0.16.5 (a8f50413)
Updated on 19/04/2025, 08:49:57.

08 - Transforming an ELF executable into a library

In this tutorial, we will see how to convert a **PIE** executable into a library

Scripts and materials are available here: materials

By Romain Thomas - @rh0main , updated by Adrien Guinet - @adriengnt

Introduction

Actually, if we look at the header of a ELF PIE executable one can notice that it has the same type as a shared object (i.e. library)

```
$ readelf -h /usr/bin/ssh|grep Type
Type:  DYN (Shared object file)

$ readelf -h /usr/lib/libm.so|grep Type
Type:  DYN (Shared object file)
```

Using LIEF we can access this information through the `file_type` attribute

```
>>> libm = lief.parse("/usr/lib/libm.so.6")
>>> print(libm.header.file_type)
E_TYPE.DYNAMIC

>>> ssh = lief.parse("/usr/bin/ssh")
>>> print(ss.header.file_type)
E_TYPE.DYNAMIC
```

The main difference between a PIE binaries and a shared libraries is how symbols are exported.

A shared library aims to expose functions so that executable can bind to it whereas executables shouldn't not expose functions [1]

It's confirmed with the number of exported functions in the two different objects:

```
>>> print(len(libm.exported_functions))
572
>>> print(len(ssh.exported_functions))
10
```

In this tutorial we will see how we can transform raw function addresses into exported functions associated with a symbol, thus thus exposing internal functions of the executable.

Exporting functions

Such transformation can be useful if we found a function at a given address and want to instrument it (using `dlopen`/`dlsym` for example). Once the target function is exported we can link it as we would do for a *normal* library.

For example in a fuzzing scenario if one identifies a function that is a parser, we can export it and then we can feed its inputs with AFL. Thus we cut the path from the normal entrypoint to reach the function.

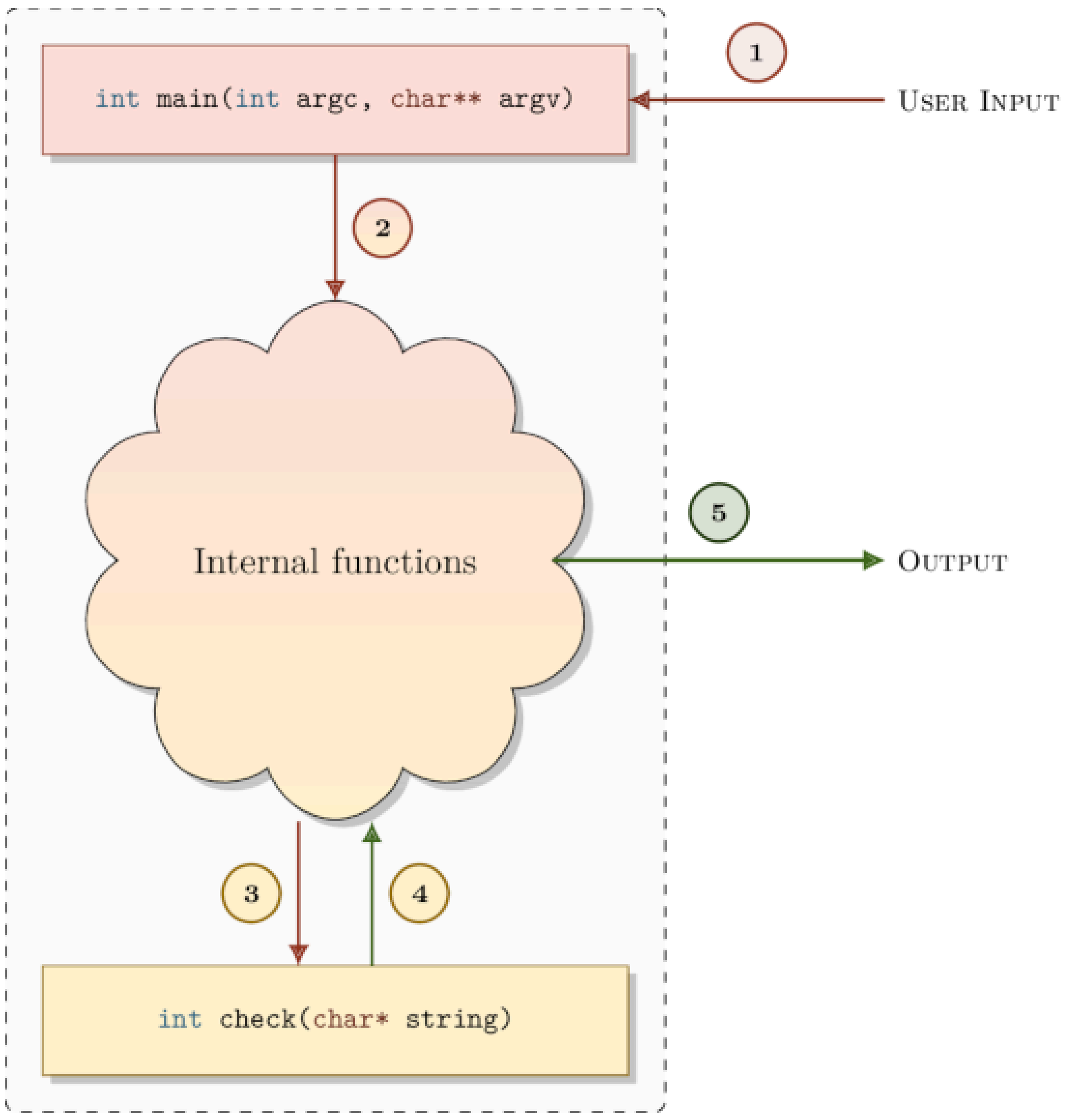
Let's see how it works on a basic *crackme*:

