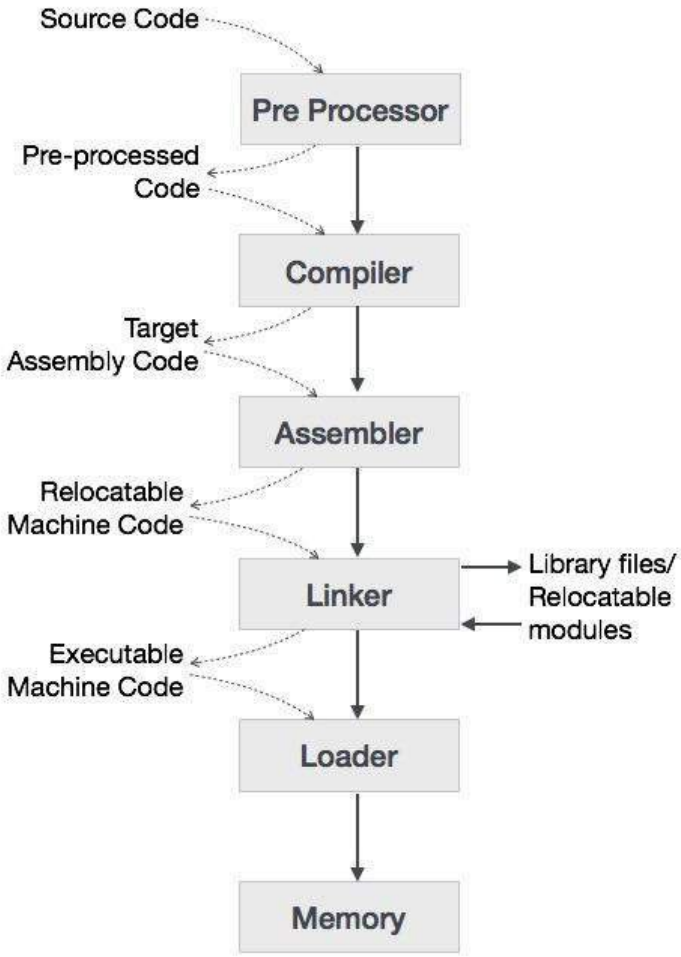
Ensamblador orientado a la creación y análisis de Malware

Para poder realizar análisis de malware es importante conocer los procesos por los que pasa el código del software hasta llegar a conseguir los archivos binarios. Entender estas transformaciones nos ayudara en a la hora de aplicar ingeniería inversa para analizar las muestras.

El siguiente diagrama muestra una visión simplificada de las fases.



Se pueden clasificar los lenguajes de la siguiente forma, en orden de abstracción:

- Lenguajes Interpretados
- Lenguajes de alto nivel
- Lenguajes a bajo nivel
- Código maquina (opcodes)
- Microcódigo (firmware)
- hardware

Siendo estos últimos los menos portables, al estar cada vez mas ligados a la arquitectura del ordenador.

Para el presente informe nos interesa especialmente los archivos en ensamblador, ya que es el lenguaje de más alto nivel que se puede recuperar de un código maquina, de forma fiable y consistente

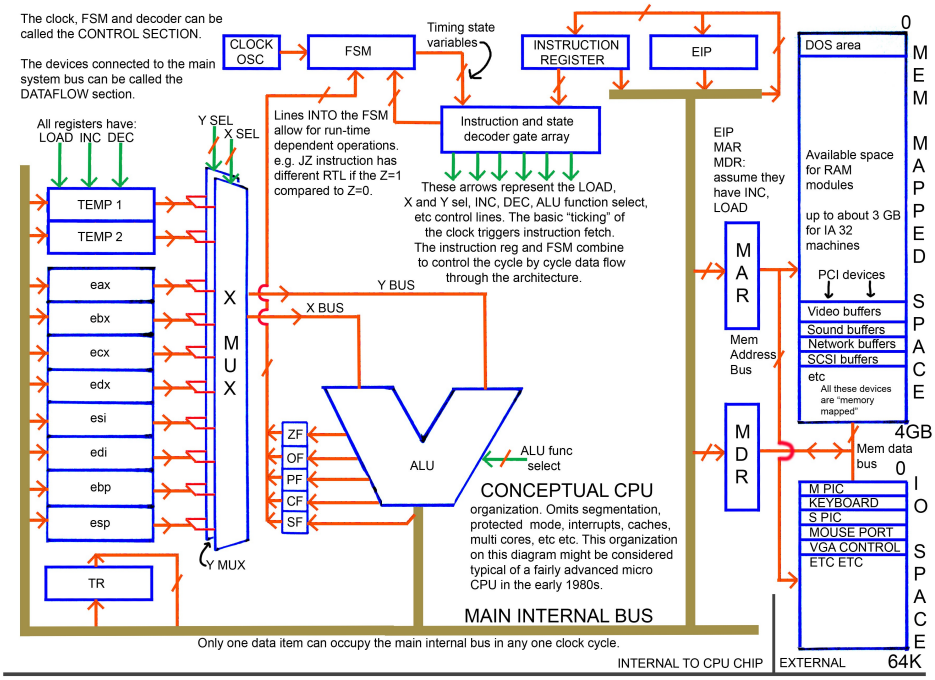
Cada dialecto de ensamblador se asocia a una familia de microprocesadores (x86, x64, SPARC, PowerPC, MIPS...) Siendo x86 la arquitectura mas común en pcs.

