con una protección anti-debugging ***IsDebuggerPresent,*** la función está protegiendo un algoritmo que determina cuál es el número más alto en una secuencia de números generados al azar.

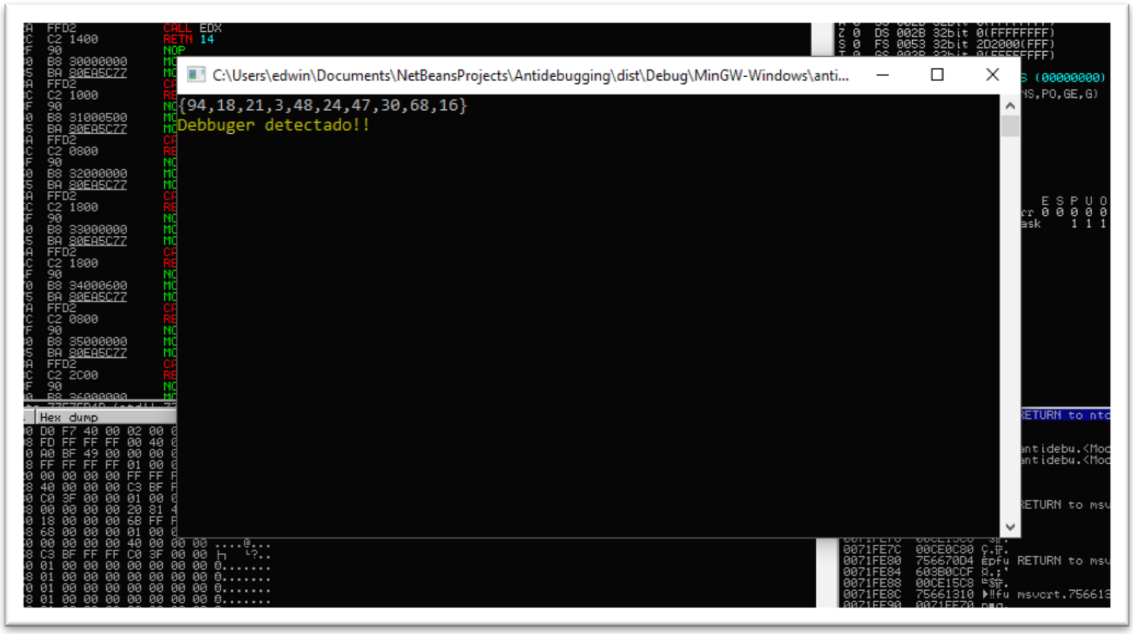
```
/*
 * File:   main.cpp
 * Author: edwin
 * Created on 01 de mayo de 2019, 10:02 AM
 */
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <cstdlib>
#include <windows.h>
#include  <winbase.h>//librería necesaria para pode invocar la función
IsDebuggerPresent()
using namespace std;
int* generar_arreglo() {//generamos e imprimimos un arreglos con 10 números
    static int r[10];
    srand((unsigned) time(NULL));
    cout << "{";
    for (int i = 0; i < 10; ++i) {
        r[i] = rand() % (101 - 1);
        if (i == 9) {
            cout << r[i] << "}" << endl;
        } else {
            cout << r[i] << ",";
        }
    }
    return r;
}
int main(int argc, char** argv) {
    int*p;
    p = generar_arreglo();
    int highNum = 0;
    int m;
    if (IsDebuggerPresent()) {//cuando el código se ejecute bajo un  debugger no se
                        // lanzara  el algoritmo para identificar el valor mas alto
        cout << "Debbuger detectado!!";
    } else {
        cout << "El numero mas alto del arreglo es :";
        for (m = 0; m < 10; m++) {
            int valor = ((p[m]));
            if (valor > highNum)
                highNum = (valor);
        }
        cout << highNum << endl;
    }
}
```

Ilustración 12:ejecución normal del programa ejemplo.



Como podemos ver la ejecución del programa en condiciones normales, nos da como salida el número más alto del arreglo.
Ahora, realizaremos la misma prueba para el mismo programa, pero esta vez lo ejecutaremos bajo un debugger.
Para nuestro ejemplo utilizaremos OllyDbg.

Ilustración 13: Ejecución bajo un debugger del programa ejemplo.



Vemos que al poner el programa bajo el debugger éste no retorna la salida esperada. Una forma en que un analista de código o un programador puede utilizar para evadir esta protección, una vez cargado el programa en el debugger (nosotros usaremos OllyDbg), buscaremos la función **IsdebuggerPresent** dentro de todos los módulos que hacen referencia a la API de Windows, para ello usamos "Ctrl+n" en Ollydbg (ver Ilustración 14). Luego damos click en "find references" para obtener un listado donde esta siendo instanciada dicha API (ver Ilustración 15) y colocamos un punto de interrupción en la primera llamada.

Ilustración 14:buscando las llamadas a la API IsDebuggerPresent.

```
00400308          Analyser  <STRUCT IMAGE_SECTION_HEADER>
00400330          Analyser  <STRUCT IMAGE_SECTION_HEADER>
00400358          Analyser  <STRUCT IMAGE_SECTION_HEADER>
00400380          Analyser  <STRUCT IMAGE_SECTION_HEADER>
004003A8          Analyser  <STRUCT IMAGE_SECTION_HEADER>
004003D0          Analyser  <STRUCT IMAGE_SECTION_HEADER>
00401250 .text    Export    <ModuleEntryPoint>
004E809C .bss     Analyser  <TLSindex>
004E921C .idata   Import    &KERNEL32.CloseHandle
004E9220 .idata   Import    &KERNEL32.CreateSemaphoreW
004E9224 .idata   Import    &KERNEL32.DeleteCriticalSection
004E9228 .idata   Import    &KERNEL32.EnterCriticalSection
004E922C .idata   Import    &KERNEL32.ExitProcess
004E9230 .idata   Import    &KERNEL32.FindClose
004E9234 .idata   Import    &KERNEL32.FindFirstFileA
004E9238 .idata   Import    &KERNEL32.FindNextFileA
004E923C .idata   Import    &KERNEL32.FreeLibrary
004E9240 .idata   Import    &KERNEL32.GetCommandLineA
004E9244 .idata   Import    &KERNEL32.GetCurrentThreadId
004E9248 .idata   Import    &KERNEL32.GetLastError
004E924C .idata   Import    &KERNEL32.GetModuleHandleA
004E9250 .idata   Import    &KERNEL32.GetProcAddress
004E9254 .idata   Import    &KERNEL32.InitializeCriticalSection
004E9258 .idata   Import    &KERNEL32.InterlockedDecrement
004E925C .idata   Import    &KERNEL32.InterlockedExchange
004E9260 .idata   Import    &KERNEL32.InterlockedIncrement
004E9264 .idata   Import    &KERNEL32.IsDBCSLeadByteEx
004E9268 .idata   Import    &KERNEL32.IsDebuggerPresent
004E926C .idata   Import    &KERNEL32.LeaveCriticalSection
004E9270 .idata   Import    &KERNEL32.LoadLibraryA
004E9274 .idata   Import    &KERNEL32.MultiByteToWideChar
004E9278 .idata   Import    &KERNEL32.ReleaseSemaphore
004E927C .idata   Import    &KERNEL32.SetLastError
004E9280 .idata   Import    &KERNEL32.SetUnhandledExceptionFilter
004E9284 .idata   Import    &KERNEL32.Sleep
004E9288 .idata   Import    &KERNEL32.TlsAlloc
004E928C .idata   Import    &KERNEL32.TlsFree
004E9290 .idata   Import    &KERNEL32.TlsGetValue
004E9294 .idata   Import    &KERNEL32.TlsSetValue
004E9298 .idata   Import    &KERNEL32.VirtualProtect
004E929C .idata   Import    &KERNEL32.VirtualQuery
004E92A0 .idata   Import    &KERNEL32.WaitForSingleObject
004E92A4 .idata   Import    &KERNEL32.WideCharToMultiByte
004E92AC .idata   Import    &msvcrt._fdopen
004E92B0 .idata   Import    &msvcrt._fstat
004E92B4 .idata   Import    &msvcrt._lseek
004E92B8 .idata   Import    &msvcrt._read
004E92BC .idata   Import    &msvcrt._strdup
```

Ilustración 15:en la lista de llamadas colocamos un punto de interrupción.

```
References to <&KERNEL32.IsDebuggerPresent>
Address   Command                                          Comments
0040156E  CALL <JMP.&KERNEL32.IsDebuggerPresent>
0041A780  JMP DWORD PTR DS:[<&KERNEL32.IsDebugger
Found 2 references
```

Existen muchas formas para saltarnos esta protección, en este ejemplo usaremos el cambio de instrucción.
Después de la llamada a la API **IsDebuggerPresent**(ver Ilustración 16) tenemos una comparación TEST del registro EAX.
Con TEST comprobamos si el valor de EAX es igual a 0, en cuyo caso el flag ZF se pondría a 1 y se ejecutaría el salto 'JE SHORT' a la posición 00401592. Si ejecutamos el código paso a paso (F7), podemos ver como EAX tiene el valor 1 en la comparación y el salto no se produce (ZF = 0), continuando con la ejecución de código secuencial.

Ilustración 16: secciones de código en modo debugg y ejecución normal.

```
00401573  .  85C0          TEST EAX,EAX
00401575  .  0F95C0        SETNE AL
00401578  .  84C0          TEST AL,AL
0040157A  v  74 16         JE SHORT 00401592
0040157C  .  C74424 04 6B  MOV DWORD PTR SS:[ESP+4],OFFSET 004A406  ASCII "Debbuger detectado!!"
00401584  .  C70424 80374  MOV DWORD PTR SS:[ESP],OFFSET 004A3780
0040158B  .  E8 706B0900   CALL 00498100
00401590  .v EB 6D         JMP SHORT 004015FF
00401592  >  C74424 04 80  MOV DWORD PTR SS:[ESP+4],OFFSET 004A408  ASCII "El numero mas alto del arreglo es :"
0040159A  .  C70424 80374  MOV DWORD PTR SS:[ESP],OFFSET 004A3780
004015A1  .  E8 5A6B0900   CALL 00498100
```

cuando no se produce el salto o se detecta un debugger

Ejecución normal del programa.

Ahora para conseguir que el salto se realice, basta con cambiar la instrucción JE SHORT y hacer que muestre el segundo mensaje saltando así esta protección, por ejemplo, por un salto JNZ, que se produce si ZF = 0, es decir, si el resultado de la comparación anterior es distinta a 0.

Ilustración 17:registros durante la ejecución del programa ejemplo.

