The Poor Man's Obfuscator

Pass The Salt 2022

Romain Thomas

me@romainthomas.fr

Contents:

1 Introduction 1

2 __unwind_info & .eh_frame 2

3 Exported Symbols 4
- 3.1 Creating Fake Exports Names 4
- 3.2 Confusing the Exports Names 5
- 3.3 Exports Addresses 7

4 Sections Transformations 9
- 4.1 ELF 9
- 4.2 Mach-O 11

5 Specific Transformations 14
- 5.1 LC_FUNCTIONS_STARTS 14
- 5.2 .dynsym section 16

6 Conclusion 17

1. Introduction

The purpose of this paper is to present ELF and Mach-O transformations which impact or hinder disassemblers like IDA, BinaryNinja, Ghidra, and Radare2. In particular, these transformations do not modify the assembly code or the data of the binary. The transformations are focused on modifying the executable file format structures like the sections or the symbols. As a result, the modified binaries look obfuscated as it will be shown through different examples.

All the modifications presented in this paper are based on LIEF (commit : `f8c711d`) which is a library for parsing and modifying executable file formats. The binaries referenced in the examples are based on the Mbed TLS test suite and more precisely, `programs/test/selftest.c`. This testing program is convenient to verify that the ELF/Mach-O modifications do not break the binary and do not introduce side effects.

> The aim of this work **is not** to highlight or point out the limits of the disassemblers mentioned in this paper. Actually, all of them have pros and cons with different design decisions which enable us to enjoy reverse engineering at different levels.
>
> Being not an expert on all these disassemblers, I might also miss some loading options for which the transformation would not impact the disassembler. I'll take care of updating the content of this paper based on the feedback (if any).

The content of this paper has been presented at Pass The Salt 2022 on July, 5th using on the following releases of the disassemblers :

Tools	Version	Release Date
IDA	7.7.211224	January, 2022
BinaryNinja	3.0	April 2022
Radare2	5.7.0	June 2022
Ghidra	10.1.4	May 2022

2. __unwind_info & .eh_frame

The ELF `.eh_frame` section and the Mach-O `__unwind_info` section contain information that can be used to get a list of function addresses present in the binary.

These sections are supported and parsed by LIEF which exposes the list of functions through the `Binary.functions` attribute :

```python
import lief

target = lief.parse("/bin/ls")
for function in target.functions:
    print(function.address)
```

These sections are also used by the disassemblers to get an initial *worklist* of functions where they can start the disassembling.

Thus, we can prevent the disassemblers from using the content of these sections by shuffling their content. Here is an example for the ELF sections :

```python
for section in [".eh_frame", ".eh_frame_hdr"]:
    section = target.get_section(name)
    section_content = list(section.content)
    random.shuffle(section_content)
    section.content = section_content
```

IDA, BinaryNinja, and Radare2 do not seem to be impacted by this modification but Ghidra raises an exception when trying to analyse the binary :

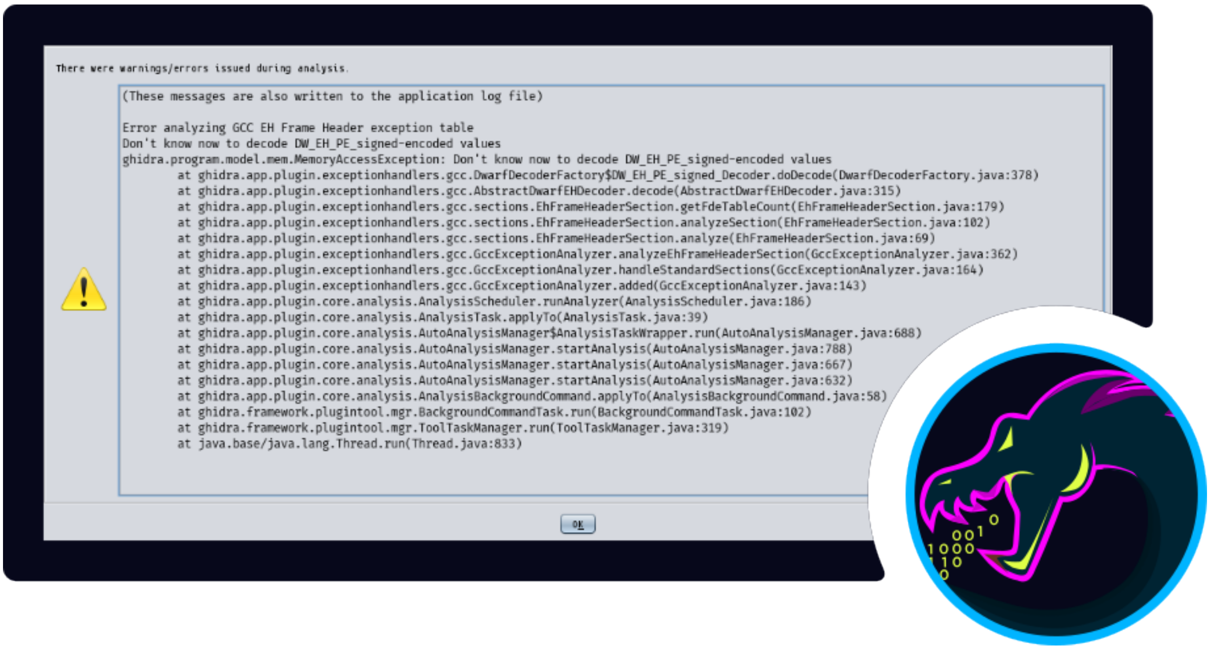


FIGURE1 – Error with Ghidra when shuffling the `__unwind_info` section

Tools	Impacted
IDA	No
BinaryNinja	No
Radare2	No
Ghidra	Yes

3. Exported Symbols

The first simple, efficient and universal modification we can do on executable formats is creating fake exports. Thanks to the ELF `.eh_frame` section and the Mach-O `__unwind_info` section, we can get a – more or less – accurate list of functions' start addresses and thus, create relevant exports.

