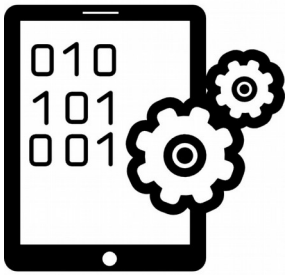
Ingeniería Inversa
Clase 3

Binarios Ejecutables

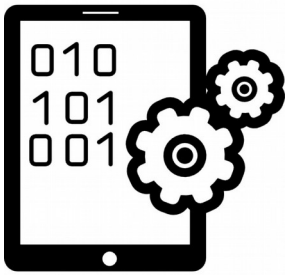
martin.uy
Open by default.

Análisis de Binarios



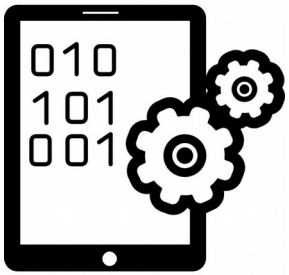
- Análisis estático basado en el formato del ejecutable
  - Funciones y variables exportadas
  - Funciones y variables importadas
  - Tablas de símbolos y strings
  - Información de debug
- Pero, ¡no todo está exportado ni tiene símbolos!

Análisis de Binarios



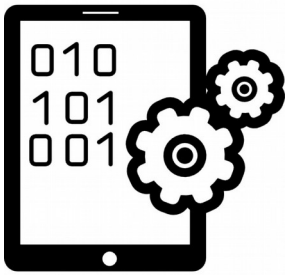
- Al compilar y linkear se pierde información
  - Nombres de funciones, variables, comentarios
  - Tipos de las variables
  - Ubicación de funciones no exportadas (estáticas) e información de relocalización
  - Parámetros de las funciones
  - Parte de esta pérdida puede ser intencional: strippear un binario en modo release
  - Compilar es una operación muchos-a-muchos
    - Mismo código assembly, diferente código fuente (o viceversa)

Análisis de Binarios



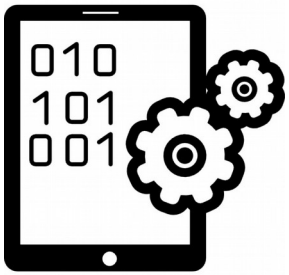
- Análisis estático del código ejecutable
  - Heurísticas de desensamblado
  - Identificación de funciones
  - Identificación de parámetros de las funciones
  - Identificación de variables locales y globales
  - Identificación de “basic blocks” (flujo de la función)
  - Identificación de referencias cruzadas
  - ¡Todo esto se puede automatizar!

Análisis de Binarios



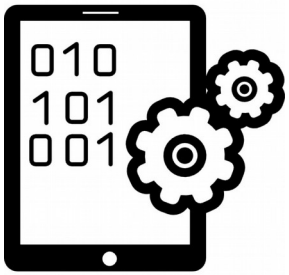
- Heurísticas de desensamblado
  - Linear Sweep
    - Desde un punto de partida (ej. símbolo de una función, comienzo del segmento .text o entry point del binario) se desensambla linealmente
      - Intrucciones y operandos de largo variable pero conocido (x86) o de largo fijo (ARM)
    - Ej. mov, add, push, etc.

Análisis de Binarios



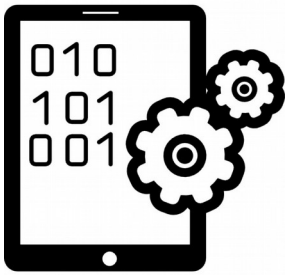
- Heurísticas de desensamblado
  - Recursive Descent
    - Branching condicional (if, while, for, switch)
      - Se desensambla una rama y se marca la otra para desensamblado futuro
    - Branching incondicional (jmp, call)
      - Problema: ¿conocemos el destino del salto?

Análisis de Binarios



- Heurísticas de desensamblado
  - Recursive Descent
    - Branching incondicional (jmp, call)
      - Si lo conocemos, desensamblamos el target. Sino, tenemos un problema.
      - En un call asumimos que existirá un “return” a la dirección siguiente. Por lo tanto, la dirección siguiente queda marcada como pendiente de ser desensamblada.

Análisis de Binarios

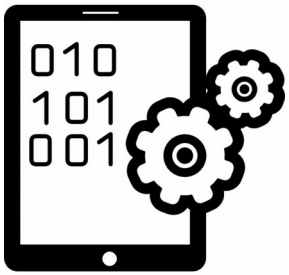


Entry point (conocido) en el stream binario

```
.text:0040101A 68 18 80 41 00
.text:0040101F 8B 45 FC
.text:00401022 50
.text:00401023 FF 15 00 10 41 00
.text:00401029 89 45 F8
.text:0040102C 83 7D F8 00
.text:00401030 74 17
.text:00401032 FF 55 F8
.text:00401035 89 45 F4
.text:00401038 8B 4D F4
.text:0040103B 51
.text:0040103C 68 20 80 41 00
.text:00401041 E8 4A 00 00 00
```

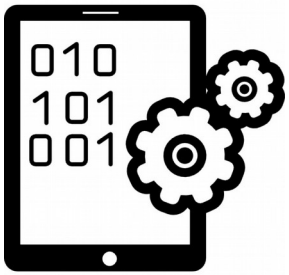
Opcodes y operandos de largo variable pero conocido para la arquitectura

Análisis de Binarios




```
.text:0040101A 68 18 80 41 00        push    offset ProcName ;
.text:0040101F 8B 45 FC              mov     eax, [ebp+hModule]
.text:00401022 50                    push    eax             ;
.text:00401023 FF 15 00 10 41 00     call    ds:GetProcAddress
.text:00401029 89 45 F8              mov     [ebp+var_8], eax
.text:0040102C 83 7D F8 00           cmp     [ebp+var_8], 0
.text:00401030 74 17                 jz      short loc_401049
.text:00401032 FF 55 F8              call    [ebp+var_8]
.text:00401035 89 45 F4              mov     [ebp+var_C], eax
.text:00401038 8B 4D F4              mov     ecx, [ebp+var_C]
.text:0040103B 51                    push    ecx
.text:0040103C 68 20 80 41 00        push    offset aReturnD ;
.text:00401041 E8 4A 00 00 00        call    sub_401090
```

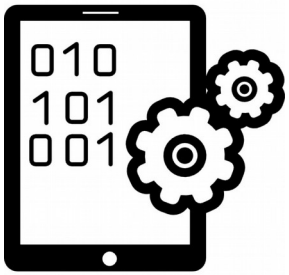
Análisis de Binarios




```
.text:0040101A 68 18 80 41 00        push    offset ProcName ;
.text:0040101F 8B 45 FC              mov     eax, [ebp+hModule]
.text:00401022 50                    push    eax             ;
.text:00401023 FF 15 00 10 41 00     call    ds:GetProcAddress
.text:00401029 89 45 F8              mov     [ebp+var_8], eax
.text:0040102C 83 7D F8 00           cmp     [ebp+var_8], 0
.text:00401030 74 17                 jz      short loc_401049
.text:00401032 FF 55 F8              call    [ebp+var_8]
.text:00401035 89 45 F4              mov     [ebp+var_C], eax
.text:00401038 8B 4D F4              mov     ecx, [ebp+var_C]
.text:0040103B 51                    push    ecx
.text:0040103C 68 20 80 41 00        push    offset aReturnD ;
.text:00401041 E8 4A 00 00 00        call    sub_401090
```

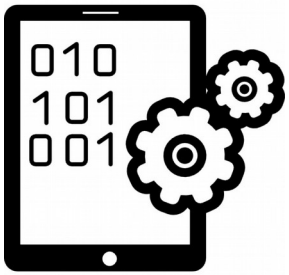
¿Dónde continuar desensamblando? CALL a variable local, solo conocido en tiempo de ejecución

Análisis de Binarios



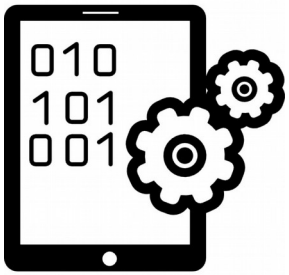
- En arquitecturas CISC como x86/x86_64 (con sets de instrucciones extendidos), muchos opcodes pueden ser válidos.
- Sin embargo, no todas las instrucciones son igualmente probables o frecuentes. El tipo de binario ejecutable nos puede dar pistas: ¿estamos esperando instrucciones de punto flotante?
- ¿Podemos diferenciar un binario ejecutable escrito en assembly a mano de uno generado por un compilador? ¿Podemos identificar idioms o patrones?

Análisis de Binarios



- Los compiladores tienden a utilizar con mayor frecuencia ciertas instrucciones y generan ciertos patrones que siguen convenciones o interfaces binarias (ABIs).
- Es importante poder hacer un juicio acerca de la probabilidad de que un código haya sido desensamblado correctamente.
  - Darle una pista al desensamblador por dónde comenzar a desensamblar.