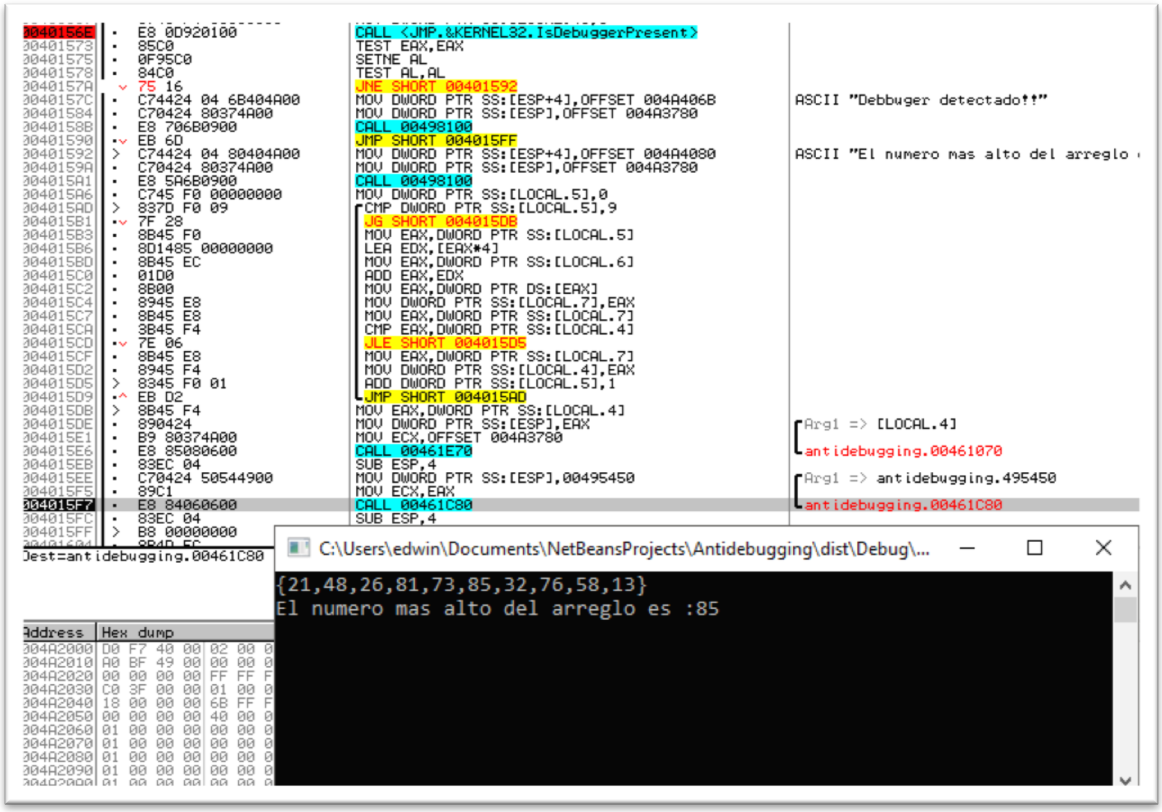
```
EAX  00000001
ECX  756933C6  msvcrt.756933C6
EDX  003A6000
EBX  003A3000
ESP  0071FF00
EBP  0071FF28
ESI  004012E0  antidebugging.<ModuleEntryPoint>
EDI  004012E0  antidebugging.<ModuleEntryPoint>
EIP  0040157C  antidebugging.0040157C
C 0   ES 002B 32bit 0(FFFFFFFF)
P 0   CS 0023 32bit 0(FFFFFFFF)
A 0   SS 002B 32bit 0(FFFFFFFF)
Z 0   DS 002B 32bit 0(FFFFFFFF)
S 0   FS 0053 32bit 3A6000(FFF)
T 0   GS 002B 32bit 0(FFFFFFFF)
D 0
O 0   LastErr 00000000 ERROR_SUCCESS
EFL  00000202  (NO,NB,NE,A,NS,PO,GE,G)
```

Para cambiar la instrucción presionamos la tecla espacio y cambiamos JE por JNZ, quedando como lo muestra la Ilustración 18. En el código se ve como JNE, pero es equivalente, y finalmente ejecutaría todo el bloque de código que concierte a la búsqueda del número más alto del arreglo y mostrándolo en pantalla (ver Ilustración 19).

Ilustración 18: código después de cambiar JE por JNZ.

```
00401573  .  85C0          TEST EAX,EAX
00401575  .  0F95C0        SETNE AL
00401578  .  84C0          TEST AL,AL
0040157A  v  75 16         JNE SHORT 00401592
0040157C  .  C74424 04 6B404A00  MOV DWORD PTR SS:[ESP+4],OFFSET 004A406B  ASCII "Debbuger detectado!!"
00401584  .  C70424 80374A00     MOV DWORD PTR SS:[ESP],OFFSET 004A3780
0040158B  .  E8 706B0900         CALL 00498100
00401590  .v EB 6D               JMP SHORT 004015FF
00401592  >  C74424 04 80404A00  MOV DWORD PTR SS:[ESP+4],OFFSET 004A4080  ASCII "El numero mas alto del arreglo es :"
0040159A  .  C70424 80374A00     MOV DWORD PTR SS:[ESP],OFFSET 004A3780
004015A1  .  E8 5A6B0900         CALL 00498100
```

Ilustración 19: saltando la protección IsDebuggerPresent.



Otra forma más fácil de evadir esta técnica, sería cambiando el valor 0 del flag Z en Ilustración 16 por el valor 1 al momento de ejecutar la instrucción original de la Ilustración 15.
Es importante destacar que existen variaciones de esta técnica anti-debugging una es consultar directamente el valor del flag "**BeingDebugged**" del PEB mediante la inserción de código ASM en el propio código del malware o software al que se le quiere proteger.
Una segunda opción es utilizar la función "**Checkremotedebuggerpresent**", que comprueba si el debugger se está ejecutando en otro proceso separado, utilizando además, la función **IsDebuggerPresent**.

4.1.2. Consultando el Flag **NtGlobalFlag**.

Esta técnica se basa principalmente en revisar el valor del campo **NtGlobalFlag** de la estructura PEB, que describimos anteriormente. Los valores de este campo vienen determinados por otros flags:

- FLG_HEAP_ENABLE_TAIL_CHECK.
- FLG_HEAP_ENABLE_FREE_CHECK.
- FLG_HEAP_VALIDATE_PARAMETERS.

Cuando se crea un proceso a través del debugger, el valor de **NtGlobalFlag** es 0x70, esto como resultado de los flag que lo componen: 0x10, 0x20, 0x40 respectivamente. Cabe destacar que la forma de crear un proceso en

un debugger está determinado por la forma en que abrimos el archivo a debuggear, o en otras palabras, cuando utilizamos la función “attached” en un proceso a debuggear. El valor del Flag **NtGlobalFlag,** no es modificado y mantiene su valor por defecto, que es 0, lo que indica que el proceso no esta siendo debuggeado así lo indica (Ferrie, 2011) es su investigación sobre técnicas anti-debugging basadas en la API de Windows.

4.1.2.1. Un analista de malware puede evadir **NtGlobalFlag**.

En un lenguaje de alto nivel es más difícil obtener el valor del Flag **NtGlogalFlag** directamente, por lo que es necesario obtenerlo a un nivel mas bajo. En lenguaje ensamblador, se obtiene accediendo al PEB luego capturando el valor del Flag en cuestión, después asignamos ese valor a una variable para su evaluación. Esto se logra a través del código siguiente para arquitecturas de 32 bits.

Ilustración 20:estructura PEB con sus respectivos desplazamientos para ensamblados de 32 bits.

```
struct _PEB {
    0x000 BYTE InheritedAddressSpace;
    0x001 BYTE ReadImageFileExecOptions;
    0x002 BYTE BeingDebugged;
    0x003 BYTE SpareBool;
    0x004 void* Mutant;
    0x008 void* ImageBaseAddress;
    0x00c _PEB_LDR_DATA* Ldr;
    0x010 _RTL_USER_PROCESS_PARAMETERS* ProcessParameters;
    0x014 void* SubSystemData;
    0x018 void* ProcessHeap;
    0x01c _RTL_CRITICAL_SECTION* FastPebLock;
    0x020 void* FastPebLockRoutine;
    0x024 void* FastPebUnlockRoutine;
    0x028 DWORD EnvironmentUpdateCount;
    0x02c void* KernelCallbackTable;
    0x030 DWORD SystemReserved[1];
    0x034 DWORD ExecuteOptions:2; // bit offset: 34, len=2
    0x034 DWORD SpareBits:30; // bit offset: 34, len=30
    0x038 _PEB_FREE_BLOCK* FreeList;
    0x03c DWORD TlsExpansionCounter;
    0x040 void* TlsBitmap;
    0x044 DWORD TlsBitmapBits[2];
    0x04c void* ReadOnlySharedMemoryBase;
    0x050 void* ReadOnlySharedMemoryHeap;
    0x054 void** ReadOnlyStaticServerData;
    0x058 void* AnsiCodePageData;
    0x05c void* OemCodePageData;
    0x060 void* UnicodeCaseTableData;
    0x064 DWORD NumberOfProcessors;
    0x068 DWORD NtGlobalFlag;
    0x070 _LARGE_INTEGER CriticalSectionTimeout;
    0x078 DWORD HeapSegmentReserve;
    0x07c DWORD HeapSegmentCommit;
    0x080 DWORD HeapDeCommitTotalFreeThreshold;
    0x084 DWORD HeapDeCommitFreeBlockThreshold;
    0x088 DWORD NumberOfHeaps;
```

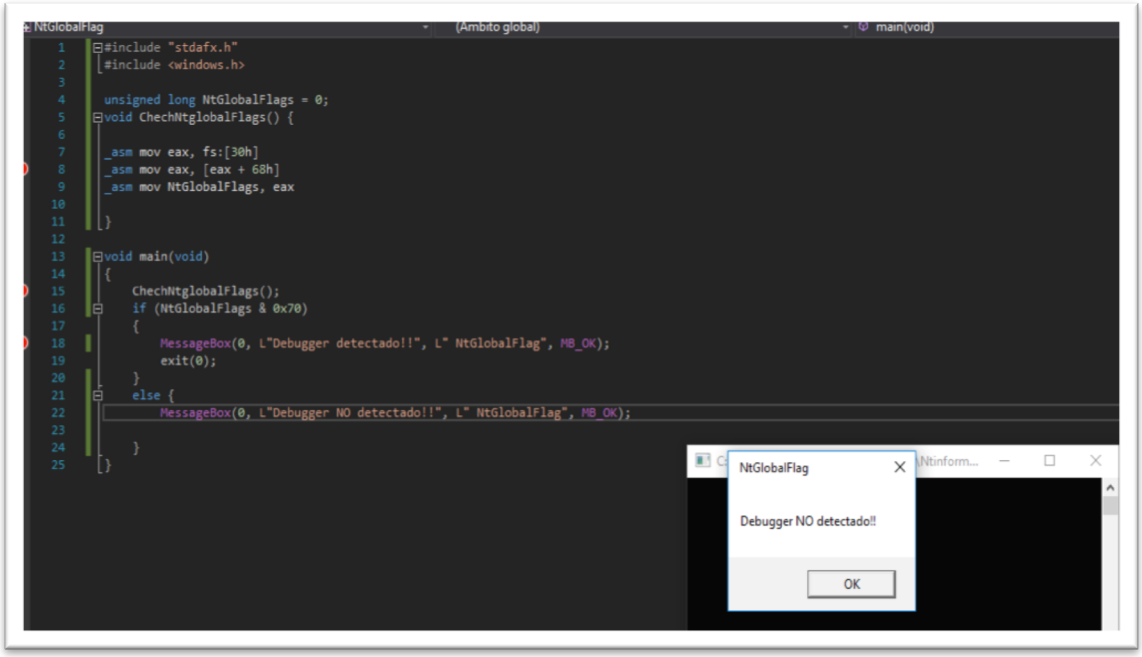
```
    mov eax, fs:[30h] //se accede a la estructura PEB
    mov eax, [eax + 68h] // se obtiene el valor del  flag NtGlobalFlag del PEB
    mov NtGlobalFlags, eax // se asigna el valor del EAX al la variable NtGlobalFlags.
```

```
#include "stdafx.h"
#include <windows.h>

unsigned long NtGlobalFlags = 0;//NO CONFUNDIR CON EL FLAG NtGlobalFlag del PEB.
```

```
void ChechNtglobalFlags() {

_asm mov eax, fs:[30h]
_asm mov eax, [eax + 68h]
_asm mov NtGlobalFlags, eax
}
void main(void)
{
          ChechNtglobalFlags();
          if (NtGlobalFlags & 0x70)
          {
                    MessageBox(0, L"Debugger detectado!!", L" NtGlobalFlag", MB_OK);
                    exit(0);
          }
          else {
                    MessageBox(0, L"Debugger NO detectado!!", L" NtGlobalFlag", MB_OK);

          }
}
```

