Lecture 10

Reverse engineering 3: debugging and anti

Today

- Reverse Engineering HLL programs
- AntiDbg: Debugger detection methods
- AntiAntiDbg: Anti-debugger detection methods (+exercise)
- Packers: Packing binaries to save space & make analysis harder
- Unpackers: Unpacking binaries to make analysis easier (+exercise)
- Generic RE: common algorithms (+exercise)
- Exceptions under the hood (+exercise)

HLL Programs

Note: In this context, C is a "High Level Language"

GCC Prologue

```
; Attributes: noreturn

public _start
_start proc near
xor     ebp, ebp
pop     esi
mov     ecx, esp
and     esp, 0FFFFFFF0h
push    eax
push    esp                 ; stack_end
push    edx                 ; rtld_fini
push    offset __libc_csu_fini ; fini
push    offset __libc_csu_init ; init
push    ecx                 ; ubp_av
push    esi                 ; argc
push    offset main         ; main
call    ___libc_start_main
hlt
_start endp
```

MinGW Prologue

```
                 public start
start            proc near
                 sub     esp, 0Ch
                 mov     ds:dword_407398, 0
                 call    sub_4028E0
                 add     esp, 0Ch
                 jmp     sub_401170
start            endp
```

```
mov     eax, ds:argv
mov     [esp+88h+Src], eax ; argv
mov     eax, ds:argc
mov     [esp+88h+dwMilliseconds], eax ; argc
call    main
mov     ecx, ds:dword_407008
mov     ds:dword_40700C, eax
test    ecx, ecx
jz      loc_4014AB
mov     edx, ds:dword_407004
test    edx, edx
jnz     short loc_40140A
call    _cexit
mov     eax, ds:dword_40700C
```

if

```
push    ebp
mov     ebp, esp
sub     esp, 28h
cmp     dword ptr [ebp+8], 3
setz    al
mov     [ebp-25], al
cmp     dword ptr [ebp+8], 4
jz      short loc_8048833
cmp     dword ptr [ebp+8], 3
jz      short loc_8048833
mov     eax, [ebp+12]
mov     eax, [eax]
sub     esp, 0Ch
push    eax
call    usage
add     esp, 10h
jmp     locret_8048987
```

```
memory[ebp-25] = [ebp+8] == 3;

if (memory[ebp+8] == 4) {
   // 8048833
} else if (memory[ebp+8] == 3) {
   // 8048833
} else {
   usage(...);
}
```

for

```
                mov     [ebp+var_C], 0
                jmp     short loc_804843C
; ---------------------------------------------------------------------------

loc_8048425:                            ; CODE XREF: main+35↓j
                sub     esp, 8
                push    [ebp+var_C]
                push    offset format   ; "%d"
                call    _printf
                add     esp, 10h
                add     [ebp+var_C], 1

loc_804843C:                            ; CODE XREF: main+18↑j
                cmp     [ebp+var_C], 9
                jle     short loc_8048425
```

```c
for (int i = 0; i < 10; i++) {
    printf("%d", i);
}
```

functions

```
push    ebp
mov     ebp, esp
sub     esp, 28h
```

```
void function(int a) {

// 0x28 bytes on stack reserved
```

```
cmp     dword ptr [ebp+8], 3
```

```
a == 3;
```

```
leave
retn
```

```
}
```

AntiDbg

Techniques

- WinAPI features and “features”
- Thread/Process internals
- Time-based checks
- OEP obfuscation
- Trap detection
- Attach prevention
- Exploiting debugger vulns and bugs
- ...

WinAPI features

```
BOOL WINAPI IsDebuggerPresent(void);

BOOL WINAPI CheckRemoteDebuggerPresent(
  _In_    HANDLE hProcess,
  _Inout_ PBOOL  pbDebuggerPresent
);
```

WinAPI features

```
NtSetInformationThread(
        GetCurrentThread(),
        0x11,  // ThreadHideFromDebugger
        0, 0);

NtQueryInformationProcess(
        ...,
        0x07,  // ProcessDebugPort
        ...)
```

WinAPI “features”

```
void WINAPI OutputDebugString(
  _In_opt_ LPCTSTR lpOutputString
);
```

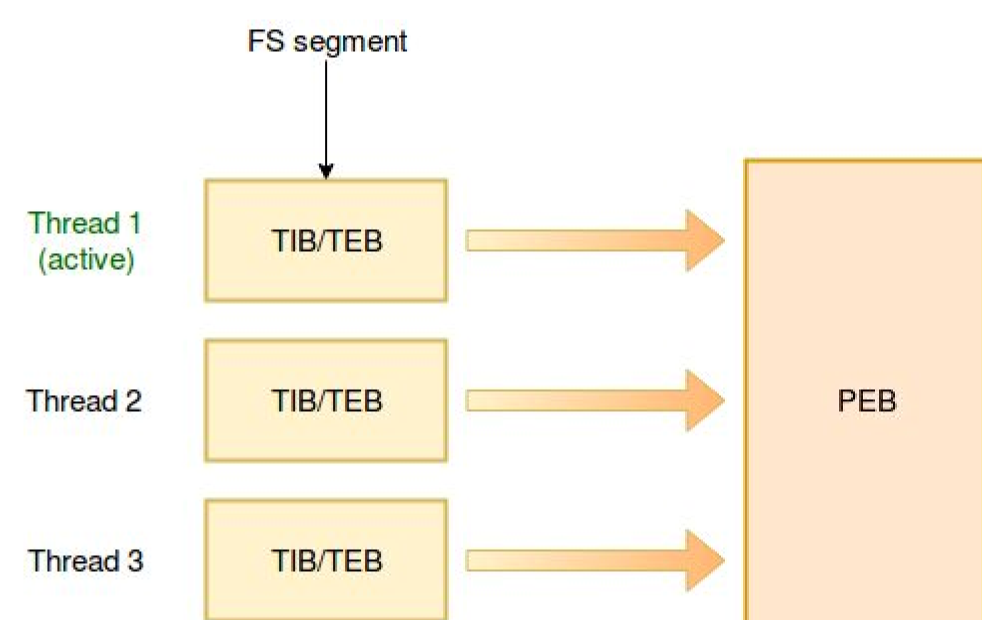
- “Silently” fails if debugger is not attached (GLE set)

Looking for debugger

- FindWindow
- CreateToolhelp32Snapshot
- CreateFile

Process Internals

ntdll.dll	77C02000	77C03000	R	W	.	D	.	byte	0000
ntdll.dll	77C03000	77C04000	R	W	.	D	.	byte	0000
ntdll.dll	77C04000	77C08000	R	W	.	D	.	byte	0000
ntdll.dll	77C08000	77C09000	R	W	.	D	.	byte	0000
ntdll.dll	77C10000	77C67000	R	.	.	D	.	byte	0000
ntdll.dll	77C70000	77C75000	R	.	.	D	.	byte	0000
debug010	7EFB0000	7EFD3000	R	.	.	D	.	byte	0000
TIB[00000318]	7EFD8000	7EFDB000	R	W	.	D	.	byte	0000
TIB[000005E4]	7EFDB000	7EFDE000	R	W	.	D	.	byte	0000
PEB	7EFDE000	7EFDF000	R	W	.	D	.	byte	0000
debug012	7EFDF000	7EFE0000	R	W	.	D	.	byte	0000
debug038	7EFE0000	7EFE5000	R	.	.	D	.	byte	0000


FS segment
Thread 1
(active)
TIB/TEB
Thread 2
TIB/TEB
Thread 3
TIB/TEB
PEB

Win32 Thread Information Block (TIB)

FS:[0x00]	**Current Structured Exception Handling (SEH) frame**
FS:[0x18]	Linear address of TEB
FS:[0x30]	**Linear address of Process Environment Block (PEB)**

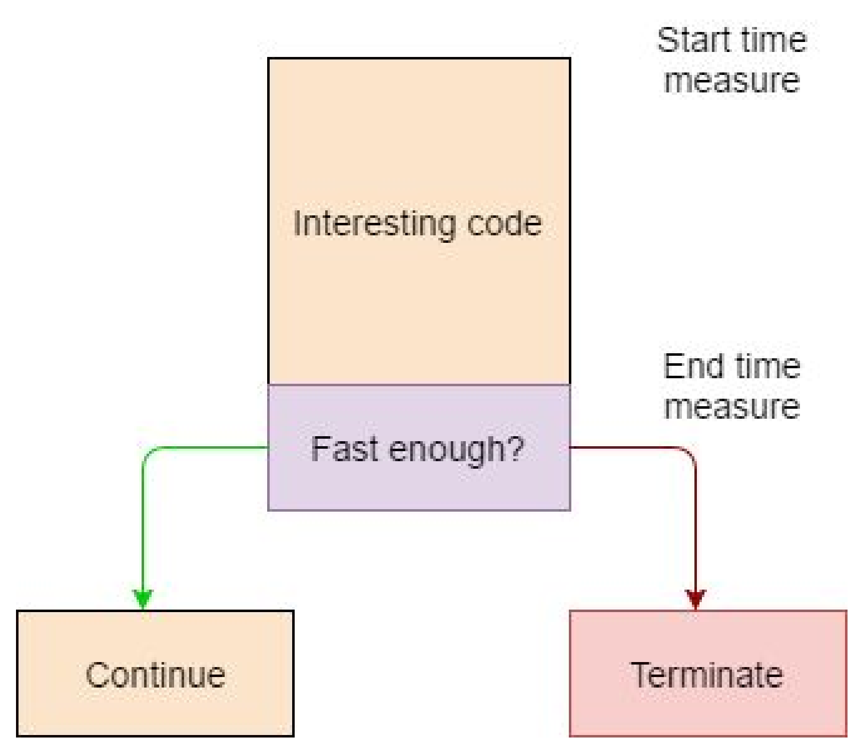
Process Environment Block (PEB)