3.1 Creating Fake Exports Names

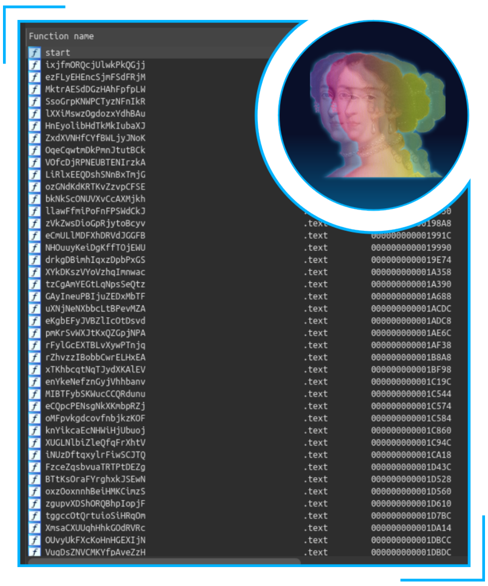
We can start to *"obfuscate"* the binary by creating export names for which the name is randomized. Using this LIEF, such exports can be created by using the `add_exported_function` function.

> This function is only available for the ELF and Mach-O formats and is not implemented for the PE format.

The random symbol's name can be generated through the Python random module :

```
import lief
import random
import string

target = lief.parse("mbedtls_self_test.arm64.macho")

for function in target.functions:
    name = "".join(random.choice(string.ascii_letters) for i in range(20))
    target.add_exported_function(function.address, name)

target.write("01-mbedtls_self_test.arm64.macho")
```

In the output of IDA, BinaryNinja, etc we can observe the list of the random names as we could expect. This transformation is not very *fancy* except if the binary to protect has been compiled with a *wrong* visibility (like `-fvisibility=default`) and in which, we would like to remove or change the symbols after the compilation.



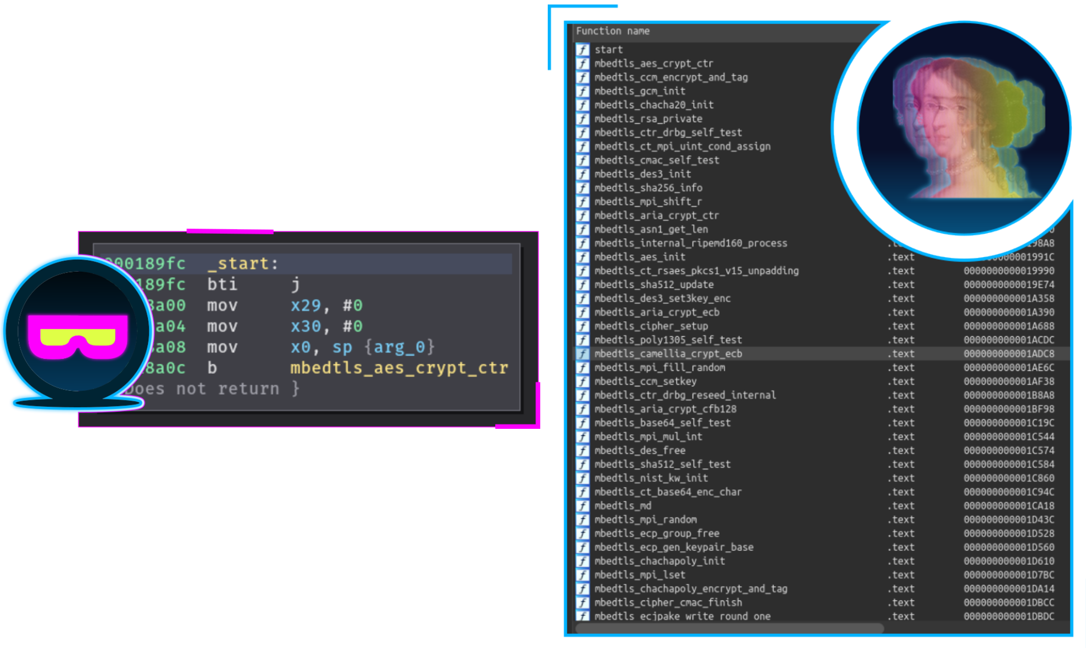
3.2 Confusing the Exports Names

Creating random export names is confusing but a reverse engineer can immediately identify that the binary has been modified or aims at being protected. In addition, it does not bring more protection compared to a regular strip of the functions.

Actually, we can still leverage the exports table to add new entries but instead of using random strings, we can pick real and consistent function names from the original Mbed TLS binary.

```
target     = lief.parse("mbedtls_self_test.arm64.elf")
nostripped = lief.parse("mbedtls_self_test.nostrip.arm64.elf")

SYMBOLS = [s.name for s in non_striped.symbols \
           if s.name.startswith("mbedtls_")]

for function in target.functions:
    name = random.choice(SYMBOLS)
    SYMBOLS.remove(name)
    target.add_exported_function(function.address, name)
```

With such exports, it creates more confusion as it becomes difficult to distinguish the real Mbed TLS functions from the fake ones.



As we can observe in the previous figure, one function not related to `mbedtls_aes_crypt_ctr` has been renamed with this name.