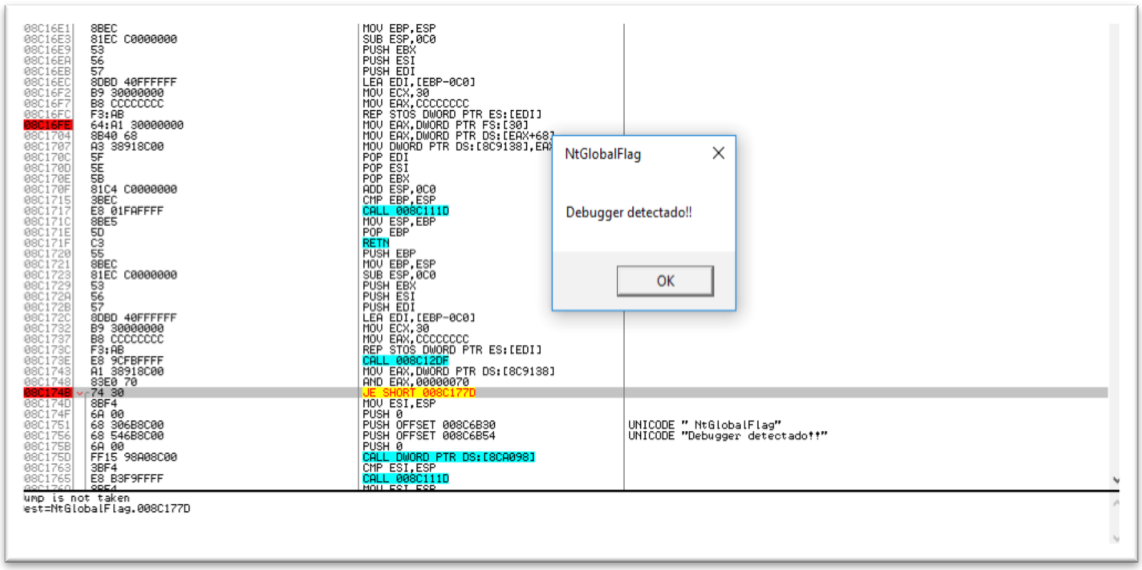
Si analizamos el código anterior, tenemos dos mensajes, uno que es lanzado cuando se detecta un debugger, y el otro cuando se ejecuta el proceso de manera normal. Esta técnica es muy interesante debido a que se modifica el valor del Flag si el proceso es abierto directamente por el debugger, y no en forma de "Attached". Si echamos un vistazo al código anterior y lo ejecutamos en el debugger interno del IDE, que para este caso en particular utilizamos Visual Studio 2017, nos dice que el proceso no corre bajo un debugger, aunque hayamos colocado puntos de interrupción y efectivamente se está debuggeando. Como vemos a continuación, el mensaje indica lo contrario.

Ilustración 21:NtGlobalFlags, ejecutando un programa desde el debugger de Visual Studio 2017.



Una vez que se ha compilado y generado el archivo binario , lo abrimos con un debugger, consecuentemente el debugger será detectado.

Ilustración 22: debuggeando un programa con proteccion anti-debugging: NtGlobalFlags



Para que el analista de malware pueda saltarse esta técnica, se debe identificar en primer lugar, si el código está siendo protegido por esta técnica. En OllyDbg pulsamos la tecla ctrl+f para buscar comandos e introducimos el siguiente código en ensamblador:

```
mov eax, [eax + 68h] // se obtiene el valor del NtGlobalFlag
```

y si lo encontramos es porque se está validando el valor del **NtGlobalFlag**.
La técnica se puede evadir de las siguientes maneras:

a) Antes de asignar el valor del registro eax a la variable ***NtGlobalFlag*** de nuestro programa, cambiamos el valor 0X70 por 0 (1), indicando que el programa no se está debuggeando(2).

Ilustración 23:cambiando el valor de asignado a la variable NtGlobalFlag.

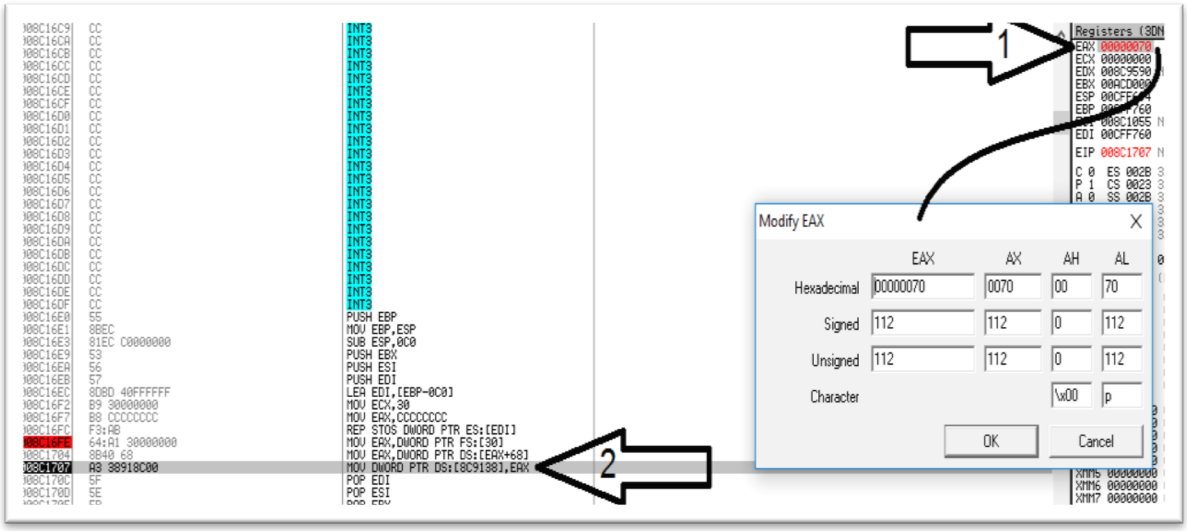
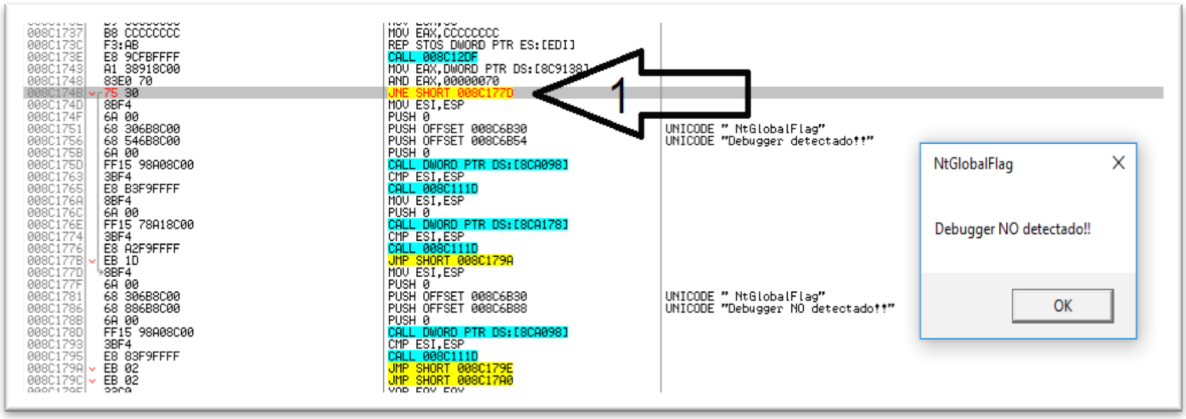


Ilustración 24:evacion de la tecnica anti-debugging que consulta el NtGlobalFlag.

b) Cambiando la instrucción JE por JNE. Esto hará que el salto se efectúe, debido a que JNE se da cuando ZF=0.

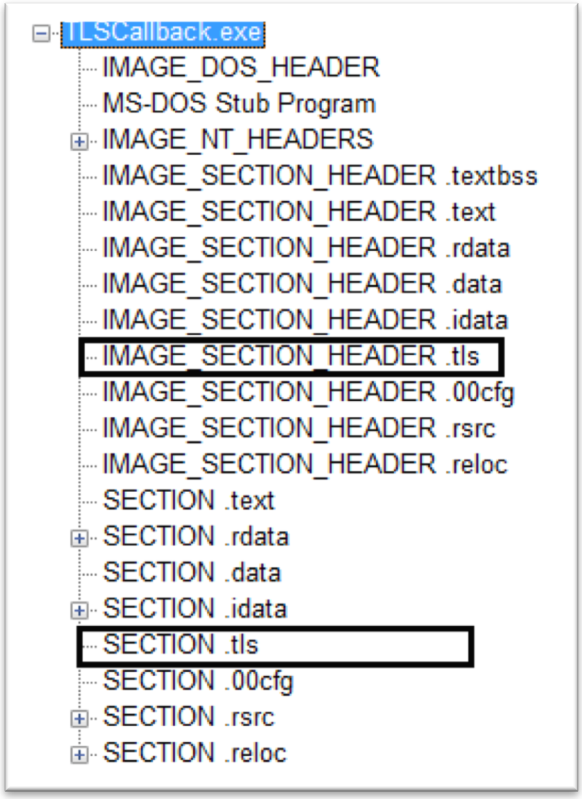
Ilustración 25:cambiando una instrucción en ensamblador para evadir NtGlobalFlag.



4.1.3. **BeingDebugger** implementado con **TLS Callback** como técnica anti-debugging.

TLS (por sus siglas en inglés Thread Local Storage) es utilizado por Windows para definir objetos de datos que son locales para cada uno de los hilos de un proceso que se ejecuta bajo un programa, en otras palabras, cada hilo puede mantener un valor diferente para cada una de las variables declaradas usando TLS. Esta información está contenida en el PE (Portable Ejecutable) El encabezado PE es utilizado por Windows para almacenar meta-información, cargar y ejecutar un programa.

Ilustración 26: Archivo ejecutable inspeccionado con PEView.



En la imagen anterior se pueden identificar las cabeceras TLS, que indican que el programa implementa esta función en su código, pero no necesariamente es una técnica anti-debugging.

Desde la perspectiva de un programador de malware, esta técnica le permite ejecutar un fragmento de código, antes que el debugger se detenga en un punto de entrada tradicional, permitiendo así que el sistema quede infectado o se desactive el debugger antes que el analista pueda revisar el código. Veamos el siguiente código.

```
#include "stdafx.h"
#include <stdio.h>
#include <Windows.h>
```

```
#include <iostream>
#pragma comment(lib,"ntdll.lib")
#pragma comment(linker,"/include:__tls_used")
#pragma section(".CRT$XLB",read)
extern "C" NTSTATUS NTAPI NtQueryInformationProcess(HANDLE hProcess, ULONG InfoClass,
PVOID Buffer, ULONG Length, PULONG ReturnLength);
#define NtCurrentProcess() (HANDLE)-1
void WINAPI TlsCallback(PVOID Module, DWORD Reason, PVOID Context)//función TLS
{
          PBOOLEAN BeingDebugged = (PBOOLEAN)__readfsdword(0x30) + 2;//obtenemos del PEB
el valor del flag que indica si el proceso entró en un estado de debugging, el +2 es
para obtener el valor del desplazamiento 0x20 (ver PEB).
          HANDLE DebugPort = NULL;
          if (*BeingDebugged) //usando el flag del PEB
          {
                    std::cout << "Debugger detectado" << std::endl;
          }

          else
          {
                    std::cout << "No existe Debugger!" << std::endl;
          }

}
__declspec(allocate(".CRT$XLB")) PIMAGE_TLS_CALLBACK CallbackAddress[] = {
TlsCallback,NULL }; //llamado a la función TlsCallback
int main()
{
          printf("Hello world");
          getchar();

          return 0;
}
```

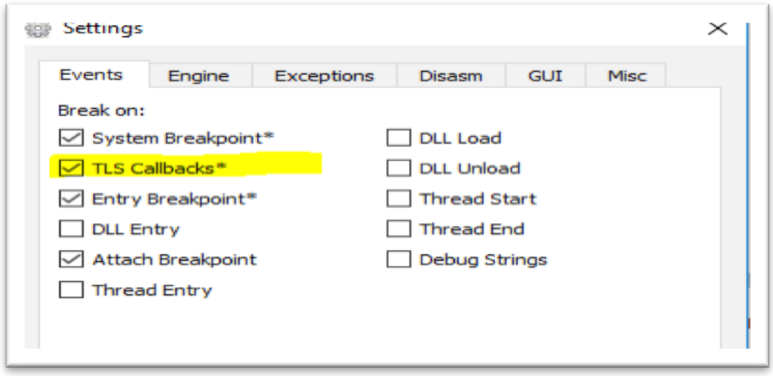
Vemos que dentro de la función TLS se implementa una técnica anti-debugging llamada **Beingdebugged** (similar a **IsDebuggerPresent**) que consiste en consultar el valor del Flag ***Beingdebugged*** de la estructura **PEB** (Ilustración 20:estructura PEB con sus respectivos desplazamientos para ensamblados de 32 bits.). Este Flag del tipo byte, contiene un 1 si el proceso se está debuggeando y cero en caso contrario.