`mov eax, fs:[0x30]`

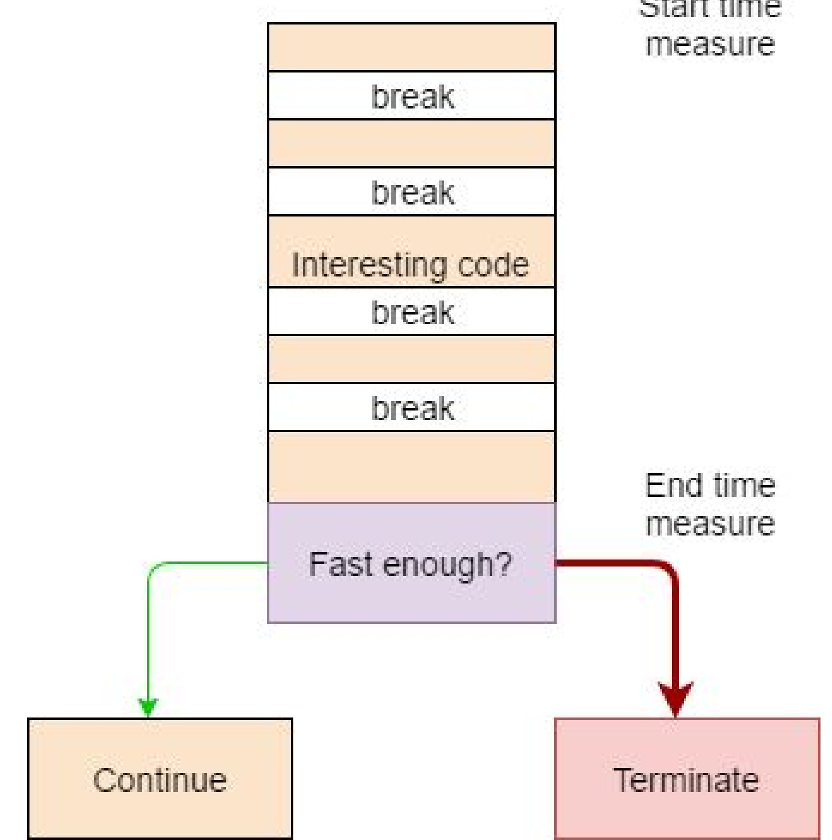
```
typedef struct _PEB
{
      // ...
      UCHAR BeingDebugged;
      // ...
      PVOID ProcessHeap;
      // ...
      ULONG NtGlobalFlag;
     // ...
} PEB, *PPEB;
```

Time based

Time-based dbg detection



Time-based dbg detection

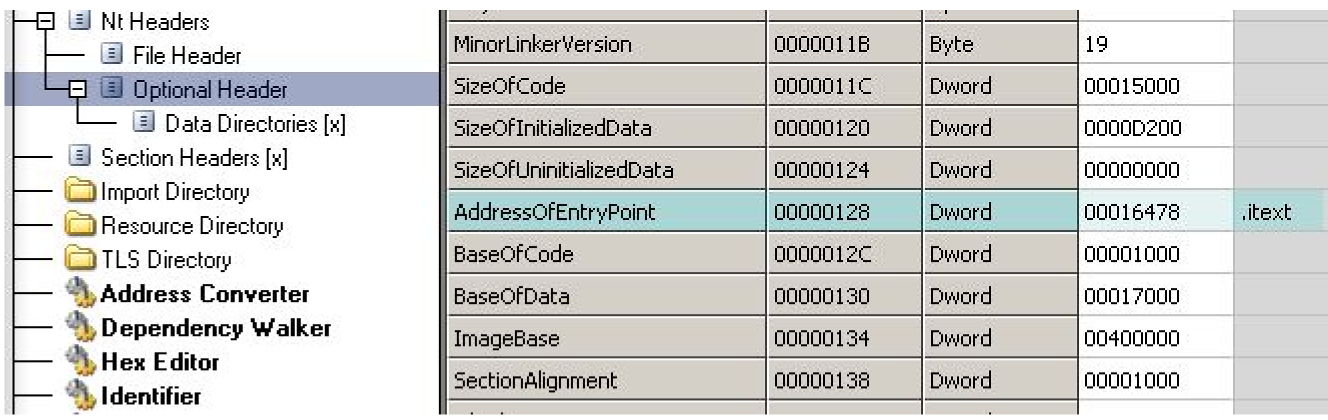


Time-based dbg detection

- GetLocalTime
- GetSystemTime
- GetTickCount
- QueryPerformanceCounter
- ….
- RDPMC/RDTSC instructions

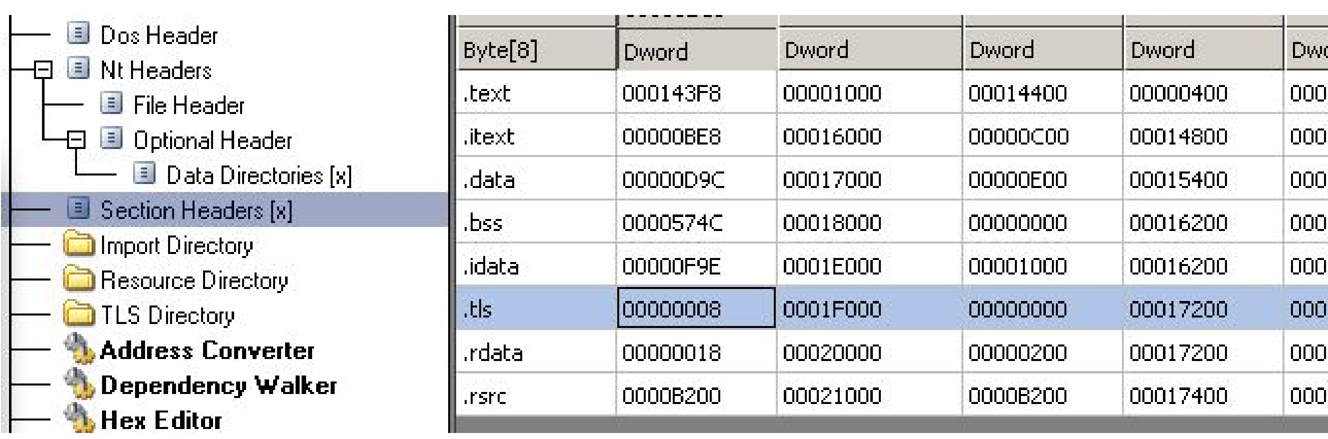
OEP obfuscation
(TLS callback)

Main entrypoint



- RVA of entry point defined in PE Optional Header

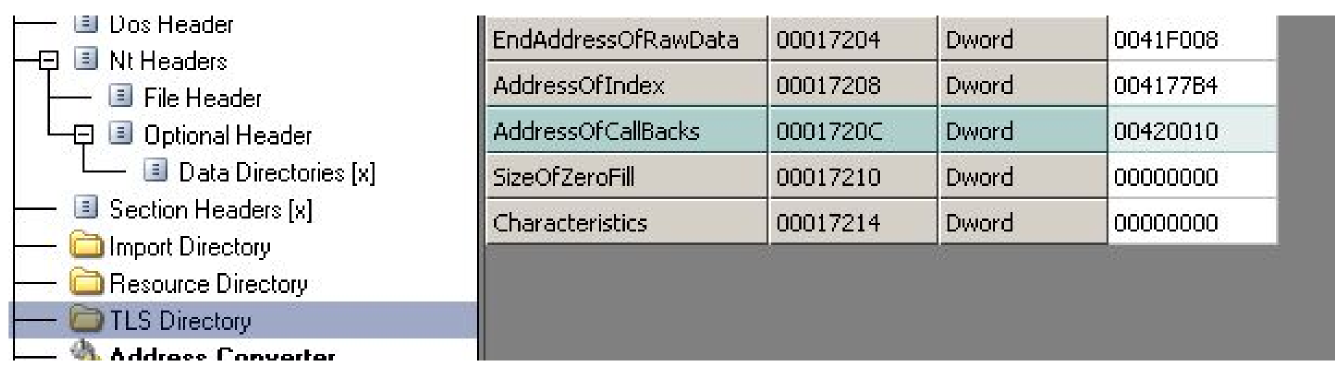
.tls section



Byte[8]	Dword	Dword	Dword	Dword	Dw
.text	000143F8	00001000	00014400	00000400	000
.itext	00000BE8	00016000	00000C00	00014800	000
.data	00000D9C	00017000	00000E00	00015400	000
.bss	0000574C	00018000	00000000	00016200	000
.idata	00000F9E	0001E000	00001000	00016200	000
.tls	00000008	0001F000	00000000	00017200	000
.rdata	00000018	00020000	00000200	00017200	000
.rsrc	0000B200	00021000	0000B200	00017400	000

