HITB SECCONF SIN-2021

IDA2Obj: Static Binary Instrumentation On Steroids

Mickey Jin

Security Researcher, Trend Micro

TRACK 2

# whoami

- Security Researcher from Trend Micro
- Malware Analyst
- Vulnerability Hunter
- 50+ CVEs since last year
- Reverse engineering and debugging enthusiasts
- @patch1t

About This Talk

- Many popular fuzzers are Code Coverage Guided
  - afl, honggfuzz, syzkaller, ...
- Easy for open source project
  - https://clang.llvm.org/docs/SanitizerCoverage.html
- How about the close sourced binaries ?
  - DBI is the most choice
    - Dynamorio, Frida stalker, ...
  - SBI is cooler, and faster
    - There are some existing SBI tools, seems no perfect solutions yet
    - I have new ideas for the implementation

What is SBI/DBI ?

[Static | Dynamic] Binary Instrumentation

Static: Analysing programs at compile/build-time

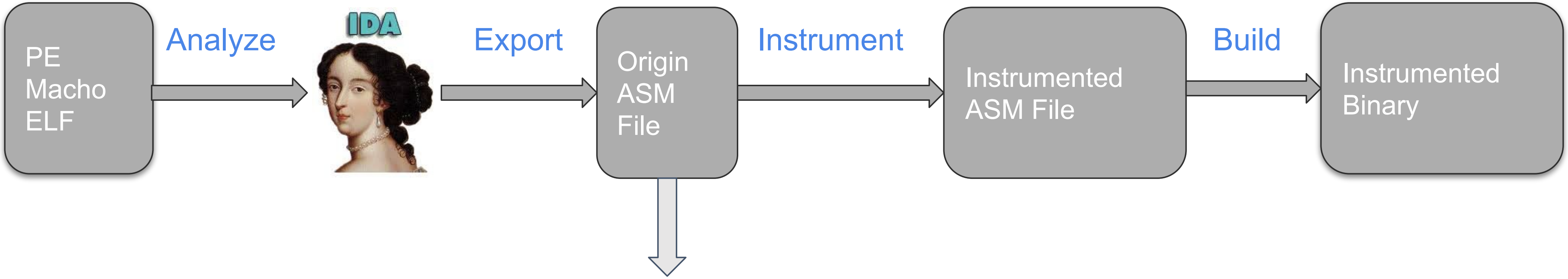
Dynamic: Analysing programs at run-time

Binary: Analysing programs at machine code level, without having access to source code

Instrumentation: The act of adding extra code to a program to measure its performance, diagnose errors and write trace information

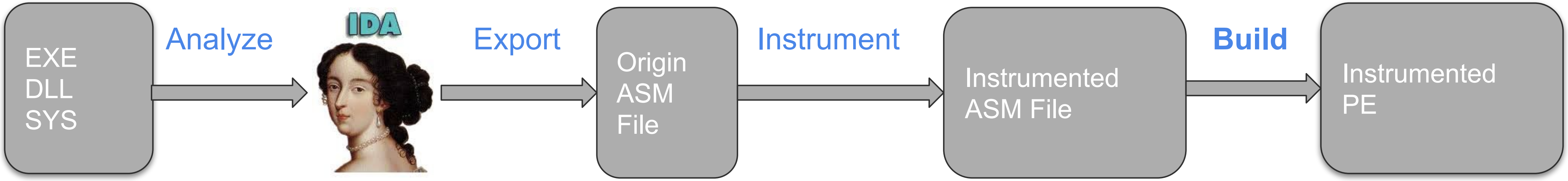
Well, start from scratch now.

My First Idea



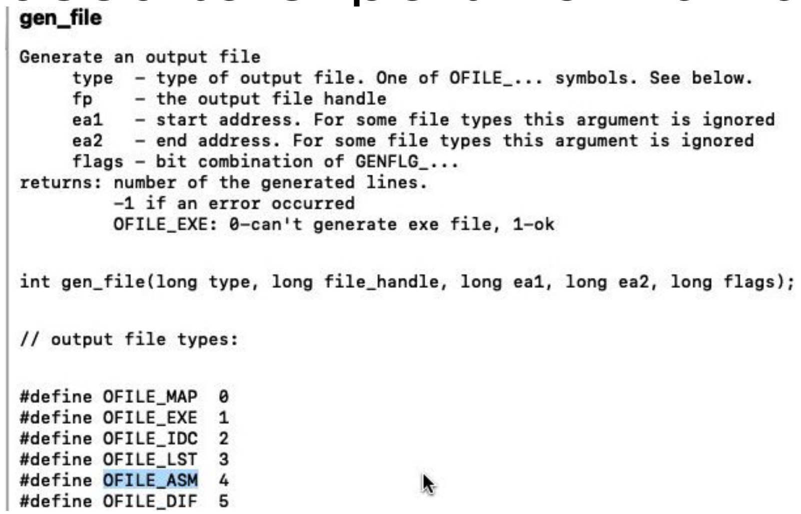
“Source Code” at
Assembly Level

IDA2MASM: My First Solution For PE



Export ASM File

- Export ALL to one ASM file by using IDA menu “File -> Produce File -> Create ASM File”
  - Shortcut “Alt+F10”
  - MASM may cost many hours/days to assemble one ASM file
- The script API can be used to export from an address range



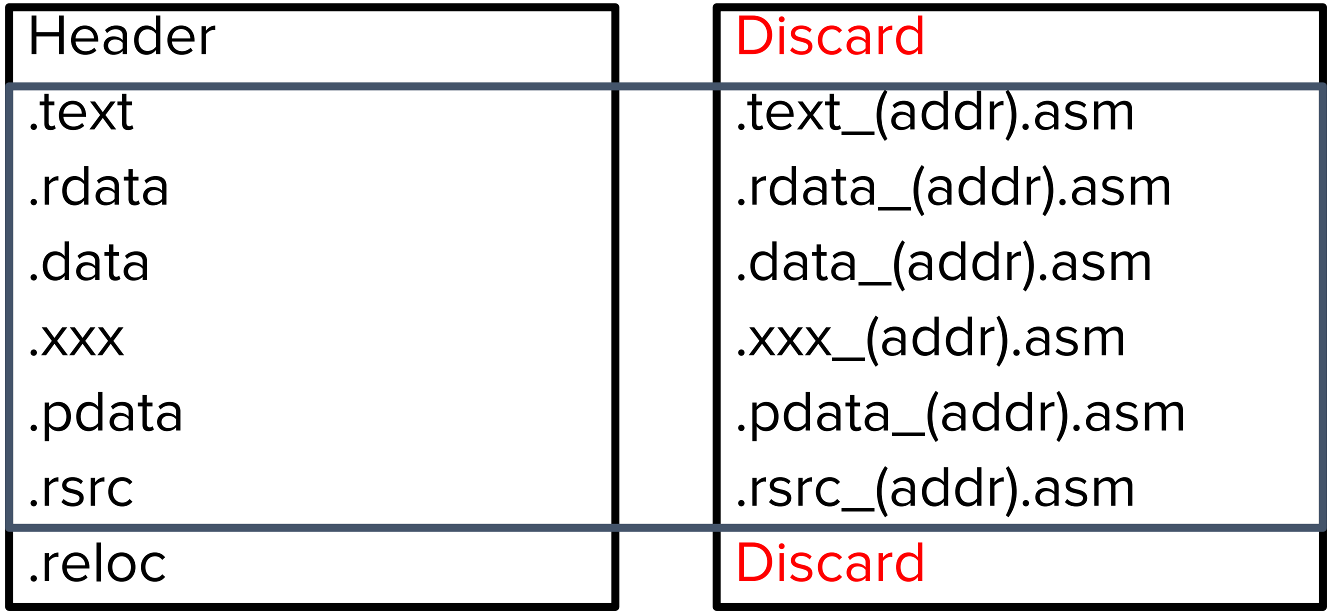
gen_file

```
Generate an output file
     type   - type of output file. One of OFILE_... symbols. See below.
     fp     - the output file handle
     ea1    - start address. For some file types this argument is ignored
     ea2    - end address. For some file types this argument is ignored
     flags - bit combination of GENFLG_...
returns: number of the generated lines.
         -1 if an error occurred
         OFILE_EXE: 0-can't generate exe file, 1-ok

int gen_file(long type, long file_handle, long ea1, long ea2, long flags);

// output file types:

#define OFILE_MAP  0
#define OFILE_EXE  1
#define OFILE_IDC  2
#define OFILE_LST  3
#define OFILE_ASM  4
#define OFILE_DIF  5
```

TIP: Before exporting, “Unhide All” first

Split By Segments

- Manually load all segments, including **Header** and **.rsrc**
- Enum all segments to dump, except for **Header** and **.reloc**
- Segments may share the same name, append their address to be unique



Header	Discard
.text	.text_(addr).asm
.rdata	.rdata_(addr).asm
.data	.data_(addr).asm
.xxx	.xxx_(addr).asm
.pdata	.pdata_(addr).asm
.rsrc	.rsrc_(addr).asm
.reloc	Discard

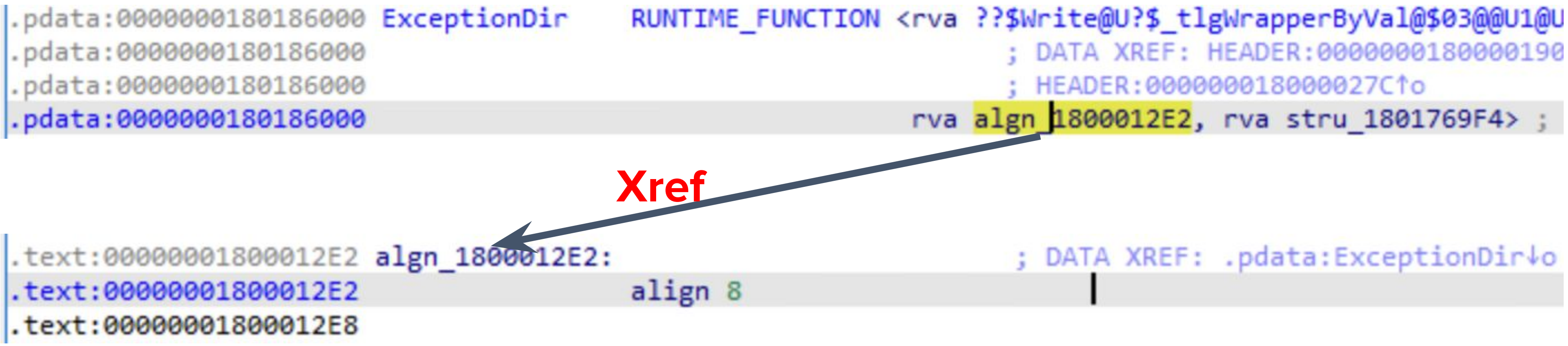
Symbol References

- Expose a symbol to linker explicitly
  - public xxx_symbol
  - xxx_label ::

Symbol References

- Expose a symbol to linker explicitly
  - public xxx_symbol
  - xxx_symbol ::
- Declare the external symbols
  - extern sub_xxx:proc
  - extern byte_xxx:byte
  - extern qword_xxx:qword

Symbol References



```
.pdata:0000000180186000 ExceptionDir    RUNTIME_FUNCTION <rva ??$Write@U?$_tlgWrapperByVal@$03@@U1@U
.pdata:0000000180186000                                              ; DATA XREF: HEADER:0000000180000190
.pdata:0000000180186000                                              ; HEADER:000000018000027C↑o
.pdata:0000000180186000                                           rva algn_1800012E2, rva stru_1801769F4> ;

.text:00000001800012E2 algn_1800012E2:                              ; DATA XREF: .pdata:ExceptionDir↓o
.text:00000001800012E2                 align 8
.text:00000001800012E8
```

.pdata_180186000.asm

…

extern algn_1800012E2:proc

…

…

.text_180001000.asm

public algn_1800012E2

…

algn_1800012E2**::** align (8)

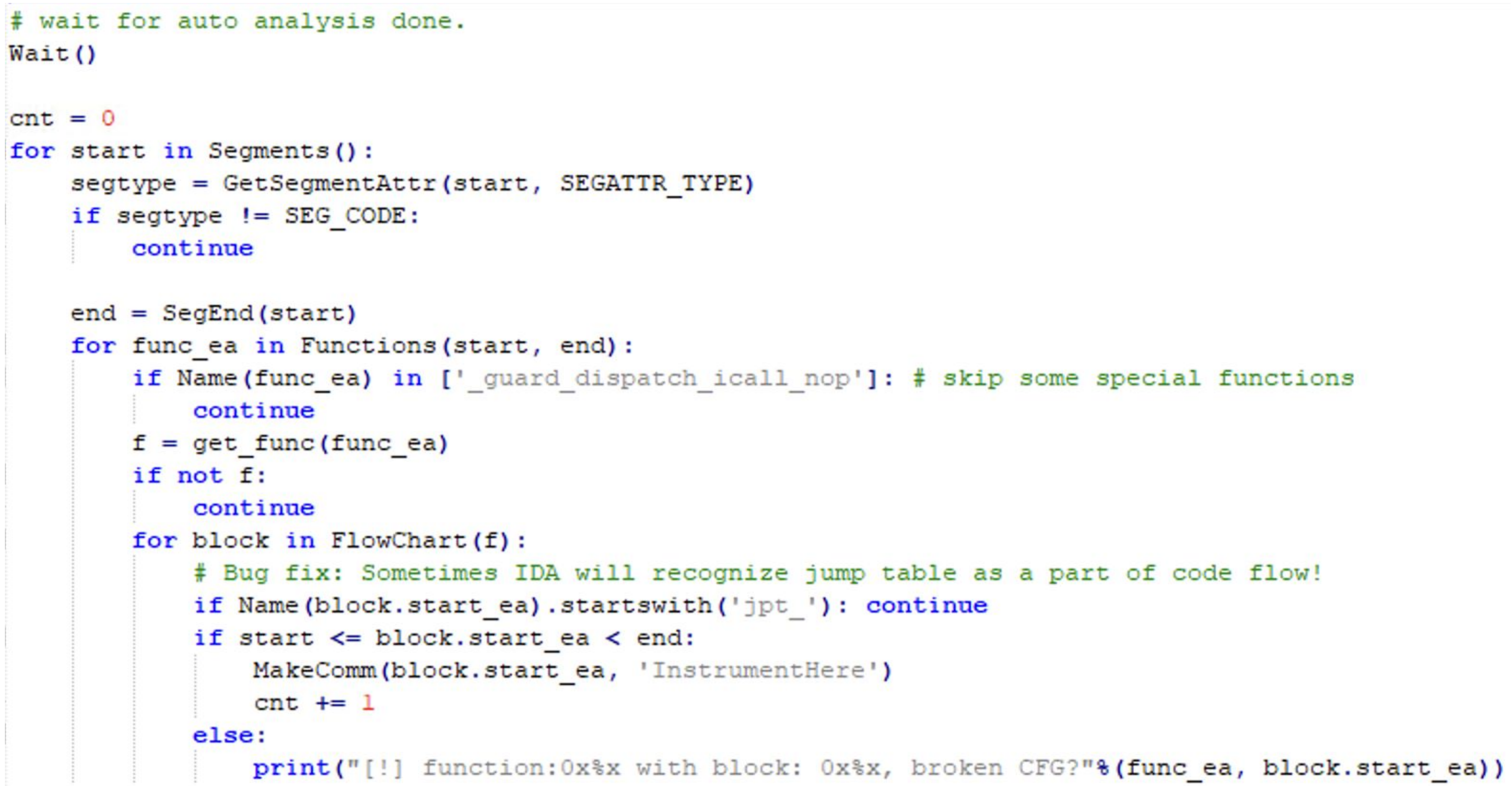
…

Symbol References

- Scan all the items from MinEA() to MaxEA()
- For each item, get all Xrefs list To its address
    - If no Xref, skip the item
    - If has Xrefs, make its name **public**
        - For each item in the Xrefs list, if not in the same segment, add an **extern** declaration for that item.