4.1.3.1. Si un analista se encuentra con una función **TLS Callback** implementando una técnica anti-debugging dentro de su código podría:

iniciar el debuggeo en el verdadero lanzamiento del programa, basta con realizar algunas configuraciones al debugger más que nada. El *OllyDbg* se queda un poco corto o es más difícil identificar el punto de retorno de la función TLS. Existe un proyecto llamado **XDbg** basado en *OllyDbg* y con funcionalidades extra que nos ayudarán a evadir esta técnica de una manera más simple.

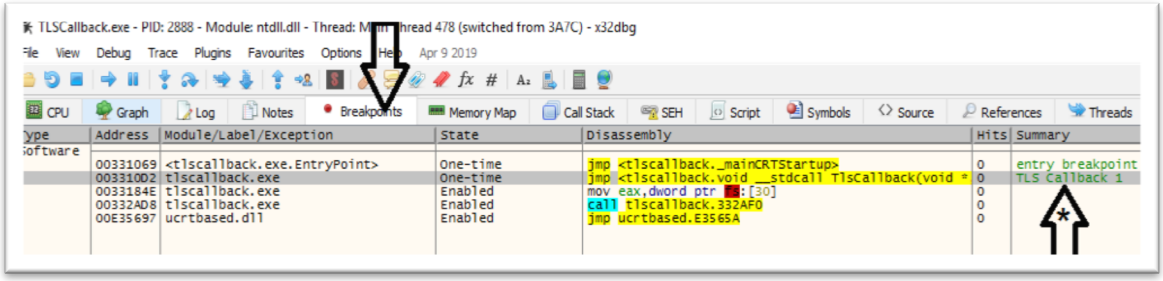
1- Para el caso de **Xdbg** se debe configurar en el menú opciones-preferencias, para que el debugger identifique funciones TLS y pueda colocar breakpoints en éstas.

Ilustración 27:Configurando Xdbg para detectar funciones TLS Callback.



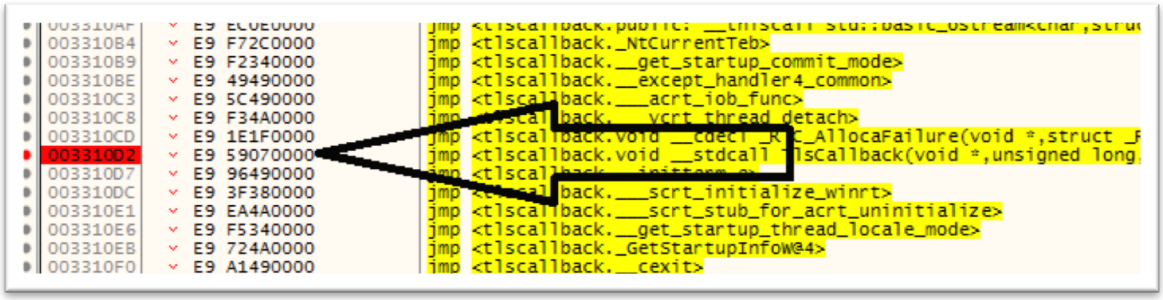
2- Se debe identificar la dirección de la función de devolución de llamada TLS. En **Xdbg** esto se hace en la pestaña Breakpoints.

Ilustración 28:Identificando la función TLS dentro de Xdbg.



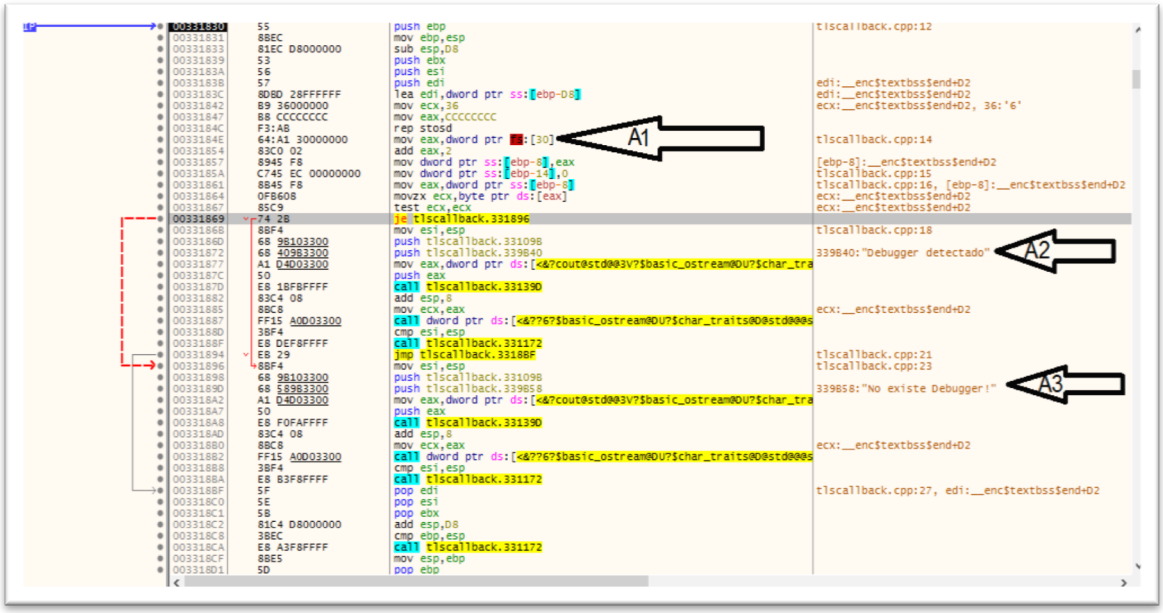
* Como vemos en la imagen anterior, los módulos marcados con TLS Callback en la columna Summary integran esta función. Una vez que tenemos el punto de interrupción, damos doble click en ese módulo y nos llevará al punto de entrada TSL.

Ilustración 29:Identificanto el punto de entrada de la función TLS en Xdbg.



3- Se debe de establecer el punto de interrupción en la función de devolución de llamada TLS, luego ejecute el programa bajo el debugger normalmente.

Ilustración 30:Debuggeado el programa de forma normal luego de encontrar el punto de entrada TLS.



A1: A partir de este punto el código es conocido, y se puede analizar como hemos hecho anteriormente. El código:

```
mov eax dword ptr fs[30]
```

lo que hace es consultar el valor del Flag **beingDebugged** del PEB.

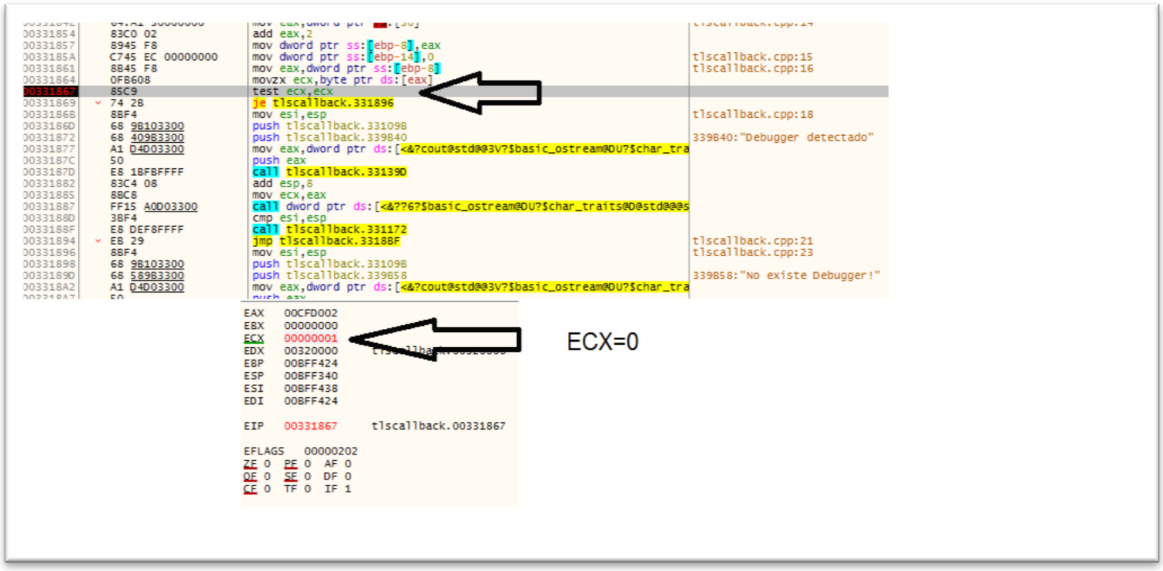
A2: Ocurre cuando se detecta el debugger.

A3: Ocurre cuando el valor del registro **ECX** es 0, o no se detecta un debugger.

Una vez que hemos llegado a este punto, podríamos saltar la protección anti-debugging de dos maneras:

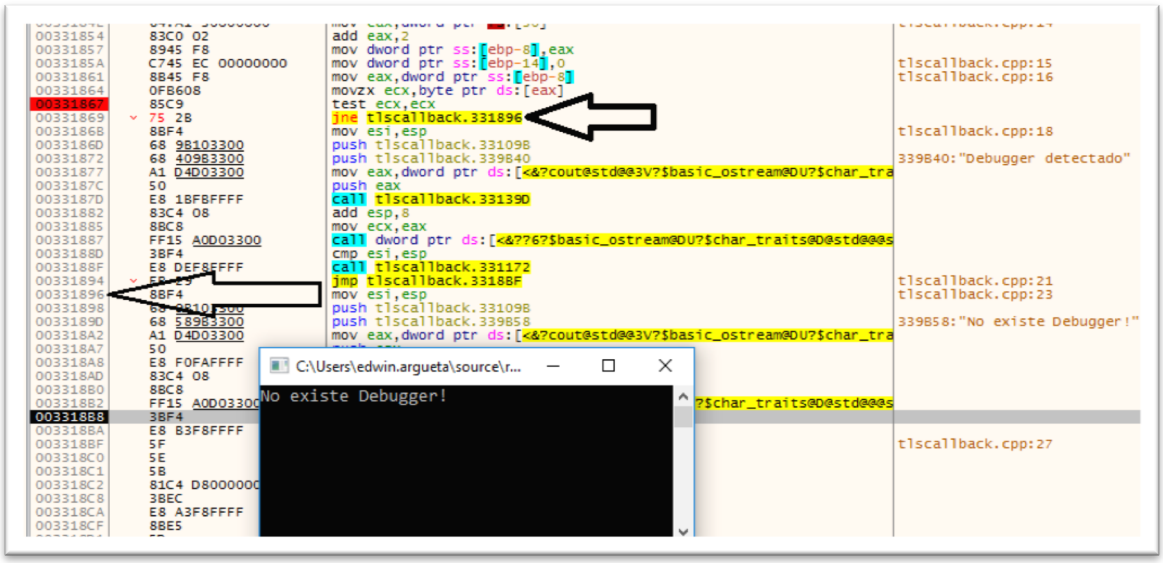
1. Antes de que se ejecute la instrucción 00331867 o test ECX,ECX. Cambiamos el valor del registro de 1 a 0, indicándole al Flag que no existe un debugger, haciendo que el salto **JE** se realice.

Ilustración 31:Evadiendo la técnica BeingDebugged.



2. O simplemente cambiando el salto JE por JNE haciendo que se de el salto, ya que el valor del ZF es igual a 0.

Ilustración 32: evadiendo la técnica BeingDebugged mediante cambio de instrucciones JNE.



También existen software especializado para poder visualizar las llamadas TLS dentro de un ejecutable. IDA pro, un desensamblador empleado para ingeniería inversa muy potente que facilita enormemente el análisis de para este tipo de funciones.