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

#define NOINLINE __attribute__ ((noinline))

NOINLINE int check_found(char* input) {
  if (strcmp(input, "easy") == 0) {
    return 1;
  }
  return 0;
}

int main(int argc, char** argv) {

  if (argc != 2) {
    printf("Usage: %s flag\n", argv[0]);
    exit(-1);
  }

  if (check_found(argv[1])) {
    printf("Well done!\n");
  } else {
    printf("Wrong!\n");
  }
  return 0;
}
```

This code takes a string as input and call the `check_found` function on this string, then it returns `1` if the input is `easy`. `0` otherwise.

The `__attribute__ ((noinline))` is used to make sure the `check_found` function won't be inlined by the compiler. Indeed, if the function check is inlined, there won't be an address associated to this function.

This figure sump-up the execution flow:



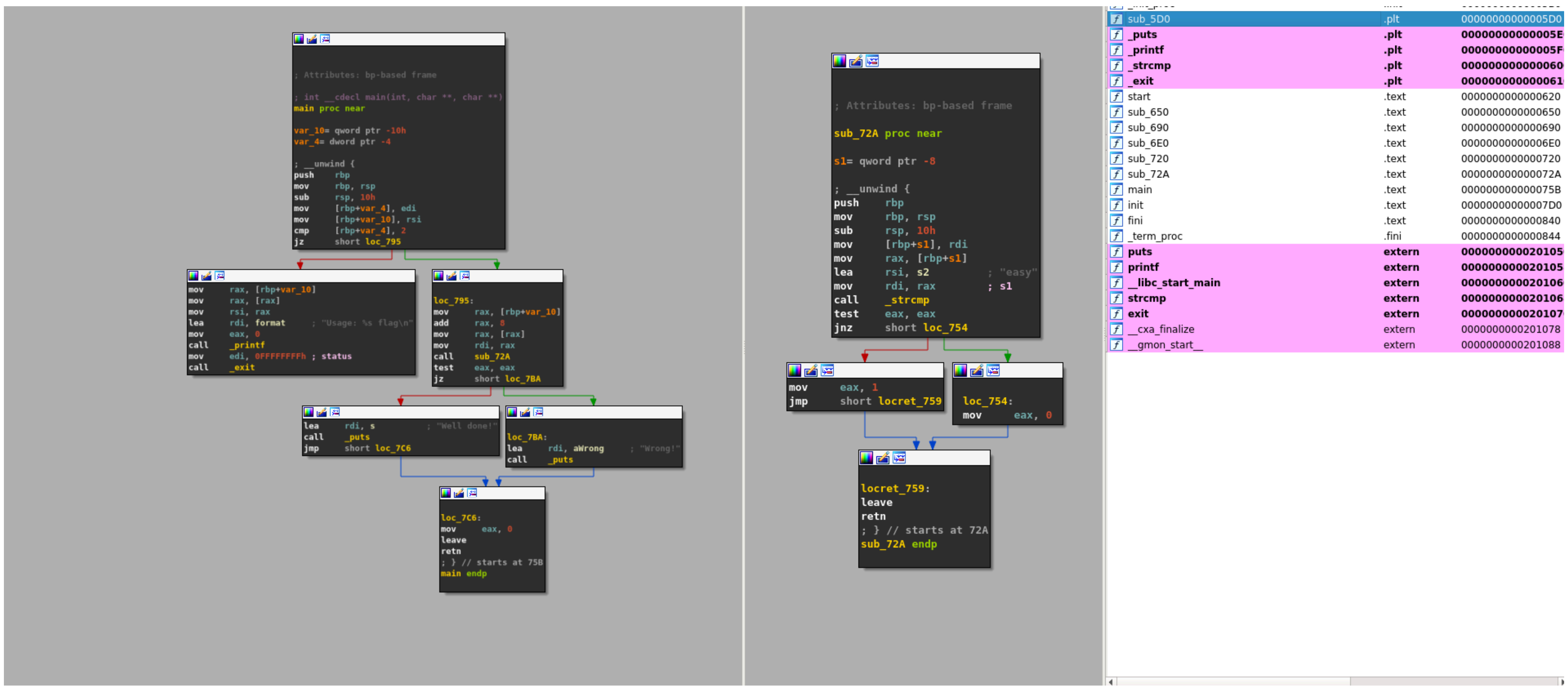
The *crackme* can be compiled with:

```
$ gcc crackme101.c -O0 -fPIE -pie -Wl,-strip-all,--hash-style=sysv -o crackme101.bin -fvisibility=hidden
$ ./crackme101.bin foo
Wrong!
$ ./crackme101.bin easy
Well done!
```

Note the usage of the `-fvisibility=hidden` flag. It makes the compiler not automatically export functions, like the `check_found` one. By opening `crackme101.bin` with LIEF, we can check that no functions are exported:

```
>>> import lief
>>> crackme101 = lief.parse("./crackme101.bin")
>>> print(len(crackme101.exported_functions))
0
```

Using a disassembler we can quickly identify the check function address:



```
; Attributes: bp-based frame

; int __cdecl main(int, char **, char **)
main proc near

var_10= qword ptr -10h
var_4= dword ptr -4

; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     [rbp+var_4], edi
mov     [rbp+var_10], rsi
cmp     [rbp+var_4], 2
jz      short loc_795
```

```
mov     rax, [rbp+var_10]
mov     rax, [rax]
mov     rsi, rax
lea     rdi, format     ; "Usage: %s flag\n"
mov     eax, 0
call    _printf
mov     edi, 0FFFFFFFFh ; status
call    _exit
```

```
loc_795:
mov     rax, [rbp+var_10]
add     rax, 8
mov     rax, [rax]
mov     rdi, rax
call    sub_72A
test    eax, eax
jz      short loc_7BA
```

```
lea     rdi, s          ; "Well done!"
call    _puts
jmp     short loc_7C6
```

```
loc_7BA:
lea     rdi, aWrong     ; "Wrong!"
call    _puts
```

```
loc_7C6:
mov     eax, 0
leave
retn
; } // starts at 75B
main endp
```

```
; Attributes: bp-based frame

sub_72A proc near

s1= qword ptr -8

; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     [rbp+s1], rdi
mov     rax, [rbp+s1]
lea     rsi, s2         ; "easy"
mov     rdi, rax        ; s1
call    _strcmp
test    eax, eax
jnz     short loc_754
```