- .tls section contains informations about static Thread Local Storage

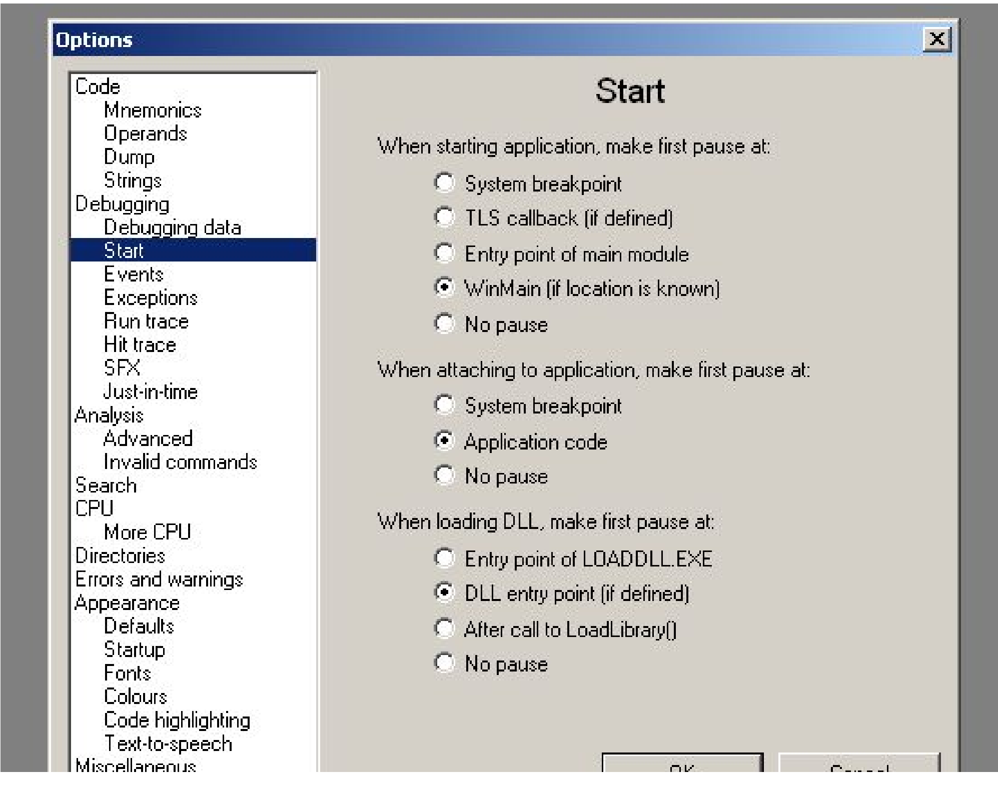
TLS callbacks



- *Address of Callbacks* field points to null-terminated array of TLS callback function pointers

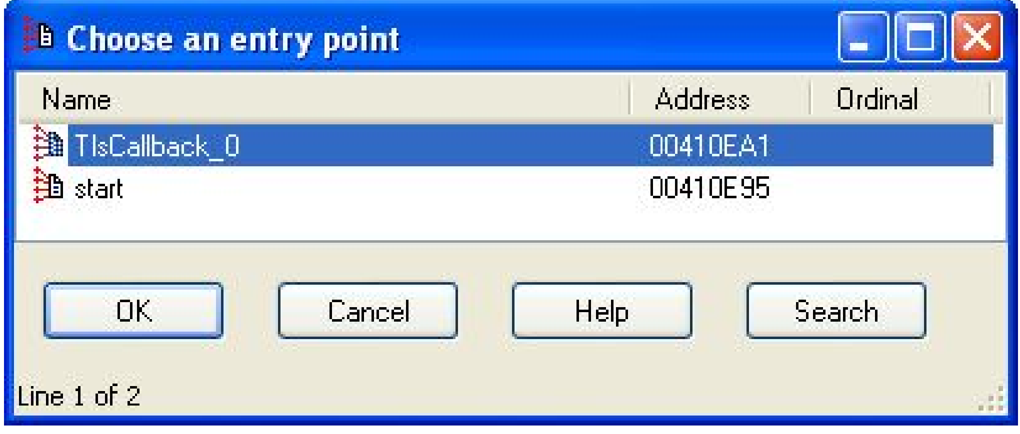
OllyDbg & TLS callbacks

- Default: break at WinMain or PE entry point
- We can choose pausing on TLS callback in Options



IDA Pro

- IDA Pro is able to locate additional entry points in loaded file
- We can jump to specified entry point using CTRL+E



Trap detection

Types of breakpoints

- Software breakpoint (SW BP)
- Hardware breakpoint (HW BP)
- Single-step mode

Software breakpoints

- Debugger temporarily inserts 0xCC byte at breakpoint target
- When bp is reached: dbg restores original byte.
- 0xCC - opcode for INT 3h instruction
  - INT xx - Call to INTerrupt procedure

Software breakpoints

IVT Offset	INT #	Description
0x0000	0x00	Divide by 0
0x0004	0x01	Reserved
0x0008	0x02	NMI Interrupt
0x000C	0x03	Breakpoint (INT3)
0x0010	0x04	Overflow (INTO)
0x0014	0x05	Bounds range exceeded (BOUND)
0x0018	0x06	Invalid opcode (UD2)
0x001C	0x07	Device not available (WAIT/FWAIT)
0x0020	0x08	Double fault
0x0024	0x09	Coprocessor segment overrun
0x0028	0x0A	Invalid TSS
0x002C	0x0B	Segment not present
0x0030	0x0C	Stack-segment fault
0x0034	0x0D	General protection fault
0x0038	0x0E	Page fault
0x003C	0x0F	Reserved
0x0040	0x10	x87 FPU error
0x0044	0x11	Alignment check
0x0048	0x12	Machine check
0x004C	0x13	SIMD Floating-Point Exception
0x00xx	0x14-0x1F	Reserved
0x0xxx	0x20-0xFF	User definable

Hardware breakpoints

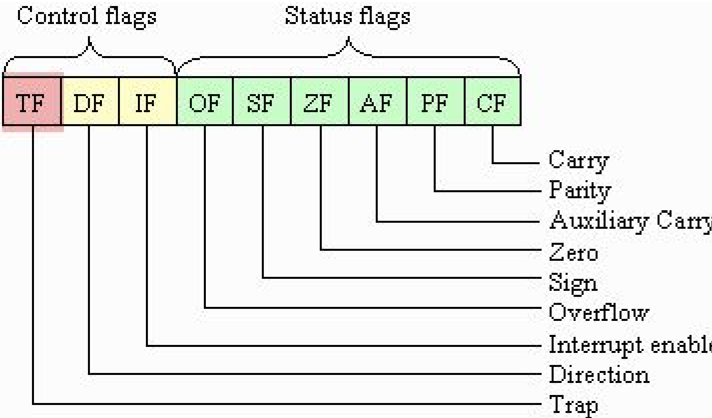
- Breakpoints handled internally by CPU
- In x86 limited to 4 breakpoints
- We can set HW breakpoints on:
  - code (on execute)
  - memory (on access, on write)

Hardware breakpoints

- Special set of x86 registers:
  - DR0-DR3 - linear addresses of breakpoints (max 4)
  - DR6 - debug status (which breakpoints were fired)
  - DR7 - debug control, specifies breakpoint condition (on read/write/execute)

Single-step mode

FLAGS register



Detection techniques

Memory scan

- Spawn another thread/process, which periodically looks for 0xCC or evaluates checksum of monitored block (e.g. CRC32)
- Effective against patching or software breakpoints

Checking debug registers

- MOV instructions from/to DRx are privileged
- We're allowed to access debug registers from ring3 via GetThreadContext/SetThreadContext

```
BOOL WINAPI GetThreadContext(
  _In_    HANDLE    hThread,
  _Inout_ LPCONTEXT lpContext
);
```

Exception-driven control flow

Structured Exception Handling

- Usually exceptions in debugged code occurs, when something bad happens:
    - Access violation
    - Division by zero
    - Hardcoded breakpoint

Structured Exception Handling

- SEH - linked list of handler pointers
- When exception occurs: each handler is executed (in order), until one will handle exception.
- If reached end of list (0xFFFFFFFF) - exception is passed to system handler (which usually terminates application)

Structured Exception Handling

Registering exception handler (start of **try** block)

```
push ExceptionHandler
push fs:[0]
mov [fs],esp
```

Unregistering exception handler (end of **try** block)

```
mov eax,[esp]
mov fs:[0],eax
add esp,8
```

Structured Exception Handling

```
int main()
{
    int* p = 0x00000000; // pointer to NULL

    __try
    {
      *p = 13; // causes an access violation exception
    } __except(filter(GetExceptionCode(), GetExceptionInformation()))
    {
      puts("Something went wrong!\n");
    }
}
```

Structured Exception Handling

```
int filter(unsigned int code, struct _EXCEPTION_POINTERS *ep)
{
    if (code == EXCEPTION_ACCESS_VIOLATION)
    {
      // caught ACCESSV
      return EXCEPTION_EXECUTE_HANDLER;
    }
    // Something else.. not interested
    return EXCEPTION_CONTINUE_SEARCH;
}
```

Structured Exception Handling

- Debugger usually tries to handle exception on its own, bypassing SEH chain
- Correct flow control may rely on SEH callbacks
- Debuggers usually ask how to handle exception:
  - OllyDbg - Shift+F9 “pass to application”
  - IDA Pro shows dialog box