4.1.4. La función **CheckRemoteDebbugerPresent**.

Esta técnica básicamente comprueba si existe un proceso "remoto", o en paralelo, no implica que el debugger resida necesariamente en una

computadora diferente; en cambio, indica que el debugger reside en un proceso separado y paralelo.

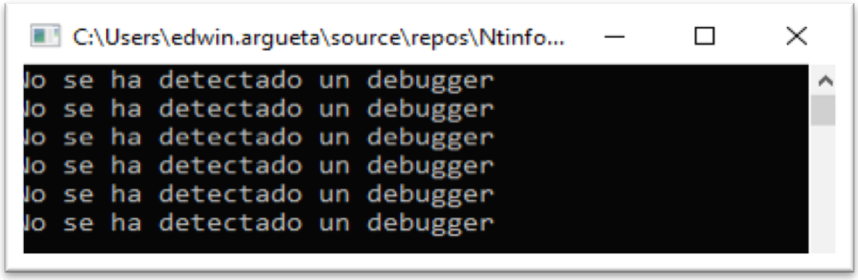
4.1.4.1. Como analista de seguridad podemos evadir **CheckRemoteDebbugerPresent**.

Supongamos que tenemos el siguiente programa que implementa esta protección. En el siguiente código tenemos un siclo *while* que se encargara de comprobar cada 5 segundos la presencia de un debugger en un proceso paralelo.

```
#include "stdafx.h"
#include  <ntstatus.h>
#include <windows.h>
#include <stdlib.h>
#include <winternl.h>
#include <iostream>
int main()
{BOOL isDebuggerPresents = FALSE;
   for (int y = 0; y <= 5; y++) {
         if (CheckRemoteDebuggerPresent(GetCurrentProcess(), &
         isDebuggerPresents)) {
                  if (isDebuggerPresents) {
                           std::cout << "Se ha detectado un debugger!\n";
                                    Sleep(5000);

                           }
                           else {

                                    std::cout << "No se ha detectado un debugger\n";
                                    Sleep(5000);                          }
                  }
         }
         return 0;
}
```

Si ejecutamos el programa de manera normal vemos que, no se detecta la presencia de ningún debugger.

Ilustración 33:Programa que verifica si existe un debugger en un proceso paralelo.



Ahora corremos el archivo ejecutable, y desde el debugger abrimos el proceso con la opción “attach”, en ese instante veremos como cambia el mensaje que se muestra en la consola.

Ilustración 34:Atacando un proceso para debuggear con Xdbg.

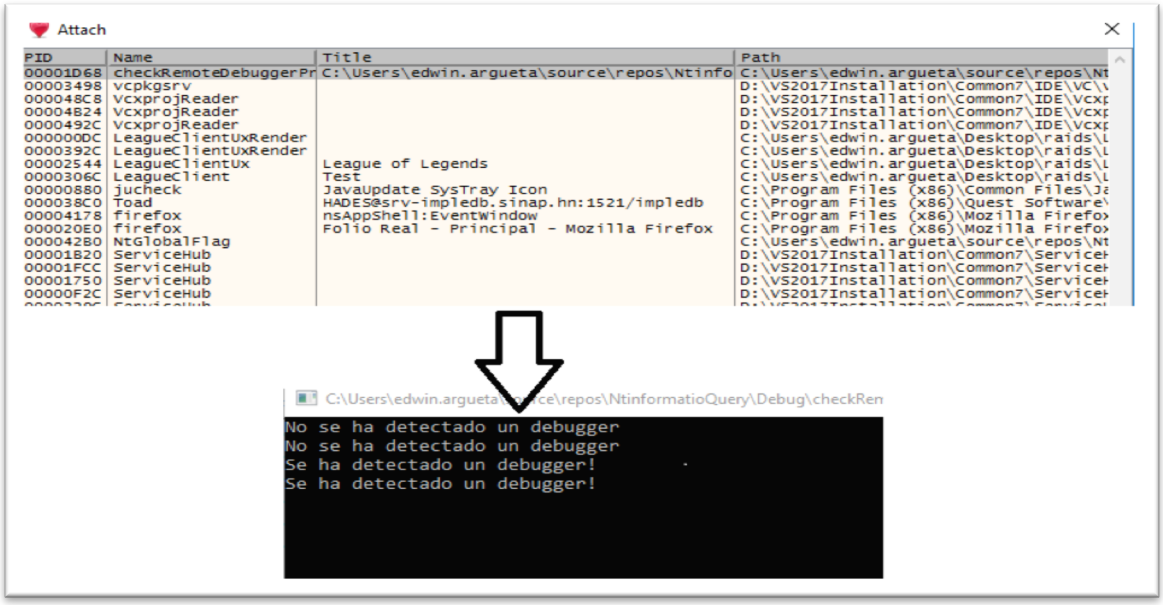


Ilustración 35:evadiendo la técnica RemoteDebuggerPresents.

Para saltarnos este tipo de protección, hacemos lo mismo que hemos hecho con las técnicas anteriores; y es la de sustituir la instrucción JE por un salto JNE y ensamblar el ejecutable nuevamente.

4.1.5. **NtQueryInformationProcess** para anti-debugging.

NtQueryInformationProcess. Es una función de la librería ntdll. Si se usa el parámetro **ProcessDebugPort**, devuelve el puerto asignado, si el proceso está siendo debuggeado, o cero en caso contrario.

También se puede llamar con el parámetro **ProcessDebugFlags**, en este caso, si el valor de retorno es cero indica que hay debugger. Por último, si se llama con el parámetro **ProcessDebugObjectHandle** y devuelve un valor distinto a cero, indica que el proceso está siendo debuggeado.

La función se compone de la siguiente manera:

Ilustración 36:Estructura de la función NtQueryInformationProcess.

```
C++

__kernel_entry NTSTATUS NtQueryInformationProcess(
  IN HANDLE           ProcessHandle,
  IN PROCESSINFOCLASS ProcessInformationClass,
  OUT PVOID           ProcessInformation,
  IN ULONG            ProcessInformationLength,
  OUT PULONG          ReturnLength
);
```

Donde **HANDLE,** es un parámetro de entrada que identifica el proceso del cual queremos obtener información.

PROCESSINFOCLASS, es un parámetro de entrada y puede tomar diferentes valores, según la información que necesitemos acerca del proceso definido. Los valores pueden ser **ProcessDebugPort**, **ProcessDebugFlags** o **ProcessDebugObjectHandle** este último es un parámetro que le indica a la función que queremos obtener información, si el proceso se está ejecutando bajo el control de un debugger.

PVOID, es un parámetro o apuntador donde se almacenará la información que estamos solicitando.

ULONG y **PULONG**, indica un tamaño de tipo **DWORD,** que es un tipo de dato definido para Windows, básicamente estos parámetros son para inicializar un búfer del parámetro anterior, que pueda contener toda la información solicitada.

4.1.5.1. Para evadir **NtQueryInformationProcess** un analista puede usar la siguiente técnica.

Tenemos el código siguiente, donde se implementa la función **NtQueryInformationProcess**

```
#include  "stdafx.h"
#include  <ntstatus.h>
#include <windows.h>
#include <stdlib.h>
#include <winternl.h>
#include <iostream>
typedef NTSTATUS(NTAPI *pfnNtQueryInformationProcess)(
          _In_      HANDLE           ProcessHandle,
          _In_      UINT             ProcessInformationClass,
          _Out_     PVOID            ProcessInformation,
          _In_      ULONG            ProcessInformationLength,
          _Out_opt_ PULONG           ReturnLength
          );
const UINT ProcessDebugPortw = 7;
int main(int argc, char *argv[])
{
          pfnNtQueryInformationProcess NtQueryInformationProcess = NULL;
          NTSTATUS status;
          DWORD isDebuggerPresent = 0;
          HMODULE hNtDll = LoadLibrary(TEXT("ntdll.dll"));

          if (NULL != hNtDll)
```

```
        {
                NtQueryInformationProcess =
(pfnNtQueryInformationProcess)GetProcAddress(hNtDll, "NtQueryInformationProcess");
                if (NULL != NtQueryInformationProcess)
                {
                        status = NtQueryInformationProcess(
                                GetCurrentProcess(),
                                ProcessDebugPortw,
                                &isDebuggerPresent,
                                sizeof(DWORD),
                                NULL);
                        if (status == 0x00000000 && isDebuggerPresent != 0)
                        {
                                std::cout << "Debugger detectado" << std::endl;
                                exit(-1);
                        }
                        else {
                                std::cout << "No existe Debugger!" << std::endl;

                        }
                }
        }
                return 0;
}
```

Ilustración 37: Evadiendo NtQueryInformationProcess mediante una instrucción JNE o JE.

```
013324E9  75 42        JNE SHORT 0133252D
013324EB  837D DC 00   CMP DWORD PTR SS:[EBP-24],0
013324EF  74 3C        JE SHORT 0133252D
013324F1  8BF4         MOV ESI,ESP
013324F3  68 A0103301  PUSH OFFSET 013310A0
013324F8  68 688B3301  PUSH OFFSET 01338B68          ASCII "Debugger detectado"
013324FD  A1 9CC03301  MOV EAX,DWORD PTR DS:[133C09C]
01332502  50           PUSH EAX
01332503  E8 86EEFFFF  CALL 0133138E
01332508  83C4 08      ADD ESP,8
0133250B  8BC8         MOV ECX,EAX
0133250D  FF15 B0C03301 CALL DWORD PTR DS:[133C0B0]
01332513  3BF4         CMP ESI,ESP
01332515  E8 53ECFFFF  CALL 0133116D
0133251A  8BF4         MOV ESI,ESP
0133251C  6A FF        PUSH -1
0133251E  FF15 E8C13301 CALL DWORD PTR DS:[133C1E8]
01332524  3BF4         CMP ESI,ESP
01332526  E8 42ECFFFF  CALL 0133116D
0133252B  EB 29        JMP SHORT 01332556
0133252D  8BF4         MOV ESI,ESP
0133252F  68 A0103301  PUSH OFFSET 013310A0
01332534  68 848B3301  PUSH OFFSET 01338B84          ASCII "No existe Debugger!"
01332539  A1 9CC03301  MOV EAX,DWORD PTR DS:[133C09C]
```

Si vemos el código en el OllyDbg podemos identificar la instrucción con dirección **013324E9,** donde existe una instrucción **JNE SHORT** a la dirección **0133252D,** también existe otra **JE** a la misma dirección, ambas dan un salto, según sea el caso del flag **ZF** a la porción del código donde no se ha identificado un debugger. Para hacer que nuestro programa continúe su ejecución de forma normal bajo el debugger, basta con cambiar una de las dos instrucciones antes mencionadas, por su contraparte, como por ejemplo, cambiar la instrucción JNE por JE o al momento de ejecutar la instrucción en la dirección de memoria **013324EF** colocar el registro ZF=1 y se dará el salto.

4.1.6. Utilizando la función **OutputDebugString** para anti-debugging.

Es una función de la librería kernel32, que se comporta de manera diferente dependiendo si hay o no un debugger capturando excepciones.

En su blog Michael Colling [12], explica detalladamente cómo trabaja esta función dentro de la API de Windows, al momento de capturar eventos que se generan durante la ejecución de un proceso. Existen dos prototipos de funciones: **OutputDebugStringA** utilizada para mensajes con codificación ANSII y **OutputDebugString** para mensajes con codificación UNICODE.

OutputDebugString funciona de manera que: si llamamos a esta función para pasarle una cadena de texto a un debugger adjunto, al momento de seguir ejecutando instrucciones del programa el valor del registro EAX será una dirección de memoria válida dentro del espacio de direcciones del proceso en ejecución. En cambio, si no existe un debugger adjunto el valor del registro EAX será 0 entonces, en ese caso, si intentamos leer el contenido de una dirección de memoria no válida, se generará una excepción (EXCEPTION_ACCESS_VIOLATION - 0xc0000005) y sabremos que no hay un depurador adjunto. Por otro lado, si no se produce una excepción, entonces sabemos que hay un depurador adjunto.