To continue walking along the path of using meaningful names, we can also take names from a standard library like the *libc.so*. Compared to the `mbedtls_*` names, `libc`'s symbols are usually recognized by the disassemblers which provide type libraries for such functions. We could also be a bit more sneaky by taking C++ mangled symbols, from the LLVM libraries (for instance).

But let's continue with the libc's symbols. For Android, we can collect libc's symbols from the NDK and we also need to avoid exporting symbols with libc's symbols already imported by the target :

```
libc = lief.parse("[ ... ]toolchains/llvm/prebuilt/linux-x86_64"
                  "/sysroot/usr/lib/aarch64-linux-android/23/libc.so")

libc_symbols = {s.name for s in libc.exported_symbols}
libc_symbols -= {s.name for s in target.imported_symbols}
libc_symbols = list(libc_symbols)

for function in target.functions:
  sym = random.choice(libc_symbols)
  libc_symbols.remove(sym)

  target.add_exported_function(function.address, sym)
```

This transformation produces the following output where the figure on the left-hand side is the original unstripped function and the figure on the right-hand side, the function with the libc's symbols exported.

This transformation could be automatically defeated in different ways, like checking if the libc's symbols points in the `.plt` section. It could also be an interesting use case with binaries statically linked with the libc (which is forbidden on iOS and Android).

We also need to make sure that the newly exported libc's symbols are not used by the loader to actually resolve libc's functions imported by other libraries or the library itself. This can be accomplished by tweaking the symbol's visibility and the symbol's binding :

```
export = target.add_exported_function(function.address, sym)

export.binding    = lief.ELF.SYMBOL_BINDINGS.GNU_UNIQUE
export.visibility = lief.ELF.SYMBOL_VISIBILITY.INTERNAL
```

If the binary targets Linux, the binding must be set to `lief.ELF.SYMBOL_BINDINGS.WEAK` instead of `lief.ELF.SYMBOL_BINDINGS.GNU_UNIQUE`

3.3 Exports Addresses

One of the challenges when doing static analysis is to identify functions present in the binary. The binary entrypoint is obviously a good start for disassembling the code but disassemblers rely on other information from the executable file format like the exports table. It turns out that the exports table is strongly trusted by the disassemblers, whilst we can actually create entries with arbitrary symbols and **arbitrary addresses**.

The previous paragraphs only dealt with the export names. Since exports are always tied with an address, we can also trick this value. One of these tricks consists in creating exports with a delta value at the beginning of the function :

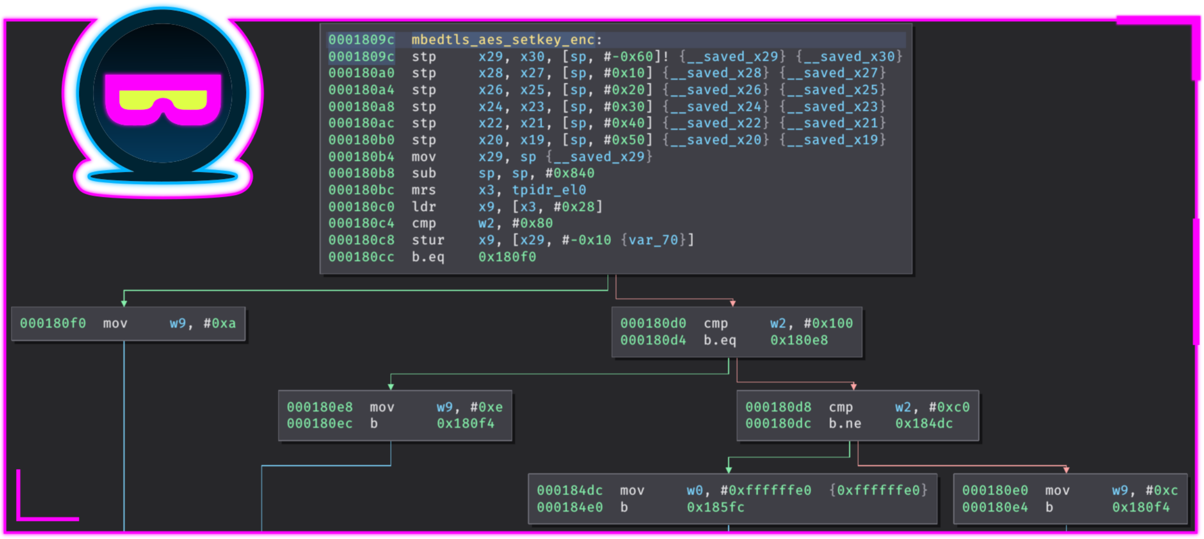
```python
for function in target.functions:
    address = function.address
    address += random.randint(16, 32)
```

To avoid unaligned instructions, we need to make sure that the delta is aligned on four bytes :

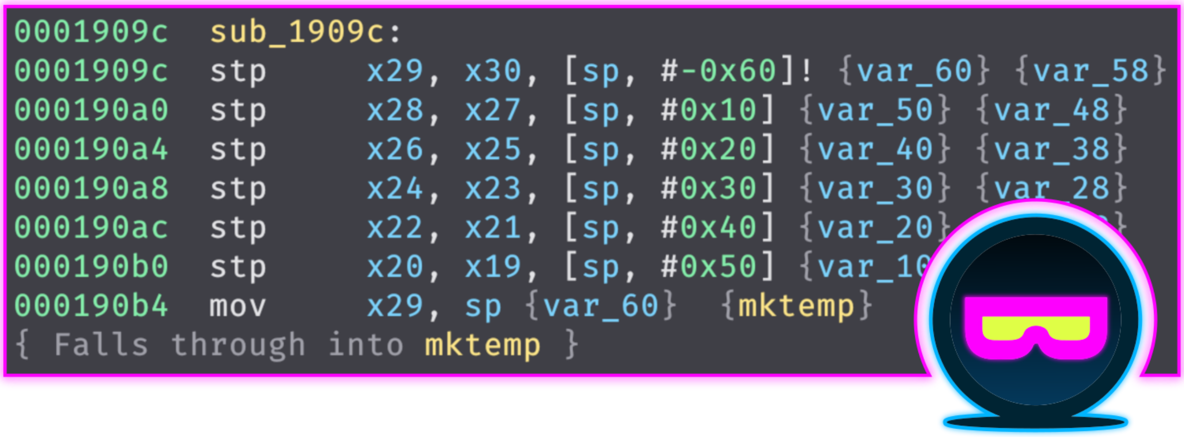
```python
for function in target.functions:
    address = function.address
    address += random.randint(16, 32)
    address -= address % 4
```