Instrumentation points

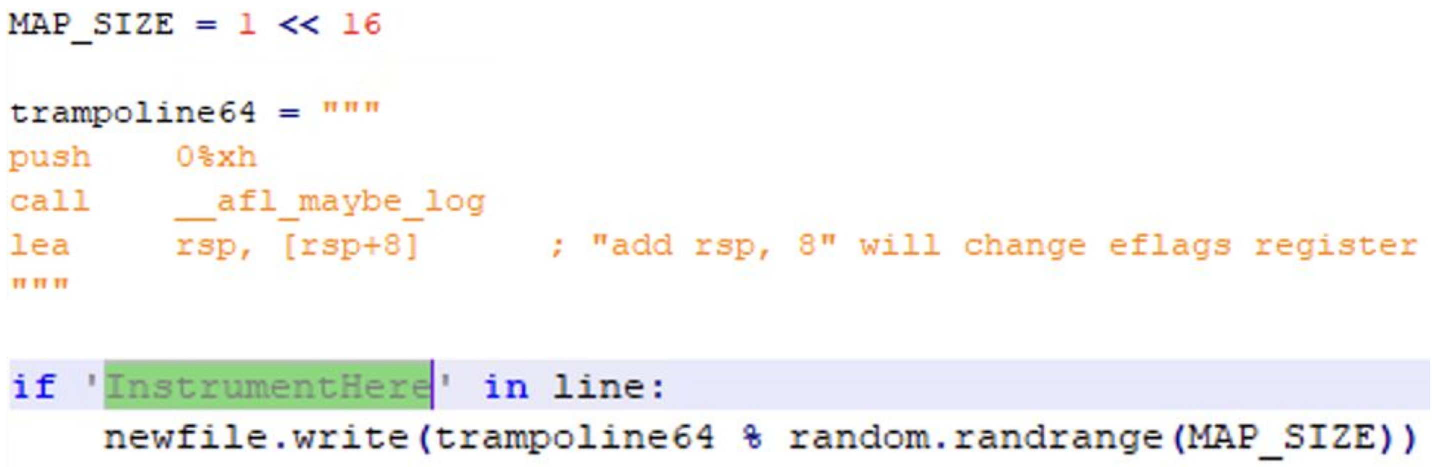
- Scan all functions from all segments
- For each function, scan all code blocks
- For each block, make a comment “InstrumentHere” as a hint



```
# wait for auto analysis done.
Wait()

cnt = 0
for start in Segments():
    segtype = GetSegmentAttr(start, SEGATTR_TYPE)
    if segtype != SEG_CODE:
        continue

    end = SegEnd(start)
    for func_ea in Functions(start, end):
        if Name(func_ea) in ['_guard_dispatch_icall_nop']: # skip some special functions
            continue
        f = get_func(func_ea)
        if not f:
            continue
        for block in FlowChart(f):
            # Bug fix: Sometimes IDA will recognize jump table as a part of code flow!
            if Name(block.start_ea).startswith('jpt_'): continue
            if start <= block.start_ea < end:
                MakeComm(block.start_ea, 'InstrumentHere')
                cnt += 1
            else:
                print("[!] function:0x%x with block: 0x%x, broken CFG?"%(func_ea, block.start_ea))
```

Instrumentation

During the post-processing of the asm files, insert the trampoline instructions before the comment string “InstrumentHere”



```
MAP_SIZE = 1 << 16

trampoline64 = """
push    0%xh
call    __afl_maybe_log
lea     rsp, [rsp+8]       ; "add rsp, 8" will change eflags register
"""

if 'InstrumentHere' in line:
    newfile.write(trampoline64 % random.randrange(MAP_SIZE))
```

Re-Assemble

Damn MASM !

- Too many grammar errors (Cost me lots of time 😑)
  - Tune later
  - Fixed by a python script during ASM file pre-processing stage
- Symbol max length limitation
  - rename to a short name
- MASM is too ancient, maybe I should try other assemblers 🤔

Tune Grammar List (Partial)

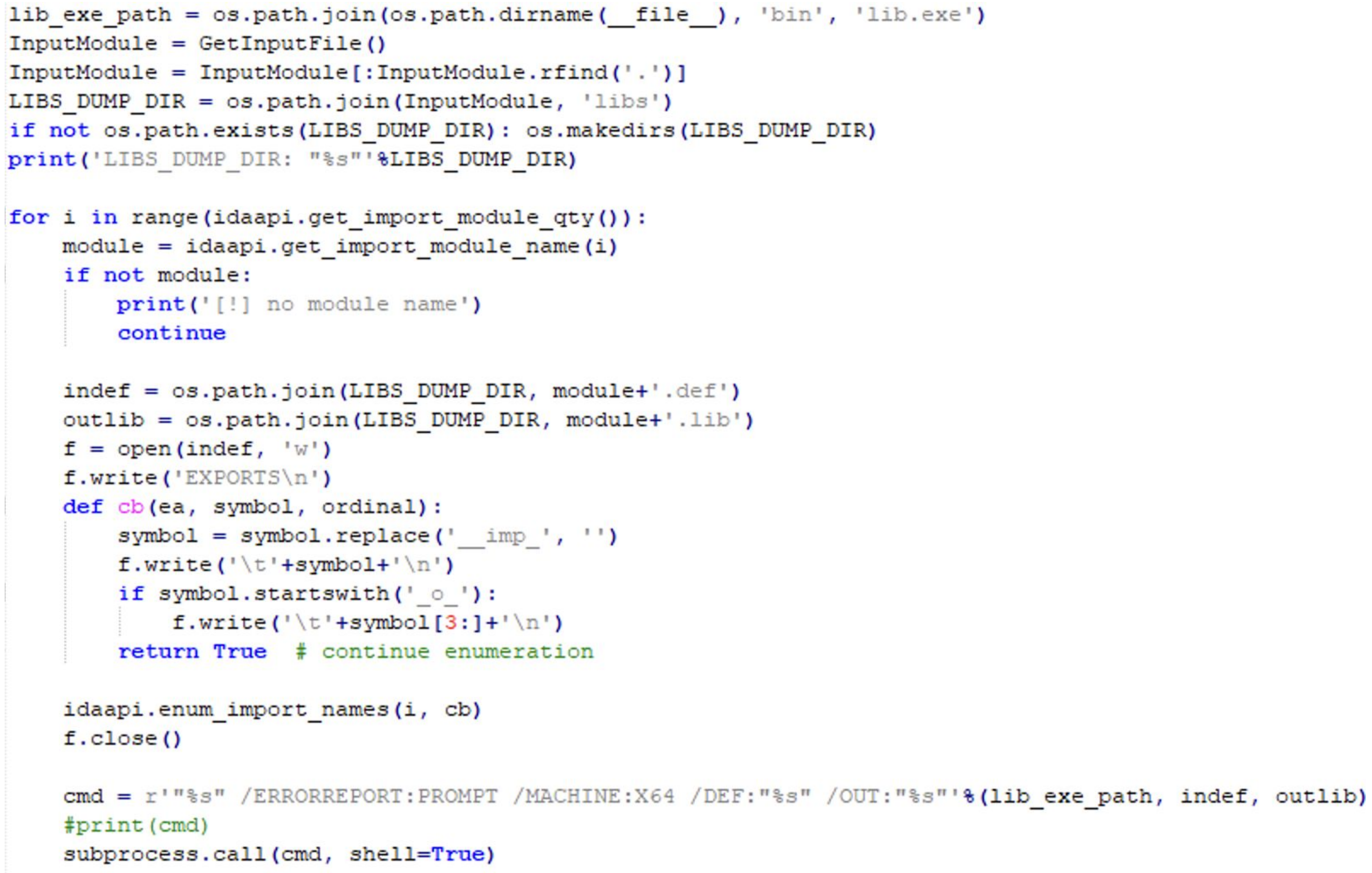
IDA ASM	MASM
retn	ret
dd rva xxx_symbol	dd imagerel xxx_symbol
align 10h	align (10h)
movq	movd
xmmword	oword
call cs:xxx_symbol	call qword ptr xxx_symbol
jmp **short** xxx_symbol	jmp xxx_symbol
…...	…...

Link Issue

- The API symbols from the imports table are undefined
- Don’t know what’s the lib file to link with
- Maybe some import symbols are from private SDKs

Link Solution

- Enum all import modules
- Create a **def** file for each module
- call **lib.exe** to generate the lib file from **def** file



```
lib_exe_path = os.path.join(os.path.dirname(__file__), 'bin', 'lib.exe')
InputModule = GetInputFile()
InputModule = InputModule[:InputModule.rfind('.')]
LIBS_DUMP_DIR = os.path.join(InputModule, 'libs')
if not os.path.exists(LIBS_DUMP_DIR): os.makedirs(LIBS_DUMP_DIR)
print('LIBS_DUMP_DIR: "%s"'%LIBS_DUMP_DIR)

for i in range(idaapi.get_import_module_qty()):
    module = idaapi.get_import_module_name(i)
    if not module:
        print('[!] no module name')
        continue

    indef = os.path.join(LIBS_DUMP_DIR, module+'.def')
    outlib = os.path.join(LIBS_DUMP_DIR, module+'.lib')
    f = open(indef, 'w')
    f.write('EXPORTS\n')
    def cb(ea, symbol, ordinal):
        symbol = symbol.replace('__imp_', '')
        f.write('\t'+symbol+'\n')
        if symbol.startswith('_o_'):
            f.write('\t'+symbol[3:]+'\n')
        return True  # continue enumeration

    idaapi.enum_import_names(i, cb)
    f.close()

    cmd = r'"%s" /ERRORREPORT:PROMPT /MACHINE:X64 /DEF:"%s" /OUT:"%s"'%(lib_exe_path, indef, outlib)
    #print(cmd)
    subprocess.call(cmd, shell=True)
```

Patch The New Built Binary

- Patch the PE header, such as data directory
  - export data directory points to the location of the symbol **ExportDir**
  - exception data directory points to the location of the symbol **ExceptionDir**
  - ...
- Fix the data entry in the **.rsrc** segment
  - data entry value is relative to image base address
- All is in one script for automation

Run & Test, Crash

Crash Root Cause

- unrecognized pointer
  - dq offset xxx_symbol
- unrecognized image based relative value
  - dd rva xxx_symbol
- unrecognized function based relative value
  - a compression-encoded value for exception handling
  - refer to: https://devblogs.microsoft.com/cppblog/making-cpp-exception-handling-smaller-x64

I will talk how to fix these issues later

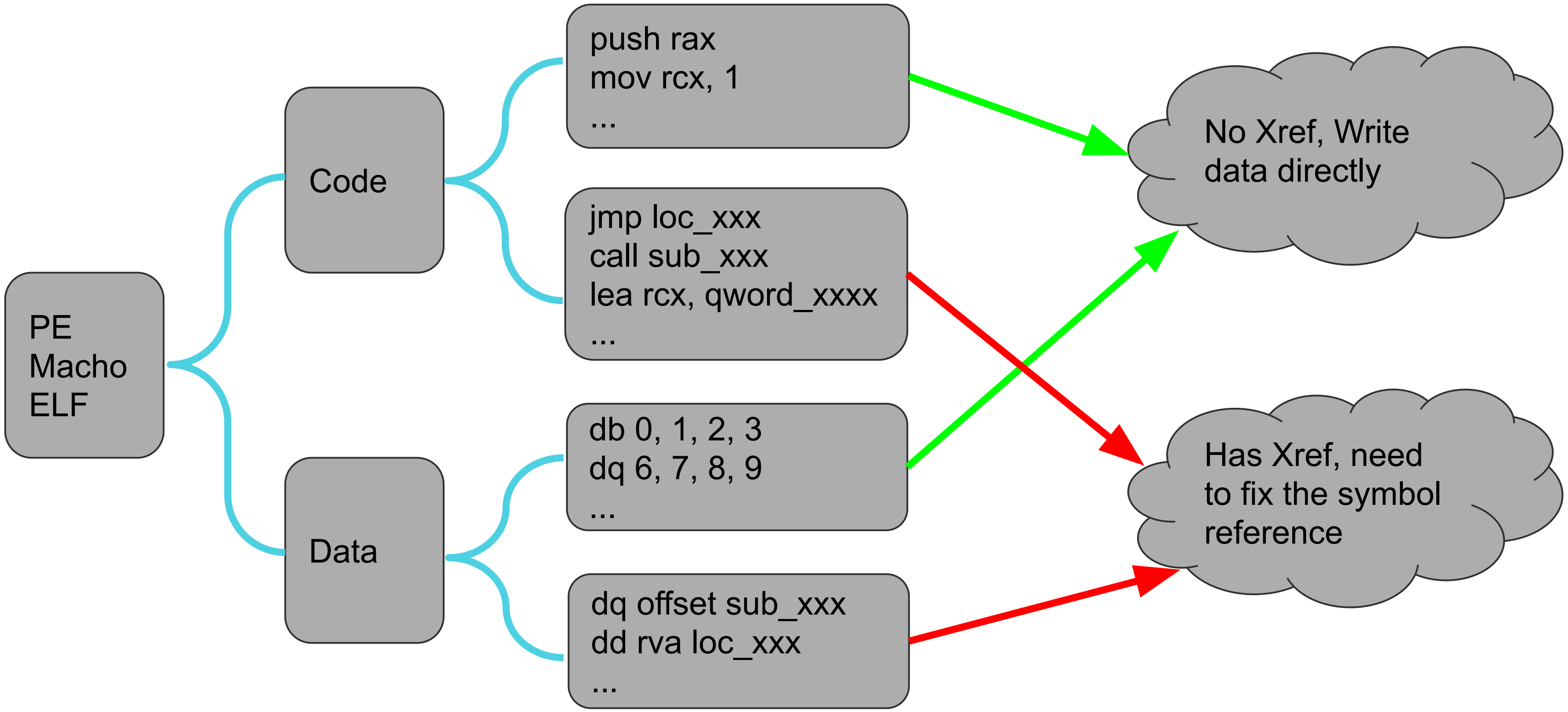
IDA2MASM works fine now

But it’s not suitable for full-automation

Due to some corner cases of grammar tuning

Thinking deeper,
I have another idea

Think Of The Essence



PE
Macho
ELF

Code

Data

push rax
mov rcx, 1
...

jmp loc_xxx
call sub_xxx
lea rcx, qword_xxxx
...

db 0, 1, 2, 3
dq 6, 7, 8, 9
...