Objetivo del proyecto:
Vamos a dar un pequeño repaso de la arquitectura x86 para conocer el terreno de juego. A continuación un resumen del formato de los archivos ejecutables en windows ya que son lo que queremos crear. Finalmente realizaremos una breve introducción a MASM, lenguaje que utilizaremos de herramienta para la creación de los scripts de muestra adjuntos.

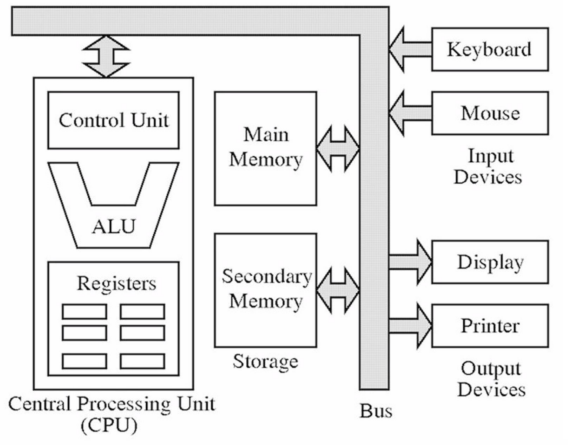
x86

Un gran porcentaje del malware hoy en día se compila para x86 (32 bits) y parece razonable para empezar a aprender ensamblador y el análisis de malware.

La siguiente imagen muestra gran detalle sobre la arquitectura del micro x86



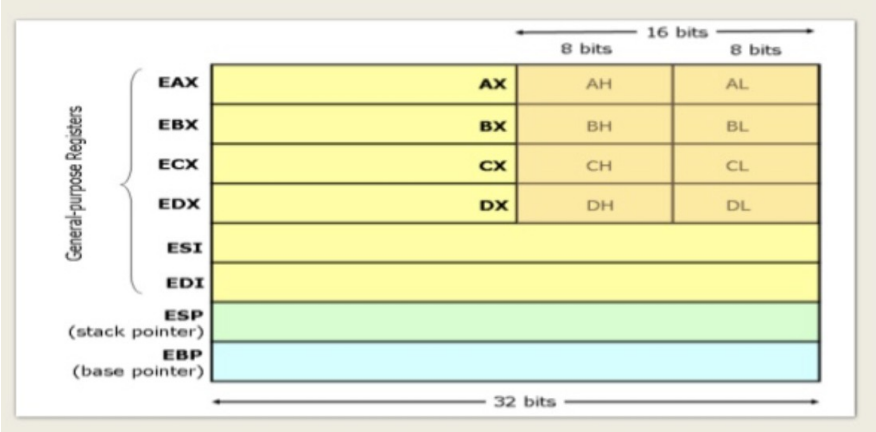
Simplifiquemos un poco el diagrama y vayamos añadiendo elementos y conceptos. Estamos tratando con una arquitectura de Vonn Neumann:



Esta arquitectura tiene disponibles varios tipos de registro:

- generales
- de segmento
- de estado
- puntero de instrucción
-

Remarquemos los que nos resulta imprescindibles conocer.



Tenemos 8 registros de propósito general. Aunque no todos se deben usar libremente ya que la cpu hace uso de ellos en ciertas situaciones.

- EAX se utiliza para almacenar el valor devuelto por las funciones
- EBP para la dirección en la pila donde empieza el frame de memoria de la funciona actual
- ESP apunta al final de la pila

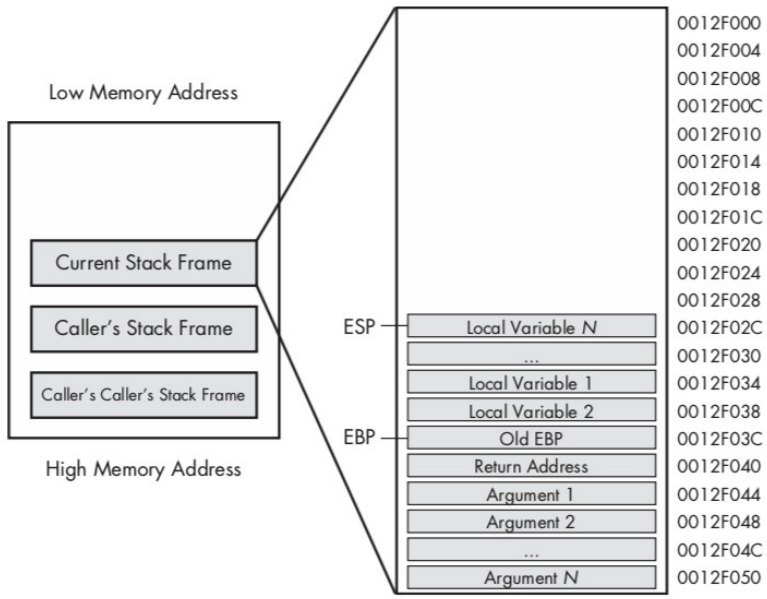
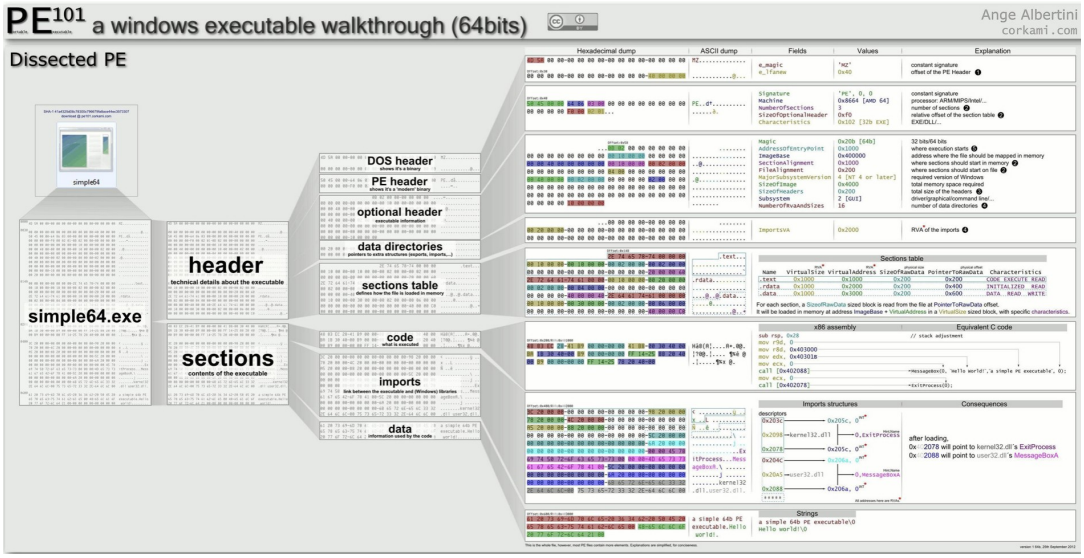