With such a transformation, **all the tools** : IDA, BinaryNinja, Radare2, Ghidra are wrongly disassembling the code and produce and incomplete control flow graph.

Here is the impact of this transformation when opening the modified binary in BinaryNinja :

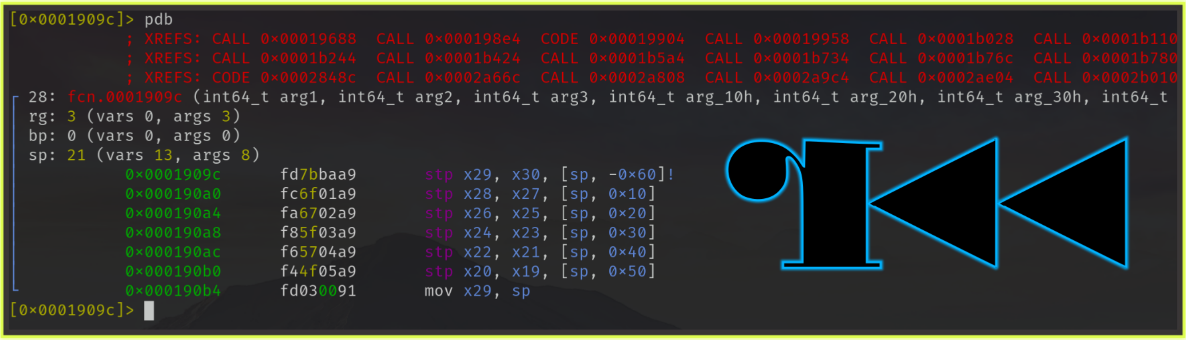


The addresses between the original binary and the modified binary are shifted by `0×1000` as a consequence of the LIEF internal mechanism to extend the exports table.



```
0001909c  sub_1909c:
0001909c  stp     x29, x30, [sp, #-0x60]! {var_60} {var_58}
000190a0  stp     x28, x27, [sp, #0x10] {var_50} {var_48}
000190a4  stp     x26, x25, [sp, #0x20] {var_40} {var_38}
000190a8  stp     x24, x23, [sp, #0x30] {var_30} {var_28}
000190ac  stp     x22, x21, [sp, #0x40] {var_20}
000190b0  stp     x20, x19, [sp, #0x50] {var_10
000190b4  mov     x29, sp {var_60}  {mktemp}
{ Falls through into mktemp }
```

We can observe a similar output in Radare2 :



```
[0x0001909c]> pdb
            ; XREFS: CALL 0x00019688  CALL 0x000198e4  CODE 0x00019904  CALL 0x00019958  CALL 0x0001b028  CALL 0x0001b110
            ; XREFS: CALL 0x0001b244  CALL 0x0001b424  CALL 0x0001b5a4  CALL 0x0001b734  CALL 0x0001b76c  CALL 0x0001b780
            ; XREFS: CODE 0x0002848c  CALL 0x0002a66c  CALL 0x0002a808  CALL 0x0002a9c4  CALL 0x0002ae04  CALL 0x0002b010
┌ 28: fcn.0001909c (int64_t arg1, int64_t arg2, int64_t arg3, int64_t arg_10h, int64_t arg_20h, int64_t arg_30h, int64_t
│ rg: 3 (vars 0, args 3)
│ bp: 0 (vars 0, args 0)
│ sp: 21 (vars 13, args 8)
│           0x0001909c      fd7bbaa9       stp x29, x30, [sp, -0x60]!
│           0x000190a0      fc6f01a9       stp x28, x27, [sp, 0x10]
│           0x000190a4      fa6702a9       stp x26, x25, [sp, 0x20]
│           0x000190a8      f85f03a9       stp x24, x23, [sp, 0x30]
│           0x000190ac      f65704a9       stp x22, x21, [sp, 0x40]
│           0x000190b0      f44f05a9       stp x20, x19, [sp, 0x50]
└           0x000190b4      fd030091       mov x29, sp
[0x0001909c]>
```

Tools	Impacted
IDA	Yes
BinaryNinja	Yes
Radare2	Yes
Ghidra	Yes

4. Sections Transformations

The transformations based on the exports table have an important impact on the efficiency and the accuracy of the disassemblers. To enhance our *Poor Man's Obfuscator* we can also apply transformations on the sections of the ELF and Mach-O binary. Whilst both formats have sections and segments, they are used differently by the loaders. Nevertheless, we can leverage the confusion made by the disassemblers between segments and sections to mislead the output of the disassemblers.

4.1 ELF

The ELF format is likely the most error-prone format to parse compared to the PE and the Mach-O format. One of the challenges when parsing an ELF binary is the duality between sections and segments. Basically, sections are used by the compilers/linker while segments are used by the loader to run the executable[1]. This means that sections should not be used to get the executable point of view of the binary. On the other hand, segments have a rough granularity over the data while sections give a better precision about the location and the meaning of the data.