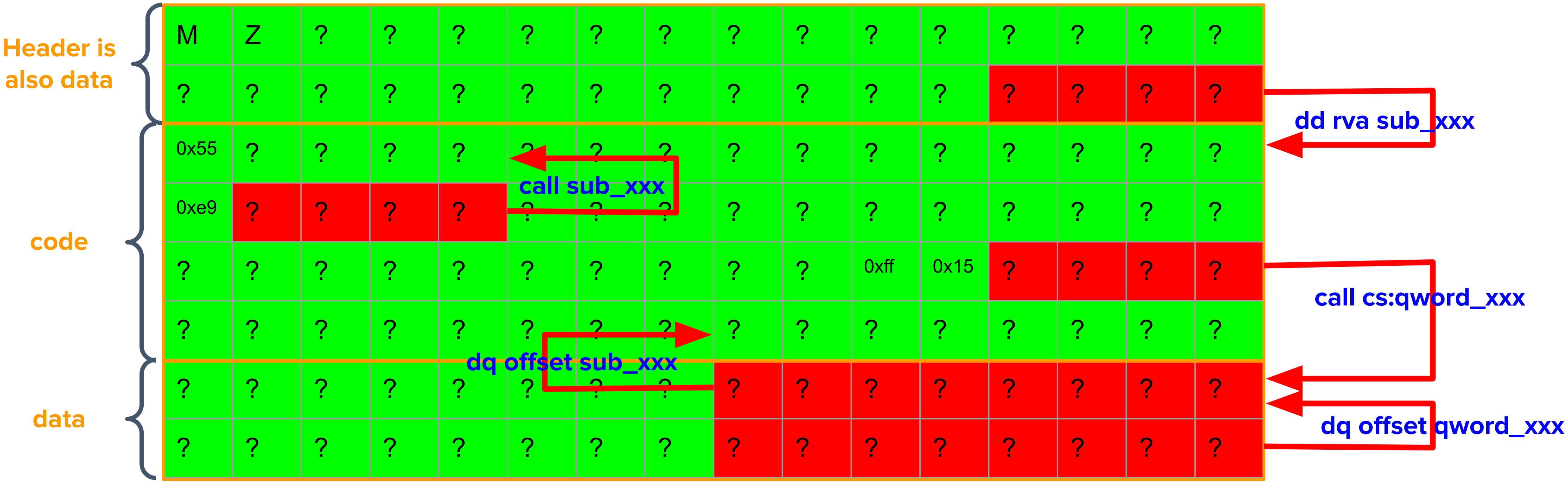
dq offset sub_xxx
dd rva loc_xxx
...

No Xref, Write data directly

Has Xref, need to fix the symbol reference

Think Of The Essence

All is Xrefs



M	Z	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
0x55	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
0xe9	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	0xff	0x15	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

My Second Idea (Algorithm)

1. Scan all the instructions from MinEA() to MaxEA(), record their addresses as the old coordinate system.

My Second Idea (Algorithm)

1. Scan all the instructions from MinEA() to MaxEA(), record their addresses as the old coordinate system.
2. During the scanning, output the machine code for each instruction.
    a. If the instruction has no reference, output its machine code directly.
    b. Otherwise, output its opcode first, then output dummy bytes as place holder and record its address and reference type into a fix table.

My Second Idea (Algorithm)

1. Scan all the instructions from MinEA() to MaxEA(), record their addresses as the old coordinate system.
2. During the scanning, output the machine code for each instruction.
   a. If the instruction has no reference, output its machine code directly.
   b. Otherwise, output its opcode first, then output dummy bytes as place holder and record its address and reference type into a fix table.
3. During the scanning, insert the trampoline instructions before the instruction with comment “InstrumentHere”.

My Second Idea (Algorithm)

1. Scan all the instructions from MinEA() to MaxEA(), record their addresses as the old coordinate system.
2. During the scanning, output the machine code for each instruction.
    a. If the instruction has no reference, output its machine code directly.
    b. Otherwise, output its opcode first, then output dummy bytes as place holder and record its address and reference type into a fix table.
3. During the scanning, insert the trampoline instructions before the instruction with comment “InstrumentHere”.
4. After the scanning, the size of temporary output binary file will be larger.

My Second Idea (Algorithm)

1. Scan all the instructions from MinEA() to MaxEA(), record their addresses as the old coordinate system.
2. During the scanning, output the machine code for each instruction.
   a. If the instruction has no reference, output its machine code directly.
   b. Otherwise, output its opcode first, then output dummy bytes as place holder and record its address and reference type into a fix table.
3. During the scanning, insert the trampoline instructions before the instruction with comment “InstrumentHere”.
4. After the scanning, the size of temporary output binary file will be larger.
5. Compute the new coordinate of each instruction in the temporary binary file, record them as the new coordinate system.

My Second Idea (Algorithm)

1. Scan all the instructions from MinEA() to MaxEA(), record their addresses as the old coordinate system.
2. During the scanning, output the machine code for each instruction.
   a. If the instruction has no reference, output its machine code directly.
   b. Otherwise, output its opcode first, then output dummy bytes as place holder and record its address and reference type into a fix table.
3. During the scanning, insert the trampoline instructions before the instruction with comment “InstrumentHere”.
4. After the scanning, the size of temporary output binary file will be larger.
5. Compute the new coordinate of each instruction in the temporary binary file, record them as the new coordinate system.
6. Fix the place holder according to the fix table and the new coordinate system.

My Second Idea (Algorithm)

1. Scan all the instructions from MinEA() to MaxEA(), record their addresses as the old coordinate system.
2. During the scanning, output the machine code for each instruction.
   a. If the instruction has no reference, output its machine code directly.
   b. Otherwise, output its opcode first, then output dummy bytes as place holder and record its address and reference type into a fix table.
3. During the scanning, insert the trampoline instructions before the instruction with comment “InstrumentHere”.
4. After the scanning, the size of temporary output binary file will be larger.
5. Compute the new coordinate of each instruction in the temporary binary file, record them as the new coordinate system.
6. Fix the place holder according to the fix table and the new coordinate system.
7. Finally, we got a new instrumented binary file.

My Second Idea (Algorithm)

- It rewrites the binary directly, regardless of the file format
- It could be cross-platform in theory
- The key point is fixing all the symbol references (relocations), and it seems too complicated to implement …

Thinking Of IDA2MASM Again

- What does the MASM do ?

Thinking Of IDA2MASM Again

- What does the MASM do ?
- What happens during the build process ?

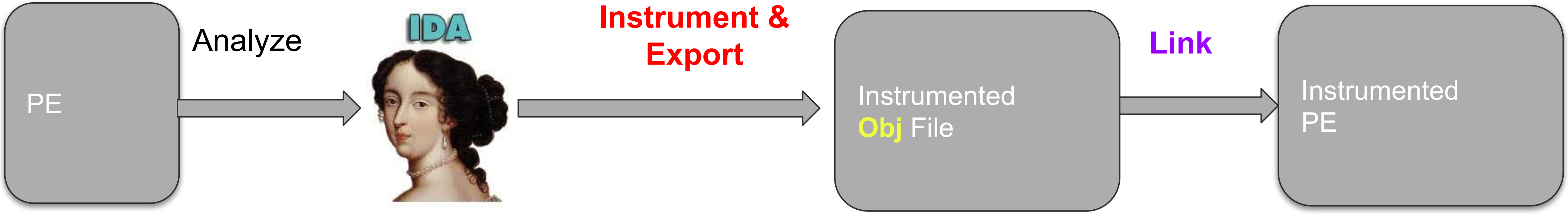
Thinking Of IDA2MASM Again

- What does the MASM do ?
- What happens during the build process ?
- Why don’t have to fix the references manually during the process of IDA2MASM ?

Linker Does The Magic

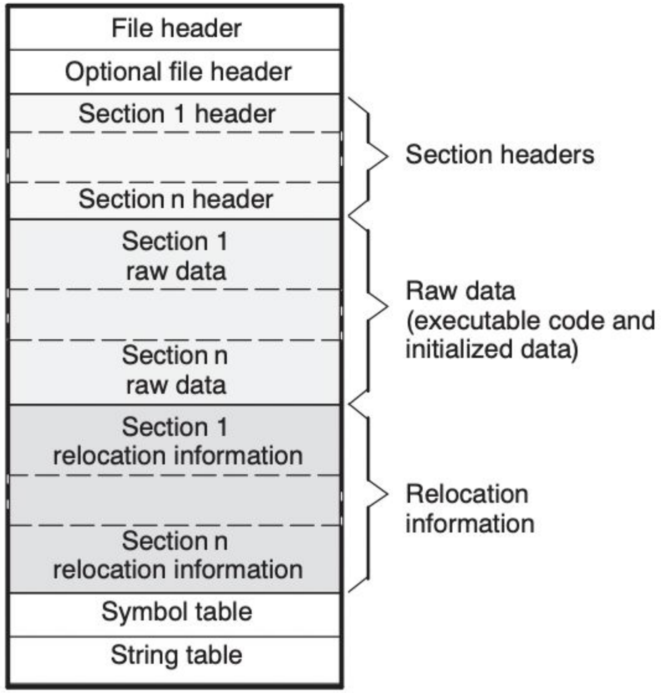
- MASM just translates the ASM to machine code, and adds the symbols to the object file :
  - public xxx_symbol -> One symbol record
  - extern xxx_symbol:type -> One symbol record + One relocation record
- It is the linker that helps to fix the symbol references in the final binary file
- So, can I directly generate the object files and make the linker help me do the fix ?

IDA2Obj: My Second Solution



Object File Format

- Object file is COFF (Common Object File Format)
  - Details: refer to https://www.ti.com/lit/an/spraao8/spraao8.pdf



File header

Optional file header

Section 1 header

Section n header

Section headers

Section 1 raw data

Section n raw data

Raw data (executable code and initialized data)