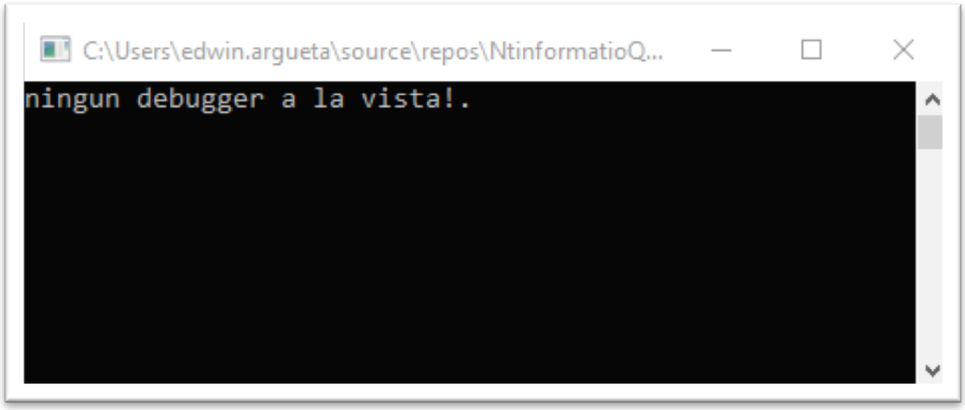
4.1.6.1. Cómo un analista de seguridad puede evadir **OutputDebugString.**

Supongamos que tenemos el siguiente código:

```
#include "stdafx.h"
#include  <ntstatus.h>
#include <windows.h>
#include <stdlib.h>
#include <winternl.h>
#include <iostream>
int main()
{OutputDebugStringA("--mensaje al debugger");
 _try{ _asm mov ebx, dword ptr [eax]//leemos el registro eax, despues de enviarle un
mensaje al debugger.
       printf("se ha detectado un debugger!");
 }_except(EXCEPTION_EXECUTE_HANDLER) {
         printf("ningun debugger a la vista!.");
  } getchar();
}
```

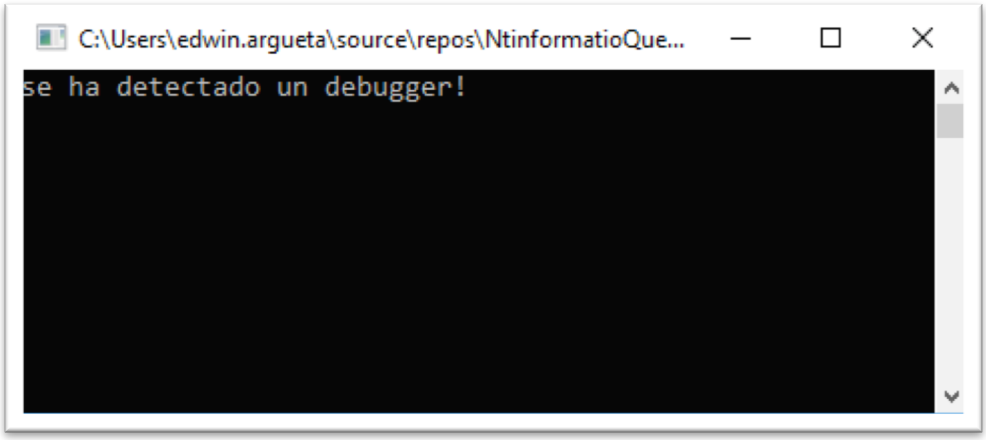
Lo que hace es enviar un mensaje al debugger, inmediatamente se le asigna el valor del registro EBX a EAX, si esto es posible, es porque el registro tiene un valor válido, en caso contrario se lanzará una excepción a nivel de sistema operativo ya que la excepción no es capturada por ningún debugger.

Ilustración 38:Ejecucion de un programa con la función OutputDebugStringA y sin ningún debugger adjunto al Sistema Operativo.



La imagen anterior muestra cuando se ejecutó el programa sin presencia de ningún debugger en ordenador.

Ilustración 39:Ejecución de un programa con un debugger adjunto al Sistema Operativo.



En la imagen anterior podemos ver cuando se ejecutó el programa, y se tenia adjunto un debugger al sistema operativo. No es necesario siquiera ejecutarlo en OllyDbg o en otro debugger para que se detecte que existe uno a la espera de excepciones.

Una manera para evadir esta protección de código es, al momento de leer el registro EAX colocar una dirección inválida haciendo que se genere una excepción y salte esta protección.

Ilustración 40:Saltando la técnica OutputDebugString.

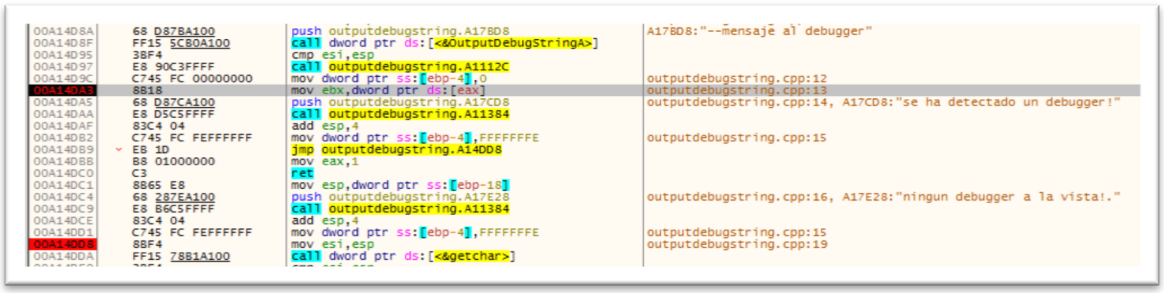


Ilustración 41:Resgistros al momento de ejecutar la instrucción mov ebx, dword ptr [eax].

```
EAX    006E5000
EBX    006E2000
ECX    73BC780B      kernelbase.73BC780B
EDX    00000000
EBP    008FF738
ESP    008FF650
ESI    008FF650
EDI    008FF720

EIP    00A14DA3      outputdebugstring.00A14DA3

EFLAGS    00000344
ZF 1   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 1   IF 1
```

Al momento de ejecutar la línea con dirección **00A14DA3** que contienen el código ASM que hemos incluido en nuestro ejecutable, cambiamos el valor del registro EAX por un una dirección de memoria inválida, consecuentemente se dará el salto mostrando el mensaje esperado.

Ilustración 42:Cambiando el valor del registro EAX por una dirección invalida.

```
EAX    0000001C
EBX    00000000
ECX    BC9928DD
EDX    0FF40BA4      ucrtbased.0FF40BA4
EBP    008FF738
ESP    008FF650
ESI    00A14DC1      outputdebugstring.00A14DC1
EDI    00000000
```

C:\Users\edwin.argueta\source\repos\NtinformatioQuery\Debug\OutPutDebugStr...

```
ningun debugger a la vista!.
```

4.2. Métodos anti-debugging basados en breakpoint por hardware.

Con anterioridad identificamos y definimos un breakpoint, cuál es su función durante la etapa de análisis de código. En este apartado definiremos y enumeraremos los breakpoint por hardware y cómo pueden ser utilizados como una técnica anti-debugging.

En la arquitectura X86 (o 32 bits), existen 8 registros que pueden ser utilizados para realizar tareas de debugging.

1. DR0- RD3: Son breakpoint lineales de entrada y van de 0 a 3 respectivamente. Son registros asociados a direcciones físicas durante el proceso de debuggeo.
2. DR4,DR5: son registros reservados.
3. DR6:Informa del estado de un punto de interrupción o cual es el punto de interrupción que esta activo en ese momento.
4. DR7:Informa sobre cómo debe activarse un punto de interrupción, si al leer, escribir o ejecutar.

Supongamos que tenemos el código siguiente expuesto en la web de ***Apriorit*** por Oleg kulchytskyy[8].

```
int CheckHardwareBreakpoints()
{
    CONTEXT ctx;
    ZeroMemory(&ctx, sizeof(CONTEXT));
    ctx.ContextFlags = CONTEXT_DEBUG_REGISTERS;
    HANDLE hThread = GetCurrentThread();

    // Get the registers
    if(GetThreadContext(hThread, &ctx) == 0)
        return -1;
    if(ctx.Dr0 != 0)
        ++NumBps;
    if(ctx.Dr1 != 0)
           ++NumBps;
    if(ctx.Dr2 != 0)
           ++NumBps;
    if(ctx.Dr3 != 0)
        ++NumBps;

    return NumBps;
}
```

Donde se evalúan uno a uno los registros DRX para determinar si existen algún breakpoint activo que pueda detener el proceso normal y alterne al modo debuggin.

4.2.1. Para evadir esta técnica basada en hardware, un programador puede utilizar la siguiente solución:

La línea :

```
ctx.ContextFlags = CONTEXT_DEBUG_REGISTERS;
```

identificara el valor de cada uno de los registros DRX como vemos en su representación en ensamblador.

```
    xor eax, eax
    mov ecx, [esp + 0ch] ; This is a CONTEXT structure on the stack
     mov dword ptr [ecx + 04h], eax ; Dr0
     mov dword ptr [ecx + 08h], eax ; Dr1
     mov dword ptr [ecx + 0ch], eax ; Dr2
     mov dword ptr [ecx + 10h], eax ; Dr3
     mov dword ptr [ecx + 14h], eax ; Dr6
     mov dword ptr [ecx + 18h], eax ; Dr7
     add dword ptr [ecx + 0b8h], 2 ;
     ret
```

una vez identificado los registros, bastara con reiniciar los valores (EAX=0) de cada uno de ellos en cada comprobación.

7) Conclusiones

7.1 Como apasionado de la seguridad informática, y después de finalizar esta tesis enfocada en demostrar las técnicas anti-debugging, utilizadas en sistemas operativos Windows concluyo que:

- Me fue posible entender el proceso que se lleva a cabo, para la ejecución de un archivo binario, los registro y estructuras que intervienen a la hora de crear un hilo de ejecución.
- Pude darme cuenta, que detrás de un debugger existe una capa llamada API proporcionada por el sistema operativo, la cual puede ser consultada para manejar los eventos que genera un programa cuando es cargado por el "loader" de Windows, para posteriormente ser ejecutado en el procesador del ordenador.
- Pude comprender lo importante que es un debugger para los analistas de malware necesario para determinar el comportamiento y las intensiones de un programa del cual no tenemos su código fuente.
- Logre entender e identificar algunas instrucciones generales, a la hora de leer código ensamblador, que consultan módulos y estructuras propias de las librerías del sistema operativo.
- Me informe sobre un gran numero de técnicas anti-debugging que se pueden implementar para evitar o dificultar el análisis de programas o malware que intentan dañar el sistema
- Gracias a la metodología abordada y objetivos propuestos, me fue posible exponer un número considerable de técnicas anti-debugging, siendo estas las más comunes y fáciles de implementar. Existen muchas otras técnicas anti-debuggin que no fueron posibles estudiar en este material, debido al énfasis que se dio en explicar de una manera muy clara y precisa las aquí expuestas.
- Espero que mi trabajo incentive al lector, a indagar y estudiar muchas otras técnicas como SEH(Structured Exception Handling por sus siglas en inglés), NtSetInformationThread, Debugging messages, Stack Segment Manipulation, Handle Tracing las cuales no fueron posibles cubrir en este documento.

- Personalmente considero que el trabajo de un analista de código, es un trabajo arduo y tedioso, por la dificultad que lleva el analizar el código ensamblador y mas aun cuando está presente alguna técnica que dificulte o interfiera dicho análisis.
- Finalmente, estoy satisfecho con el desarrollo de esta trabajo, por que amplié mis conocimientos en el área de programación, análisis de código, manejo de debuggers, implementación de técnicas anti-debugging y más que eso, pude exponer los que he aprendido a lo largo del desarrollo de mi master. Sintetizando los conocimientos en un trabajo que sirva de base a nuevas investigaciones.

8) Glosario

API: Una API es un conjunto de funciones y procedimientos que cumplen una o muchas funciones con el fin de ser utilizadas por otro software. Las siglas API vienen del inglés Application Programming Interface. En español sería Interfaz de Programación de Aplicaciones por (Comunidad_Tecnologica, 2019) [13].