For instance, it is less trivial to identify the location of the GOT based on the segments, while using the sections, the GOT is usually mirrored by the `.got` section.

We could completely get rid of the sections by removing the ELF sections table but disassemblers like IDA learned to handle such a case.

> Removing the sections table could be used as an anti-debug on Linux. Indeed, gdb is not able to debug a sectionless binary.

So instead of completely removing the sections table or arbitrarily corrupting the section attributes, the idea is to swap some sections, so that we keep a certain level of consistency while still breaking the overall layout.

This *swap* configuration is pretty efficient against the different disassemblers :

```
SWAP_LIST = [
    (".rela.dyn", ".data.rel.ro"),
    (".got",      ".got.plt"),
    (".plt",      ".text"),
    (".dynsym",   ".gnu.version"),
]
```

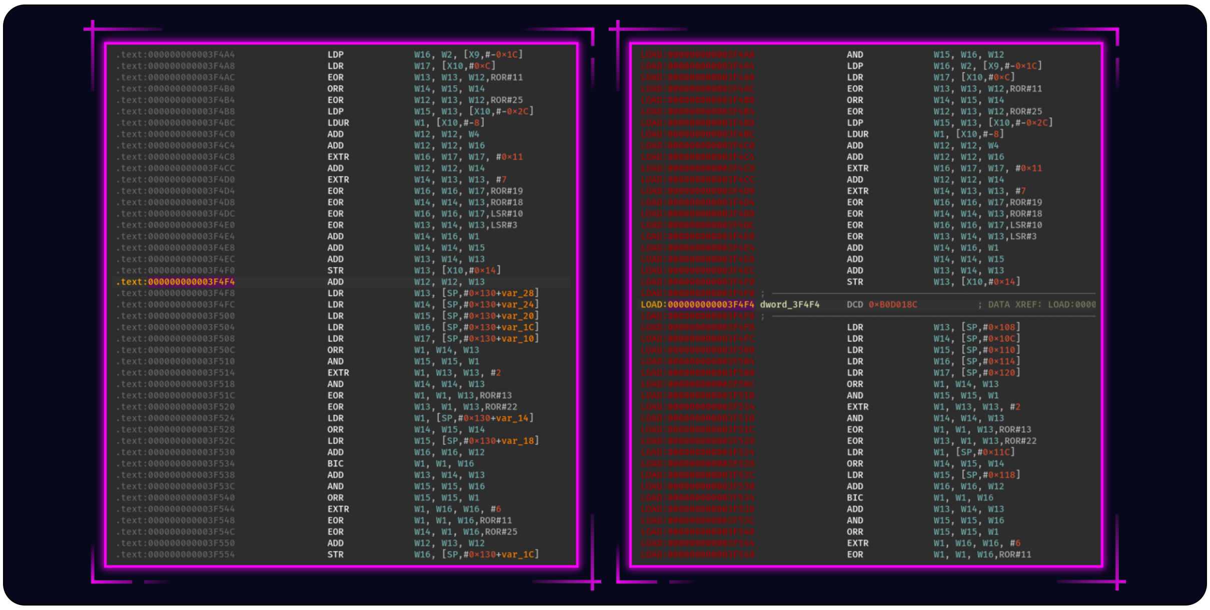
Radare2 and BinaryNinja faced difficulties with the imported symbols but the assembly code seems consistent with the original one. On the other hand, Ghidra is not able **to import the binary** even before launching the analysis. Lastly, IDA is able to load the binary but it corrupts the assembly code.

The swap list can be applied to the ELF binary with the following script :

1. This is **not exactly** true in particular on Android, where the loader enforces consistency between the `.dynamic` section and the `PT_DYNAMIC` segment.

```python
target = lief.parse("mbedtls_self_test.arm64.elf")

for (lhs_name, rhs_name) in SWAP_LIST:
    print(lhs_name, rhs_name)

    lhs = target.get_section(lhs_name).as_frame()
    rhs = target.get_section(rhs_name).as_frame()
    tmp = lhs.offset, lhs.size, lhs.name, lhs.type, lhs.virtual_address

    lhs.offset          = rhs.offset
    lhs.size            = rhs.size
    lhs.name            = rhs.name
    lhs.type            = rhs.type
    lhs.virtual_address = rhs.virtual_address

    rhs.offset          = tmp[0]
    rhs.size            = tmp[1]
    rhs.name            = tmp[2]
    rhs.type            = tmp[3]
    rhs.virtual_address = tmp[4]

target.write("swapped_alt_mbedtls_self_test.arm64.elf")
```

The figures below show the differences in IDA between the original ELF binary and the binary with swapped sections :



This case is pretty interesting because it exists obfuscation techniques which consist in adding junk code between instructions and jumping on the real instructions with an opaque predicate. With the swapped sections, IDA is adding itself (likely because of confusion on the relocations) the junk code, making the function corrupted.

Loading the binary with the option `Use SHT` disabled does not prevent the corruption.