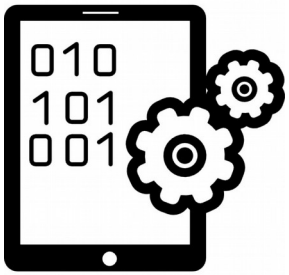
Análisis de Binarios



- ¿Por dónde comenzaríamos a desensamblar?

```
.text:004012FF                  db    0
.text:00401300                  db  89h
.text:00401301                  db  15h
.text:00401302                  db 0D4h
.text:00401303                  db  87h
.text:00401304                  db  41h
.text:00401305                  db    0
.text:00401306                  db 0E8h
.text:00401307                  db    5
.text:00401308                  db 0FFh
.text:00401309                  db 0FFh
.text:0040130A                  db 0FFh
.text:0040130B                  db  83h
.text:0040130C                  db 0F8h
.text:0040130D                  db 0FFh
.text:0040130E                  db  75h
.text:0040130F                  db    5
```

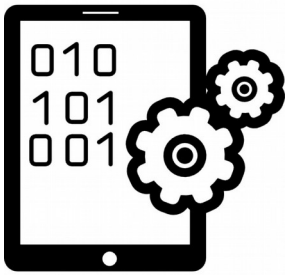
Análisis de Binarios



- Parece bien desensamblado?

```
.text:004012FE                  db  41h ; A
.text:004012FF                  db    0
.text:00401300                  db  89h ; ë
.text:00401301                  db  15h
.text:00401302 ; ---------------------------------------
.text:00401302                  aam     87h
.text:00401304                  inc     ecx
.text:00401305                  add     al, ch
.text:00401307                  add     eax, 83FFFFFFh
.text:0040130C                  clc
.text:0040130D                  push    dword ptr [ebp+5]
.text:00401310                  or      eax, 0FFFFFFFFh
.text:00401313                  jmp     short loc_401370
.text:00401313 ; ---------------------------------------
.text:00401315                  db  6Ah ; j
.text:00401316                  db    0
.text:00401317                  db  6Ah ; j
```

Análisis de Binarios



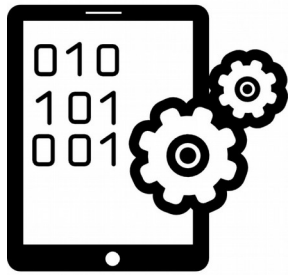
- Parece bien desensamblado?: no ✗

```
.text:004012FE                  db  41h ; A
.text:004012FF                  db    0
.text:00401300                  db  89h ; ë
.text:00401301                  db  15h
.text:00401302 ; ---------------------------------------
.text:00401302                  aam     87h
.text:00401304                  inc     ecx
.text:00401305                  add     al, ch
.text:00401307                  add     eax, 83FFFFFFh
.text:0040130C                  clc
.text:0040130D                  push    dword ptr [ebp+5]
.text:00401310                  or      eax, 0FFFFFFFFh
.text:00401313                  jmp     short loc_401370
.text:00401313 ; ---------------------------------------
.text:00401315                  db  6Ah ; j
.text:00401316                  db    0
.text:00401317                  db  6Ah ; j
```

Instruccion rara: ASCII Adjust AX After Multiply

Los compiladores a veces hacen cosas “tontas” pero no “tan tontas”

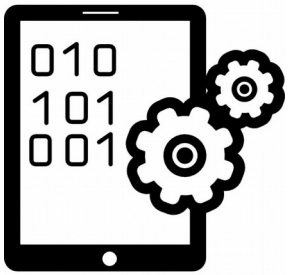
Análisis de Binarios



- Parece bien desensamblado?

```
.text:00401300 ; ---------------------------------------
.text:00401300                 mov     dword_4187D4, edx
.text:00401306                 call    sub_401210
.text:0040130B                 cmp     eax, 0FFFFFFFFh
.text:0040130E                 jnz     short loc_401315
.text:00401310                 or      eax, 0FFFFFFFFh
.text:00401313                 jmp     short loc_401370
.text:00401315 ; ---------------------------------------
```

Análisis de Binarios



- Parece bien desensamblado?: sí 

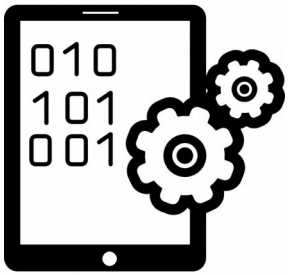
1er parámetro de un call (x86_64 ABI)

Call a una función verificable

Compara contra -1 el resultado de la función

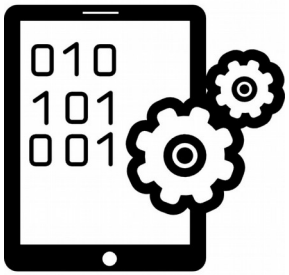
```
.text:00401300 ; ---------------------------------------
.text:00401300                 mov     dword_4187D4, edx
.text:00401306                 call    sub_401210
.text:0040130B                 cmp     eax, 0FFFFFFFFh
.text:0040130E                 jnz     short loc_401315
.text:00401310                 or      eax, 0FFFFFFFFh
.text:00401313                 jmp     short loc_401370
.text:00401315 ; ---------------------------------------
```

Análisis de Binarios



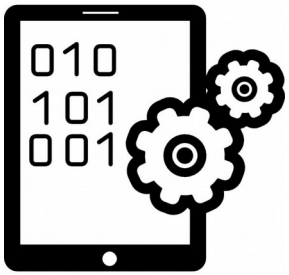
- En los ejemplos anteriores asumimos que el binario no está obfuscado / packeado, y que es assembly legítimo de un compilador.
  - Ejemplo de caso de uso: diff de DLLs o SYS modules para parches de seguridad
  - Para analizar malware hay que tener cuidado con estas suposiciones
- Hay una buena parte de “oficio”

Análisis de Binarios



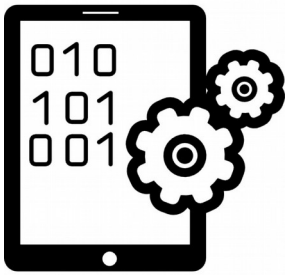
- Identificación de funciones
  - Funciones exportadas
  - Target de instrucciones CALL
  - Epílogos (ABIs)
- Identificación de parámetros de funciones
  - Calling conventions (ej. x86 ABI) para el número de parámetros
  - Instrucciones “mov” para el tamaño

Análisis de Binarios



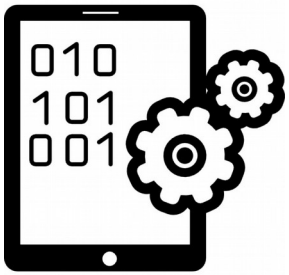
- Identificación de parámetros de funciones
  - Es tarea del reverser determinar:
    - Significado de los punteros
    - Estructuras
      - ¿Cuando se leen y escriben las estructuras? Eso da valor semántico a sus miembros.
    - Tipos de dato
      - Ej: ¿se realizan operaciones de punto flotante sobre un parámetro?