Structured Exception Handling

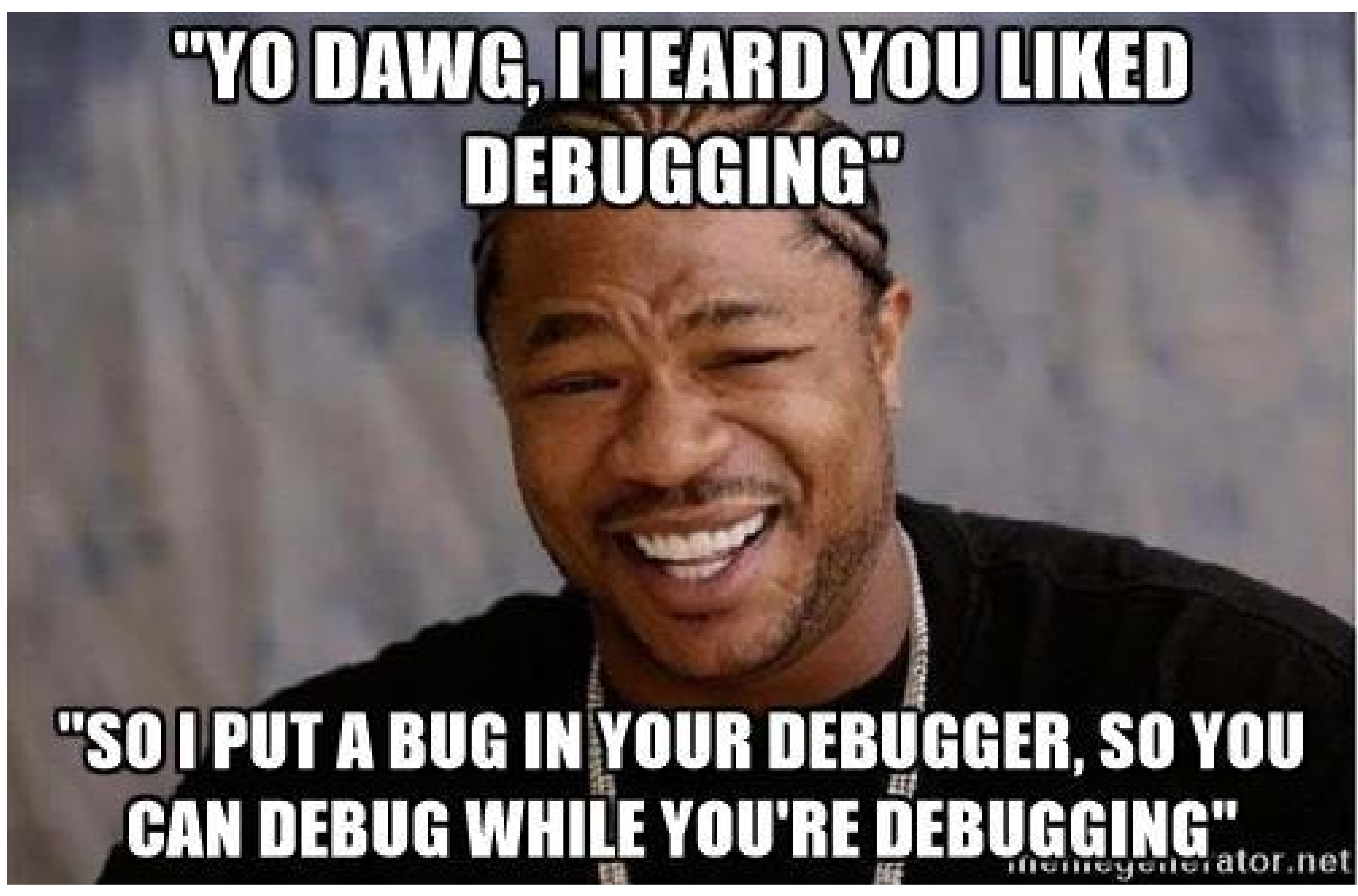
Attach prevention

Self-debug

- You can't be debugged, if you “debug yourself”
- Create another process which will attach to its parent. Both processes could also monitor each other, to prevent detach without termination.

Attach side-effects

- DebugActiveProcess internally creates remote thread in debuggee context with ntdll::DbgUiRemoteBreakin as entrypoint.
- Antidbg: hook DbgUiRemoteBreakin and pass call to ExitProcess
- NtContinue also can be hooked this way.


"YO DAWG, I HEARD YOU LIKED DEBUGGING"
"SO I PUT A BUG IN YOUR DEBUGGER, SO YOU CAN DEBUG WHILE YOU'RE DEBUGGING"

Debugger bugs and vulns

- OllyDbg OutputDebugString format string bug
- OllyDbg export name buffer overflow
- CVE-2011-1051: Integer overflow in the COFF/EPOC/EXPLOAD input file loaders in Hex-Rays IDA Pro 5.7 and 6.0

AntiAntiDbg

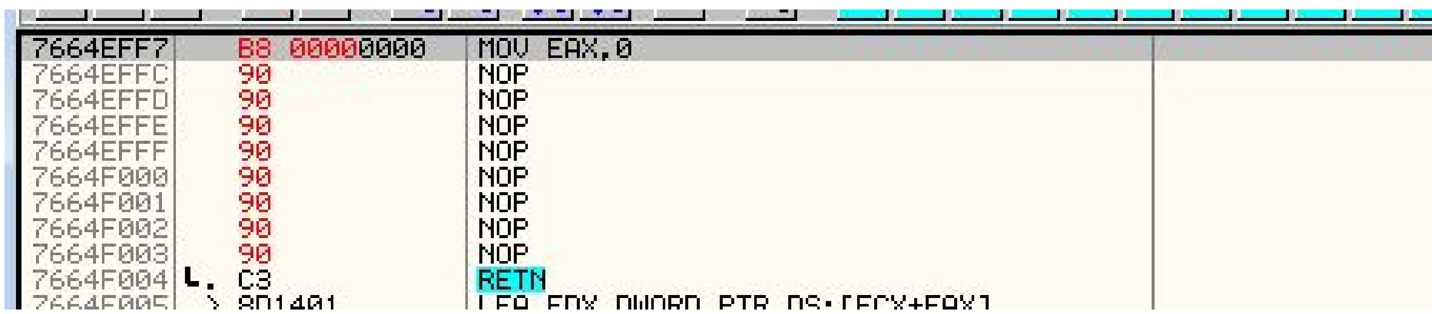
Patching, patching, patching...

```
00402385  $  CC              INT3
00402386     E8 A5350176     CALL IsDebuggerPresent              Jump to KE
0040238B     85C0            TEST EAX,EAX
0040238D  ^  0F84 6DECFFFF   JE <ModuleEntryPoint>
00402393     E8 18190176     CALL ExitProcess
00402398     90              NOP
00402399     90              NOP
```

```
00402385  $  CC              INT3
00402386     31C0            XOR EAX,EAX
00402388     90              NOP
00402389     90              NOP
0040238A     90              NOP
0040238B     85C0            TEST EAX,EAX
0040238D  ^  0F84 6DECFFFF   JE <ModuleEntryPoint>
```

Patching, patching, patching...

Patch IsDebuggerPresent



Patching, patching, patching...

Patch PEB.BeingDebugged

```
00395000 | 00 00 00 00 | FF FF FF FF | 00 00 40 00 | 00 7C AB 77 | aaaaaaaaaa@aa|źw
00395010 | 78 22 7F 00 | 00 00 00 00 | 00 00 7F 00 | C0 79 AB 77 | x¨⌂aaaaaaa⌂a└yźw
00395020 | 00 00 00 00 | 00 00 00 00 | 15 00 00 00 | 00 10 55 74 | aaaaaaaa§aaaa►Ut
```

Sometimes patching isn't enough

- User-mode sometimes isn't enough:
  - e.g. RDTSC interception needs access to Control Registers and IDT hooking (ring0)
- Anti-debug checkups are usually obfuscated

OllyDbg anti-anti-debug plugins

- Phantom
- ScyllaHide/TitanHide

Exercise 0x1

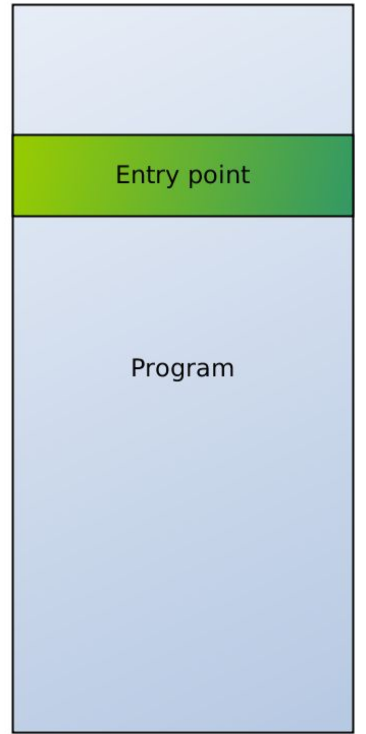
- Found nice flag generator, but also need a password.
- I've tried some debugging, but it doesn't work.
- Can you help me?