Figure 4-8: Individual stack frame

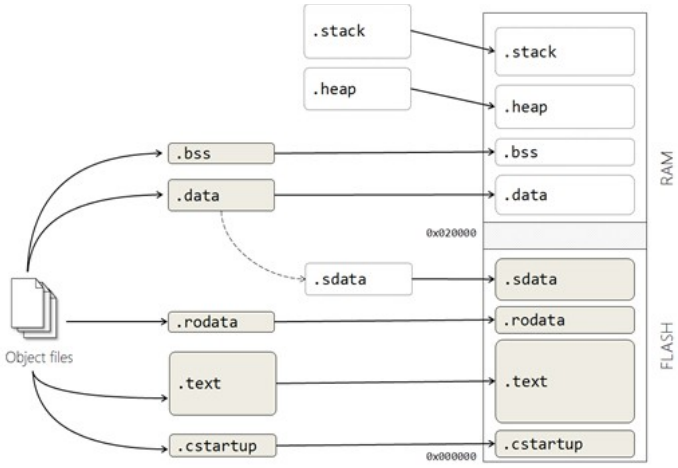
PE formato (Portable Executable)

El formato de los programas en Windows se denomina Portable Ejecutables (abreviados como PE). Estos archivos tienen un formato estructurado y contiene la información necesaria para que Windows cargue, lance y manipule el código ejecutable.
Resulta interesante tener una idea de su estructura tanto para poder construir nuestros pequeños programas en ensamblador, como para poder entender aquellas muestras que queramos analizar posteriormente.

Los archivos PE (tanto dll como ejecutables) se caracterizan por comenzar su cabecera con los caracteres 0x4D5a que corresponde en ascii a las siglas “MZ”.



Entender todo el formato del archivo PE es complicado y se sale del objetivo del proyecto. Nos interesan las secciones que lo forman



A continuación un pequeño resumen de las secciones existentes, su contenido y uso. Necesario comentar que los nombres pueden variar.

- .text: contiene las instrucciones a ejecutar.
- .rdata: contiene información de importaciones y exportaciones.
- .data: contiene datos globales
- .rsrc: contiene recursos usados por el ejecutable.
- .stack: bien conocida pila
- .heap: memoria dinámica a disposición del usuario.

Los archivos PE son compilados con una dirección base predefinida para cargarse en memoria.

Microsoft Assembler (comúnmente conocido como **MASM**)

MASM es un lenguaje ensamblador para windows que utiliza sintaxis intel. Resulta de los mas fáciles para comenzar, así que será el elegido para este proyecto.

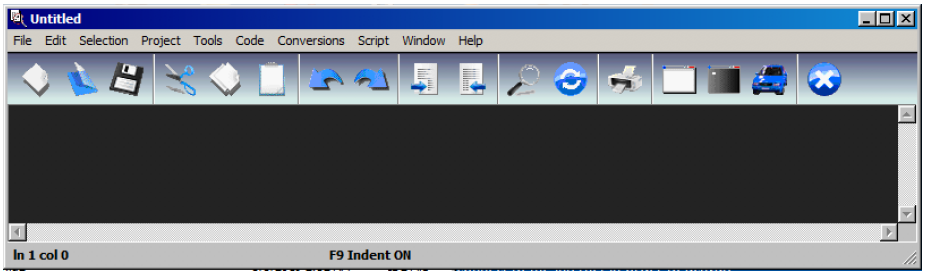
A partir de ahora, nuestra pagina de referencia será MSDN donde podemos encontrar todas las funciones que Microsoft ha querido documentar, con parámetros de entrada, valor de salida y requisitos como las librerías que necesitamos para poder utilizar cada función.