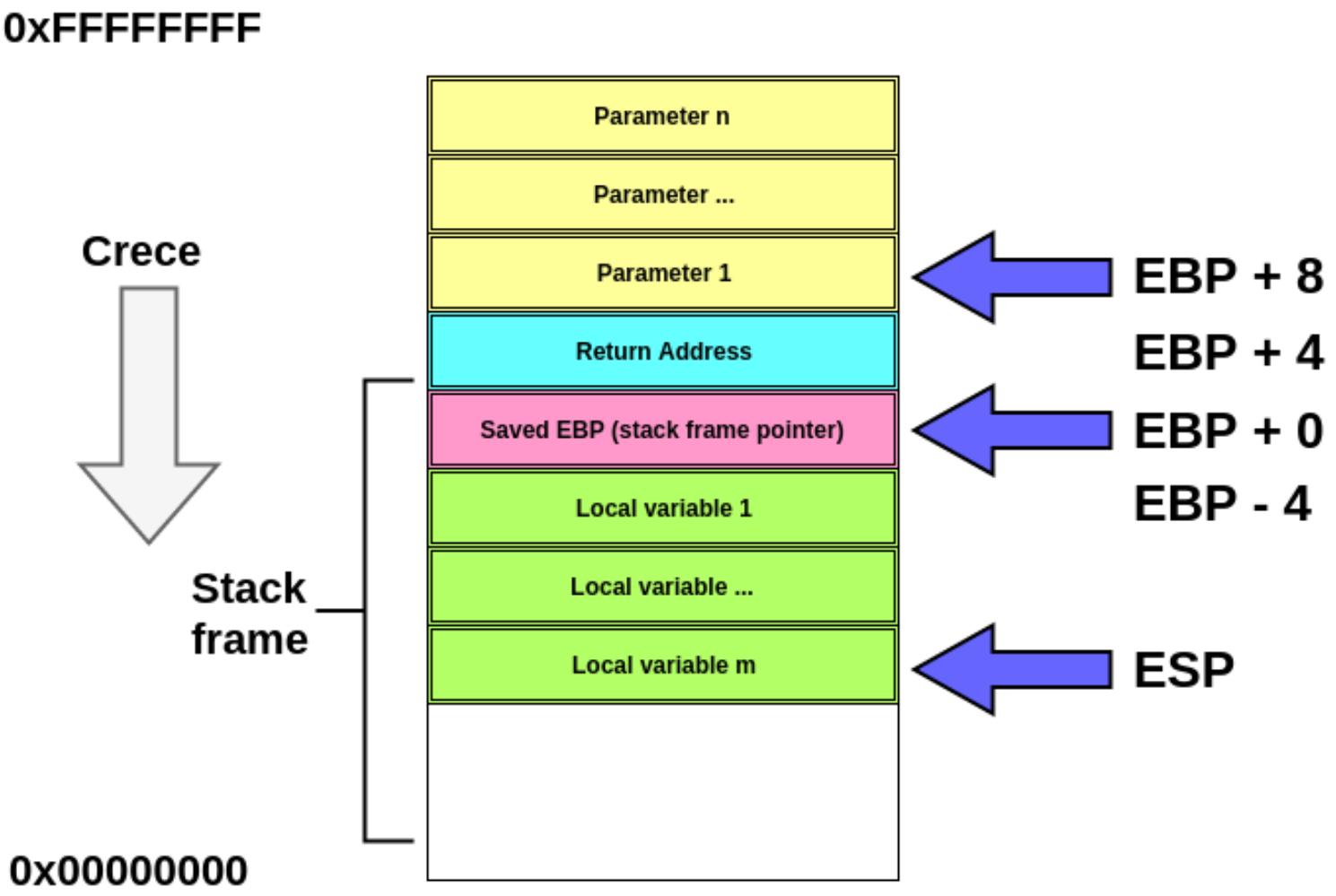
Análisis de Binarios



- Calling conventions – Application Binary Interface (ABI)
- ¿Cómo es llamada una función a nivel assembly?
  - Enviar parámetros (valores, alineación, estructuras)
  - Dirección de retorno
  - Valor de retorno
  - Balance del stack
  - ¿Qué registros se conservan? ¿Quién los conserva?
- Es necesaria una convención: código generado por un compilador puede llamar a librerías generadas por otro compilador.
- Estas convenciones dependen de la arquitectura del CPU y de la plataforma (Windows, Unix, etc.)

Análisis de Binarios



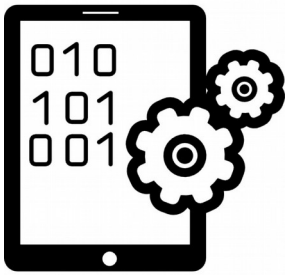


Stack

1 en espacio de usuario por thread

x86

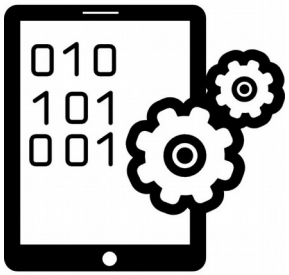
Análisis de Binarios




```
sub_401060 proc near

arg_0= dword ptr  8
arg_4= dword ptr  0Ch
arg_8= dword ptr  10h
arg_C= dword ptr  14h

push    ebp
mov     ebp, esp
mov     eax, [ebp+arg_C]
push    eax
mov     ecx, [ebp+arg_8]
push    ecx
mov     edx, [ebp+arg_4]
push    edx
mov     eax, [ebp+arg_0]
push    eax
```

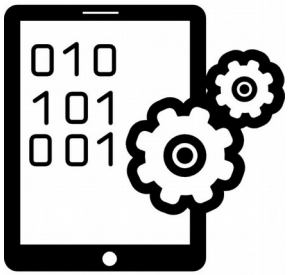
x86

Análisis de Binarios



- Calling conventions x86
  - Cdecl
    - La función que llama balancea el stack (parámetros)
  - Stdcall
    - La función llamada balancea el stack (parámetros)
    - Común en la Windows API
  - Fastcall
    - Parámetros por registros

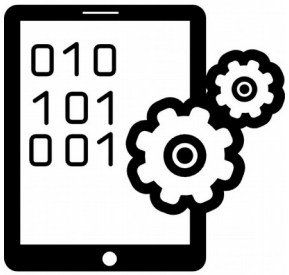
Análisis de Binarios




```
int __stdcall function_a(int p1) { return ++p1; }

int __cdecl function_b(int p1) { return ++p1; }

int __fastcall function_c(int p1) { return ++p1; }

void main(void) {
    printf("function_a: %d\n", function_a(0));
    printf("function_b: %d\n", function_b(1));
    printf("function_c: %d\n", function_c(2));
}
```

MSVC calling conventions

Análisis de Binarios




```
int __stdcall function_a(int p1) { return ++p1; }
```

```
push    0
call    sub_401000
push    eax
push    offset aFunction_aD
call    sub_4010F0
```

Parámetro 1 pusheado al stack

function_a

No se balancea el stack, el llamado lo hizo

printf

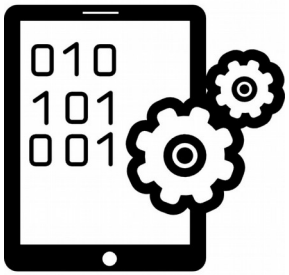
Función main

```
mov     eax, [ebp+arg_0]
pop     ebp
retn    4
```

El llamado balancea el stack, liberando el espacio utilizado para el parámetro

Función function_a

Análisis de Binarios




```
int __cdecl function_b(int p1) { return ++p1; }
```

```
push    1
call    sub_401020
add     esp, 4
push    eax
push    offset aFunction_bD ; "function_b\n"
call    sub_4010F0
```

Parámetro 1 pusheado al stack

function_b

Se balancea el stack, liberando el espacio utilizado para el parámetro

printf

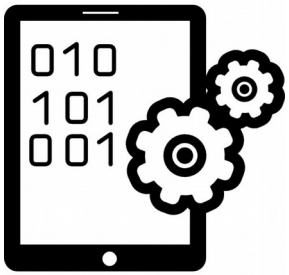
Función main

```
mov     eax, [ebp+arg_0]
pop     ebp
retn
```

El llamado no balancea el stack

Función function_b

Análisis de Binarios




```
int __fastcall function_c(int p1) { return ++p1; }
```

```
mov     ecx, 2
call    sub_401040
push    eax
push    offset aFunction_cD
call    sub_4010F0
```

Parámetro 1 cargado en registro

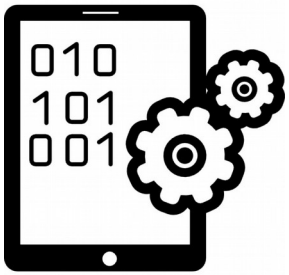
function_c

No hay desbalance del stack

printf

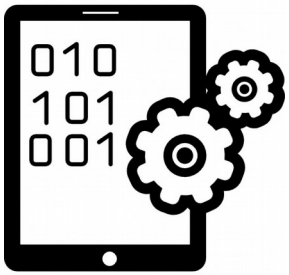
Función main

Análisis de Binarios



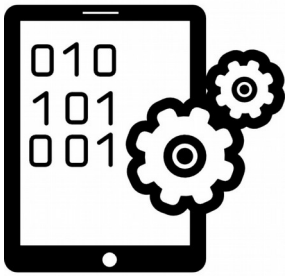
- Identificación de variables
  - Igual a la identificación de parámetros
  - Las variables locales están referidas (en x86) por EBP - offset
    - El compiler podría referirlas por ESP
    - Podrían estar en registros, según el nivel de optimización
  - Las variables globales son referencias a los segmentos .data (inicializadas) y .bss (no-inicializadas)