http://uw2017.p4.team/static/flaggen.exe

Packers

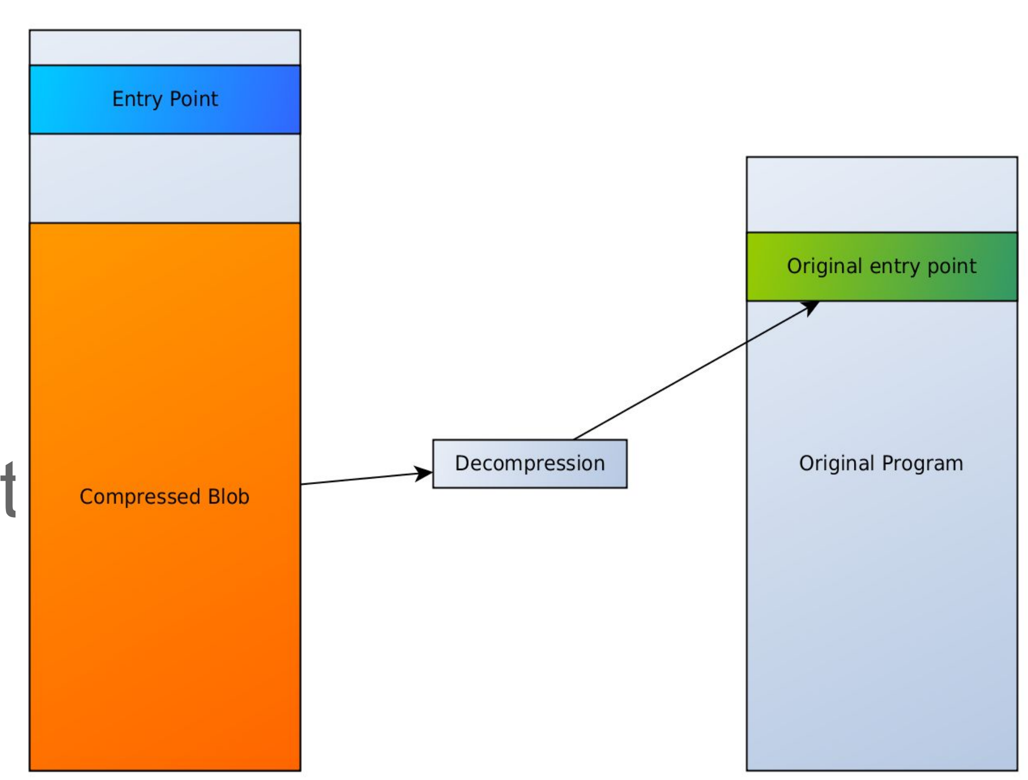
Packers?

- Normal program
- Has entry point (in PE header) and rest of it's code
- Everything visible "in plain sight"
- Standard output of any compiler



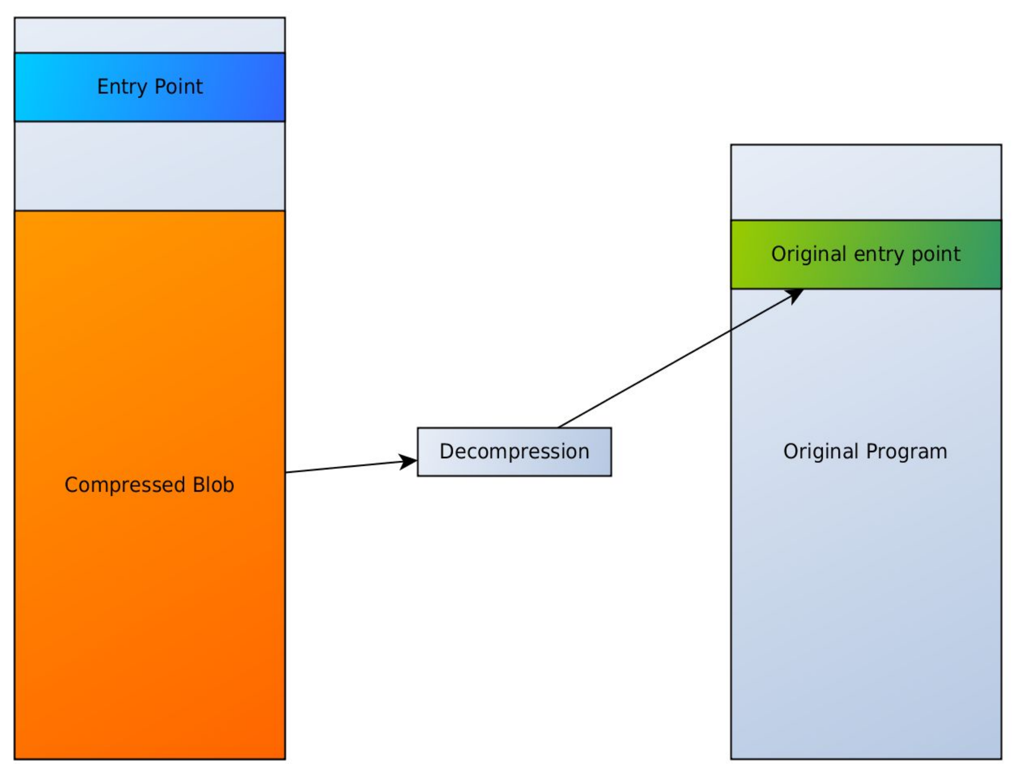
Packers

- Real program is stored in compressed form
- Not a security measure *per se*, but makes static analysis impossible



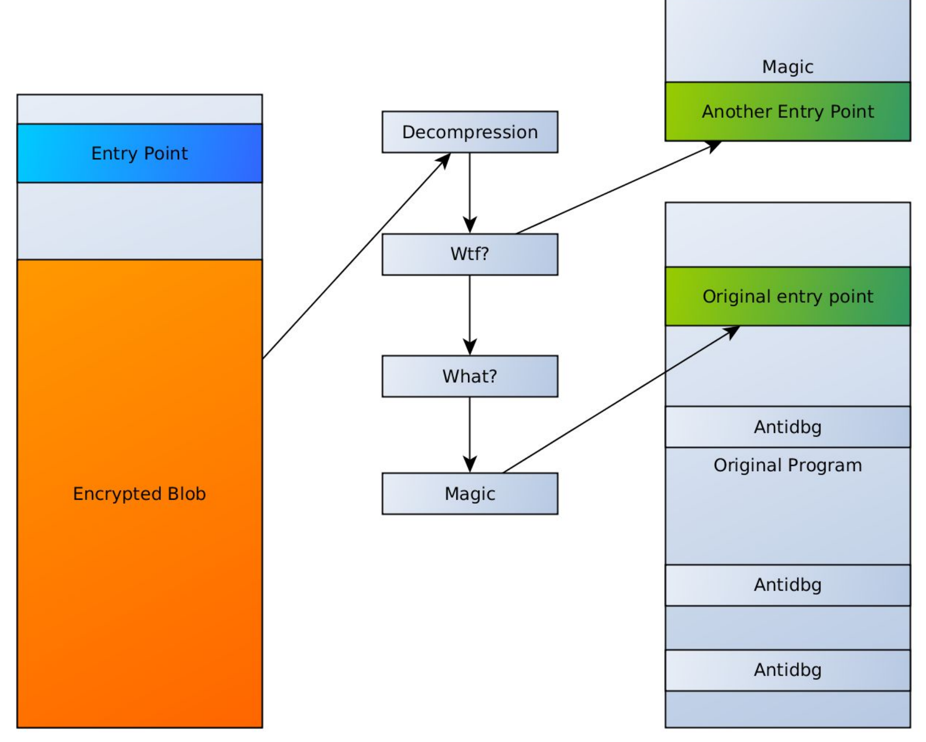
Packers

- Common packers:
  - UPX
  - FSG
  - AsPack



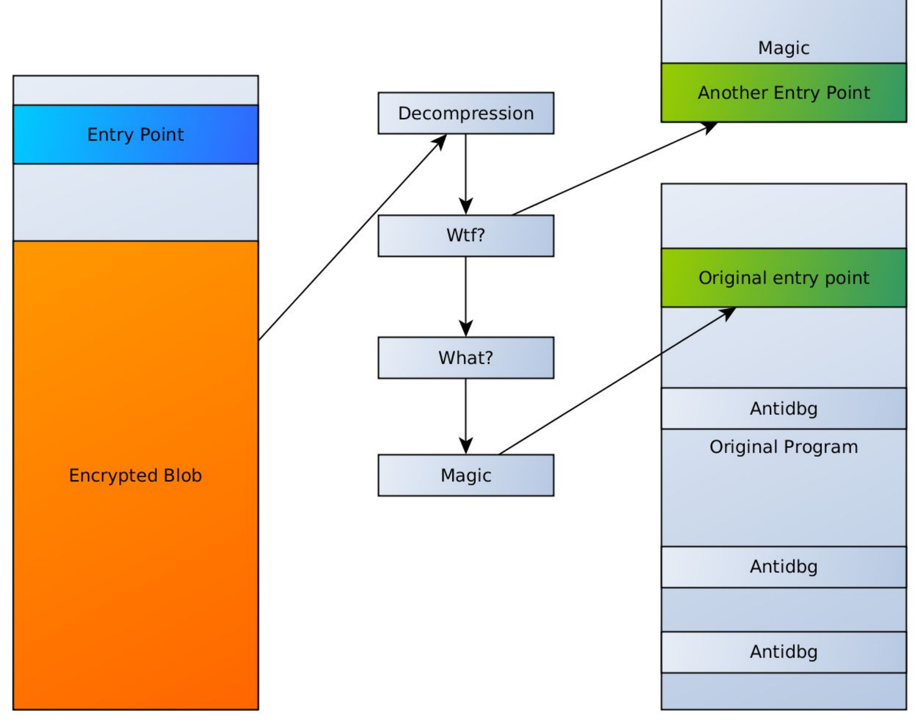
Protectors

- Real program is stored in compressed and encrypted form
- Goal: make RE as difficult as possible