Tools	Impacted	Note
IDA	Yes	The code is corrupted
BinaryNinja	Yes	The symbols are corrupted but the code seems consistent
Radare2	Yes	The imports are not recognized but the code seems consistent
Ghidra	Yes	We can't import the binary in Ghidra and especially, launch the analysis

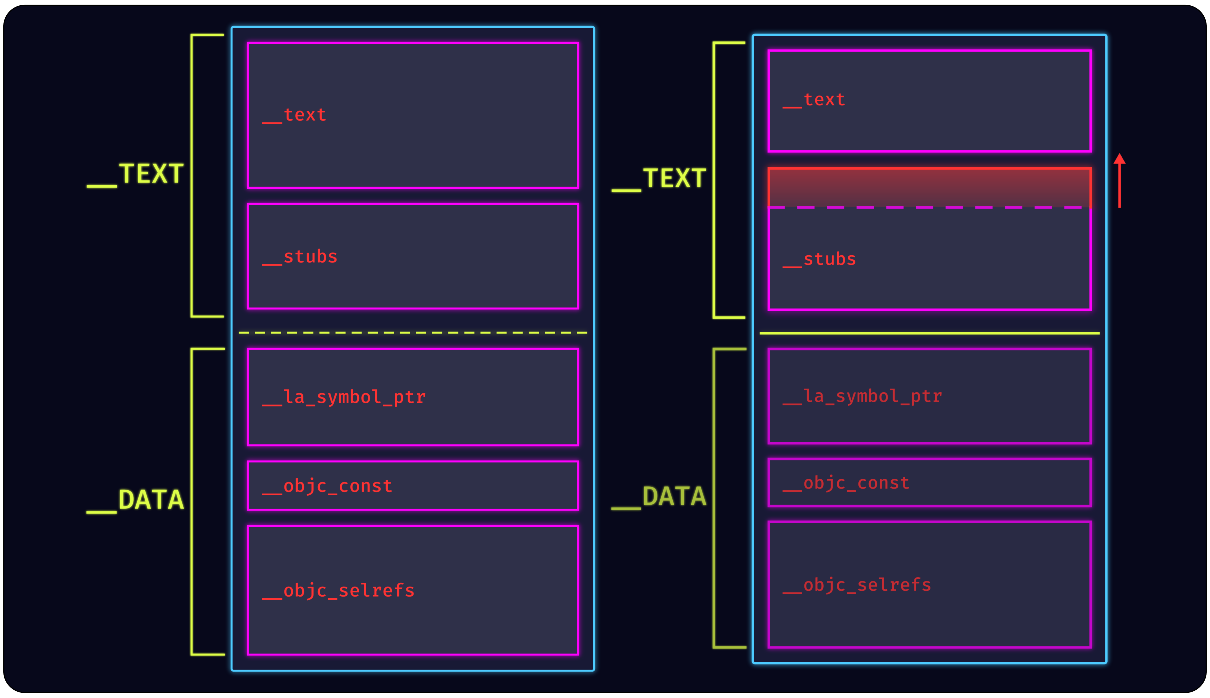
4.2 Mach-O

Compared to the ELF format, the Mach-O format and its loader `dyld` enforce a stricter layout such as it is not possible to swap sections without breaking the execution of the binary. Some of the checks performed by `dyld` on the layout of the sections are defined in `dyld/dyld3/MachOAnalyze.cpp`.

Among these checks, it verifies that :

1. The section's size is not negative (or overflow)
2. The section's virtual address and virtual size is within the segment's virtual address/virtual size
3. It enforces section size alignment for special sections like `MOD_INIT_FUNC_POINTERS`

So basically, sections are stronger bound to segments than for the ELF format. Nevertheless, within the `__TEXT` segment, we can perform a small modification which consists in *virtually* shifting the beginning of the `__stub` section over the original content of the `__text` section.



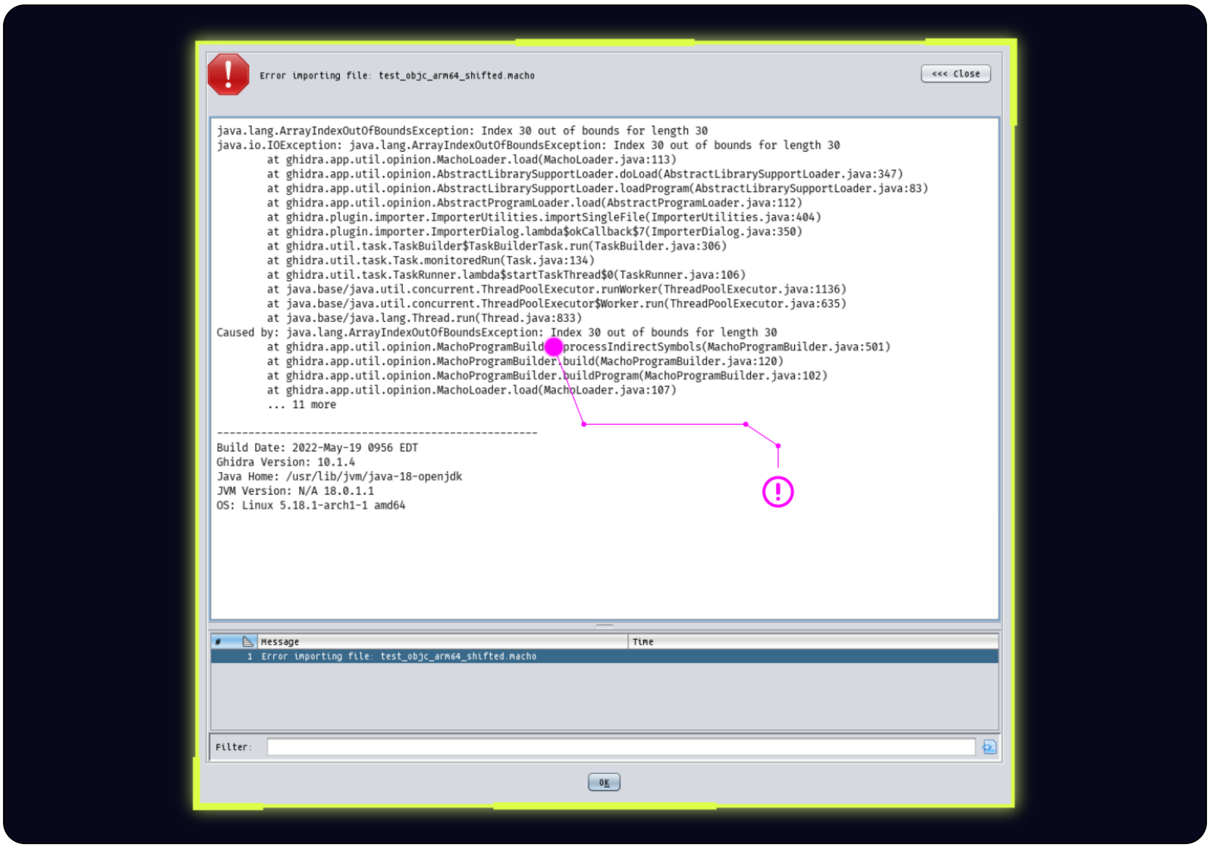
First off, these two sections are within the **same** `__TEXT` segment. Secondly, the transformation consists of a shift that keeps the **global** boundaries consistent with the original ones (i.e the size of all the sections does not change). The `__stubs` section is very similar to the ELF `.plt` section which is used for memoizing the resolution of the imports. In particular, the `__stubs` section contains **assembly code** (trampoline stubs) so it shares the same kind of content as the `__text` section.