Aclaración sobre funciones parecidas en msdn. Podremos encontrar funciones que tienen el mismo nombre pero que acaban con sufijos diferente como:

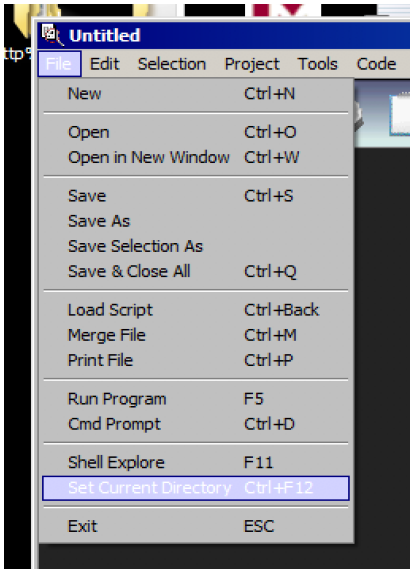
- Ex: cuando Microsoft quiere actualizar una función, y la nueva versión resulta incompatible con la existente, para continuar dando soporte a la versión actual de la función Microsoft cambia el nombre a la nueva añadiendo este sufijo “Ex”
- A: ansi version
- W: unicode version

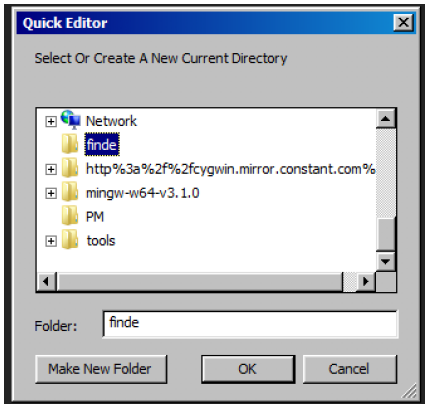
Se recomienda ignorar estos sufijos para buscar la referencia en msdn y luego utilizar el que mas se ajuste a nuestras necesidades.

Vamos a utilizar **MASM32**, una herramienta gratuita y ampliamente extendida, para poder realizar el ensamblado y enlazado de nuestras pruebas de concepto.



NOTA IMPORTANTE: Es necesario indicar a esta herramienta cuál es el directorio actual cuando se realice ensamblado y enlazado, o dará errores y problemas al no poder encontrar los archivos que ella misma crea, y que se encontraran en su directorio home por defecto.
Para esto solo necesitamos ir a “File -> Set Current Directory” como se muestra en la imagen





Hand Made SCRIPTS

1.- Hola mundo

Anexos adjuntos con código:

- hello.asm
- helloOneLine.asm

El objetivo de este primer programa es conseguir un ejecutable que imprima el mensaje *"Hola Epasan"* en una ventana de alerta.

El primer paso es definir la arquitectura e inicializar el modelo de memoria a utilizar.

```
.386
.model flat, stdcall
.stack 100h
option casemap :none
```

Asignamos espacio a la stack y especificamos que nuestro programa distinga entre mayusculas y minúsculas en el código.

A continuación definimos las varias secciones en las que se va a dividir el archivo.

- .data
- .code

Para este programa tan sencillo no necesitaremos mas por el momento.

Syntax

```
C++

int MessageBox(
  HWND    hWnd,
  LPCTSTR lpText,
  LPCTSTR lpCaption,
  UINT    uType
);
```

Como siempre para empezar con la lógica propiamente dicha acudimos a MSDN

https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-messagebox

Como podemos ver en los requisitos para esta función, necesitaremos añadir las siguientes referencias:

```
include \masm32\INCLUDE\user32.inc
include \masm32\INCLUDE\kernel32.inc

includelib \masm32\lib\user32.lib
includelib\masm32\lib\kernel32.lib
```

La librería user32 es necesaria para llamar a MessageBox y kernel32 es necesaria para la llamada a ExitProcess que realizaremos para terminar el proceso lanzado
https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/nf-processthreadsapi-exitprocess

En la sección de datos, creamos dos variables globales que van a contener el texto del mensaje y el titulo de la ventana:

```
.data
      hello_world    db 'Hello, Epasan!',0
      world_tittle   db 'Epasan Finde', 0
```

Necesitamos terminar las cadenas de caracteres con el carácter nulo, de ahí el 0 que se puede observar al final de la linea.

Y ahora toca utilizar estos valores para llamar a la función MessageBoxA que cree nuestra ventana mostrando el mensaje de Hola Mundo.

IMPORTANTE: Los argumentos que necesita la función deben meterse en la pila **de derecha a izquierda,** de esta forma, cuando la función los recoja, aparecerán en el orden correcto, ya que como sabemos la stack es una pila LIFO.

```
start:
      push 0                    ; uType - MB_OK (0x00000000L) message box contains one push button: OK.
      push offset world_tittle  ; lpCaption - dialog box title
      push offset hello_world   ; lpText - message to be displayed
      push 0                    ; hWnd - message box has no owner window
      call MessageBoxA
```

Esta misma llamada se podría realizar en una linea como se muestra a continuación: (código completo en archivo adjunto helloOneLine.asm)