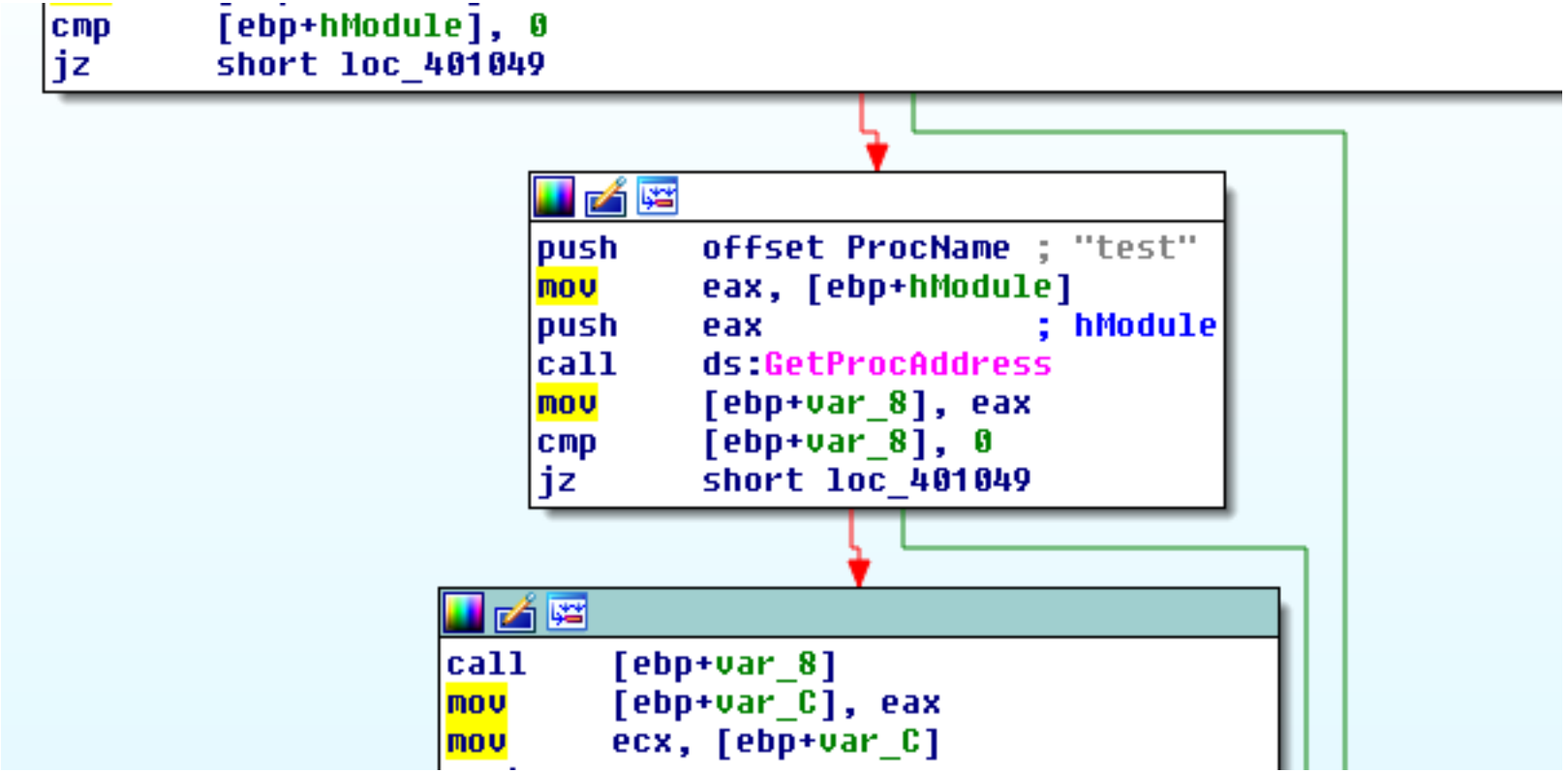
Análisis de Binarios




```
sub_4026F4 proc near

var_C= dword ptr -0Ch
var_8= dword ptr -8
var_1= byte ptr -1
arg_0= dword ptr  8
arg_4= dword ptr  0Ch

mov     edi, edi
push    ebp
mov     ebp, esp
sub     esp, 0Ch
mov     eax, [ebp+arg_0]
lea     ecx, [ebp+var_1]
mov     [ebp+var_8], eax
mov     [ebp+var_C], eax
```

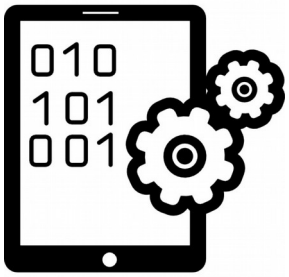
Análisis de Binarios





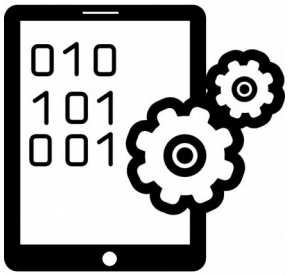
Basic blocks

Análisis de Binarios



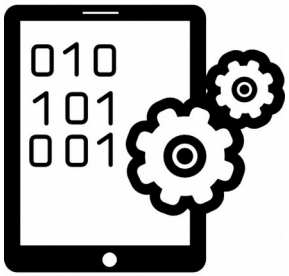
- Identificación de referencias cruzadas
  - Basada en offsets
    - + información de símbolos
    - + valor (ej. String)
  - Búsqueda bidireccional
  - Buena estrategia para “entender” sobre una función

Análisis de Binarios



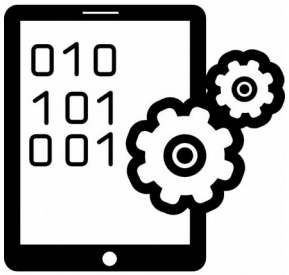

```
push    ebp
mov     ebp, esp
sub     esp, 0Ch
push    offset LibFileName ; "test.dll"
call    ds:LoadLibraryA
```

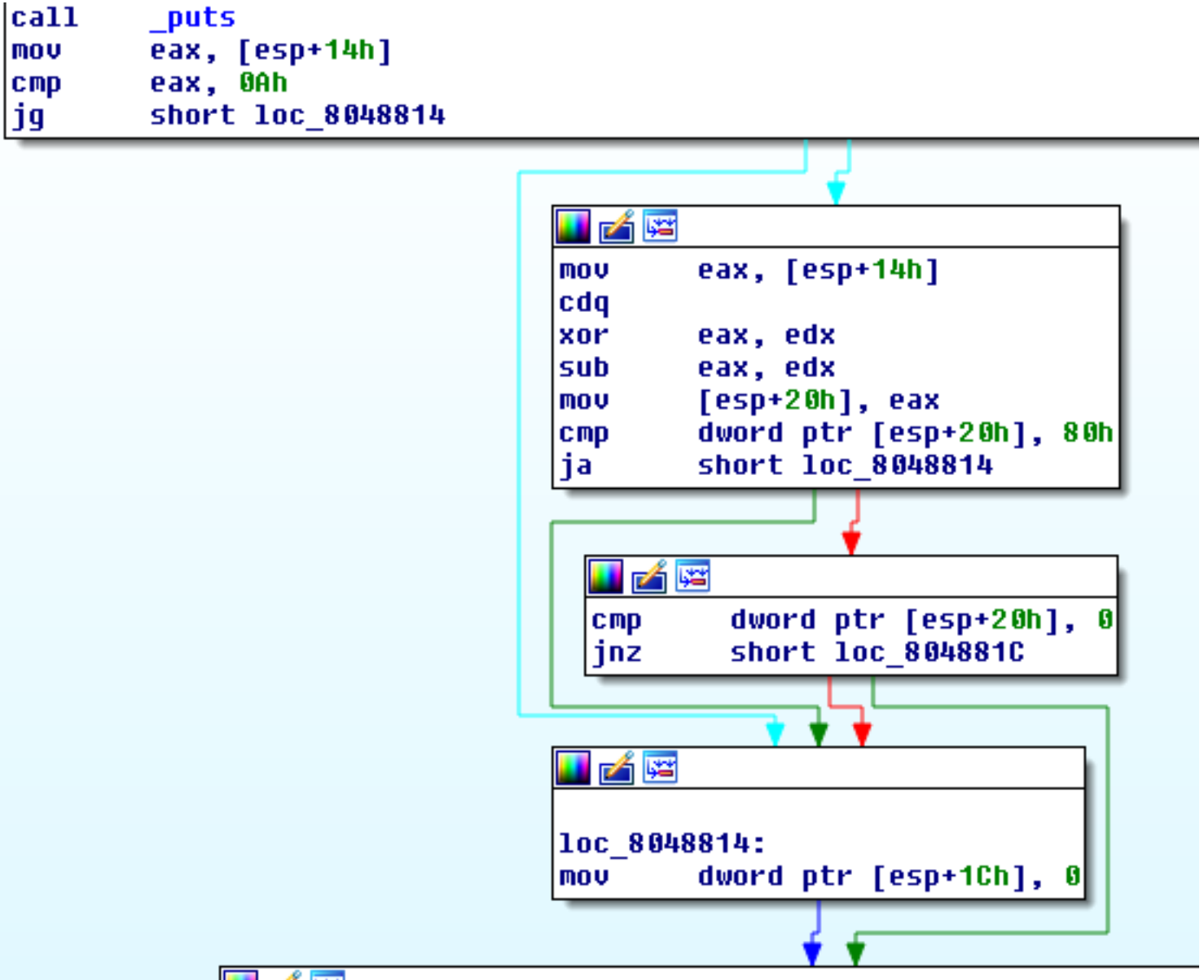
Análisis de Binarios



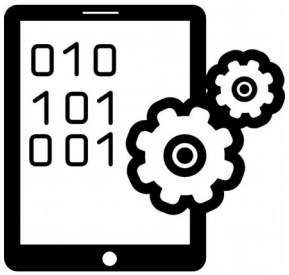
- Identificación de patrones
  - Desde assembly al código fuente
    - Un desensamblador permite identificar los opcodes y mostrar el nemotécnico en assembly
    - Un decompilador hace una abstracción de más alto nivel y permite visualizar código C o pseudocódigo