Section 1 relocation information

Section n relocation information

Relocation information

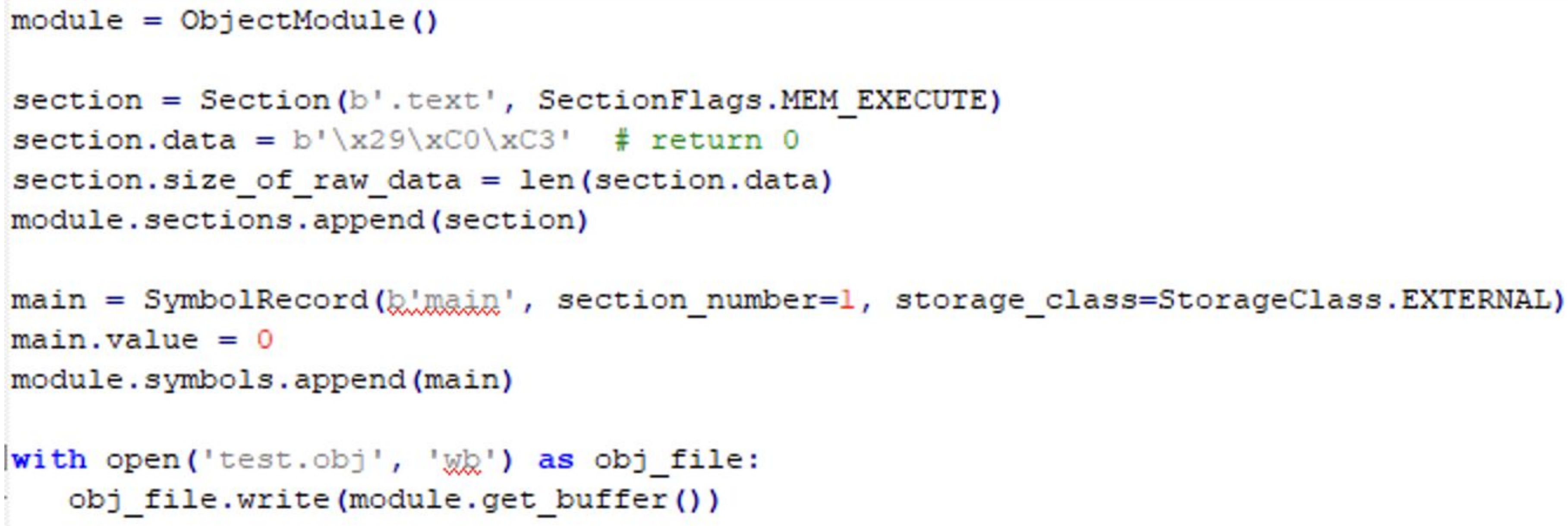
Symbol table

String table

cough: Object File Writer

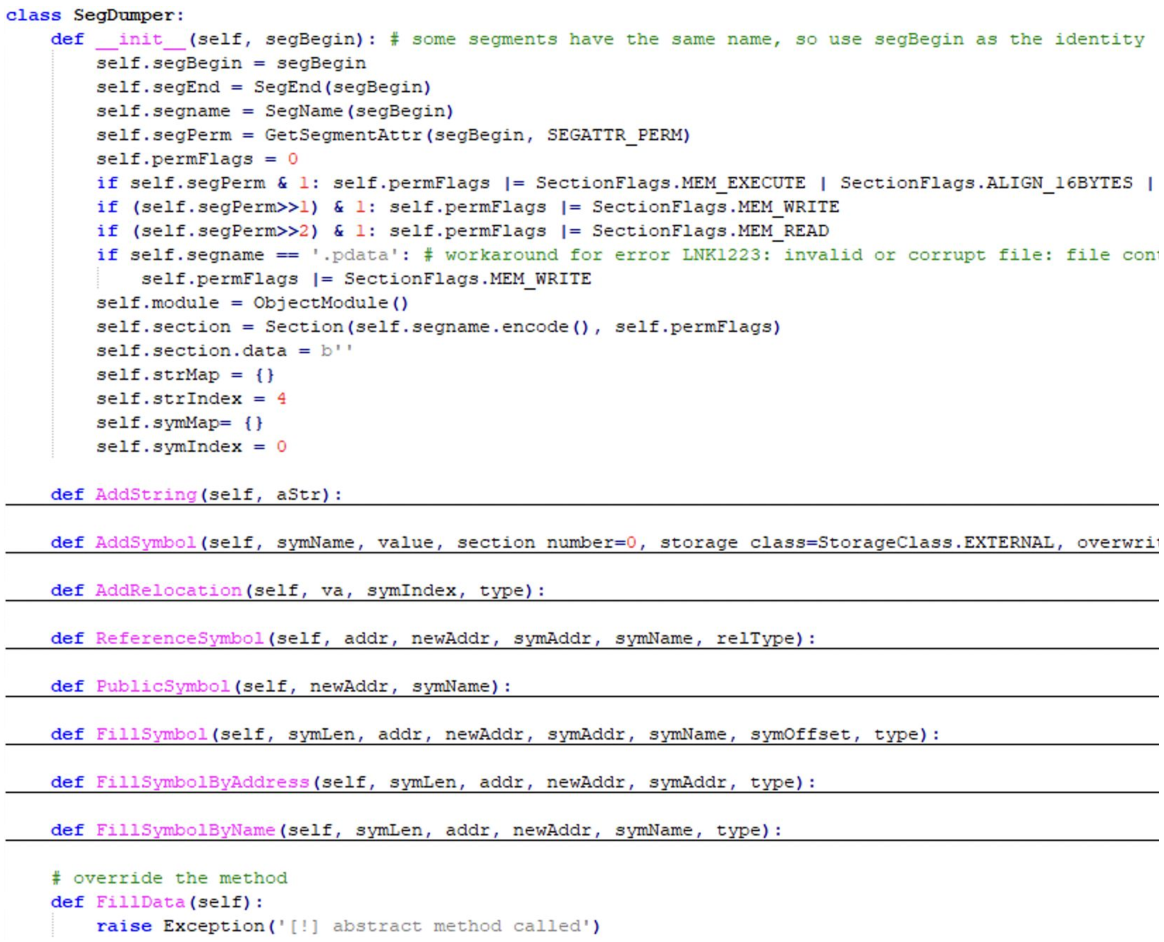
Don’t reinvent the wheels

- Repo : https://github.com/d3dave/cough
- Install: pip install cough
- Tutorial:



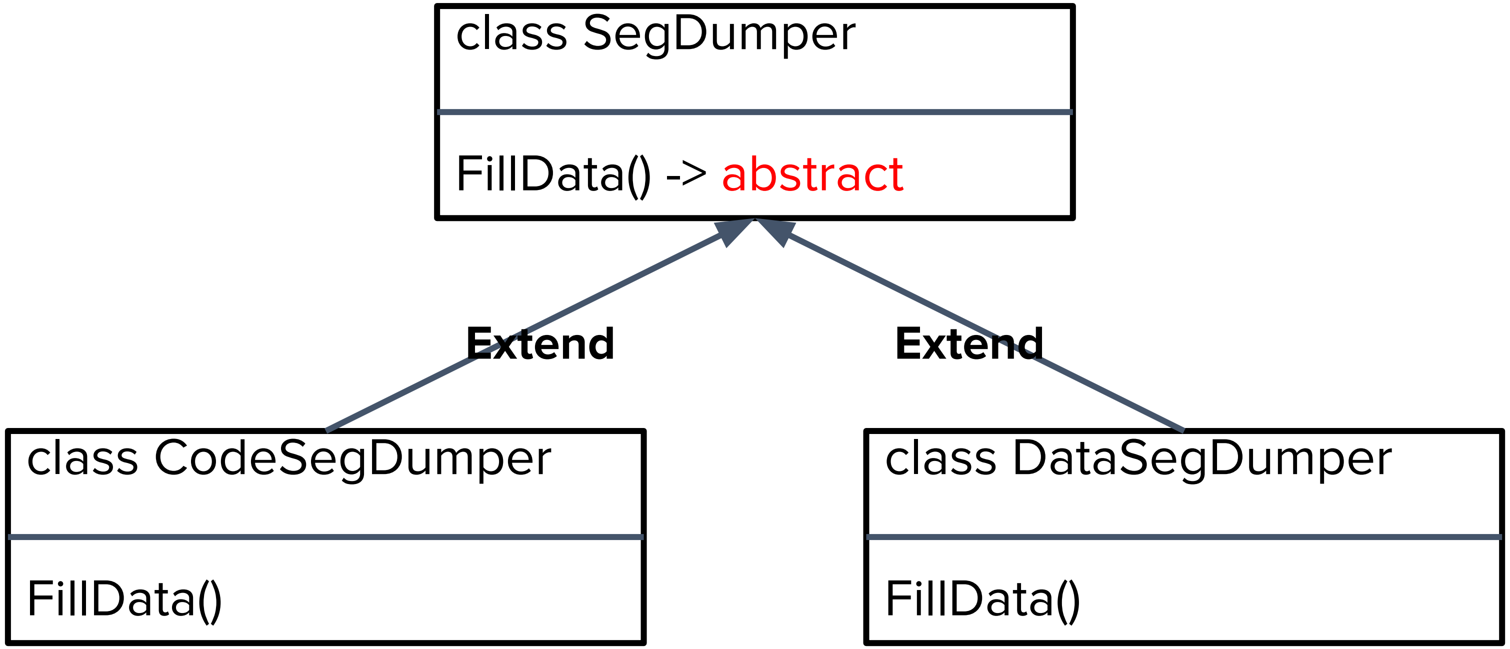
```
module = ObjectModule()

section = Section(b'.text', SectionFlags.MEM_EXECUTE)
section.data = b'\x29\xC0\xC3'  # return 0
section.size_of_raw_data = len(section.data)
module.sections.append(section)

main = SymbolRecord(b'main', section_number=1, storage_class=StorageClass.EXTERNAL)
main.value = 0
module.symbols.append(main)

with open('test.obj', 'wb') as obj_file:
    obj_file.write(module.get_buffer())
```

Encapsulate Some Primitives



```python
class SegDumper:
    def __init__(self, segBegin): # some segments have the same name, so use segBegin as the identity
        self.segBegin = segBegin
        self.segEnd = SegEnd(segBegin)
        self.segname = SegName(segBegin)
        self.segPerm = GetSegmentAttr(segBegin, SEGATTR_PERM)
        self.permFlags = 0
        if self.segPerm & 1: self.permFlags |= SectionFlags.MEM_EXECUTE | SectionFlags.ALIGN_16BYTES |
        if (self.segPerm>>1) & 1: self.permFlags |= SectionFlags.MEM_WRITE
        if (self.segPerm>>2) & 1: self.permFlags |= SectionFlags.MEM_READ
        if self.segname == '.pdata': # workaround for error LNK1223: invalid or corrupt file: file con
            self.permFlags |= SectionFlags.MEM_WRITE
        self.module = ObjectModule()
        self.section = Section(self.segname.encode(), self.permFlags)
        self.section.data = b''
        self.strMap = {}
        self.strIndex = 4
        self.symMap= {}
        self.symIndex = 0

    def AddString(self, aStr):

    def AddSymbol(self, symName, value, section_number=0, storage_class=StorageClass.EXTERNAL, overwri

    def AddRelocation(self, va, symIndex, type):

    def ReferenceSymbol(self, addr, newAddr, symAddr, symName, relType):

    def PublicSymbol(self, newAddr, symName):

    def FillSymbol(self, symLen, addr, newAddr, symAddr, symName, symOffset, type):

    def FillSymbolByAddress(self, symLen, addr, newAddr, symAddr, type):

    def FillSymbolByName(self, symLen, addr, newAddr, symName, type):

    # override the method
    def FillData(self):
        raise Exception('[!] abstract method called')
```

SegDumper



class SegDumper

FillData() -> abstract

Extend

Extend

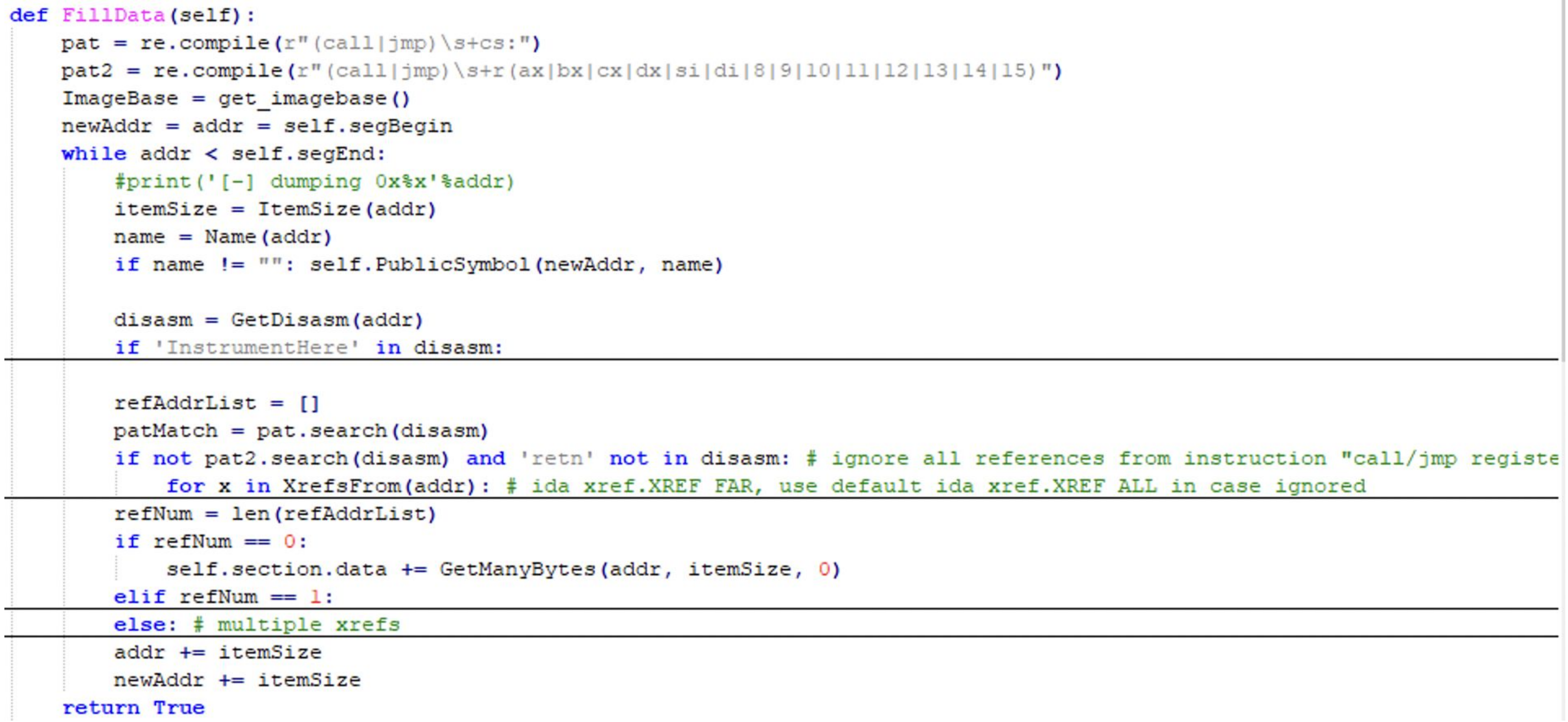
class CodeSegDumper

FillData()

class DataSegDumper

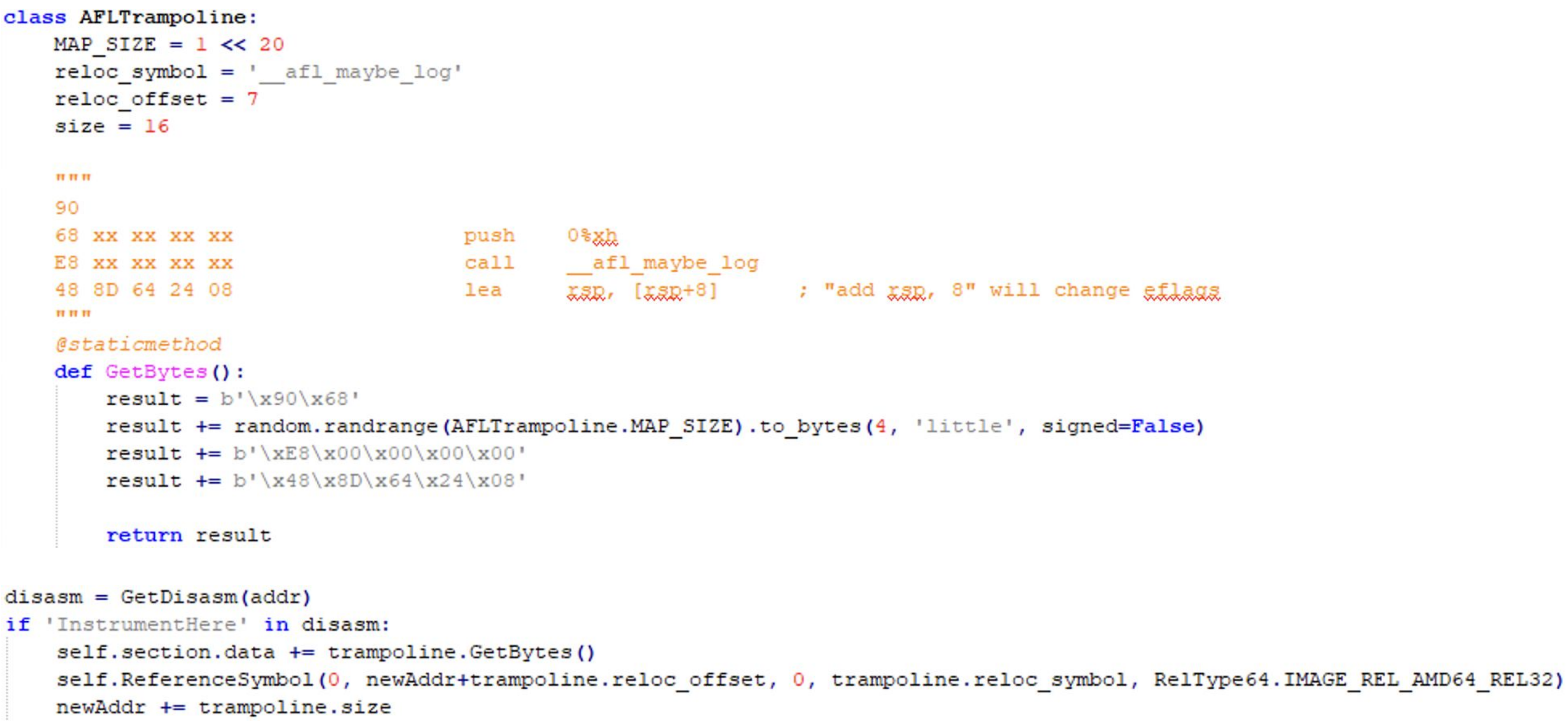
FillData()