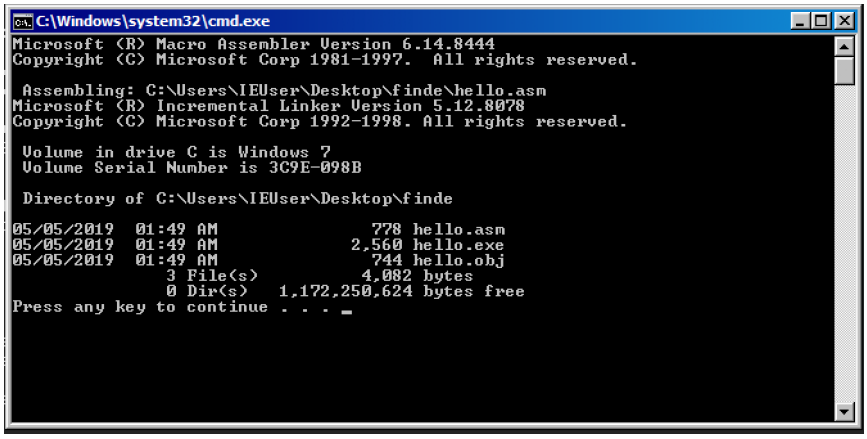
```
start:
      INVOKE MessageBoxA,0,offset hello_world,offset world_tittle,0
      INVOKE ExitProcess,0

end start
```

Para empezar resultaba mas instructivo el primer método, ya que permite entender como funcionan las llamadas a funciones en ensamblador y su interacción con la pila

Aquellos scripts que permitan incluirse en el informe por ser programas cortito se mostraran como imágenes Adicionalmente se adjuntaran al informe el código de todos los scripts como apéndices en archivos .asm independientes, para facilitar su inspección y copia.

```
; reserve space for stack
; symbol names are case-sensitive
; model statement must precede both directive

.386
.model flat, stdcall
.stack 100h
option casemap :none

include \masm32\INCLUDE\user32.inc
include \masm32\INCLUDE\kernel32.inc

includelib \masm32\lib\user32.lib
includelib\masm32\lib\kernel32.lib

.data
      hello_world    db 'Hello, Epasan!',0
      world_tittle   db 'Epasan Finde', 0

.code

start:
      push 0                    ; uType - MB_OK (0x00000000L) message box contains one push button: OK.
      push offset world_tittle  ; lpCaption - dialog box title
      push offset hello_world   ; lpText - message to be displayed
      push 0                    ; hWnd - message box has no owner window
      call MessageBoxA

      push 0
      call ExitProcess

end start
```

Para obtener el ejecutable solo tenemos que ir a “Project -> Console Assemble & Link”
La siguiente ventana aparecerá si no hay ningún problema durante el ensamblado. Como podemos ver se ha creado un archivo *hello.exe*.



Y al ejecutarlo, como esperábamos....

2.- Crear un archivo, modificarlo, esperar y borrarlo.

El objetivo de este programa es conseguir un ejecutable que genere un archivo, espere (para que podamos corroborar su existencia) y a continuación lo borre.

A partir de ahora asumiremos que en todos los scripts se define la arquitectura inicial y el modelo de memoria como en el primero por lo que no hace falta repetirlo.

Vamos a utilizar nuevas funciones para este script:

- CreateFileA
- CloseHandle
- DeleteFileA
- Sleep

El primer paso como siempre será revisar la sintaxis y requisitos en la documentación de MSDN.
https://docs.microsoft.com/en-us/windows/desktop/api/fileapi/nf-fileapi-createfilea

Nos damos cuenta que necesitamos incluir una nueva cabecera windows.h

```
; includes for ExitProcess
include \MASM32\INCLUDE\kernel32.inc
includelib \MASM32\LIB\kernel32.lib

; include needed for CreateFileA and DeleteFileA : fileapi.h (include Windows.h)
include \MASM32\INCLUDE\windows.inc
```

Y a continuación generamos todas las variables globales que vamos a necesitar:

```
.data
    file_handler    dd  0, 0
    file_name       db  ".\temp", 0
    lpFileName      dd  0, 0
    errorCode       dd  0, 0
```

Se utiliza una nueva estructura en ensamblador, los saltos (condicionales en este caso)
Tras la llamada la función para crear el archivo, queremos comprobar si todo resulto como se esperaba, o hubo algún problema. Podría suceder que el proceso no tuviera permisos para crear un archivo en la carpeta indicada por ejemplo.

Revisando el enlace anterior de msdn

Return Value

If the function succeeds, the return value is an open handle to the specified file, device, named pipe, or mail slot.

If the function fails, the return value is **INVALID_HANDLE_VALUE**. To get extended error information, call GetLastError.