Programmatically, we can perform the shift with the following piece of code :

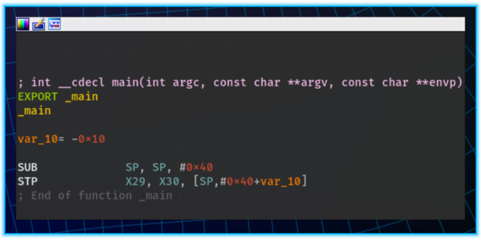
```
SHIFT = 0x100

__text  = target.get_section("__text")
__stubs = target.get_section("__stubs")

# Reduce the size of the __text section
__text.size              -= SHIFT

__stubs.offset           -= SHIFT
__stubs.virtual_address  -= SHIFT
__stubs.size             += SHIFT
```

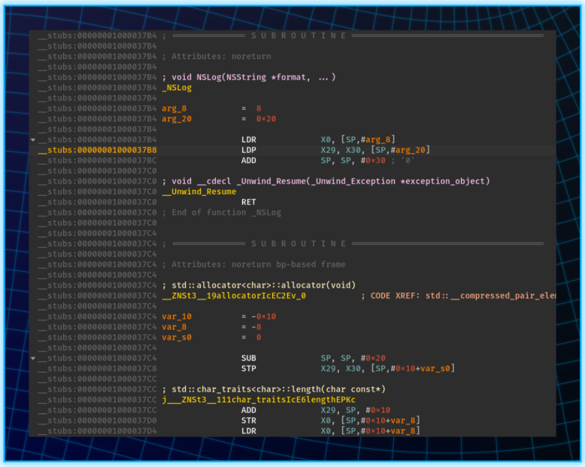
Radare2 and **BinaryNinja** seem not impacted by this transformation while Ghidra and IDA are strongly impacted.

Ghidra simply refuses to **import** the binary. The import action comes before the analysis action so this transformation prevents Ghidra from adding a binary to a project.



On the other hand, IDA is able to load the binary but its output is confusing. In particular, the main function is broken as we can observe in the following figures :





Even if the Mach-O loader enforces a stricter layout on the sections, small wisely chosen modifications enable to prevent some regular disassemblers from working correctly.

Tools	Impacted
IDA	Yes
BinaryNinja	No
Radare2	No
Ghidra	Yes

5. Specific Transformations

In the previous paragraphs, we detailed transformations based on structures shared by the two formats :

1. The exports
2. The sections

This part details other transformations that are specific to the Mach-O or the ELF format.

5.1 LC_FUNCTIONS_STARTS

The `LC_FUNCTIONS_STARTS` command is a kind of *debug* command that references a list of functions present in the binary. This command does not have an impact on `dyld` when loading the binary such as it is possible to completely corrupt its content or to modify the addresses of the functions.

Similar to the exports table, this command is used by the disassemblers to get a *trustworthy* list of functions to start disassembling.

The functions listed in this command are just a list of addresses relative to the default base address (given by the `__TEXT` segment virtual address). Using LIEF, we can apply a similar technique described in *Exports Addresses* by creating overlaps between two addresses :

```
functions = [f for f in LC_FUNCTION_STARTS.functions]
for idx, f in enumerate(functions):

    # Overlap 7 instructions
    if idx % 2 == 0:
        functions[idx] += 4 * 7
    else:
        functions[idx] -= 4 * 7

LC_FUNCTION_STARTS.functions = functions

bin.write("./fstart_mbedtls_self_test.arm64.macho")
```

This overlapping impacts all the disassemblers except Ghidra.