Dump Objects



```
def FillData(self):
    pat = re.compile(r"(call|jmp)\s+cs:")
    pat2 = re.compile(r"(call|jmp)\s+r(ax|bx|cx|dx|si|di|8|9|10|11|12|13|14|15)")
    ImageBase = get_imagebase()
    newAddr = addr = self.segBegin
    while addr < self.segEnd:
        #print('[-] dumping 0x%x'%addr)
        itemSize = ItemSize(addr)
        name = Name(addr)
        if name != "": self.PublicSymbol(newAddr, name)

        disasm = GetDisasm(addr)
        if 'InstrumentHere' in disasm:

        refAddrList = []
        patMatch = pat.search(disasm)
        if not pat2.search(disasm) and 'retn' not in disasm: # ignore all references from instruction "call/jmp registe
            for x in XrefsFrom(addr): # ida xref.XREF FAR, use default ida xref.XREF ALL in case ignored
        refNum = len(refAddrList)
        if refNum == 0:
            self.section.data += GetManyBytes(addr, itemSize, 0)
        elif refNum == 1:
        else: # multiple xrefs
        addr += itemSize
        newAddr += itemSize
    return True
```

Instrumentation & Trampoline



```
class AFLTrampoline:
    MAP_SIZE = 1 << 20
    reloc_symbol = '__afl_maybe_log'
    reloc_offset = 7
    size = 16

    """
    90
    68 xx xx xx xx                  push    0%xh
    E8 xx xx xx xx                  call    __afl_maybe_log
    48 8D 64 24 08                  lea     rsp, [rsp+8]      ; "add rsp, 8" will change eflags
    """
    @staticmethod
    def GetBytes():
        result = b'\x90\x68'
        result += random.randrange(AFLTrampoline.MAP_SIZE).to_bytes(4, 'little', signed=False)
        result += b'\xE8\x00\x00\x00\x00'
        result += b'\x48\x8D\x64\x24\x08'

        return result

disasm = GetDisasm(addr)
if 'InstrumentHere' in disasm:
    self.section.data += trampoline.GetBytes()
    self.ReferenceSymbol(0, newAddr+trampoline.reloc_offset, 0, trampoline.reloc_symbol, RelType64.IMAGE_REL_AMD64_REL32)
    newAddr += trampoline.size
```

link.bat

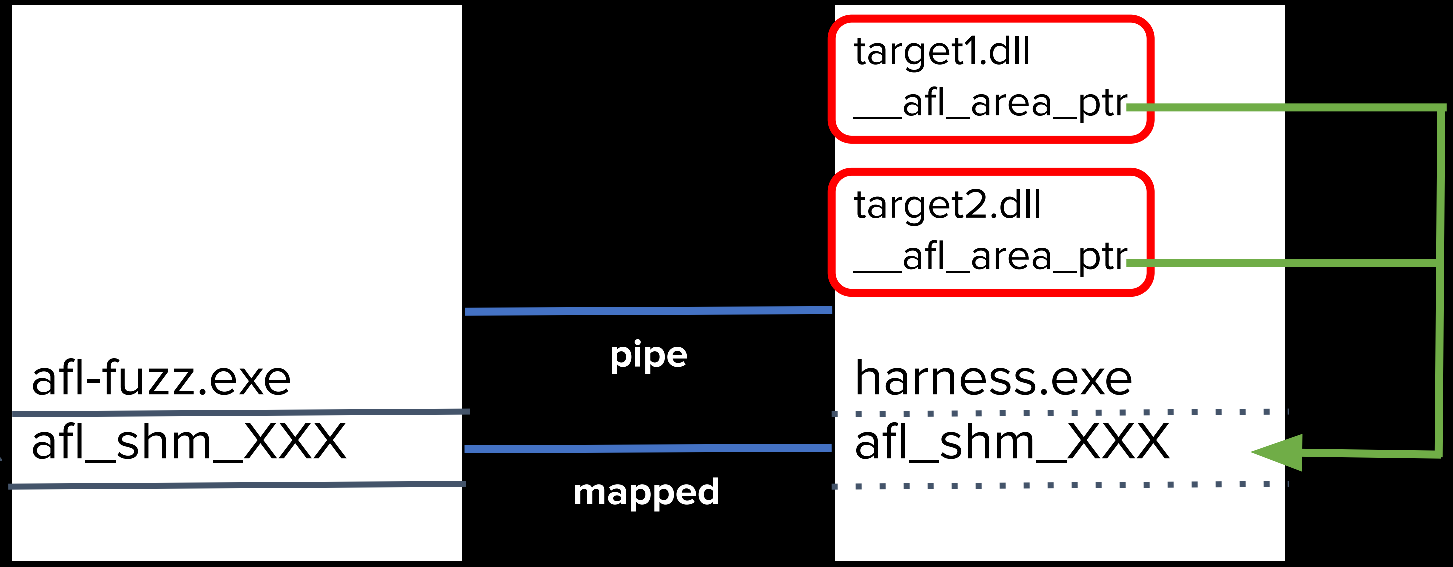
```
rem Usage: link.bat GdiPlus dll/exe/sys afl/trace [/RELEASE] [option 1] [option 2]
SET batpath=%~dp0
"%batpath%bin\link.exe" /OUT:"%1\%1.%3.%2" /PDB:"%1\%1.%3.pdb" /DEBUG /%2
/DEF:"%1\exports_%3.def" %4 %5 %6 "%1\libs\*.lib" "%1\objs\%3\*.obj"
"%batpath:~0,-5%payloads\%3_payload64.obj" /MACHINE:X64 /ERRORREPORT:PROMPT
/INCREMENTAL:NO /NOLOGO /GUARD:NO /MANIFEST:NO /NODEFAULTLIB /DYNAMICBASE /NXCOMPAT
/SECTION:.pdata,R
```

Integrate With WinAFL

- The key point is passing the bitmap area to the instrumented binary
  - afl-fuzz.exe uses CreateFileMapping to create shared memory
  - harness.exe uses OpenFileMapping to fetch the shared memory
- Do some modifications from a good example
  - refer to: https://github.com/wmliang/pe-afl/tree/master/AFL

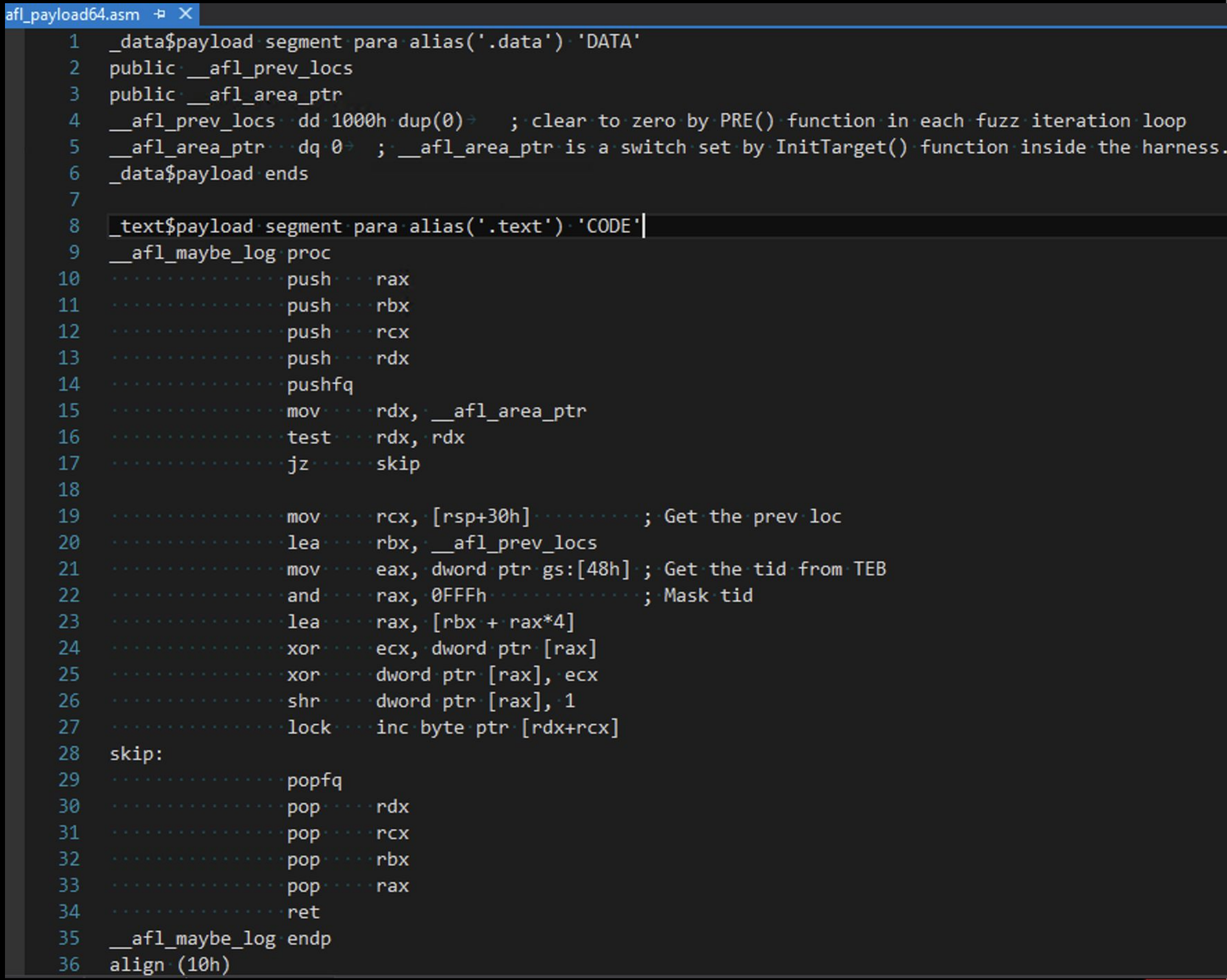
Architecture

TIP:
Enlarge the MAP_SIZE when intrumententing more modules



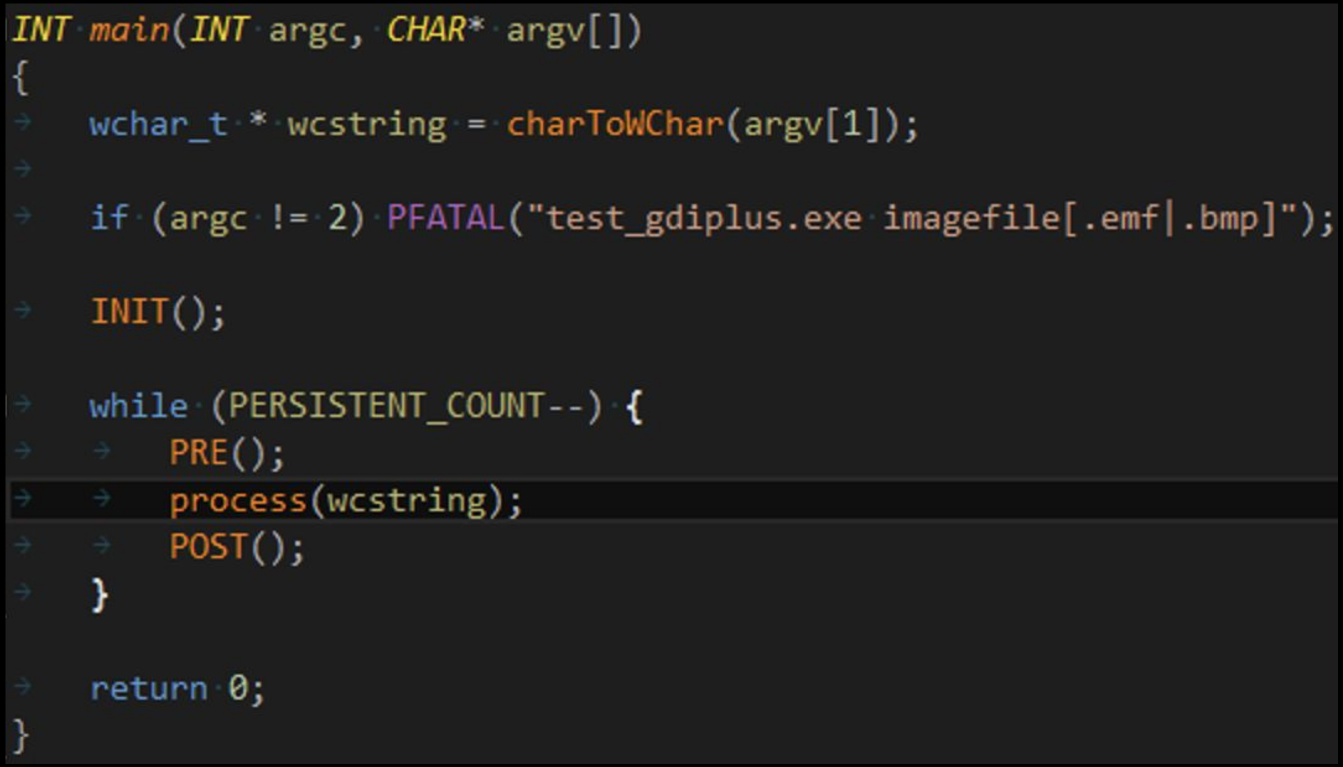
__afl_maybe_log

- Multi-threads support
  - __afl_prev_locs is an array with the tid as its index
  - clear to zero before each fuzz iteration loop
- Multi-modules support
  - __afl_area_ptr is exported
  - set to the shared memory address by harness