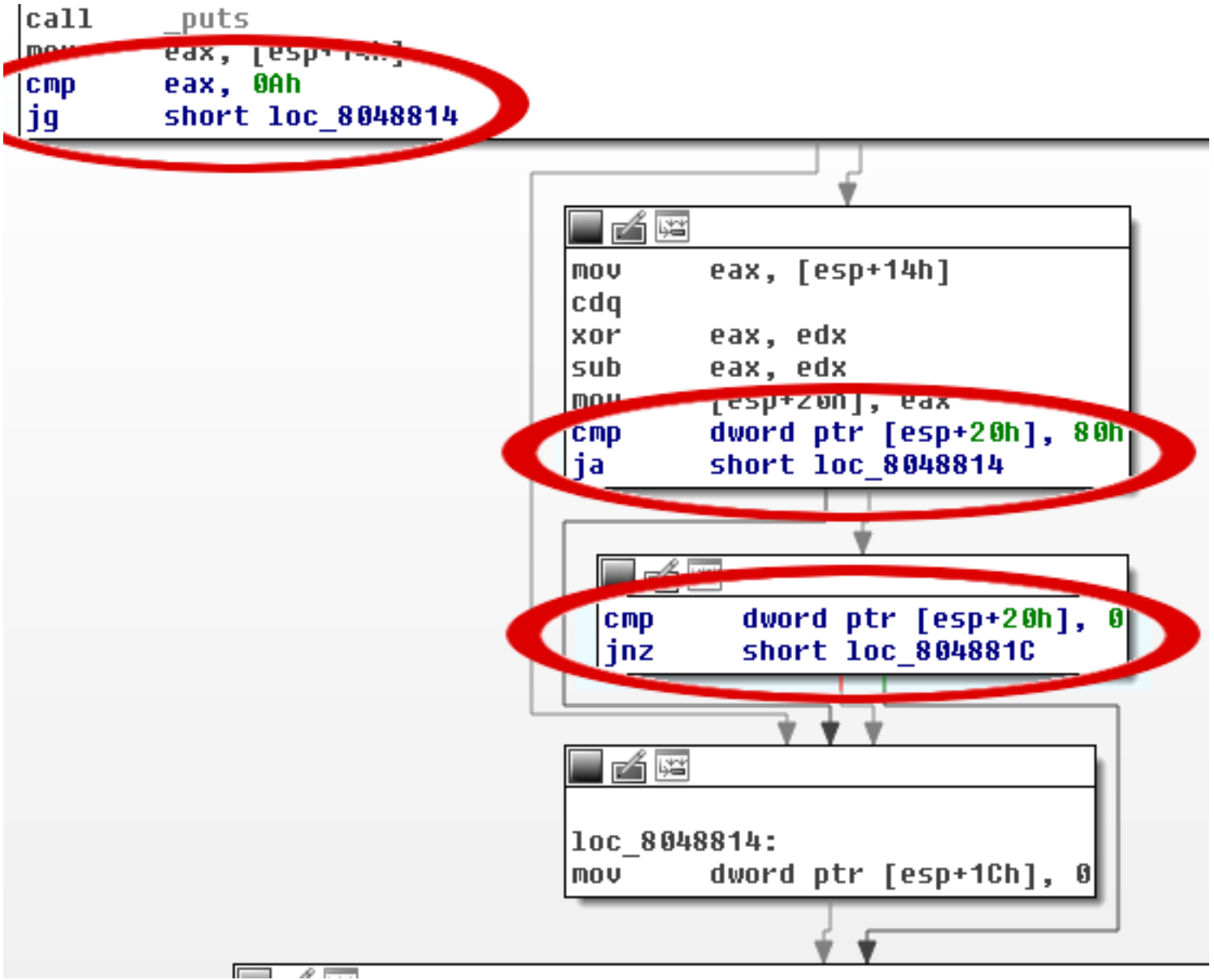
Análisis de Binarios



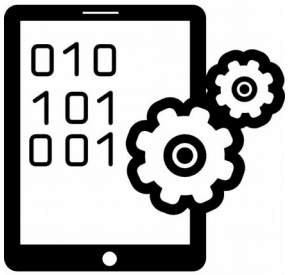


Análisis de Binarios





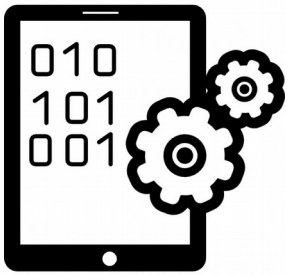
Análisis de Binarios



- Identificación de patrones

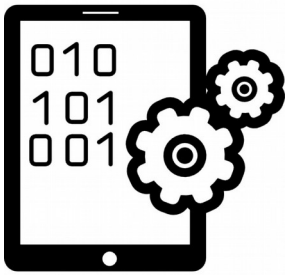
```
if ( condition_1 && condition_2 … && condition_n) {
    do;
    }
```

Análisis de Binarios



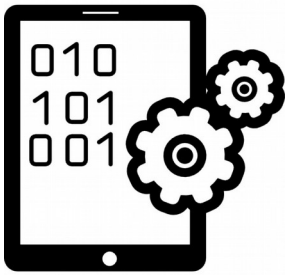

```
loc_80489C4:
lea     eax, [esp+13h]
mov     [esp+4], eax
mov     dword ptr [esp], offset aC ; "%c"
call    ___isoc99_scanf
movzx   eax, byte ptr [esp+13h]
cmp     al, 0Ah
jnz     short loc_80489C4
```

Análisis de Binarios




```
loc_80489C4:
lea     eax, [esp+13h]
mov     [esp+4], eax
mov     dword ptr [esp], offset aC ; "%c"
call    __isoc99_scanf
movzx   eax, byte ptr [esp+13h]
cmp     al, 0Ah
jnz     short loc_80489C4
```