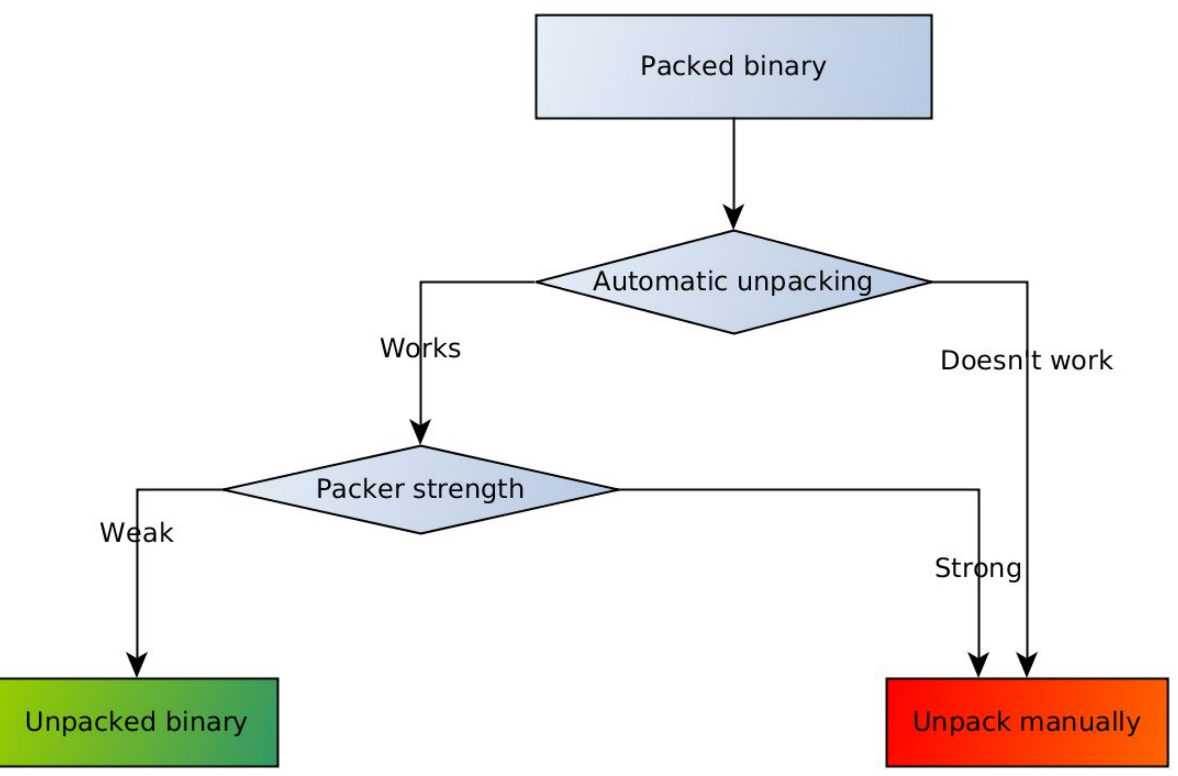
Protectors

- Common protectors:
  - Enigma
  - VmProtect
  - AsProtect

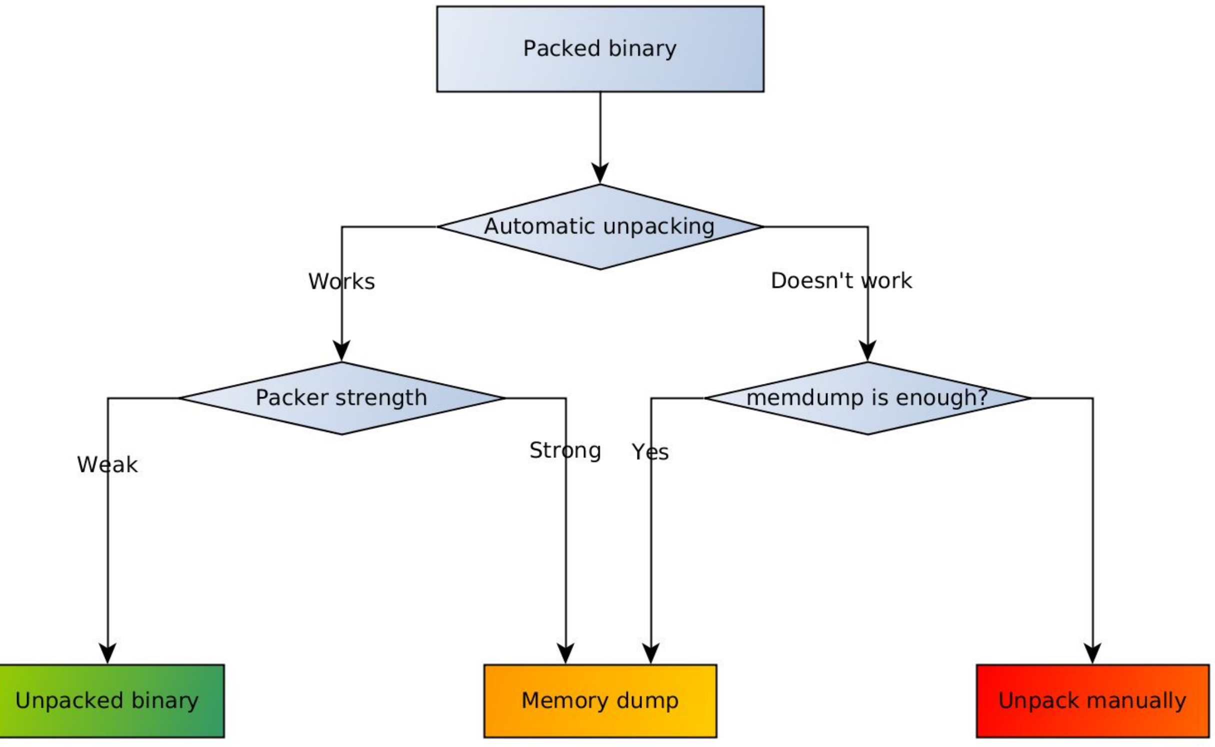


Unpackers

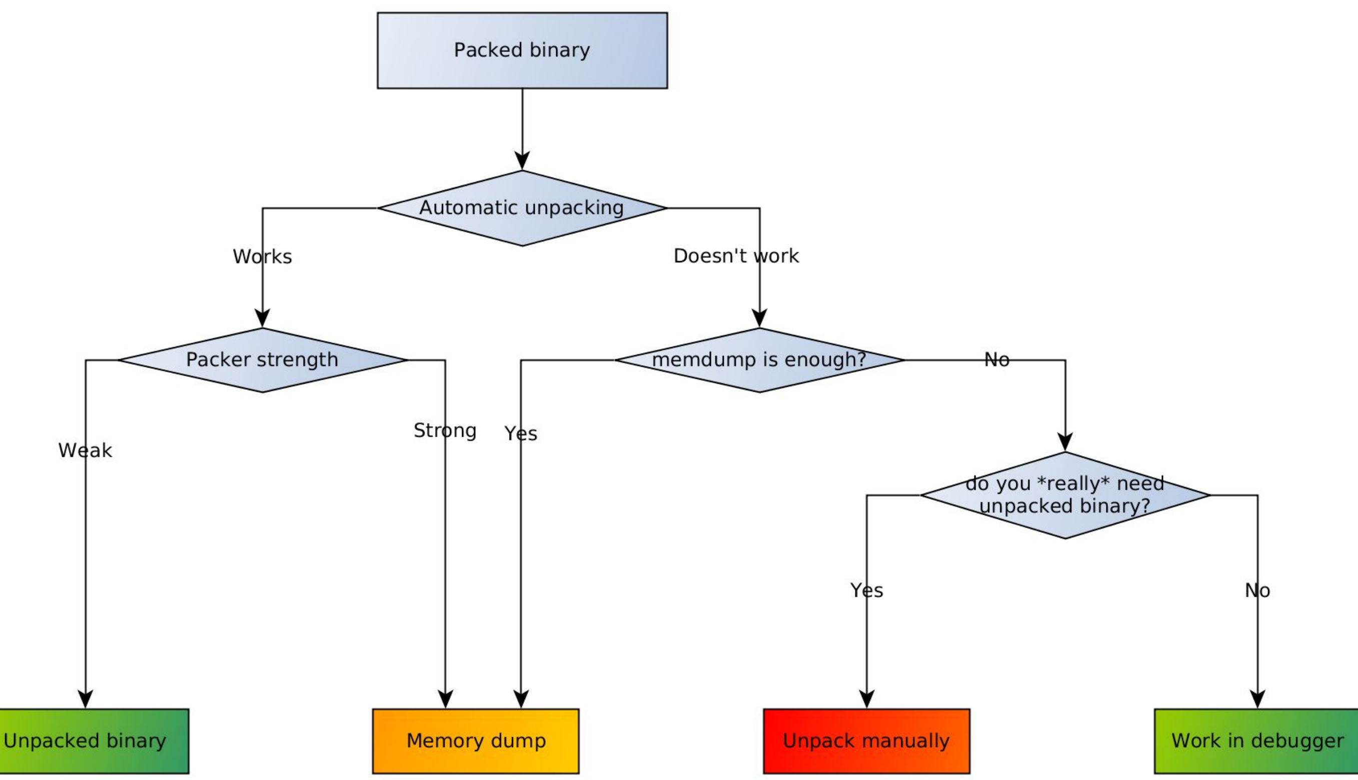
Unpacking?



Unpacking?



Unpacking?