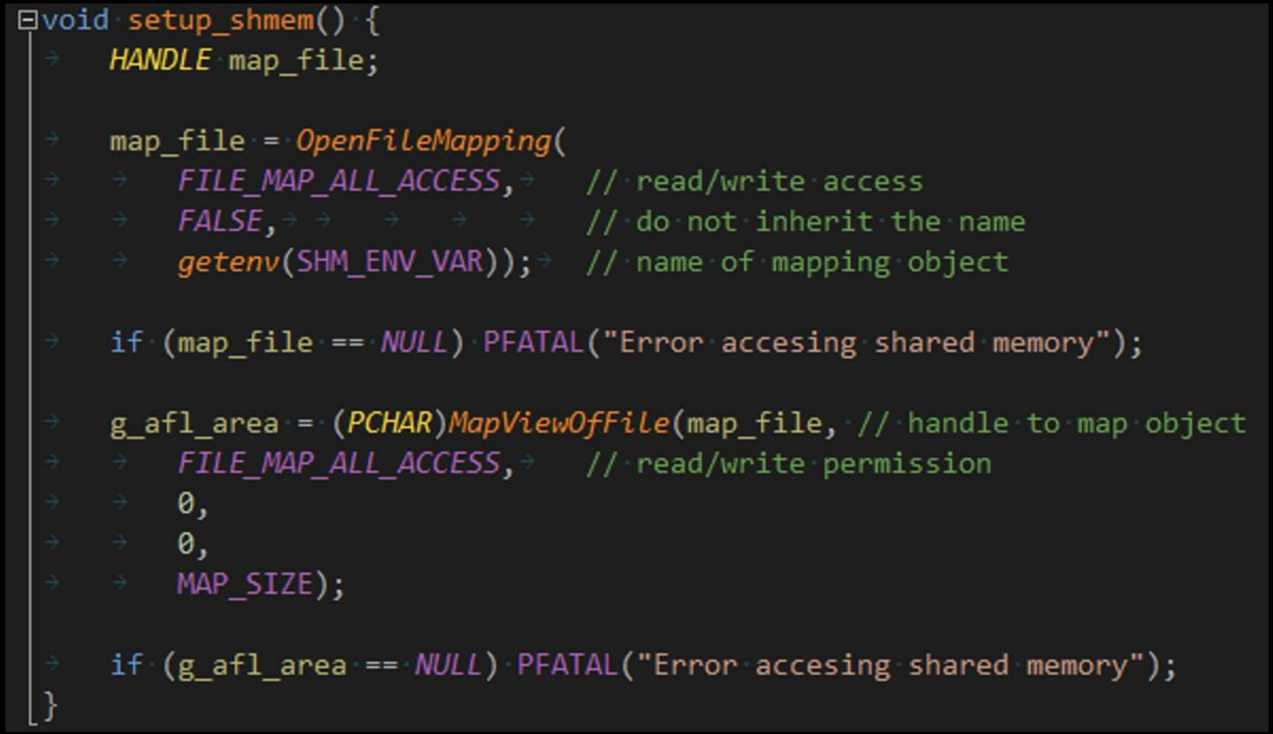
```
_data$payload segment para alias('.data') 'DATA'
public __afl_prev_locs
public __afl_area_ptr
__afl_prev_locs  dd 1000h dup(0)    ; clear to zero by PRE() function in each fuzz iteration loop
__afl_area_ptr   dq 0   ; __afl_area_ptr is a switch set by InitTarget() function inside the harness.
_data$payload ends

_text$payload segment para alias('.text') 'CODE'
__afl_maybe_log proc
                push    rax
                push    rbx
                push    rcx
                push    rdx
                pushfq
                mov     rdx, __afl_area_ptr
                test    rdx, rdx
                jz      skip

                mov     rcx, [rsp+30h]          ; Get the prev loc
                lea     rbx, __afl_prev_locs
                mov     eax, dword ptr gs:[48h] ; Get the tid from TEB
                and     rax, 0FFFh              ; Mask tid
                lea     rax, [rbx + rax*4]
                xor     ecx, dword ptr [rax]
                xor     dword ptr [rax], ecx
                shr     dword ptr [rax], 1
                lock    inc byte ptr [rdx+rcx]
skip:
                popfq
                pop     rdx
                pop     rcx
                pop     rbx
                pop     rax
                ret
__afl_maybe_log endp
align (10h)
```

Harness



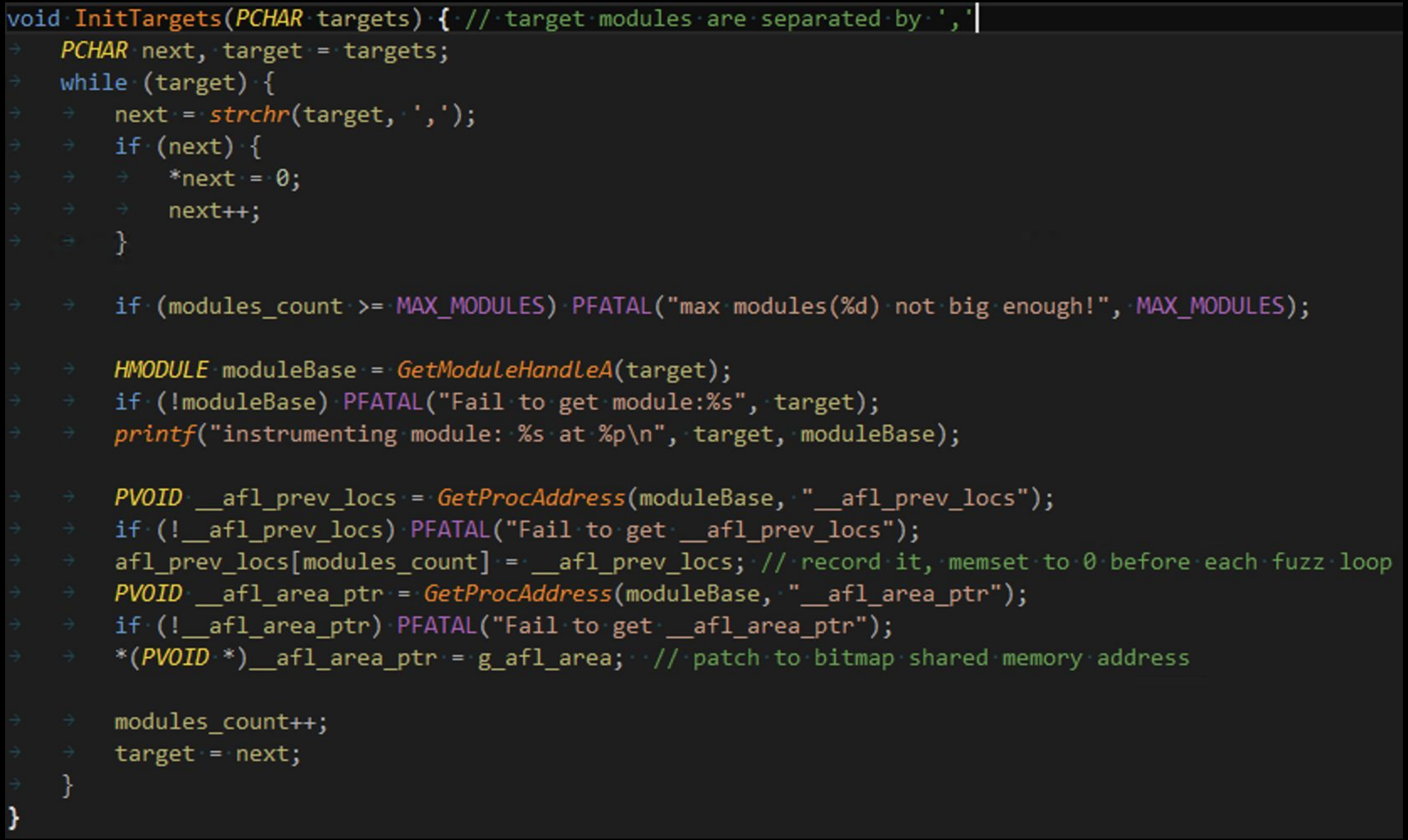
```
INT main(INT argc, CHAR* argv[])
{
    wchar_t * wcstring = charToWChar(argv[1]);

    if (argc != 2) PFATAL("test_gdiplus.exe imagefile[.emf|.bmp]");

    INIT();

    while (PERSISTENT_COUNT--) {
        PRE();
        process(wcstring);
        POST();
    }

    return 0;
}
```

Harness



```
void setup_shmem() {
    HANDLE map_file;

    map_file = OpenFileMapping(
        FILE_MAP_ALL_ACCESS,    // read/write access
        FALSE,                  // do not inherit the name
        getenv(SHM_ENV_VAR));   // name of mapping object

    if (map_file == NULL) PFATAL("Error accesing shared memory");

    g_afl_area = (PCHAR)MapViewOfFile(map_file, // handle to map object
        FILE_MAP_ALL_ACCESS,    // read/write permission
        0,
        0,
        MAP_SIZE);

    if (g_afl_area == NULL) PFATAL("Error accesing shared memory");
}
```

Harness



```
void InitTargets(PCHAR targets) { // target modules are separated by ','
    PCHAR next, target = targets;
    while (target) {
        next = strchr(target, ',');
        if (next) {
            *next = 0;
            next++;
        }

        if (modules_count >= MAX_MODULES) PFATAL("max modules(%d) not big enough!", MAX_MODULES);

        HMODULE moduleBase = GetModuleHandleA(target);
        if (!moduleBase) PFATAL("Fail to get module:%s", target);
        printf("instrumenting module: %s at %p\n", target, moduleBase);

        PVOID __afl_prev_locs = GetProcAddress(moduleBase, "__afl_prev_locs");
        if (!__afl_prev_locs) PFATAL("Fail to get __afl_prev_locs");
        afl_prev_locs[modules_count] = __afl_prev_locs; // record it, memset to 0 before each fuzz loop
        PVOID __afl_area_ptr = GetProcAddress(moduleBase, "__afl_area_ptr");
        if (!__afl_area_ptr) PFATAL("Fail to get __afl_area_ptr");
        *(PVOID *)__afl_area_ptr = g_afl_area;  // patch to bitmap shared memory address

        modules_count++;
        target = next;
    }
}
```

Demo

Summary

- I just let the linker help me fix the symbol references
  - However, the linker also generated some redundant data, such as the PE header, which makes me cannot reuse the old PE header
  - Maybe I can hijack link.exe and only exploit its function of fixing symbol relocations
- It could be cross-platform in theory
  - But I just made it come true for 64-bit PE
- It is as fast as the compiler instrumentation with source code
- The new binary could be equivalent to the old one, if all the cross references analysis is right

The Real Challenge & The Solution

Challenge :

The precondition of the solutions is that the analysis result from IDA is correct.

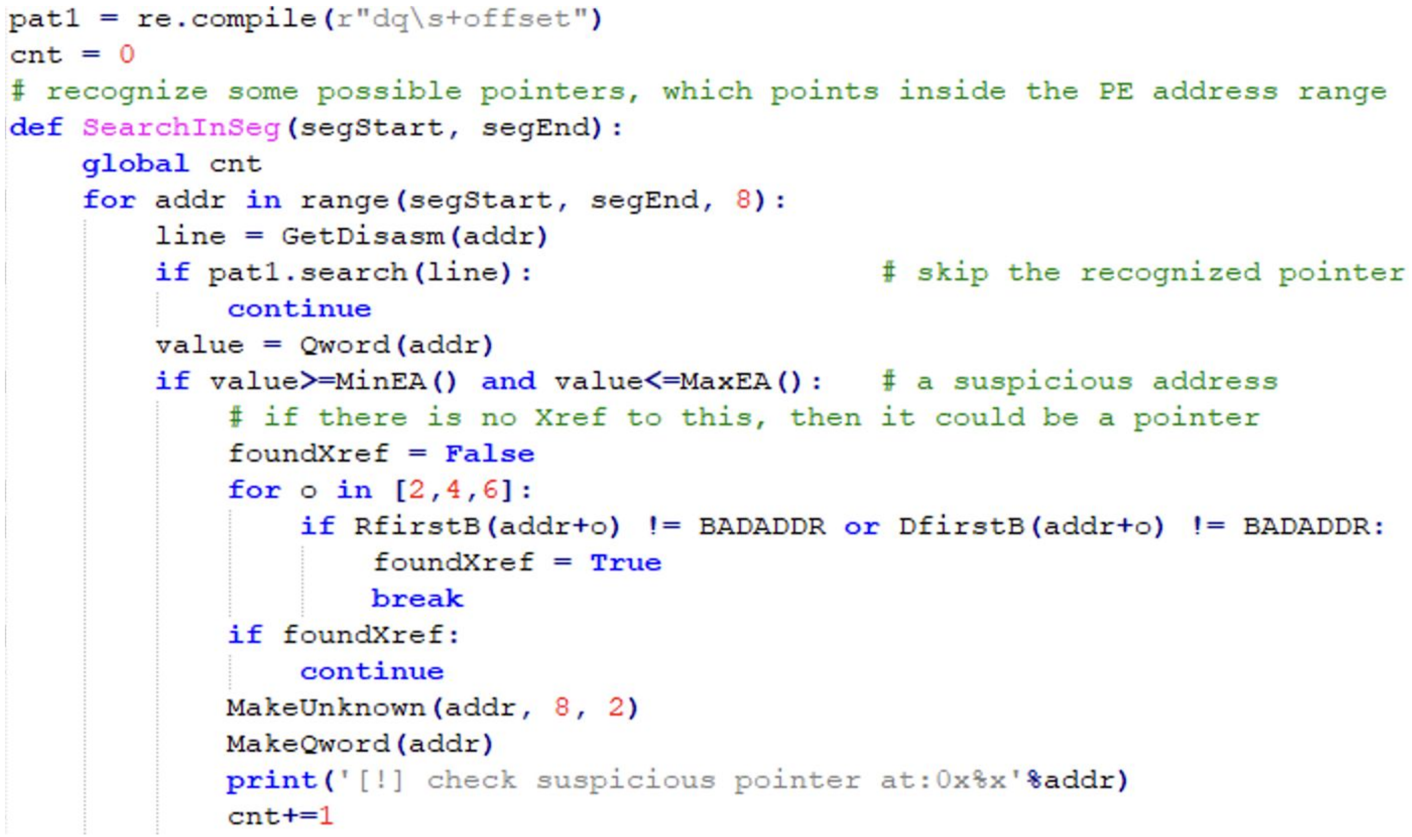
For some reasons, sometimes IDA couldn't recognize some pointers or relative values. And it may lead to the crash issues.

Solution :

Create assistant scripts to help IDA analyze before exporting.