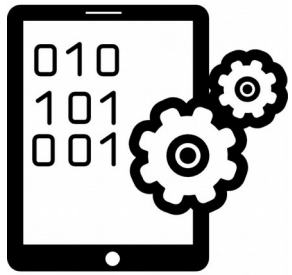
Análisis de Binarios

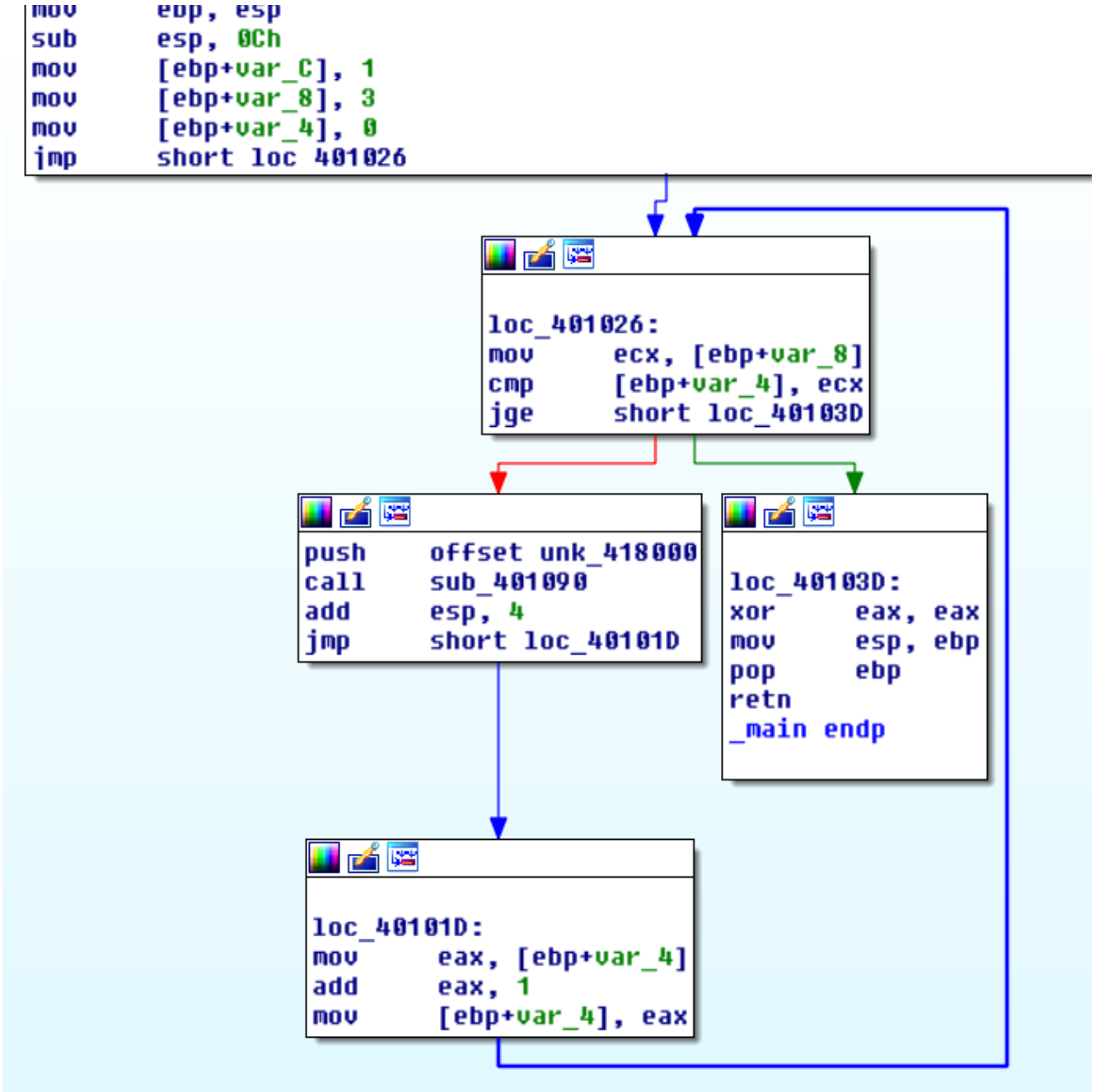


- Identificación de patrones

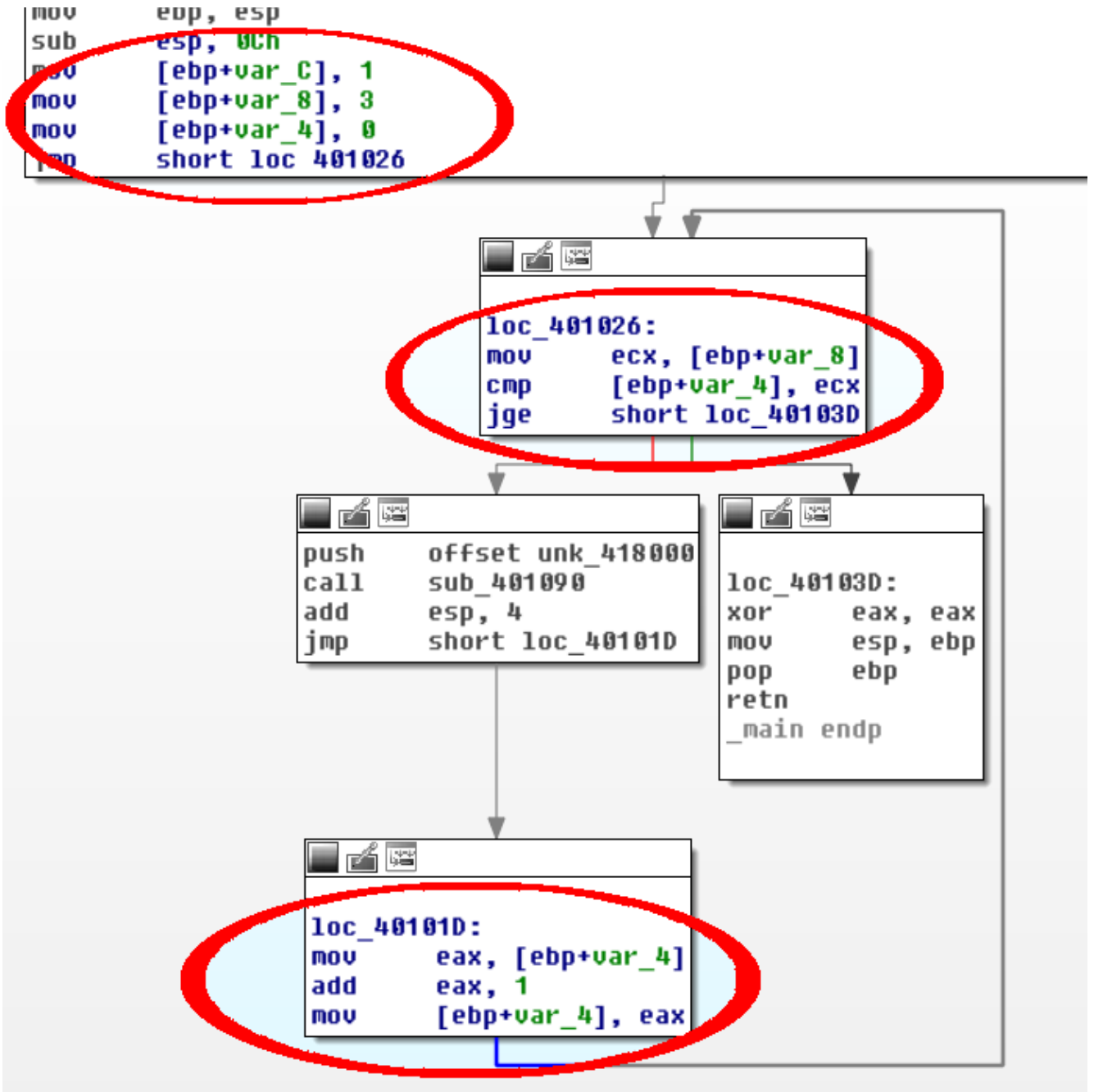
```
while ( condition_1 ) {
    do;
    }
```

Análisis de Binarios

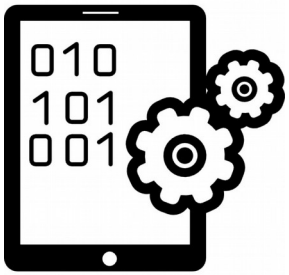




Análisis de Binarios



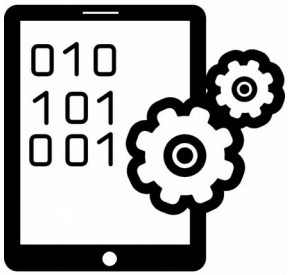
Análisis de Binarios

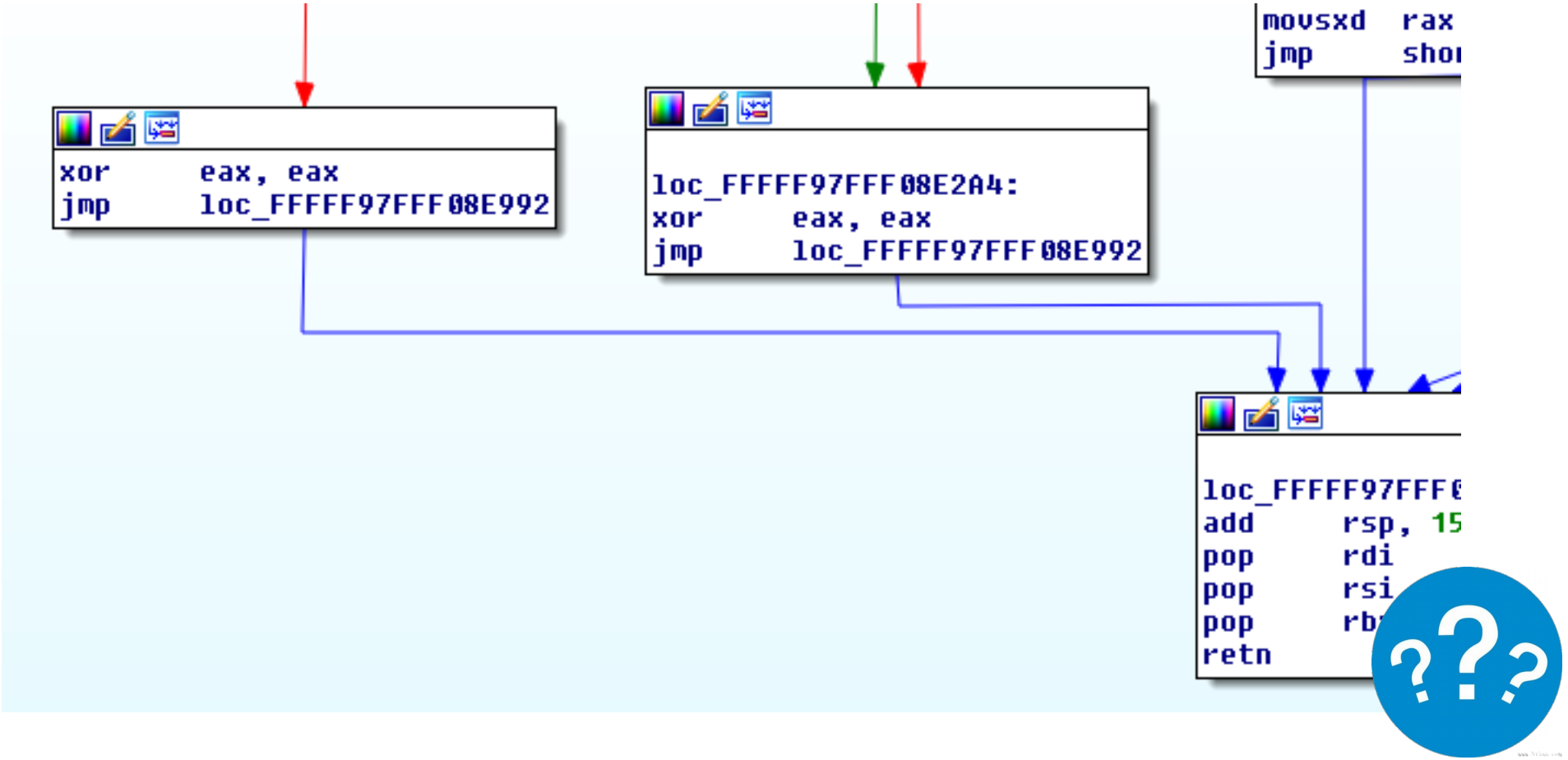


- Identificación de patrones

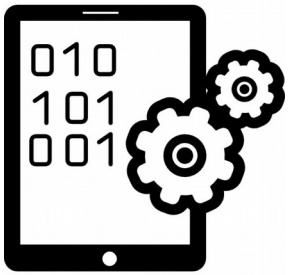
```
int max = 3;
for ( int i = 0; i < max; i++ ) {
    ...
}
```