How to unpack X

- Generic automatic unpacker (eg. cuckoo).
- Google: X unpacker version Y
- Google: X unpacking script
- Google: X unpacker
- Google: X manual unpacking
- Last resort: unpacking by hand

Unpacking manually

General technique:

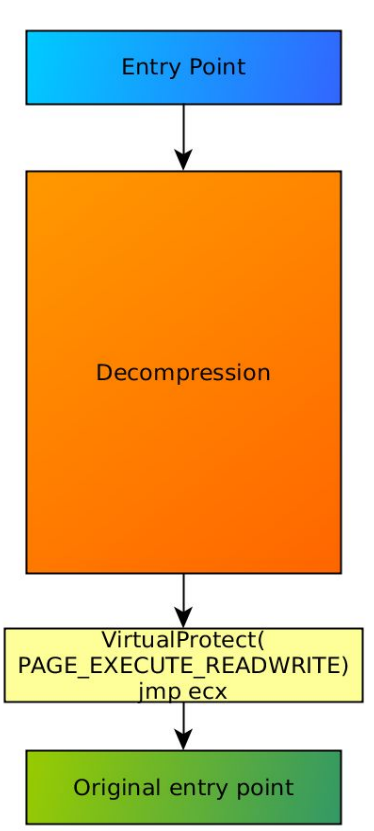
- Find decompression code
- Find jump to OEP after it
- Breakpoint on jump/call

```
cmp     [esp+14D0h+var_1484], ebx
jz      short loc_402F09

mov     eax, [esp+14D0h+wParam]
mov     ecx, [eax]
push    eax
call    dword ptr [ecx+8]

loc_402F09:
push    [esp+14D0h+var_14B0]
call    [esp+14D4h+var_14B0]
```

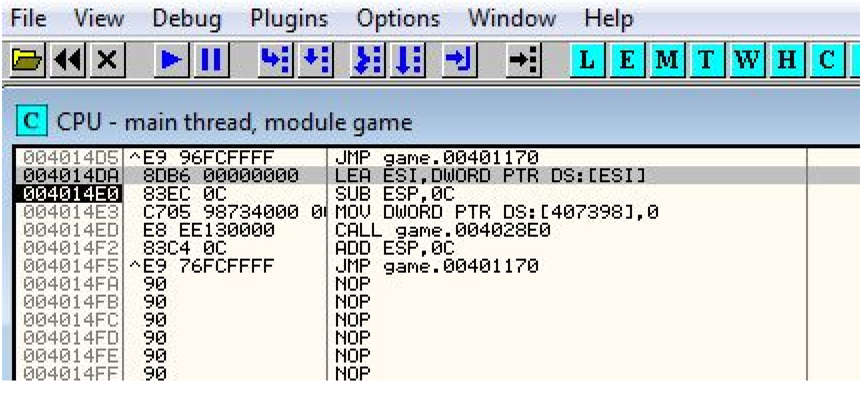
Unpacking manually

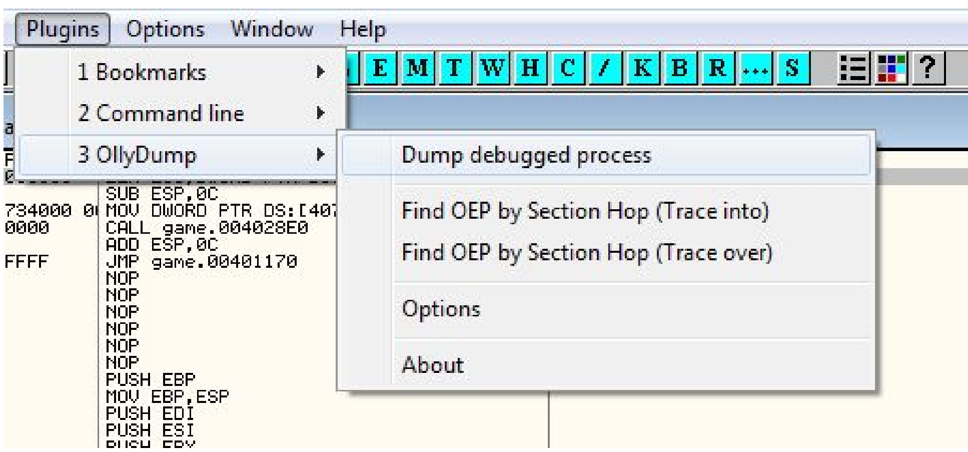
- General technique: find OEP
- Dump
- Fix imports
- Reverse engineer

`jmp reg / call reg / ret`

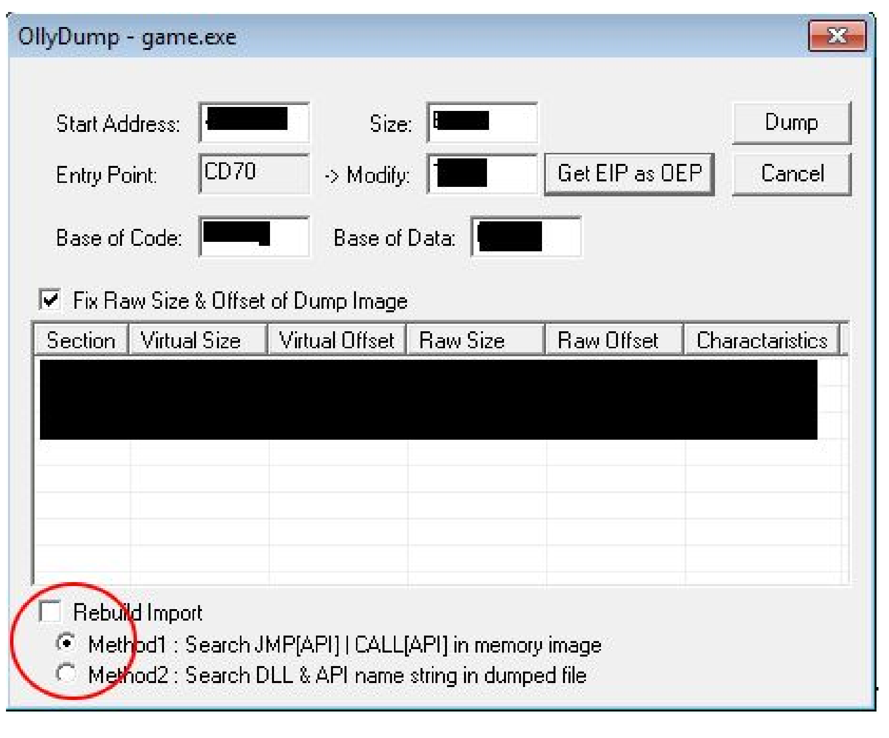


Unpacking manually



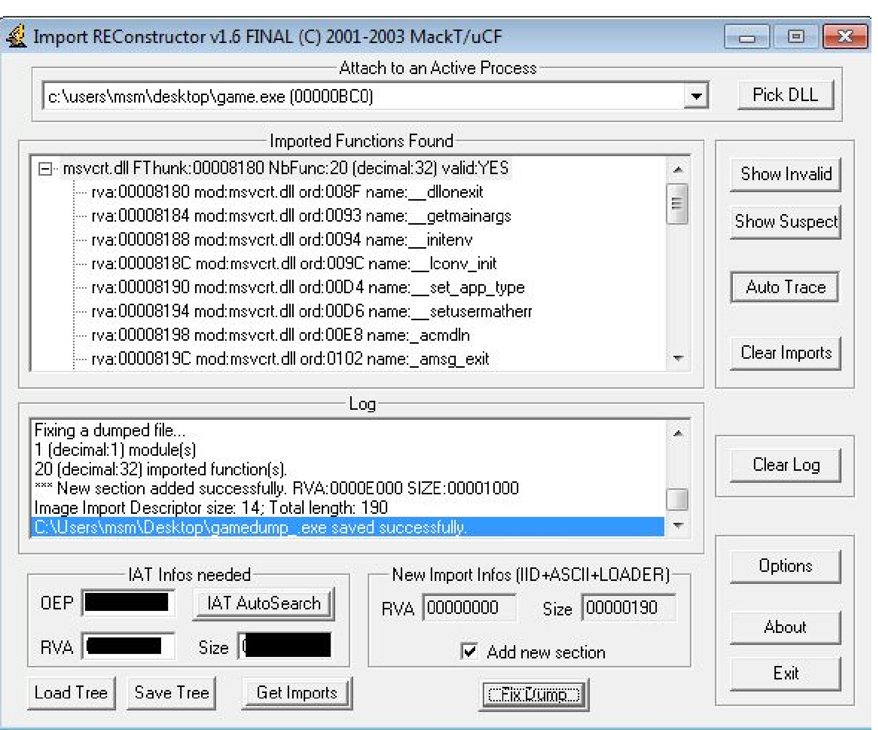


Unpacking manually

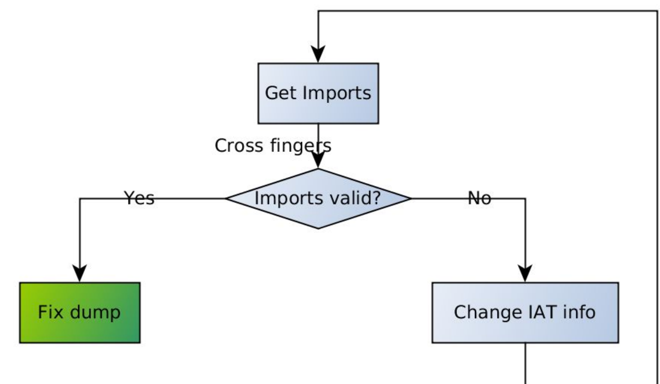


- Set correct OEP (usually EIP == OEP)
- Usually address and size is correct, but you might want to verify
- **Don't** rebuild IAT (usually fails)
- Dump

Unpacking manually



- Attach to running process
- Fill *"IAT infos needed"*
- Click "Get Imports"
- Cross fingers (optional)



Exercise 0x2

- Great computer game!
- Unfortunately, needs a license file
- But you can patch executable
- Unfortunately, executable is packed
- Goal: runnable game without license

http://uw2017.p4.team/static/game.exe

Exercise 0x2: hints

- Hint 0x0: don't try to reverse engineer packer, just find jump to EP.
- Hint 0x1: one of the first instructions is `pushad`. So one of the last instructions will be ...?
- Hint 0x2 (optional): usually OEP == first jump to different code section.