FixPointer.py

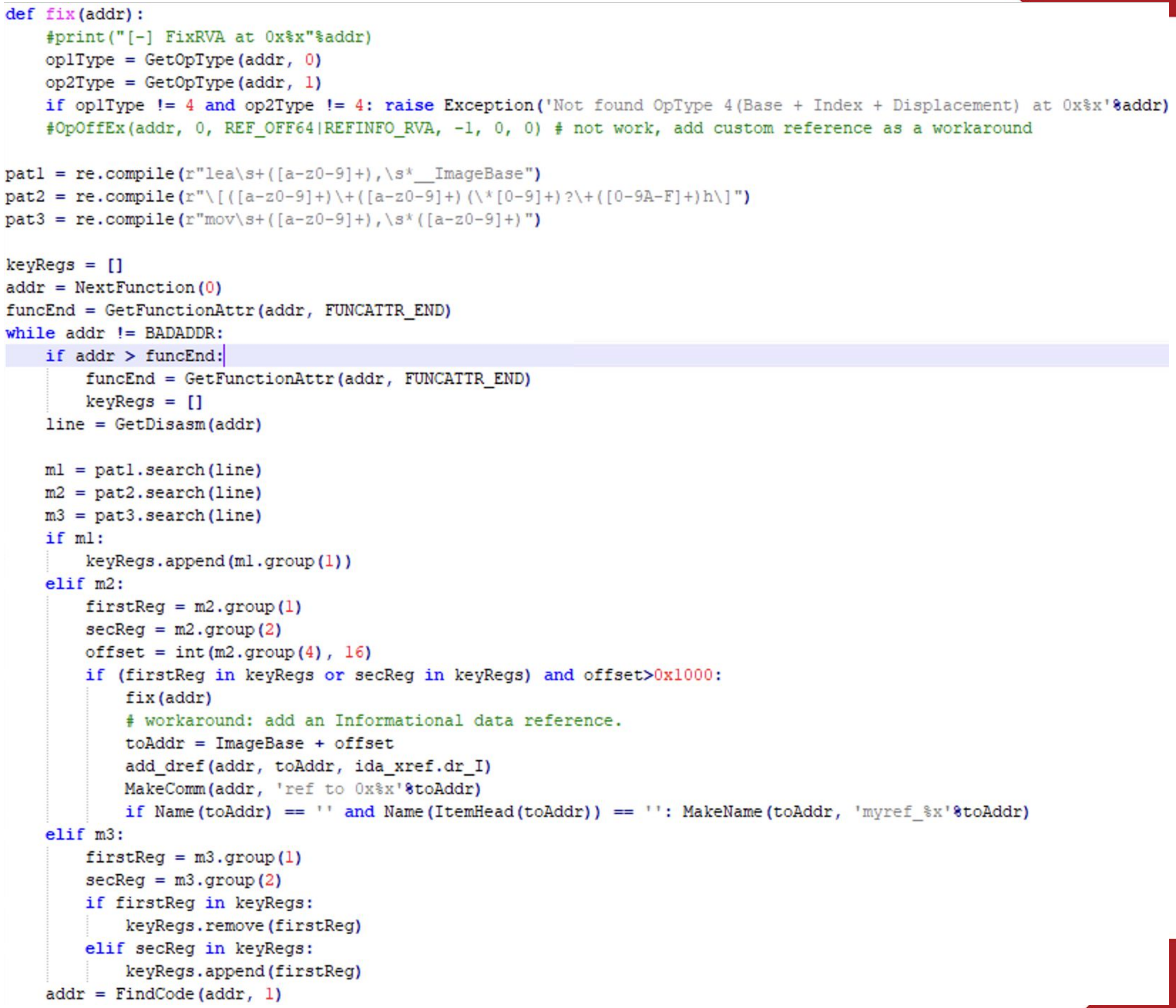
Scan suspicious pointers



```
pat1 = re.compile(r"dq\s+offset")
cnt = 0
# recognize some possible pointers, which points inside the PE address range
def SearchInSeg(segStart, segEnd):
    global cnt
    for addr in range(segStart, segEnd, 8):
        line = GetDisasm(addr)
        if pat1.search(line):                         # skip the recognized pointer
            continue
        value = Qword(addr)
        if value>=MinEA() and value<=MaxEA():    # a suspicious address
            # if there is no Xref to this, then it could be a pointer
            foundXref = False
            for o in [2,4,6]:
                if RfirstB(addr+o) != BADADDR or DfirstB(addr+o) != BADADDR:
                    foundXref = True
                    break
            if foundXref:
                continue
            MakeUnknown(addr, 8, 2)
            MakeQword(addr)
            print('[!] check suspicious pointer at:0x%x'%addr)
            cnt+=1
```

FixRVA.py

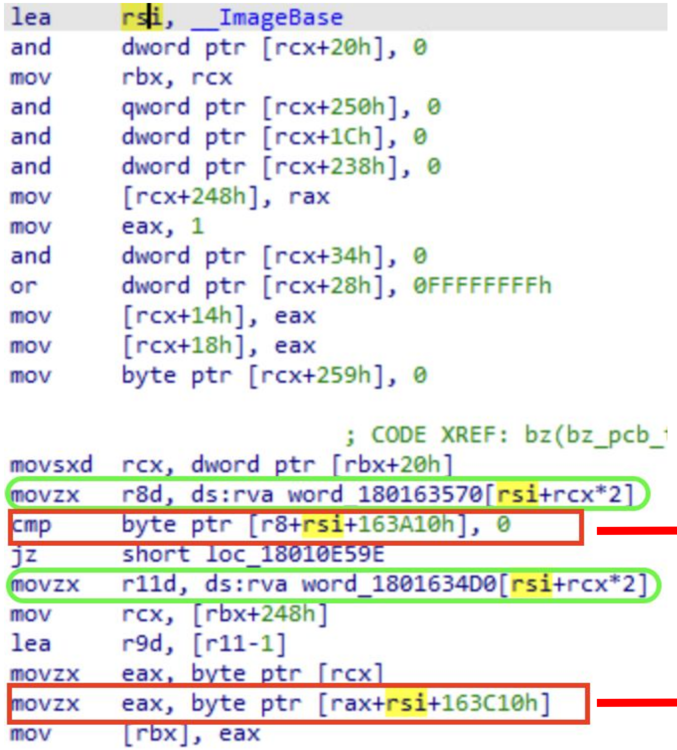
There are some image based relative values not recognized, mainly exist in the jump table of switch-case



```
def fix(addr):
    #print("[-] FixRVA at 0x%x"%addr)
    op1Type = GetOpType(addr, 0)
    op2Type = GetOpType(addr, 1)
    if op1Type != 4 and op2Type != 4: raise Exception('Not found OpType 4(Base + Index + Displacement) at 0x%x'%addr)
    #OpOffEx(addr, 0, REF_OFF64|REFINFO_RVA, -1, 0, 0) # not work, add custom reference as a workaround

pat1 = re.compile(r"lea\s+([a-z0-9]+),\s*__ImageBase")
pat2 = re.compile(r"\[(([a-z0-9]+)\+([a-z0-9]+)(\*[0-9]+)?\+([0-9A-F]+)h\]")
pat3 = re.compile(r"mov\s+([a-z0-9]+),\s*([a-z0-9]+)")

keyRegs = []
addr = NextFunction(0)
funcEnd = GetFunctionAttr(addr, FUNCATTR_END)
while addr != BADADDR:
    if addr > funcEnd:
        funcEnd = GetFunctionAttr(addr, FUNCATTR_END)
        keyRegs = []
    line = GetDisasm(addr)

    m1 = pat1.search(line)
    m2 = pat2.search(line)
    m3 = pat3.search(line)
    if m1:
        keyRegs.append(m1.group(1))
    elif m2:
        firstReg = m2.group(1)
        secReg = m2.group(2)
        offset = int(m2.group(4), 16)
        if (firstReg in keyRegs or secReg in keyRegs) and offset>0x1000:
            fix(addr)
            # workaround: add an Informational data reference.
            toAddr = ImageBase + offset
            add_dref(addr, toAddr, ida_xref.dr_I)
            MakeComm(addr, 'ref to 0x%x'%toAddr)
            if Name(toAddr) == '' and Name(ItemHead(toAddr)) == '': MakeName(toAddr, 'myref_%x'%toAddr)
    elif m3:
        firstReg = m3.group(1)
        secReg = m3.group(2)
        if firstReg in keyRegs:
            keyRegs.remove(firstReg)
        elif secReg in keyRegs:
            keyRegs.append(firstReg)
    addr = FindCode(addr, 1)
```

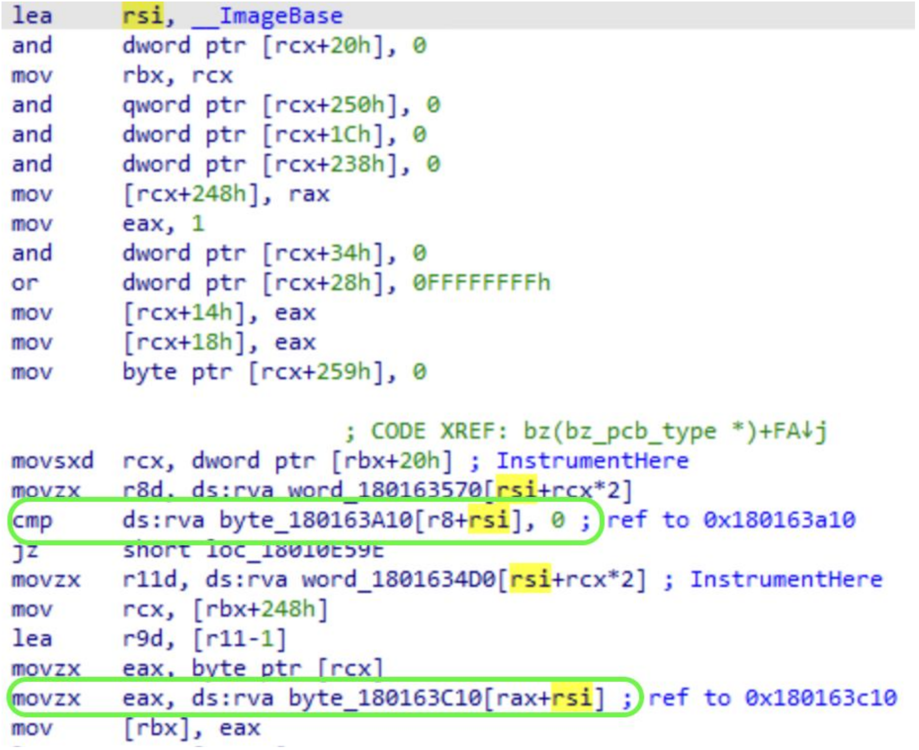
FixRVA.py

Before fix:



```
lea     rsi, __ImageBase
and     dword ptr [rcx+20h], 0
mov     rbx, rcx
and     qword ptr [rcx+250h], 0
and     dword ptr [rcx+1Ch], 0
and     dword ptr [rcx+238h], 0
mov     [rcx+248h], rax
mov     eax, 1
and     dword ptr [rcx+34h], 0
or      dword ptr [rcx+28h], 0FFFFFFFFh
mov     [rcx+14h], eax
mov     [rcx+18h], eax
mov     byte ptr [rcx+259h], 0

                        ; CODE XREF: bz(bz_pcb_
movsxd  rcx, dword ptr [rbx+20h]
movzx   r8d, ds:rva word_180163570[rsi+rcx*2]
cmp     byte ptr [r8+rsi+163A10h], 0
jz      short loc_18010E59E
movzx   r11d, ds:rva word_1801634D0[rsi+rcx*2]
mov     rcx, [rbx+248h]
lea     r9d, [r11-1]
movzx   eax, byte ptr [rcx]
movzx   eax, byte ptr [rax+rsi+163C10h]
mov     [rbx], eax
```

After fix:



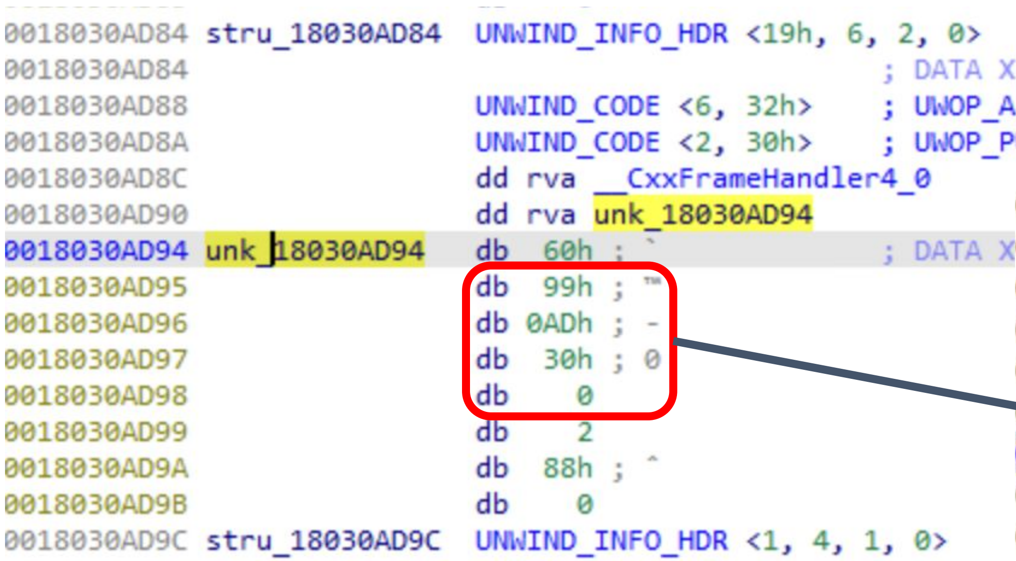
```
lea     rsi, __ImageBase
and     dword ptr [rcx+20h], 0
mov     rbx, rcx
and     qword ptr [rcx+250h], 0
and     dword ptr [rcx+1Ch], 0
and     dword ptr [rcx+238h], 0
mov     [rcx+248h], rax
mov     eax, 1
and     dword ptr [rcx+34h], 0
or      dword ptr [rcx+28h], 0FFFFFFFFh
mov     [rcx+14h], eax
mov     [rcx+18h], eax
mov     byte ptr [rcx+259h], 0

                        ; CODE XREF: bz(bz_pcb_type *)+FA↓j
movsxd  rcx, dword ptr [rbx+20h] ; InstrumentHere
movzx   r8d, ds:rva word_180163570[rsi+rcx*2]
cmp     ds:rva byte_180163A10[r8+rsi], 0 ; ref to 0x180163a10
jz      short loc_18010E59E
movzx   r11d, ds:rva word_1801634D0[rsi+rcx*2] ; InstrumentHere
mov     rcx, [rbx+248h]
lea     r9d, [r11-1]
movzx   eax, byte ptr [rcx]
movzx   eax, ds:rva byte_180163C10[rax+rsi] ; ref to 0x180163c10
mov     [rbx], eax
```