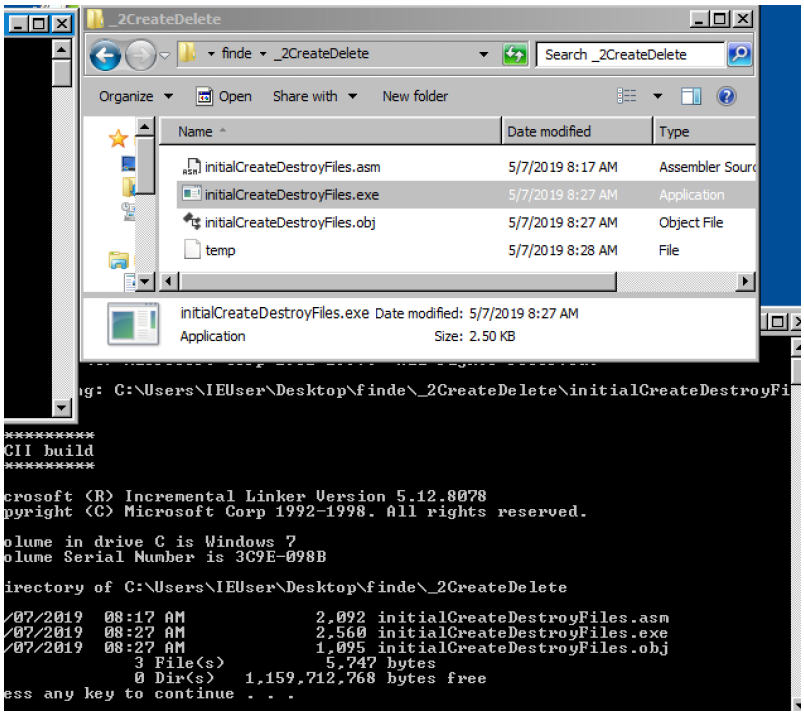
Así que necesitamos comparar el valor devuelto por la función CreateFileA y que se encuentra en el registro EAX con la constante INVALID_HANDLE_VALUE. La instrucción *cmp* pone a 1 el flag ZF cuando ambos valores comparados son iguales. A continuación la instrucción jz realiza el salto en memoria solo si el flag ZF esta levantado.

```
invoke CreateFileA, offset file_name, GENERIC_READ Or GENERIC_WRITE, TRUE, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL

cmp eax, INVALID_HANDLE_VALUE       ; If the function fails, the return value is INVALID_HANDLE_VALUE
jz _clear_leave                     ; If we face a problem creating the file do a clean exit
mov file_handler, eax               ; keep the handler of the file
invoke CloseHandle, file_handler
```

Es decir, si la función CreateFileA tiene algún problema, salta a la etiqueta de *_clear_leave* donde se obtiene el error lanzado, y se podrían realizar las labores de limpieza necesarias

```
invoke Sleep,5000                   ; sleep for 5 seconds so we can check the file was created

invoke DeleteFileA, offset file_name    ; delete file just created
invoke ExitProcess,0
```

A continuación utilizamos la llamada Sleep del sistema para para la ejecución del programa durante 5000ms mientras comprobamos que el archivo se genera, con el nombre “temp” en la misma carpeta en la que ha lanzado el ejecutable



Se ha utilizado una variable global con la cadena “./temp” aunque dicho valor podría ser recogido como parámetro de entrada al programa, o leído de una key del registro. Posteriormente veremos como acceder a keys del registro de forma programática

El código total del script para este punto se vería así:

```
;.....................................
; 2.- Program that creates and deletes files
;.....................................

; reserve space for stack
; symbol names are case-sensitive
; model statement must precede both directive

.386
.model flat, stdcall
;.stack 100h ; to find out whats the use of this
option casemap :none

; includes for ExitProcess
include \MASM32\INCLUDE\kernel32.inc
includelib \MASM32\LIB\kernel32.lib

; include needed for CreateFileA and DeleteFileA : fileapi.h (include Windows.h)
include \MASM32\INCLUDE\windows.inc


.data
    file_handler    dd  0, 0            ; Define double word. Generally 4 bytes on a typical x86 32-bit system
    file_name       db  ".\temp", 0     ; declares 7 bytes starting at the address of 'file_name'
    lpFileName      dd  0, 0
    errorCode       dd  0, 0

.code

start:
    ; Creates a new file, always.
    ; If the specified file exists and is writable, the function overwrites the file, the function succeeds, and last-error code is set to ERROR_ALREADY_EXISTS (183).
    ; If the specified file does not exist and is a valid path, a new file is created, the function succeeds, and the last-error code is set to zero.
    invoke CreateFileA, offset file_name, GENERIC_READ Or GENERIC_WRITE, TRUE, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL

    cmp eax, INVALID_HANDLE_VALUE       ; If the function fails, the return value is INVALID_HANDLE_VALUE
    jz _clear_leave                     ; If we face a problem creating the file do a clean exit
    mov file_handler, eax               ; keep the handler of the file
    invoke CloseHandle, file_handler


    invoke Sleep,5000                   ; sleep for 5 seconds so we can check the file was created

    invoke DeleteFileA, offset file_name    ; delete file just created
    invoke ExitProcess,0

_clear_leave:
    invoke GetLastError
    mov errorCode, eax                  ; Could be used to promp with error message
    invoke CloseHandle, file_handler
    invoke ExitProcess,0

end start
```

Se debería incluir mas control de errores y de limpieza de datos. Se recomiendo añadir en el momento que esta funcionalidad se utilice como una función independiente utilizada por un código más interesante.

Esta función nos seria útil para dejar una nota de rescate tras lanzar un ransomware ;)

3.- Comprobar conectividad, llamar a una url y obtener el cuerpo de la respuesta... en una función separada.