Exercise 0x2: solution

```
                push    eax
                push    1
                push    ebx
                call    ecx
                popa
                lea     eax, [esp+38h+var_B8]

loc_40CF0A:                                    ; COD
                push    0
                cmp     esp, eax
                jnz     short loc_40CF0A
                sub     esp, 0FFFFFF80h
                jmp     loc_4014E0
```

Entry point

Algorithm REconstruction

Actually reverse-engineering something

Exercise 0x3

- We are given flag right away
- Unfortunately, it's encrypted
- But we have encryptor and password
- Unfortunately, decryption is not implemented
- Goal: decrypt the flag (password = secret1234)

http://uw2017.p4.team/static/encryptor.exe

http://uw2017.p4.team/static/flag.enc

Exercise 0x3: hints 0

- Don't RE *too much*
- Important thing: find encryption function
- Somewhere in program find loop ~~ this:

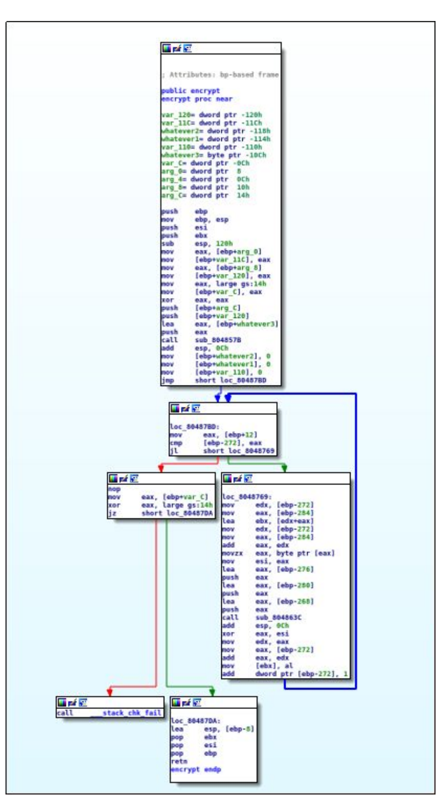
```
while (something) {
    fread(buffer, ...);
    encrypt(buffer, ...);
    fwrite(buffer, ...);
}
```

Exercise 0x3: hints 1

```
; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

var_4= dword ptr -4
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

lea     ecx, [esp+4]
and     esp, 0FFFFFFF0h
push    dword ptr [ecx-4]
push    ebp
mov     ebp, esp
push    ecx
sub     esp, 4
mov     eax, ecx
sub     esp, 4
push    offset encrypt
push    dword ptr [eax+4]
push    dword ptr [eax]
call    real_main
add     esp, 10h
mov     eax, 0
mov     ecx, [ebp+var_4]
leave
lea     esp, [ecx-4]
retn
main endp
```

```
real_main(..., ..., encrypt);
```

Exercise 0x3: hints 2



```
// whatever
// whatever
// whatever
// whatever
for (int i = 0; i < xxx; i++) {
    // real encryption!
}
// whatever
// whatever
// whatever
```

Exercise 0x3: hints 3

```
loc_80487BD:
mov     eax, [ebp+12]
cmp     [ebp-272], eax
jl      short loc_8048769
```

```
var_C]
gs:14h
80487DA
```

```
loc_8048769:
mov     edx, [ebp-272]
mov     eax, [ebp-284]
lea     ebx, [edx+eax]
mov     edx, [ebp-272]
mov     eax, [ebp-284]
add     eax, edx
movzx   eax, byte ptr [eax] ; eax = next in byte
mov     esi, eax            ; esi = eax
lea     eax, [ebp-276]
push    eax                 ; push whatever
lea     eax, [ebp-280]
push    eax                 ; push whatever
lea     eax, [ebp-268]
push    eax                 ; push whatever
call    whatever            ; eax = whatever(a, b, c)
add     esp, 0Ch
xor     eax, esi            ; eax ^= esi
mov     edx, eax            ; edx = eax
mov     eax, [ebp-272]
add     eax, edx            ; eax = [ebp-272] + edx
mov     [ebx], al           ; next output byte = al
add     dword ptr [ebp-272], 1 ; [ebp-272] += 1
```

```
eax = next_byte
esi = eax
eax = whatever()
eax ^= esi
edx = eax
eax = [ebp-272] + edx
out_byte = eax
[ebp-272] += 1
```

[ebp-272] == loop variable!

Exercise 0x3: hints 3

```
loc_80487BD:
mov     eax, [ebp+12]
cmp     [ebp-272], eax
jl      short loc_8048769
```

```
var_C]
 gs:14h
80487DA
```

```
loc_8048769:
mov     edx, [ebp-272]
mov     eax, [ebp-284]
lea     ebx, [edx+eax]
mov     edx, [ebp-272]
mov     eax, [ebp-284]
add     eax, edx
movzx   eax, byte ptr [eax] ; eax = next in byte
mov     esi, eax            ; esi = eax
lea     eax, [ebp-276]
push    eax                 ; push whatever
lea     eax, [ebp-280]
push    eax                 ; push whatever
lea     eax, [ebp-268]
push    eax                 ; push whatever
call    whatever            ; eax = whatever(a, b, c)
add     esp, 0Ch
xor     eax, esi            ; eax ^= esi
mov     edx, eax            ; edx = eax
mov     eax, [ebp-272]
add     eax, edx            ; eax = [ebp-272] + edx
mov     [ebx], al           ; next output byte = al
add     dword ptr [ebp-272], 1 ; [ebp-272] += 1
```

```
esi = next_byte
edx = whatever() ^ esi
eax = [ebp-272] + edx
out_byte = eax
[ebp-272] += 1
```

`[ebp-272] == loop variable!`

Exercise 0x3: hints 3

```
loc_80487BD:
mov     eax, [ebp+12]
cmp     [ebp-272], eax
jl      short loc_8048769
```

```
var_C]
 gs:14h
80487DA
```

```
loc_8048769:
mov     edx, [ebp-272]
mov     eax, [ebp-284]
lea     ebx, [edx+eax]
mov     edx, [ebp-272]
mov     eax, [ebp-284]
add     eax, edx
movzx   eax, byte ptr [eax] ; eax = next in byte
mov     esi, eax            ; esi = eax
lea     eax, [ebp-276]
push    eax                 ; push whatever
lea     eax, [ebp-280]
push    eax                 ; push whatever
lea     eax, [ebp-268]
push    eax                 ; push whatever
call    whatever            ; eax = whatever(a, b, c)
add     esp, 0Ch
xor     eax, esi            ; eax ^= esi
mov     edx, eax            ; edx = eax
mov     eax, [ebp-272]
add     eax, edx            ; eax = [ebp-272] + edx
mov     [ebx], al           ; next output byte = al
add     dword ptr [ebp-272], 1 ; [ebp-272] += 1
```

```
out_byte = [ebp-272] + whatever() ^ next_byte

[ebp-272] += 1
```

[ebp-272] == loop variable!

Exercise 0x3: solution

Standard RC4 stream cipher used

(bonus points if you figured that out)

```
void rc4keysched(unsigned char state[], const uint8_t *key, size_t len) {
   for (int i = 0; i < 256; ++i) {
       state[i] = i;
   }
   int j = 0;
   for (int i = 0; i < 256; ++i) {
       j = (j + state[i] + key[i % len]) % 256;
       int t = state[i];
       state[i] = state[j];
       state[j] = t;
   }
}

char rc4rand(unsigned char state[], int *i, int *j) {
    *i = (*i + 1) % 256;
    *j = (*j + state[*i]) % 256;
    int t = state[*i];
    state[*i] = state[*j];
    state[*j] = t;
    return state[(state[*i] + state[*j]) % 256];
}
```

Exercise 0x3: solution

..but recognising RC4 is not necessary

Encryption is almost symmetric

```
void encrypt(uint8_t *data, size_t data_size, const uint8_t *key, size_t
key_size) {
    unsigned char state[256];
    rc4keysched(state, key, key_size);
    int a = 0, b = 0;
    for (int i = 0; i < (int)data_size; i++) {
        data[i] = (data[i] ^ rc4rand(state, &a, &b)) + i;
    }
}

void decrypt(uint8_t *data, size_t data_size, const uint8_t *key, size_t
key_size) {
    unsigned char state[256];
    rc4keysched(state, key, key_size);
    int a = 0, b = 0;
    for (int i = 0; i < (int)data_size; i++) {
        data[i] = (data[i] - i) ^ rc4rand(state, &a, &b);
    }
}
```

Exercise 0x3: solution

Expected solutions:

- Python/ruby/(...) script
- Patching few bytes in executable

Unexpected solutions:

- Ecsm2016
- More?

Bibliography

"Praktyczna inżynieria wsteczna" - Mateusz Jurczyk, Gynvael Coldwind

"Reversing: Secrets of Reverse Engineering" - Eldad Eilam

"Practical Reverse Engineering: x86" - Alexandre Gazet, Bruce Dang, and Elias Bachaalany