```
mov     eax, 1
jmp     short locret_759
```

```
loc_754:
mov     eax, 0
```

```
locret_759:
leave
retn
; } // starts at 72A
sub_72A endp
```

sub_5D0	.plt	00000000000005D0
_puts	.plt	00000000000005E
_printf	.plt	00000000000005F
_strcmp	.plt	000000000000060
_exit	.plt	000000000000061
start	.text	0000000000000620
sub_650	.text	0000000000000650
sub_690	.text	0000000000000690
sub_6E0	.text	00000000000006E0
sub_720	.text	0000000000000720
sub_72A	.text	000000000000072A
main	.text	000000000000075B
init	.text	00000000000007D0
fini	.text	0000000000000840
_term_proc	.fini	0000000000000844
puts	extern	000000000020105
printf	extern	000000000020105
__libc_start_main	extern	000000000020106
strcmp	extern	000000000020106
exit	extern	000000000020107
__cxa_finalize	extern	0000000000201078
__gmon_start__	extern	0000000000201088

In this case, the **check** function is located at the address: `0x72A` [2]

Now that we identified the address we can export it as a named function: `check_found`

```
>>> crackme101.add_exported_function(0x72A, "check_found")
>>> crackme101.write("libcrackme101.so")
```

And that all!

`libcrackme101.so` is now a **library** that export one function: `check_found`.

```
>>> import lief
>>> libcrackme101 = lief.parse("./libcrackme101.so")
>>> print(len(crackme101.exported_functions))
1
>>> print(crackme101.exported_functions[0])
check_found
```

It turns out that `libcrackme101.so` is still an executable:

```
$ ./libcrackme101.so foo
Wrong!
$ ./libcrackme101.so easy
Well done!
```

Since we have exported a function we can now use `dlopen` on `libcrackme101.so` and `dlsym` on `check_found`

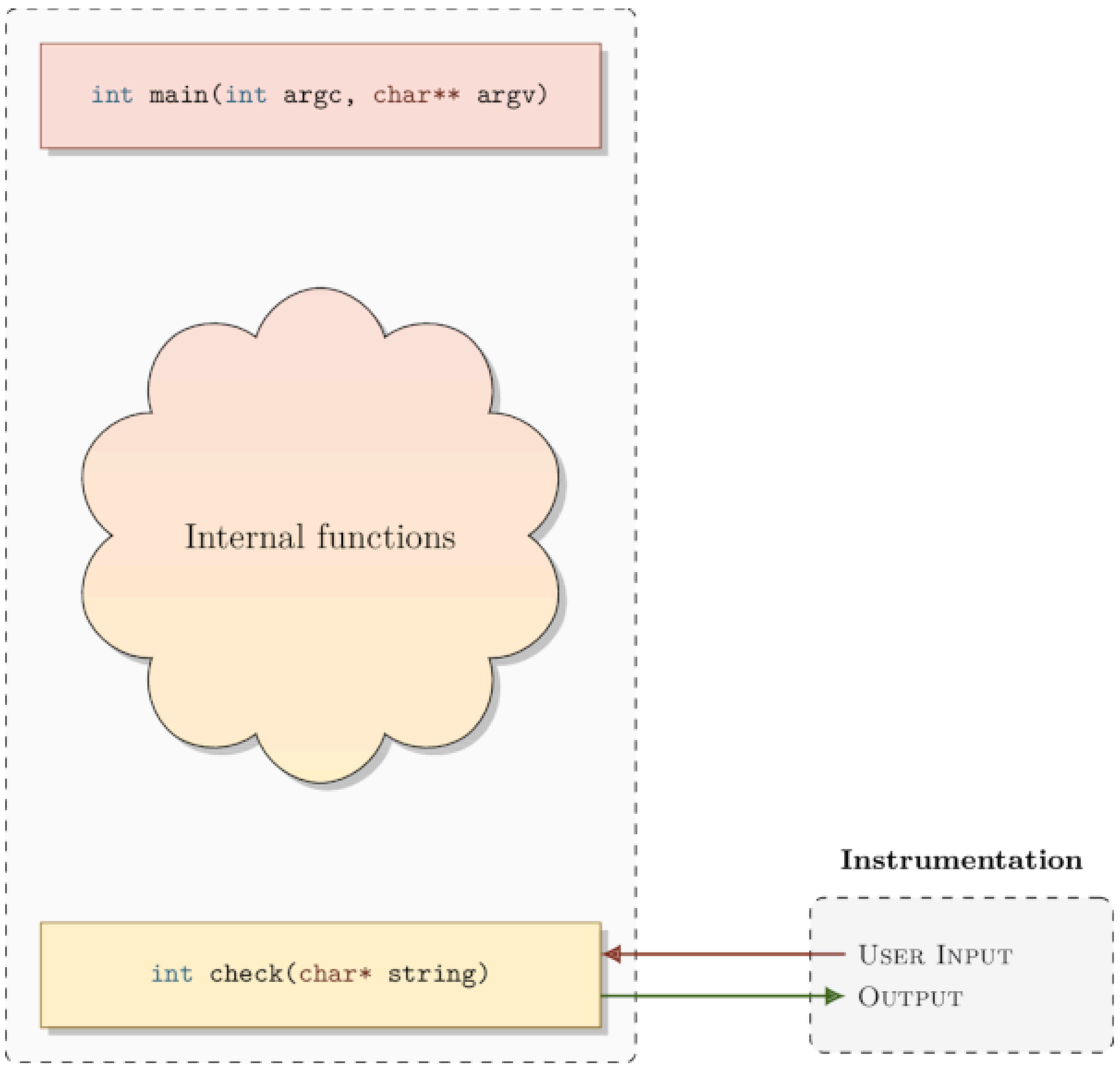
```
#include <dlfcn.h>
#include <stdio.h>
#include <stdlib.h>