FixEH.py

- IDA supports to analyze the exception handling data structures since version 7.0
- However, there are still some data structures cannot be recognized
  - Maybe because of the UNDOC data structures ?
  - There are some function relative values in FH4
    - refer to: https://github.com/light-tech/MSCpp/blob/master/include/msvc/ehdata4_export.h
  - The script to parse and fix is too long to display here
    - the core logic is writing a parser according to the referred ehdata4_export.h

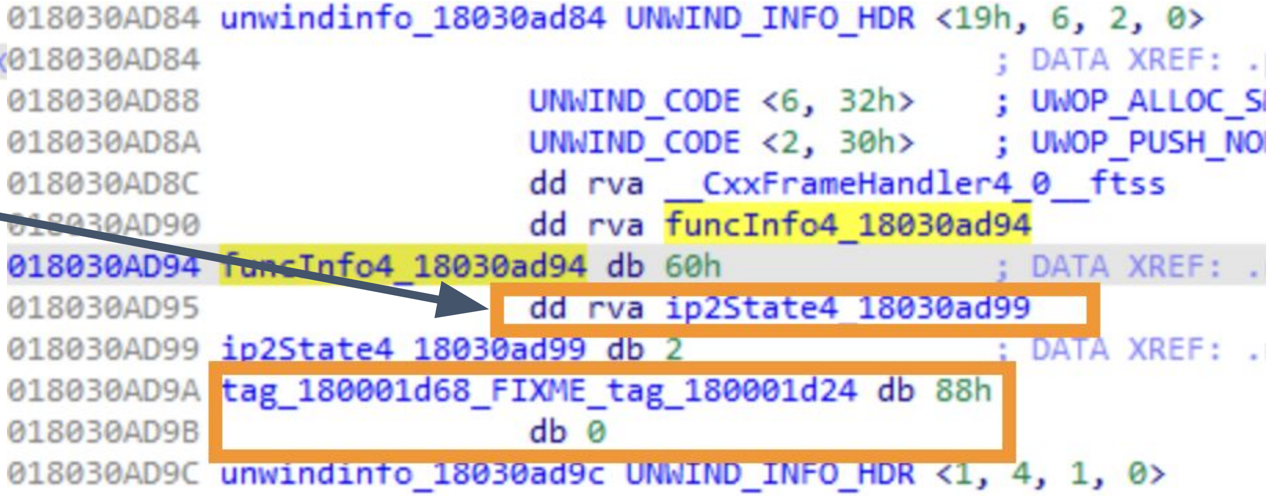
FixEH.py

Before fix:



```
0018030AD84 stru_18030AD84  UNWIND_INFO_HDR <19h, 6, 2, 0>
0018030AD84                                        ; DATA XREF: .pdata:
0018030AD88                 UNWIND_CODE <6, 32h>   ; UWOP_ALLOC_SMALL
0018030AD8A                 UNWIND_CODE <2, 30h>   ; UWOP_PUSH_NONVOL
0018030AD8C                 dd rva __CxxFrameHandler4_0
0018030AD90                 dd rva unk_18030AD94
0018030AD94 unk_18030AD94   db  60h ; `            ; DATA X
0018030AD95                 db  99h ; ™
0018030AD96                 db 0ADh ; -
0018030AD97                 db  30h ; 0
0018030AD98                 db    0
0018030AD99                 db    2
0018030AD9A                 db  88h ; ^
0018030AD9B                 db    0
0018030AD9C stru_18030AD9C  UNWIND_INFO_HDR <1, 4, 1, 0>
```

After fix:

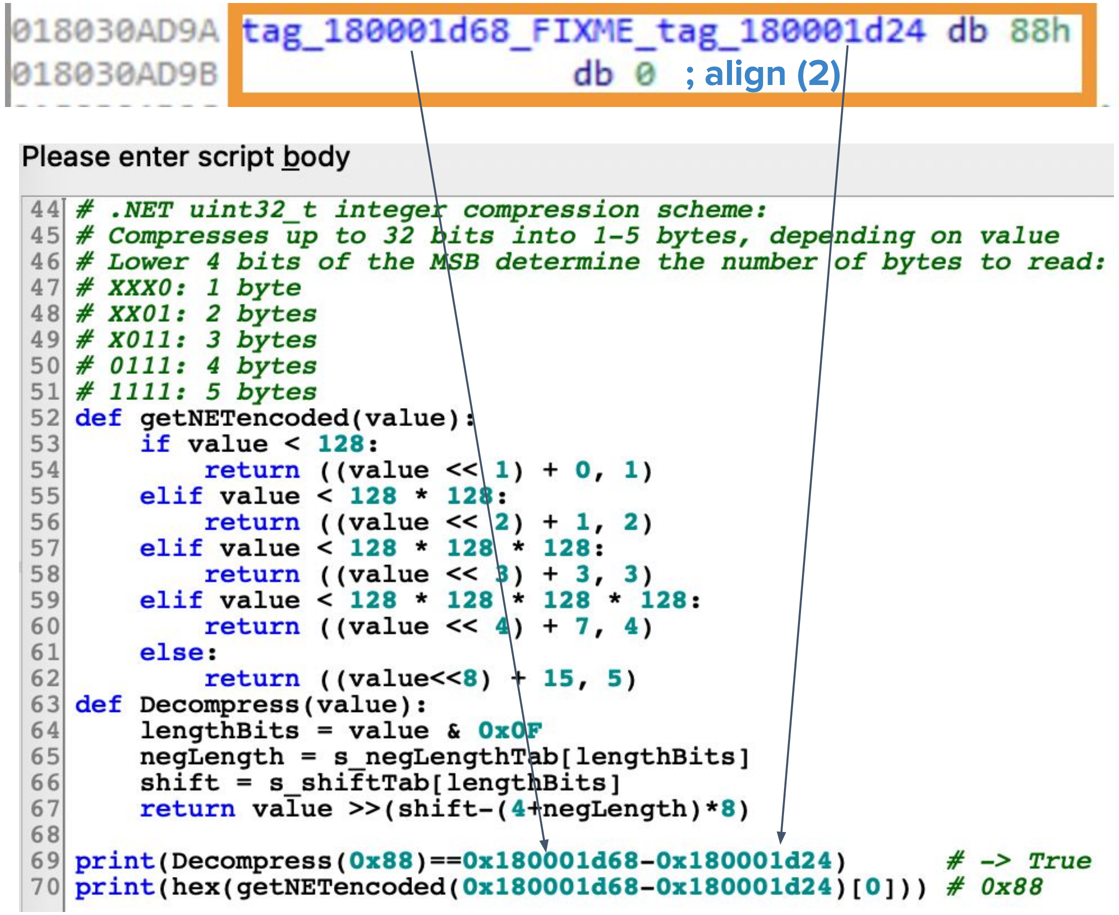


```
018030AD84 unwindinfo_18030ad84 UNWIND_INFO_HDR <19h, 6, 2, 0>
018030AD84                                                ; DATA XREF: .
018030AD88                      UNWIND_CODE <6, 32h>      ; UWOP_ALLOC_S
018030AD8A                      UNWIND_CODE <2, 30h>      ; UWOP_PUSH_NO
018030AD8C                      dd rva __CxxFrameHandler4_0__ftss
018030AD90                      dd rva funcInfo4_18030ad94
018030AD94 funcInfo4_18030ad94 db 60h                    ; DATA XREF: .
018030AD95                      dd rva ip2State4_18030ad99
018030AD99 ip2State4_18030ad99 db 2                      ; DATA XREF: .
018030AD9A tag_180001d68_FIXME_tag_180001d24 db 88h
018030AD9B                      db 0
018030AD9C unwindinfo_18030ad9c UNWIND_INFO_HDR <1, 4, 1, 0>
```

FH4

- __CxxFrameHandler4, dubbed as FH4
- A new feature to reduce the binary size of C++ exception handling on x64
- Some function relative values are compressed and saved into .rdata segment
- The relative values will be larger due to the instrumentation
- It means the .rdata segment could be enlarged too

Compression Scheme of FH4

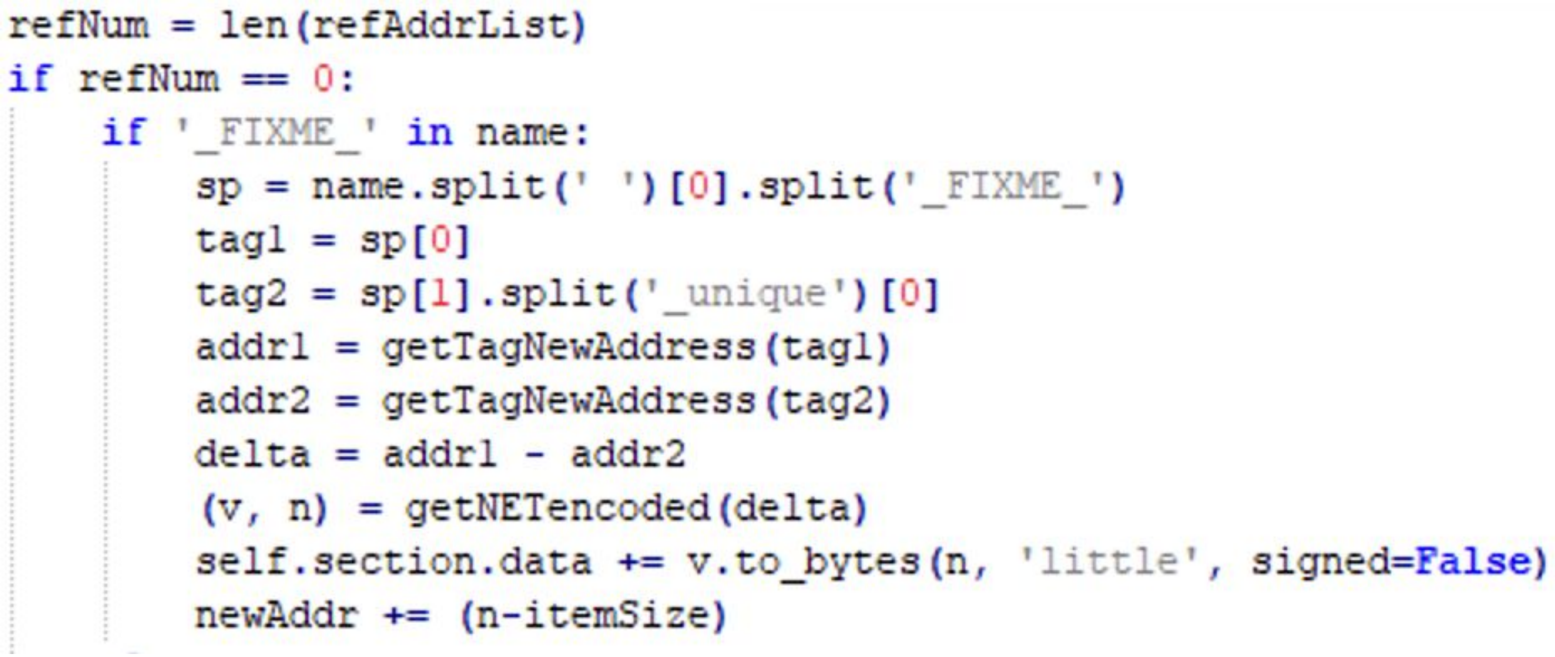


```
018030AD9A tag_180001d68_FIXME_tag_180001d24 db 88h
018030AD9B                                 db 0  ; align (2)
```

Please enter script body

```
# .NET uint32_t integer compression scheme:
# Compresses up to 32 bits into 1-5 bytes, depending on value
# Lower 4 bits of the MSB determine the number of bytes to read:
# XXX0: 1 byte
# XX01: 2 bytes
# X011: 3 bytes
# 0111: 4 bytes
# 1111: 5 bytes
def getNETencoded(value):
    if value < 128:
        return ((value << 1) + 0, 1)
    elif value < 128 * 128:
        return ((value << 2) + 1, 2)
    elif value < 128 * 128 * 128:
        return ((value << 3) + 3, 3)
    elif value < 128 * 128 * 128 * 128:
        return ((value << 4) + 7, 4)
    else:
        return ((value<<8) + 15, 5)
def Decompress(value):
    lengthBits = value & 0x0F
    negLength = s_negLengthTab[lengthBits]
    shift = s_shiftTab[lengthBits]
    return value >>(shift-(4+negLength)*8)

print(Decompress(0x88)==0x180001d68-0x180001d24)      # -> True
print(hex(getNETencoded(0x180001d68-0x180001d24)[0])) # 0x88
```

Solution For FH4



```
refNum = len(refAddrList)
if refNum == 0:
    if '_FIXME_' in name:
        sp = name.split(' ')[0].split('_FIXME_')
        tag1 = sp[0]
        tag2 = sp[1].split('_unique')[0]
        addr1 = getTagNewAddress(tag1)
        addr2 = getTagNewAddress(tag2)
        delta = addr1 - addr2
        (v, n) = getNETencoded(delta)
        self.section.data += v.to_bytes(n, 'little', signed=False)
        newAddr += (n-itemSize)
```

Takeaway

- Two SBI implementations
  - IDA2MASM : https://github.com/jhftss/IDA2MASM
  - IDA2Obj : https://github.com/jhftss/IDA2Obj
- One SBI algorithm
  - Binary rewrite directly
  - Cross-platform in theory
  - Not implemented yet
- Some IDAPython scripts to assist the analysis
- The reposities will be **open source later**, private now

Future Plan

- Bugfix
  - Welcome to report issues and pull request
- Integrate with other fuzzers
- Try to make the cross-platform idea come true

References

1. https://www.ti.com/lit/an/spraao8/spraao8.pdf
2. https://github.com/d3dave/cough
3. https://github.com/wmliang/pe-afl (@_wmliang_)
4. https://devblogs.microsoft.com/cppblog/making-cpp-exception-handling-smaller-x64
5. https://github.com/light-tech/MSCpp/blob/master/include/msvc/ehdata4_export.h
6. https://github.com/googleprojectzero/p0tools/blob/master/TrapFuzz/findPatchPoints.py

Thanks !

Mickey Jin (@patch1t) of Trend Micro