El siguiente scripts utilizara las funciones de WinINet para en primer lugar comprobar que la maquina donde se ejecuta el proceso tiene conexión a internet y a continuación realizar una llamada a una url estática y leer el contenido de la respuesta.

```
include \masm32\INCLUDE\wininet.inc
includelib\masm32\lib\wininet.lib
```

Necesitamos añadir estas librerías. Realizaremos las siguientes llamadas (en este orden) a funciones que podemos (como de costumbre) consultar en la documentación de msdn:

1. InternetGetConnectedState
2. InternetOpenA
3. InternetOpenUrlA
4. InternetReadFile

Llamamos a la función getValueFromUrl:

```
getValueFromUrl PROC resp:DWORD

    local my_local:DWORD    ; This part is not useful. Just to distract
    mov my_local, 33        ; added so we can see difference between parameter and local variables
                            ; when reversing

    invoke InternetGetConnectedState, 0,0  ; dwReserved parameter is reserved and must be 0.
    cmp eax, 0
    jz _no_internet  ; if return value is 0, we do NOT have internet

    invoke InternetOpenA, offset lpszAgent, 1, 0, 0, 0
    mov [internet_handle], eax
    cmp internet_handle, 0
    je _exit_clean

    invoke InternetOpenUrlA,internet_handle, offset hidden_url ,0,0,INTERNET_FLAG_RELOAD,0
    mov [url_handle], eax
    cmp url_handle, 0
    je _exit_clean

    invoke InternetReadFile, url_handle, resp, 100 , offset bytes_read
    cmp eax, 0
    je _exit_clean

    invoke InternetCloseHandle,  [internet_handle]
    invoke InternetCloseHandle,  [url_handle]
    ret

getValueFromUrl ENDP

_no_internet:
    no_connection       db    'Sorry, there is no connectivity',0
    invoke MessageBoxA,0,offset no_connection,offset 0,0
    jmp _exit_clean
```

Recibe un parámetro Este parámetro es un puntero a un buffer donde se almacenara el contenido de la respuesta recibida de la url
Necesitamos definir ciertas variables que esta función va a utilizar:

```
.data

    hidden_url          db    'http://192.168.119.26:83/hiddenFile.txt', 0
    lpszAgent           db    'fake Agent 1.1',0

    bytes_read          dd  ?

    internet_handle     dd  ?
    url_handle          dd  ?
    response            db 100 DUP(0)
.code
```

También se puede observar una variable local definida *my_local.* Esta variable no se usa en este momento, Se ha añadido por motivos instructivos para mostrar diferencia de variables locales y globales al referenciarse, y entre parámetros y variables locales en la pila de memoria.

Al haber separado este código en una función propia, el código del programa se queda en algo muy sencillo: realiza una llamada a este método y muestra en una ventana de alerta el contenido devuelto por la función anterior

```
start:
    invoke getValueFromUrl, offset response
    invoke MessageBoxA,0,offset response,offset 0,0

_exit_clean:
    invoke ExitProcess,0

end start
```

Para probarlo, levantamos un servidor sencillo con python en otra maquina, y le añadimos un archivo que nos proporcionara la respuesta:

```
root@kali:~/finde# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.119.26  netmask 255.255.255.0  broadcast 192.168.119.255
        inet6 fe80::a00:27ff:febe:da8a  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:be:da:8a  txqueuelen 1000  (Ethernet)
        RX packets 6  bytes 1862 (1.8 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1744  bytes 290512 (283.7 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 114  bytes 8670 (8.4 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 114  bytes 8670 (8.4 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@kali:~/finde# ls
hiddenFile.txt
root@kali:~/finde# vim hiddenFile.txt
root@kali:~/finde# cat hiddenFile.txt
arhiddenValueJejeje
root@kali:~/finde# python -m SimpleHTTPServer 83
Serving HTTP on 0.0.0.0 port 83 ...
```

Así que cuando se ejecuta el programa, podemos ver:

4.- Crear una Key en el registro (para guardar el valor del script anterior). Usar un valor de key encodeado.

Las claves de registro se pueden utilizar como mutex en la creación de malware. Para nuestro caso particular simplemente vamos a crear una key fija con un valor especifico si no existe. Posteriormente se integrara con los scripts anteriores.

Vamos a utilizar dos Funciones principalmente que buscamos en MSDN para entender como llamarlas, que requisitos necesitan, y que valores y errores nos pueden devolver

Syntax

C++ Copy

```
LSTATUS RegCreateKeyExA(
  HKEY                        hKey,
  LPCSTR                      lpSubKey,
  DWORD                       Reserved,
  LPSTR                       lpClass,
  DWORD                       dwOptions,
  REGSAM                      samDesired,
  const LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  PHKEY                       phkResult,
  LPDWORD                     lpdwDisposition
);
```

Al contrario que la función RegOpenKeyEx, RegCreateKeyEx crea la key especificada si la misma no existe en el registro.

Para darle un valor a la clave usaremos:

Syntax

C++ Copy

```
LSTATUS RegSetValueExA(
  HKEY       hKey,
  LPCSTR     lpValueName,
  DWORD      Reserved,
  DWORD      dwType,
  const BYTE *lpData,
  DWORD      cbData
);
```

Al igual que en el script anterior lo agruparemos todo en una función para que nos resulte mas reutilizable y fácil de reunir todo en un ultimo programa recopilatorio.

```
setKey PROC value:DWORD
    invoke RegCreateKeyExA,HKEY_CURRENT_USER,offset key_name,0,0,REG_OPTION_NON_VOLATILE,KEY_ALL_ACCESS,0,offset phkResult,offset hRegistryResponse
    cmp eax,ERROR_SUCCESS
    jnz _back       ; jump if not 0 == jump if not equal

    invoke RegSetValueExA,phkResult,offset key_name_2,0,REG_SZ,value, 12 ; we'll use a fixed value but length should be calculated or parameter
    cmp eax, ERROR_SUCCESS
    jnz _back

    ret

_back:
    invoke MessageBoxA,0,offset handle_problem,offset 0,0
    ret

setKey ENDP
```