Archivo binario: Es un archivo informático que contiene información de cualquier tipo codificada en binario para el propósito de almacenamiento y procesamiento en ordenadores.

C/C++: Es un lenguaje de programación diseñado en 1979 por Bjarne Stroustrup.

Compilador: Es un tipo de traductor que transforma un programa entero de un lenguaje de programación (llamado código fuente) a otro.Usualmente el lenguaje objetivo es código máquina, aunque también puede ser traducido a un código intermedio (bytecode) o a texto.

CPU: Unidad de procesamiento central (conocida por las siglas CPU, del inglés: central processing unit), es el hardware dentro de un ordenador u otros dispositivos programables, que interpreta las instrucciones de un programa informático mediante la realización de las operaciones básicas aritméticas, lógicas y de entrada/salida del sistema.

Debugger: Es una Herramienta o Aplicación que permite la ejecución controlada de un programa o código para seguir cada instrucción ejecutada y localizar así el Bug (o error) , códigos de protección.

Dump: En informática, un volcado de memoria (en inglés core dump o memory dump) es un registro no estructurado del contenido de la memoria en un momento concreto, generalmente utilizado para depurar un programa que ha finalizado su ejecución incorrectamente.

Ejecutable: Es tradicionalmente un archivo binario, cuyo contenido se interpreta por el ordenador como un programa. Generalmente, contiene instrucciones en código máquina de un procesador en concreto.

Evento: Los eventos son todas las acciones que el usuario o alguna aplicación inicia, dar clic sobre un botón, presionar las teclas del teclado.

Flag: O bandera se refiere a uno o más bits que se utilizan para almacenar un valor binario o código que tiene asignado un significado.

Handler: O manejador de eventos, son las funciones que responden a los eventos que se producen.

Hardware: las partes físicas, tangibles, de un sistema informático; sus componentes eléctricos, electrónicos, electromecánicos y mecánicos.

IDE: entorno de desarrollo interactivo, en inglés Integrated Development Environment (IDE), es una aplicación informática que proporciona servicios integrales para facilitarle al desarrollador o programador el desarrollo de software.

Instrucción: al conjunto de datos insertados en una secuencia estructurada o específica que el procesador interpreta y ejecuta.

Interfaz: es un medio común para que los objetos no relacionados se comuniquen entre sí. Estas son definiciones de métodos y valores sobre los cuales los objetos están de acuerdo para cooperar.

OllyDbg: depurador de código ensamblador de 32 bits para sistemas operativos Microsoft Windows.

Opcode: o código de operación, es la porción de una instrucción de lenguaje de máquina que especifica la operación a ser realizada. Su especificación y formato serán determinados por la arquitectura del conjunto de instrucciones.

Stack: Una pila (stack en inglés) es una lista ordenada o estructura de datos que permite almacenar y recuperar datos, el modo de acceso a sus elementos es de tipo LIFO (del inglés Last In, First Out, «último en entrar, primero en salir»).

Proceso: Puede entenderse informalmente como un programa en ejecución. Formalmente un proceso es "Una unidad de actividad que se caracteriza por la ejecución de una secuencia de instrucciones, un estado actual, y un conjunto de recursos del sistema asociados".

PEView: un software que permite ver la estructura y el contenido de los archivos portables ejecutables .

Software: se considera como software al soporte lógico de un sistema informático, que comprende el conjunto de los componentes lógicos necesarios que hacen posible la realización de tareas específicas.

9) Bibliografía

[1] Pablo. Ramos, «Welivesecurity,» 14 Enero 2014. [En línea]. Available: https://www.welivesecurity.com/la-es/2014/01/14/bases-analisis-estatico-malware-bases-desensamblado/.

[2] TR31N0RD, «TR31N0RD,» 30 Diciembre 2012. [En línea]. Available: http://tr31n0rd.blogspot.com/2012/12/estructura-pe-portable-executable-en.html.

[3] M. Collins, «The Imaginary Road,» 01 Junio 2013. [En línea]. Available: https://www.michaelfcollins3.me/blog/2013/06/01/understanding-outputdebugstring.html.

[4] Oscar. Campos. «genbeta,» 20 Abril 2012. [En línea]. Available: https://www.genbeta.com/desarrollo/como-funciona-un-depurador-de-c-c-parte-i.

[5] Miquel. Albert. Orenga y Gerard. Enrique. Manonellas, «Programación en ensamblador (x86-64),» [En línea]. Available: uoc.edu.

[6] P. Rascagneres, SEGURIDAD INFORMATICA Y MALWARES: ANALISIS DE AMENAZAS E IMPLEMENTACION DE CONTRAMEDIDAS, Barcelona: ENI ediciones, 2016.

[7] R. Kath, «Microsoft Docs,» 29 Junio 2010. [En línea]. Available: https://docs.microsoft.com/en-us/previous-versions/ms809754(v=msdn.10).

[8] Andalucia Cert, «Laboratorio para el analisis de malware,» Sevilla España, 2015.

[9] Michael Sikorki. y Andrew. Honig. , PRACTICAL MALWARE ANALYSIS, San Francisco: No Starch Press, Inc, 2012.

[10] Matias. Porolli. «welivesecurity,» 18 junio 2014. [En línea]. Available: https://www.welivesecurity.com/la-es/2014/06/18/complicando-analisis-algunas-tecnicas-anti-debugging/.

[11] Oleg Kulchytskyy, «Anti Debugging Protection Techniques With Examples,» Lunes Septiembre 2016. [En línea]. Available: https://www.apriorit.com/dev-blog/367-anti-reverse-engineering-protection-techniques-to-use-before-releasing-software.

[12] Jose Manuel Fernández, «Security Art work,» 28 mayo 2015. [En línea]. Available: https://www.securityartwork.es/2015/05/28/jugando-con-tecnicas-anti-debugging-ii/.

[13] Joshua Tully, «Code Project for those who code,» 18 Septiembre 2008. [En línea]. Available: https://www.codeproject.com/Articles/29469/Introduction-Into-Windows-Anti-Debugging.

[14] P. Ferrie, «The Ultimate Anti-Debugging Reference,» 04 Mayo 2011. [En línea]. Available: http://pferrie.host22.com/papers/antidebug.pdf.

[15] L. P. «Welivesecurity,» 4 Agosto 2017. [En línea]. Available: https://www.welivesecurity.com/la-es/2017/08/04/tacticas-de-los-cibercriminales-arruinan-dia-analistas/.

[16] C. G. Amaya, «Welivesecurity,» 13 Agosto 2014. [En línea]. Available: https://www.welivesecurity.com/la-es/2014/08/13/adelantandonos-atacantes-analisis-dinamico-de-malware/.

[17] Comunidad_Tecnologica, «pandorafms.org,» 14 febrero 2019. [En línea]. Available:

https://web.archive.org/web/20190215165536/https://blog.pandorafms.org/es/para-que-sirve-una-api/.

10) Anexos

10.1 Instalación del entorno de trabajo Netbeans 8.2.

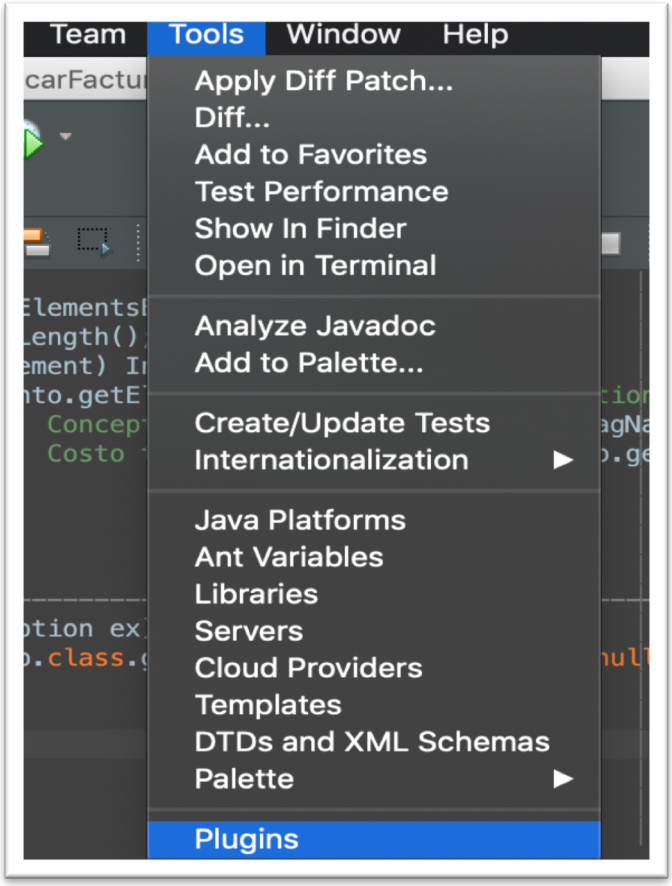
Netbeans 8.2 es un IDE de código abierto disponible en https://netbeans.org/downloads/8.2/.

Para que este software pueda correr en el ordenador es necesario tener instalada la maquina virtual de java, la cual podemos descargar de este enlace https://www.java.com/es/download/.

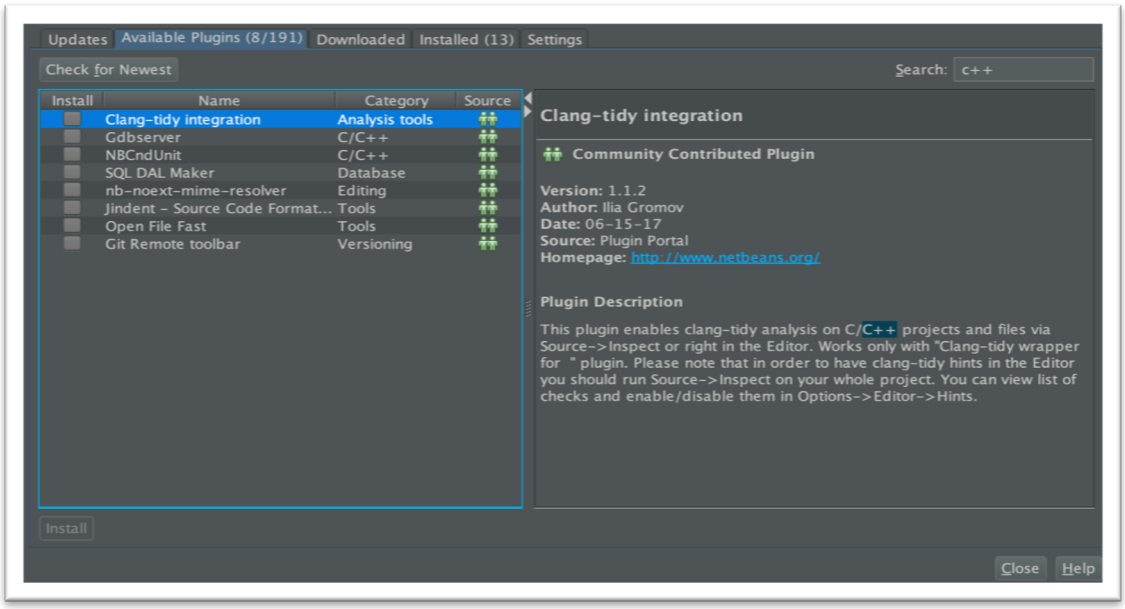
Para compilar aplicaciones para Windows es necesario tener el compilador Mingw instalado en el ordenador http://www.mingw.org/. Luego de instalarlo debemos guardamos la ruta de instalación que necesitaremos maas adelante.

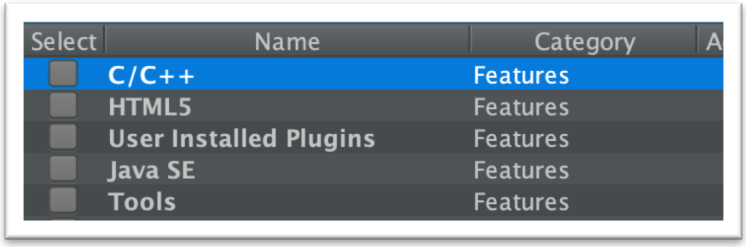
Una vez hayamos instalado Java, Netbeans y el MinGw en nuestro sistema debemos configurar el Netbeans para que use el MinGw como compilador de C/C++ como vemos a continuación.