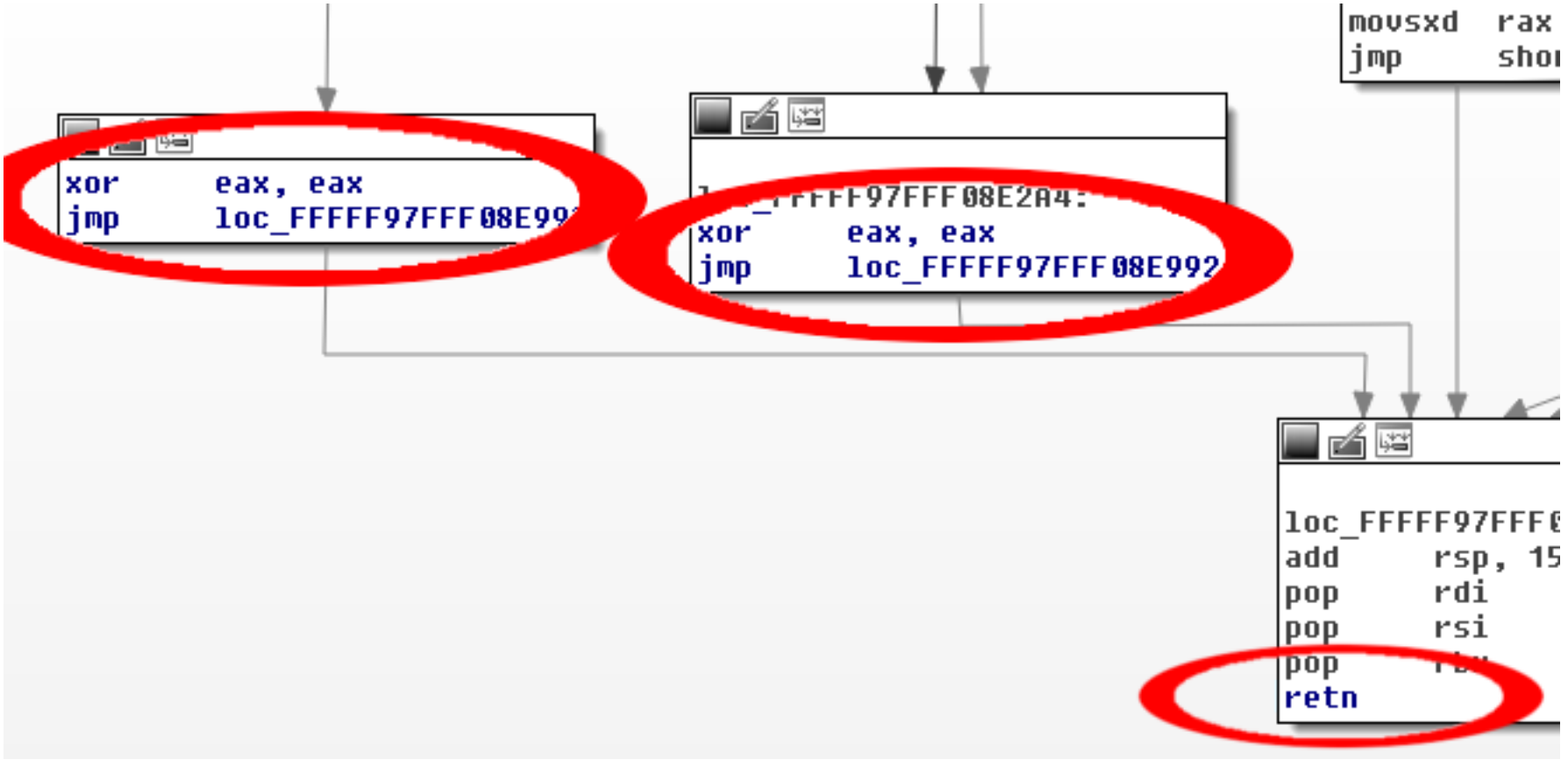
Análisis de Binarios



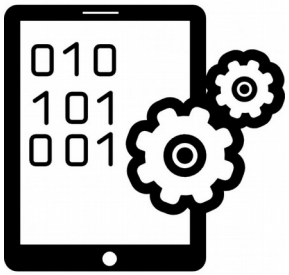


Análisis de Binarios





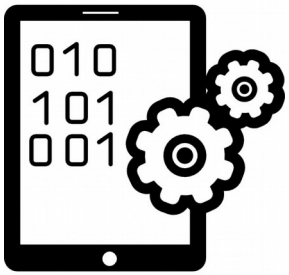
Análisis de Binarios



- Identificación de patrones

```
if ( condition_1 ) {
    goto error;
}
if ( condition_2 ) {
    goto error;
}
error:
    return 0;
```

Análisis de Binarios




```
mov     [ebp+var_C], 33h
mov     [ebp+var_8], 4
mov     eax, [ebp+var_1C]
mov     [ebp+var_18], eax
cmp     [ebp+var_18], 5 ;
ja      short loc_40106A ;
```

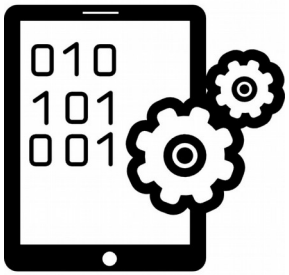
```
mov     ecx, [ebp+var_18]
jmp     ds:off_401084[ecx*4] ;
```

```
loc_401058:                ;
mov     [ebp+var_14], 3
jmp     short loc_401071
```

```
loc_401061:                ;
mov     [ebp+var_14], 6
jmp     short loc_401071
```

```
loc_401071:
```

Análisis de Binarios




```
mov     [ebp+var_C], 33h
mov     [ebp+var_8], 4
mov     eax, [ebp+var_1C]
mov     [ebp+var_18], eax
cmp     [ebp+var_18], 5 ; switch 6 cases
ja      short loc_40106A ; jumptable 004010
```

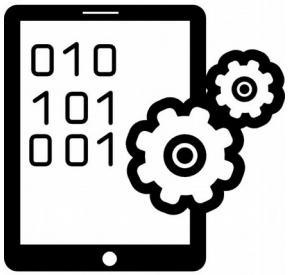
```
mov     ecx, [ebp+var_18]
jmp     ds:off_401084[ecx*4] ; switch jump
```

```
loc_401058:                             ; jumptable 0040103F case 3
mov     [ebp+var_14], 3
jmp     short loc_401071
```

```
loc_401061:                             ; jumpta
mov     [ebp+var_14], 6
jmp     short loc_401071
```

```
loc_401071:
xor     eax, eax
```

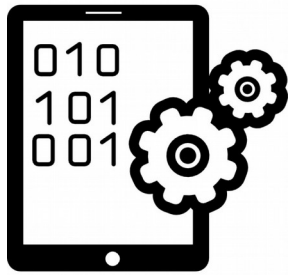
Análisis de Binarios



- Identificación de patrones

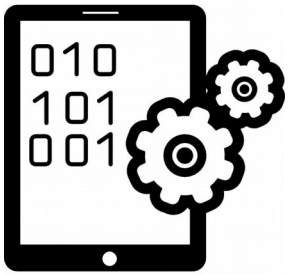
```
switch ( variable ) {
    case 0:
        ...
        break;
    case 1:
        ...
        break;
}
```

Análisis de Binarios



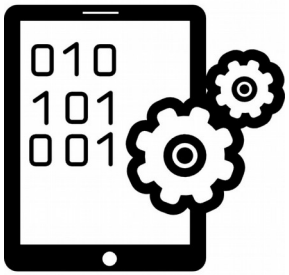

```
mov     [ebp+var_14], 1
mov     [ebp+var_10], 2
mov     [ebp+var_C], 33h
mov     [ebp+var_8], 4
mov     [ebp+var_18], offset sub_401000
call    [ebp+var_18]
xor     eax, eax
mov     ecx, [ebp+var_4]
xor     ecx, ebp
call    @__security_check_cookie@4 ; __security
mov     esp, ebp
pop     ebp
```

Análisis de Binarios




```
mov     [ebp+var_14], 1
mov     [ebp+var_10], 2
mov     [ebp+var_C], 33h
mov     [ebp+var_8], 4
mov     [ebp+var_18], offset sub_401000
call    [ebp+var_18]
xor     eax, eax
mov     ecx, [ebp+var_4]
xor     ecx, ebp
call    @__security_check_cookie@4 ; __securit
mov     esp, ebp
pop     ebp
```