typedef int(*check_t)(char*);

int main (int argc, char** argv) {

  void* handler = dlopen("./libcrackme101.so", RTLD_LAZY);
  if (!handler) {
    fprintf(stderr, "dlopen error: %s\n", dlerror());
    return 1;
  }
  check_t check_found = (check_t)dlsym(handler, "check_found");

  int output = check_found(argv[1]);

  printf("Output of check_found('%s'): %d\n", argv[1], output);

  return 0;
}
```

Running the code above should give a similar output:

```
$ gcc instrument.c -O0 -fPIE -pie -o instrument.bin -ldl
$ ./instrument.bin test
Output of check('test'): 0
$ ./instrument.bin easy
Output of check('easy'): 1
```

If `dlopen` returns an error, please read the following section about glibc >= 2.29.

The transformation of the execution flow can be represented as follow:



Warning for glibc >= 2.29 users

If you are using `glibc ⩾ 2.29` (or a close version depending on your Linux distribution), you might have encountered this error while using the *dlopen* function:

```
dlopen error: cannot dynamically load position-independent executable
```

Loading PIE binaries as shared libraries wasn't indeed really an intended use case for `dlopen`, and it used to work without really being properly supported. One of the reasons is that it does not seem that trivial to support all the possible use cases (issues with some relocations and ELF constructors).

These glibc versions now implements a check to deny calls to `dlopen` with PIE binaries. This is done by verifying the `DF_1_PIE` flag isn't present in the list of dynamic information flags.

In order to circumvent this test, LIEF can be used to remove this `DF_1_PIE` flag:

```
import lief
import sys
path = sys.argv[1]
bin_ = lief.parse(path)
bin_[lief.ELF.DynamicEntry.TAG.FLAGS_1].remove(lief.ELF.DynamicEntryFlags.FLAG.PIE)
bin_.write(path + ".patched")
```

Conclusion

Because PIE executables aim to be mapped at a random base address, they globally behave as a library. We only need to export the *interesting* functions.

For non-PIE executables such transformation would be very difficult because it requires to transform first the executable into a *relocatable* executable. It means creating relocations, patching absolute *jump*, …

LIEF only support this transformation for ELF and we need to investigate the PE and Mach-O cases [3].

Notes

[1] Some functions can be exported by the linker such as `_init`

[2] The mapped virtual address will be `BASE + 0x72A` where `BASE` is randomly choosed by the ASLR

[3] In OSX all executables are compiled with the PIE flag.

API:
- `lief.ELF.Binary.add_exported_function()`
- `lief.ELF.Binary.export_symbol()`
- `lief.ELF.Symbol.visibility`
- `lief.ELF.Symbol.name`
- `lief.ELF.Symbol.value`