Tools	Impacted
IDA	Yes
BinaryNinja	Yes
Radare2	Yes
Ghidra	No

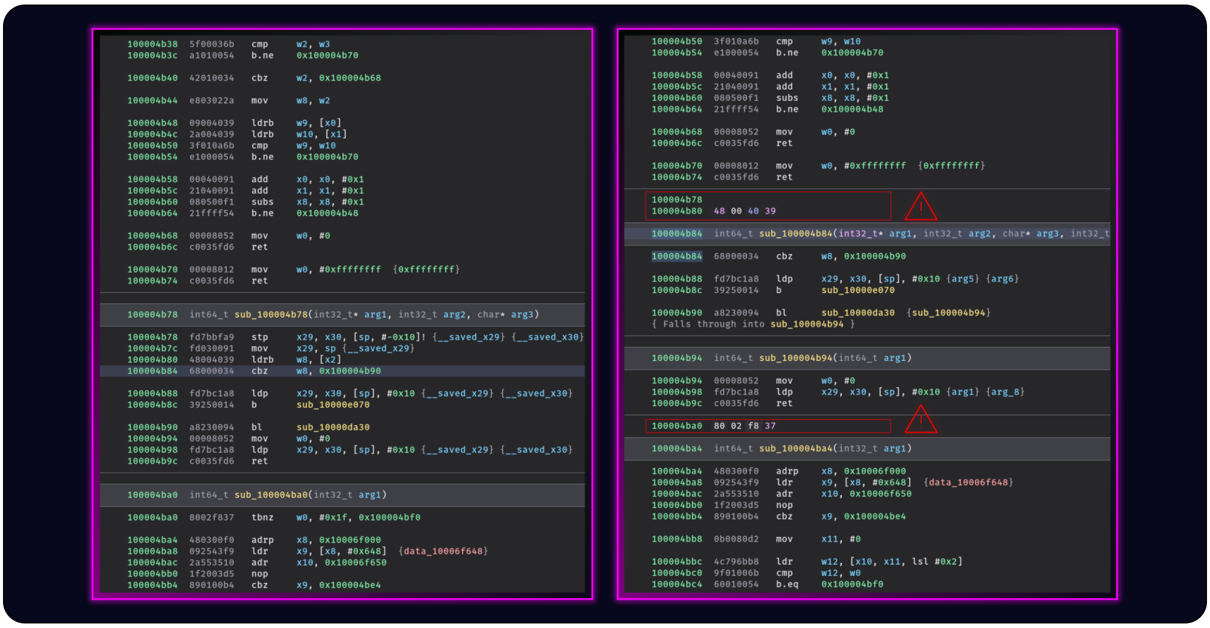


FIGURE1 – BinaryNinja

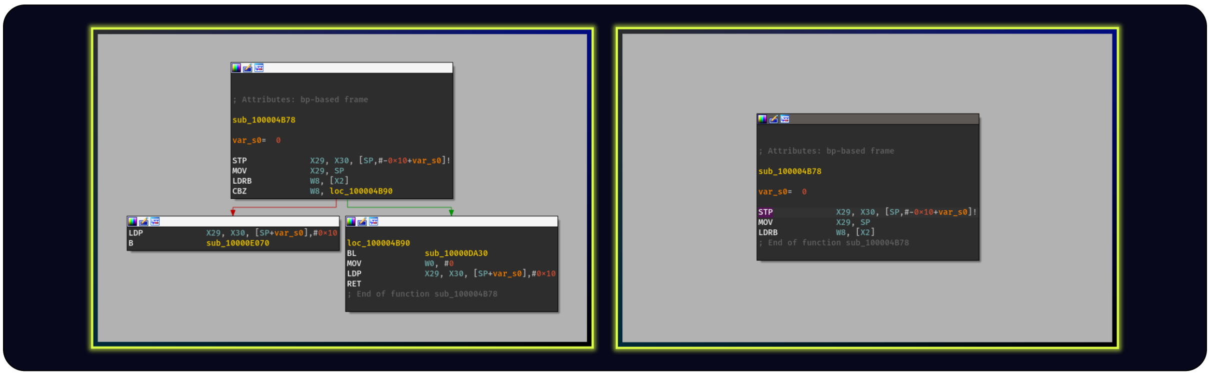


FIGURE2 – IDA

FIGURE3 – Radare2

5.2 .dynsym section

As already mentioned in the previous paragraphs : the ELF format is very tricky. In addition to the duality between sections and segments, counting the number of imported or the exported symbols referenced in the `.dynsym` section is not trivial.

The beginning of the symbols table associated with the imports and the exports is defined by the `DT_SYMTAB` entry which is located in the `PT_DYNAMIC` segment. The `DT_*` entries which reference a table are usually paired with another `DT_*SZ` entry which holds the size of the table. It turns out, there is an exception for the `DT_SYMTAB` entry which is not tied to another entry referencing its size.

On the other hand, the dynamic symbols table is mirrored by the `.dynsym` section which has a size. Therefore, it is appealing to use this section to count the number of entries in the table. As we mentioned in the paragraph *Sections Transformations*, ELF sections can't be trusted.

We can leverage this *feature* of the ELF format to artificially reduce the size of the `.dynsym` section :

```
dynsym      = target.get_section(".dynsym").as_frame()

sizeof      = dynsym.entry_size
osize       = dynsym.size
nsyms       = osize / sizeof
dynsym.size = sizeof * 3
```

This code artificially limits the size of the `.dynsym` section to 3 symbols.

Tools	Impacted	Note
IDA	Yes	The .dynsym symbols are not truncated when loading the binary with the option `'Use SHT'` disabled
BinaryNinja	No	
Radare2	Yes	The `ia` command does not show all the symbols but they are correctly referenced in the assembly code (e.g. `reloc.puts` instead of `sym.imp.puts`)
Ghidra	Yes	The symbols table is truncated and it seems there is no loading option preventing it

6. Conclusion

As it has been demonstrated through this paper, file format modifications can be powerful to prevent reverse engineering tools from working correctly. File format modifications are less resilient than classical obfuscation since the original assembly code remains unchanged. On the other hand, this is a topic that is less explored than regular obfuscation and for which, it exists less tooling, automation, and literature.

You can find the different scripts used for this work in the following GitHub repository : romainthomas/the-poor-mans-obfuscator