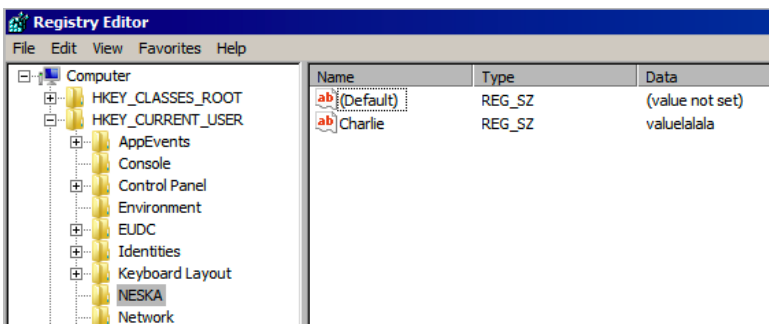
Así la función queda bastante sencilla, aunque ambas llamadas tienen muchos parámetros y opciones a las que hay que prestar atención si queremos que la key se genere a nuestro gusto. La sección de datos que necesitamos se quedaría en estas pocas variables:

```
.data
    handle_problem  db "Sorry, there was a problem editing the key",0
    key_name        db "NESKA",0
    key_name_2      db "Charlie",0
    key_value       db "valuelalala", 0
    phkResult           dd  ?
    hRegistryResponse   dd ?
.code
```

Y el método principal, el controlador del programa tan solo llamaría a la función anterior de la siguiente forma:

```
start:
    invoke setKey,offset key_value
    invoke ExitProcess,0

end start
```

El ultimo paso ya es tan solo ejecutarlo y comprobar que se genera lo que esperamos.



5.- Borrando las huellas... Como un ejecutable se puede borrar a sí mismo

Después de haber realizado las acciones, buenas o malas, nos proponemos pasar desapercibidos, desaparecer sin dejar huella aparente.

El objetivo de este programa es conseguir un ejecutable que se borre a sí mismo.

La parte mas importante a destacar de este programa es el uso de la función *ShellExecuteA*. Esta función nos permite mucha flexibilidad y un mundo de posibilidades dentro de la creación de malware. Como siempre, veamos sus requisitos dentro de la documentación de msdn

Syntax

C++ Copy

```
HINSTANCE ShellExecuteA(
  HWND   hwnd,
  LPCSTR lpOperation,
  LPCSTR lpFile,
  LPCSTR lpParameters,
  LPCSTR lpDirectory,
  INT    nShowCmd
);
```

En este caso el problema es que un proceso no se puede borrar a sí mismo mientras se está ejecutando. La belleza de esta solución reside en su simplicidad. El programa lanza una shell, con sólo dos comandos, uno para borrar el archivo ejecutable y el primero un temporizador, que le da tiempo al proceso a terminar de ejecutarse.

```
cmd_exe                 db  "cmd.exe",0
cmd_line                db  "Timeout /T 1 & /C del .\deleteMyself.exe", 0
```

El script completo primero lanza una ventana de alerta, para facilitar la visibilidad de la acción, espera 1 segundo y desde la shell, se borra el archivo ejecutable que la ha lanzado.

La parte principal del código se muestra a continuación, el archivo completo se adjunta para facilitar su uso.

```
; include needed for ShellExecuteA
include \MASM32\INCLUDE\shell32.inc
includelib \MASM32\LIB\shell32.lib

include \MASM32\INCLUDE\windows.inc


.data
    notification_message    db  "Executable file is about to delete itself...", 0
    notification_tittle     db  "delete myself poc", 0
    cmd_exe                 db  "cmd.exe",0
    cmd_line                db  "Timeout /T 1 & /C del .\deleteMyself.exe", 0

.code

start:
    invoke MessageBoxA,0,offset notification_message,offset notification_tittle,0

    ; execute cmd_line in separated shell
    invoke ShellExecuteA, NULL, NULL, offset cmd_exe, offset cmd_line, NULL, SW_SHOWDEFAULT

    invoke ExitProcess,0

end start
```

6.- Poniéndolo todo junto...

Finalmente recopilamos todos los scripts en un programa de mayor tamaño. Dicha muestra se utilizara en el siguiente documento para explicar una metodología que permita realizar reversing de malware de forma ordenada y razonada.

Propuestas de futuro....

Se podrían sugerir muchos mas pequeñas funcionalidades para empezar a tomar contacto con la programación en ensamblador:

- comprobar si estamos en una maquina virtual
- añadir argumentos de entrada
- cifrar los archivos pdf que existan en el directorio en que se ejecuta
- realizar diferentes funciones dependiendo del valor de la url
  - jugar con la estructura de mapa en memoria equivalente del switch
- comprobar si hay algún callback hook cuando se conecte un usb, y copiarse a sí mismo dentro de la unidad recién conectada
- control de errores y comprobación de valores devueltos por las funciones utilizadas
- limpieza de handlers
- ....

Se dejan como ejercicio al lector, y posiblemente al autor de este documento como objetivo para un futuro..