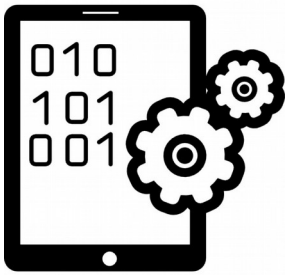
Análisis de Binarios



- Identificación de patrones

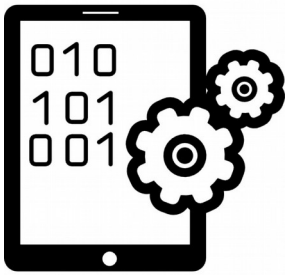
```
int ( * f_ptr ) ( ) = f;

( * f_ptr ) ( );
```

Análisis de Binarios



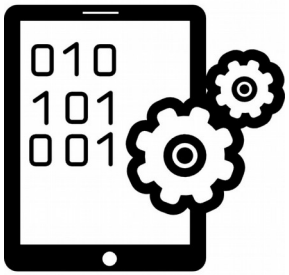
- Análisis dinámico del código ejecutable
  - IDA Pro (debugger)
  - Otros debuggers
    - Windbg, gdb, Ollydbg, etc.
  - strace (Linux)
  - API monitor (Windows)
  - Wireshark

Análisis de Binarios



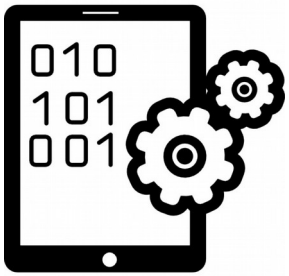
- Análisis dinámico del código ejecutable
  - Tools para monitorear cambios en el registro (Windows)
  - Tools para monitorear cambios en el filesystem
  - Suite integrada: Cuckoo

Análisis de Binarios



- Trazas de ejecución
  - No detienen la ejecución (a diferencia de los breakpoints) y registran:
    - Ejecución de instrucciones
    - Lecturas o escrituras en memoria
      - Desde qué intrucción se accedió a la memoria
    - Otros cambios de estado (ej. registros)
    - Thread que ejecutó
    - Otra información (ej. Call-graph)
  - Pueden llegar a generar demasiada información. Es necesario filtrar.

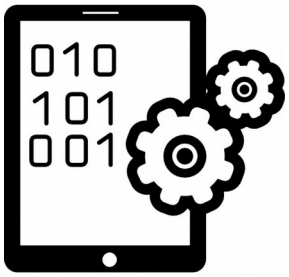
Análisis de Binarios



- Ejemplo de traza

```
00000F20                                                    ST0=FFFFFFFFFFFFFFFF ST1=FFFFFFFFFFFFFFFF ST2=FFFFFFF
00000F20  .text:sub_2F13C0+3          sub   esp, 14h        ESP=0042FA34 PF=0
00000F20  .text:sub_2F13C0+6          push  ebx             ESP=0042FA30
00000F20  .text:sub_2F13C0+7          cpuid                 EAX=00000000 EBX=00000000 ECX=00000000 EDX=00000000
00000F20  .text:sub_2F13C0+9          rdtsc                 EAX=DDA53517 EDX=000002FE
00000F20  .text:sub_2F13C0+B          mov   [ebp+var_C], eax
00000F20  .text:sub_2F13C0+E          mov   [ebp+var_8], edx
00000F20  .text:sub_2F13C0+11         mov   [ebp+var_4], 0
00000F20  .text:sub_2F13C0+18         jmp   short loc_2F13E3
00000F20  .text:sub_2F13C0:loc_2F13E3 cmp   [ebp+var_4], 8  CF=1 AF=1 SF=1
00000F20  .text:sub_2F13C0+27         jnb   short loc_2F13FE
00000F20  .text:sub_2F13C0+29         mov   ecx, 8          ECX=00000008
00000F20  .text:sub_2F13C0+2E         sub   ecx, [ebp+var_4] CF=0 AF=0 SF=0
00000F20  .text:sub_2F13C0+31         mov   edx, [ebp+var_4] EDX=00000000
00000F20  .text:sub_2F13C0+34         mov   al, [ebp+ecx+var_D] EAX=DDA53500
00000F20  .text:sub_2F13C0+38         mov   byte ptr [ebp+edx+var_14], al
00000F20  .text:sub_2F13C0+3C         jmp   short loc_2F13DA
00000F20  .text:sub_2F13C0:loc_2F13DA mov   eax, [ebp+var_4] EAX=00000000
00000F20  .text:sub_2F13C0+1D         add   eax, 1          EAX=00000001
00000F20  .text:sub_2F13C0+20         mov   [ebp+var_4], eax
00000F20  .text:sub_2F13C0:loc_2F13E3 cmp   [ebp+var_4], 8  CF=1 PF=1 AF=1 SF=1
00000F20  .text:sub_2F13C0+27         jnb   short loc_2F13FE
```

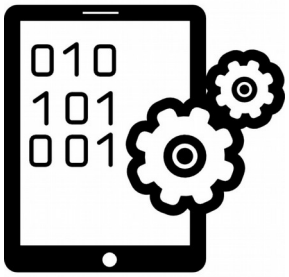
Análisis de Binarios



- Ejemplo de traza (con filtro por 0x42FA48)

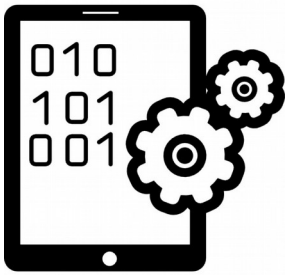
```
00000F20  .text:sub_2F13C0+45                  mov    esp, ebp                                  ESP=0042FA48
00000F20  .text:_main+16                       call   sub_2F1210                                ESP=0042FA48
00000F20  .text:sub_2F1210+1                   mov    ebp, esp                                  debug021:0042FA48: 58
00000F20  .text:sub_2F1210+1                   mov    ebp, esp                                  EBP=0042FA48
00000F20  .text:sub_2F1000+5E                  pop    ebp                                       EBP=0042FA48 ESP=0042FA24
00000F20  .text:sub_2F1000+5E                  pop    ebp                                       EBP=0042FA48 ESP=0042FA24
00000F20  .text:sub_2F1000+5E                  pop    ebp                                       EBP=0042FA48 ESP=0042FA24
00000F20  .text:sub_2F1210:loc_2F12E2          mov    esp, ebp                                  ESP=0042FA48
00000F20  .text:_main+27                       push   0                    ; bInitialOwner      ESP=0042FA48
00000F20  KERNELBASE:kernelbase_CreateMutexA+A jz     short near ptr unk_768717C8               debug021:0042FA48: 00
00000F20  .text:_main+38                       push   0                    ; dwCreationFlags    ESP=0042FA48
00000F20  kernel32:kernel32_CreateThread+D     push   dword ptr [ebp+14h]                       debug021:0042FA48: 00
00000F20  .text:_main+50                       push   0                    ; dwCreationFlags    ESP=0042FA48
00000F20  kernel32:kernel32_CreateThread+D     push   dword ptr [ebp+14h]                       debug021:0042FA48: 00
00000F20  .text:_main+6B                       push   eax                   ; hHandle           ESP=0042FA48
00000F20  kernel32:kernel32_WaitForSingleObject+D call   near ptr kernel32_WaitForSingleObjectEx debug021:0042FA48: 34
```

Análisis de Binarios



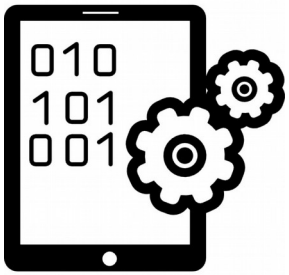
- ¿Cuál es el enfoque adecuado para analizar un...
  - binario “strippeado”? (sin símbolos)
  - binario ofuscado o packeado?
- Code-coverage en el análisis dinámico:
    - ¿cómo triggereamos todos los flujos posibles de ejecución?

Análisis de Binarios



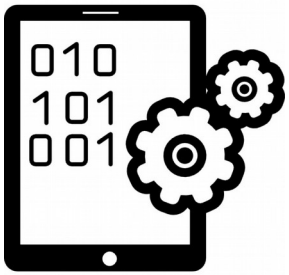
- La respuesta es caso-a-caso y probablemente implique la combinación de diferentes técnicas
  - Análisis estático puede implicar un esfuerzo excesivo: ¡hay demasiados datos para procesar!
  - Análisis dinámico basado en debugging también puede implicar un esfuerzo excesivo
  - Análisis dinámico basado en herramientas de monitoreo puede ser insuficiente

Pregunta



¿Que abordaje utilizarían para analizar un binario que cifra las comunicaciones con un algoritmo criptográfico propio?

Lab 3.1



Analizar dicho binario, describir la lógica y extraer los datos comunicados

Referencias

- https://github.com/cuckoosandbox/cuckoo
- The IDA Pro Book