Agregar el plugin para C/C++.



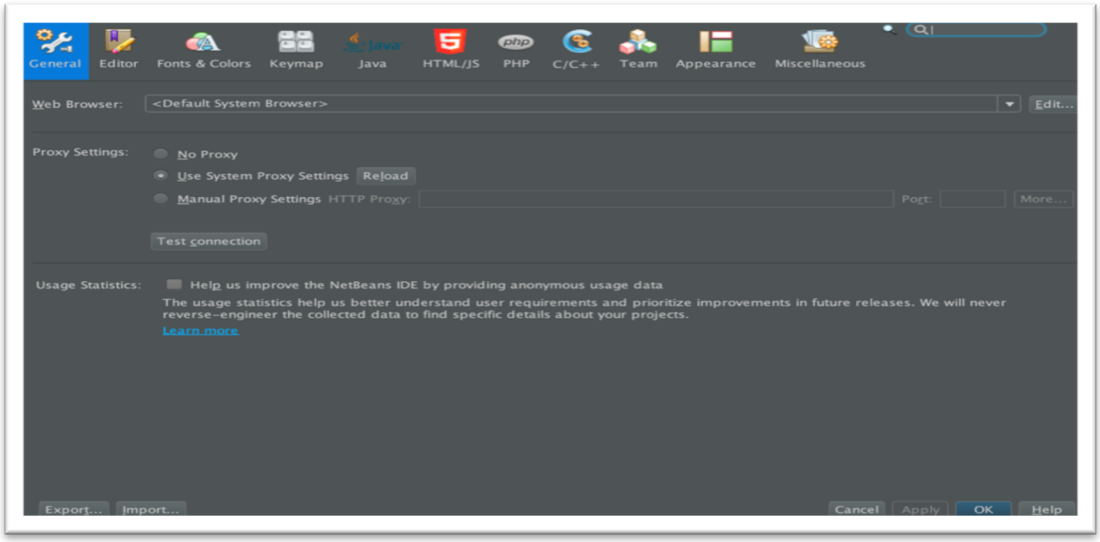
Buscamos en Avaible Plugin el de C/C++ y lo instalamos.

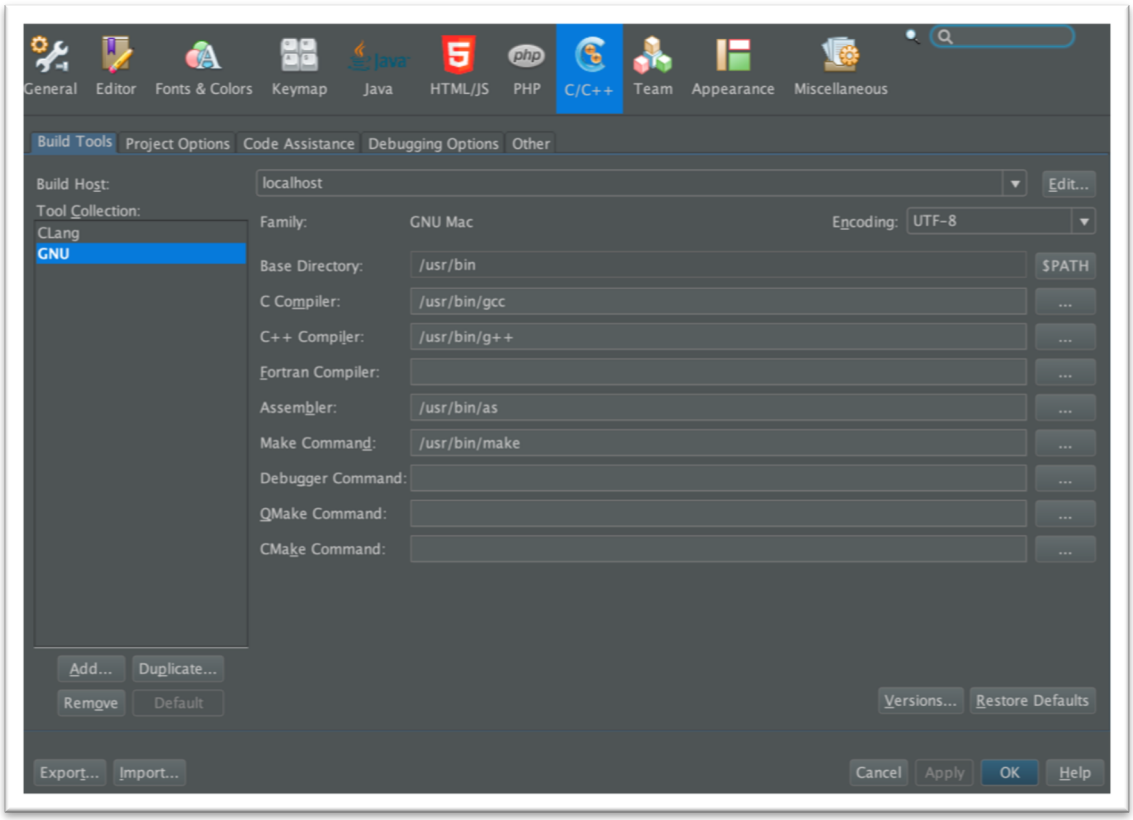




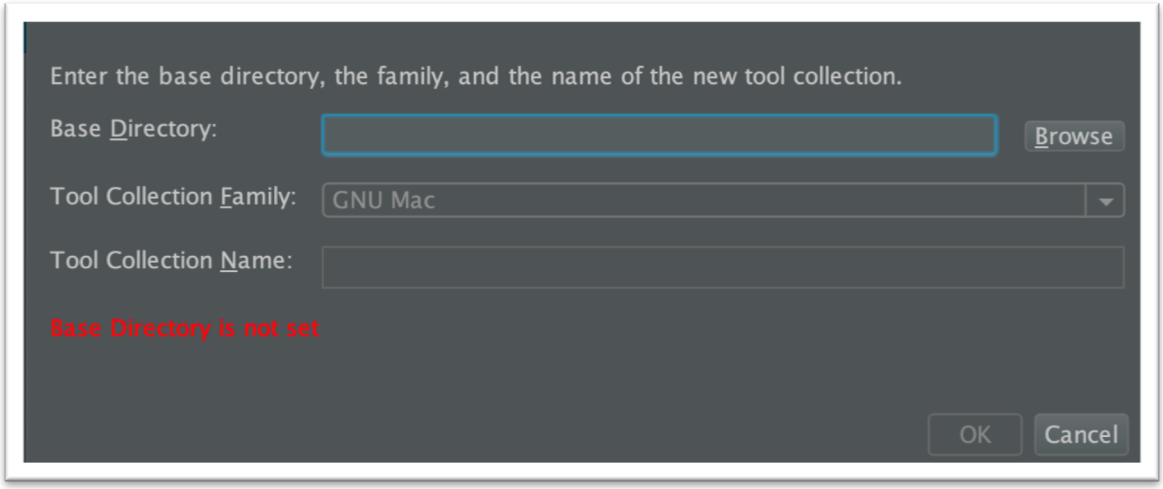
Una vez que tenemos el IDE con el plugin para C/C++ procedemos a configurar el MinGw como compilador.

Nos vamos a la barra de herramientas Tools-->Options. Y nos aparecerá la siguiente pantalla y seleccionaremos C/C++.

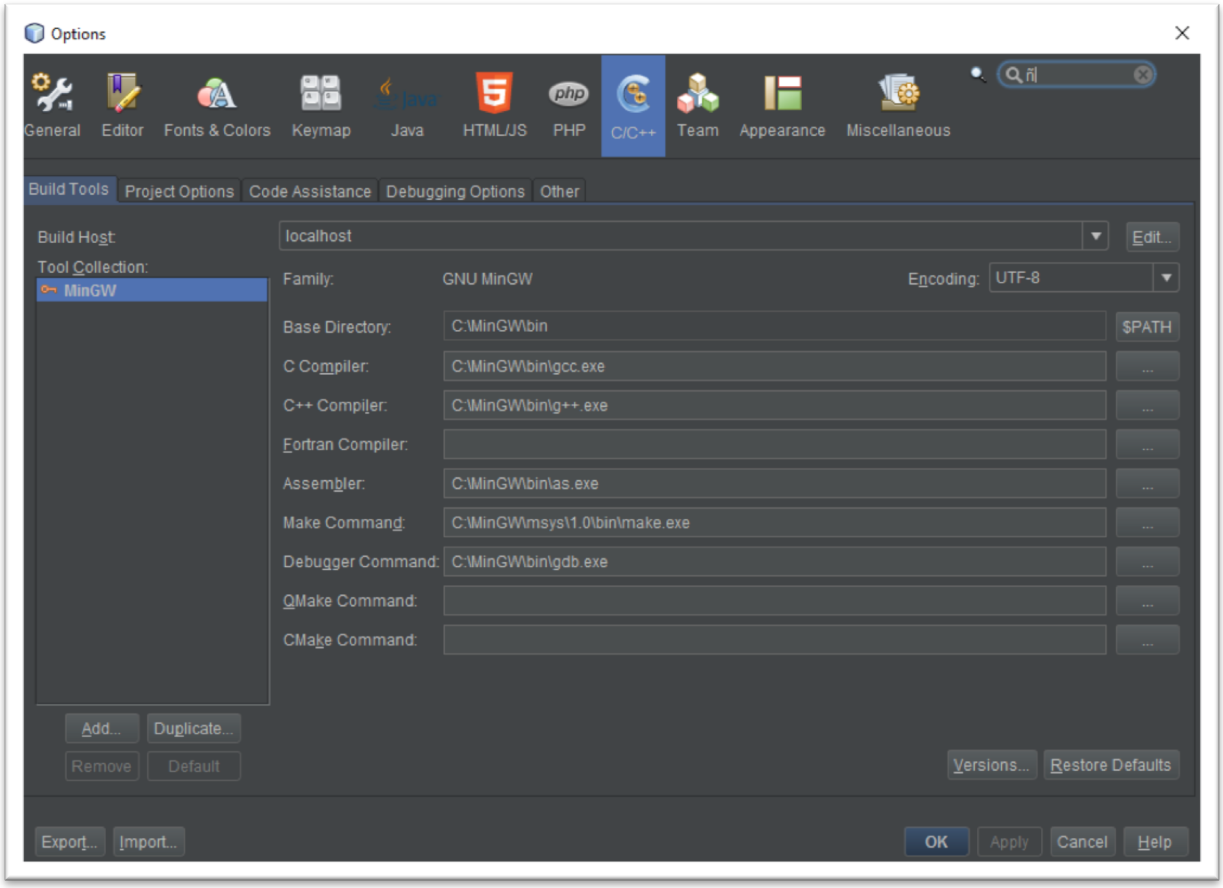




En la parte inferior Izquierda le damos al botón Add y se nos desplegara la siguiente pantalla.



Dando click al botón Browse, buscamos la ruta donde se instalo el MinGw quedando de esta manera:


Options
General
Editor
Fonts & Colors
Keymap
Java
HTML/JS
PHP
C/C++
Team
Appearance
Miscellaneous
Build Tools
Project Options
Code Assistance
Debugging Options
Other
Build Host:
localhost
Edit...
Tool Collection:
MinGW
Family:
GNU MinGW
Encoding:
UTF-8
Base Directory:
C:\MinGW\bin
$PATH
C Compiler:
C:\MinGW\bin\gcc.exe
C++ Compiler:
C:\MinGW\bin\g++.exe
Fortran Compiler:
Assembler:
C:\MinGW\bin\as.exe
Make Command:
C:\MinGW\msys\1.0\bin\make.exe
Debugger Command:
C:\MinGW\bin\gdb.exe
QMake Command:
CMake Command:
Add...
Duplicate...
Remove
Default
Versions...
Restore Defaults
Export...
Import...
OK
Apply
Cancel
Help