Control Flow Obfuscation — What happens if we modify callee-saved registers? 🙈

⋮ 17/08/2025

Posted Aug 17, 2025 Updated Aug 19, 2025
By *Elma* *13 min* read

I've always had much appreciation for all the low-level things from assembly to compilers and more. In my pursuit to better understand these mechanisms, I'm often left with many questions on what happens if we challenge the different assumptions and conventions that our compilers are built upon.

In this post, we will question the following conventions and break some assumptions made by disassemblers/decompilers to obfuscate control flow and hide code!

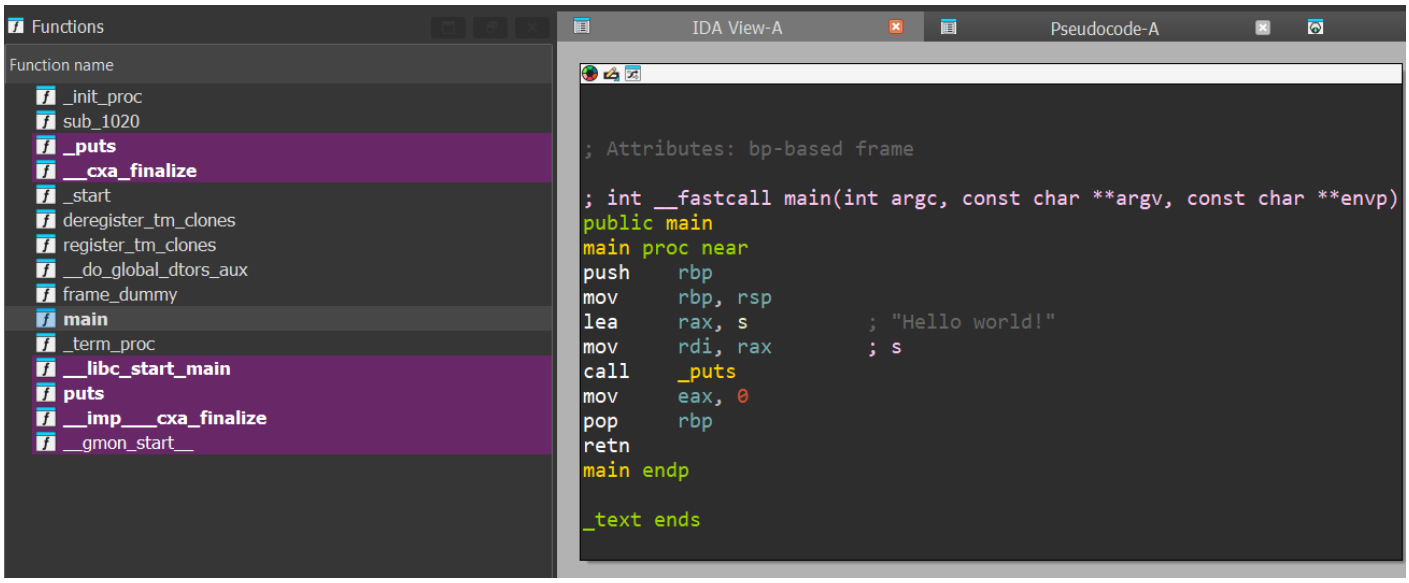
What happens when we break compiler conventions?

What happens if you modify registers that do not belong to you?

Proof of Concept (PoC)

Before we dive into the details, let me first share a snippet of the proof of concept! You can find the binary here.

As you can see, the program is pretty small and the main function does nothing except `put("Hello world!")` and it does not have many defined functions as well.



disasembly of the main function

Probably, the most “out of the ordinary” function would be `frame_dummy` that is typically overlooked since it exists in most ELF binaries as some sort of a placeholder function.

disasembly of the frame_dummy function

However, if you run the program with the correct parameters, it will print `nice!`.

```
1 ❯ ./helloworld.elf
2 Hello world!
3
4 ❯ ./helloworld.elf sctf{going_beyond_hello_world_1z2ket65cx3sdxfjb}
5 nice!
6 Hello world!
```

Where in the code does it even check for the command line argument, and prints `nice!`??

Compiler Convention on x64 Registers

For those with some assembly knowledge, we often know of registers simply as a variable that stores a value.

Some registers are known as special-purpose registers,

- `RIP`: Holds the address of the next instruction to be executed
- `RSP`: Holds the address of the top of the stack frame *(modifiable via `PUSH`/`POP`)*

while the rest are often simply called general purpose registers that we can also classify as caller/callee-saved registers.

Caller- and Callee-Saved Registers

In Lab 3, your compiler's code-generation and register allocation phases will need to distinguish between *callee-saved* and *caller-saved* registers:

- The values stored in **callee-saved registers** must be preserved across function calls. This means that your function must save and restore any callee-saved registers that it modifies.
- The values stored in **caller-saved registers** may be modified by any function call, so your compiler cannot assume that they will retain their values after calling a function. If you need those values to be preserved, you must save and restore them before and after the function call.

Note that all registers used to pass and return arguments (%rdi, %rsi, %rdx, %rcx, %r8, %r9, %rax) are also **caller-saved registers**.

In your register allocation, you will probably want to con-

Function	64-bit	32-bit	16-bit	8-bit
Return Value	**%rax**	**%eax**	%ax	%al
Callee saved	**%rbx**	**%ebx**	%bx	%bl
4th Argument	**%rcx**	**%ecx**	%cx	%cl
3rd Argument	**%rdx**	**%edx**	%dx	%dl
2nd Argument	**%rsi**	**%esi**	%si	%sil
1st Argument	**%rdi**	**%edi**	%di	%dil
Callee saved	**%rbp**	**%ebp**	%bp	%bpl
Stack Pointer	**%rsp**	**%esp**	%sp	%spl
5th Argument	**%r8**	**%r8d**	%r8w	%r8b
6th Argument	**%r9**	**%r9d**	%r9w	%r9b
Caller saved	**%r10**	**%r10d**	%r10w	%r10b
Caller saved	**%r11**	**%r11d**	%r11w	%r11b
Callee saved	**%r12**	**%r12d**	%r12w	%r12b
Callee saved	**%r13**	**%r13d**	%r13w	%r13b
Callee saved	**%r14**	**%r14d**	%r14w	%r14b
Callee saved	**%r15**	**%r15d**	%r15w	%r15b

reference: Notes from CMU Compiler Design Class

Variables in our programs are typically either stored on the stack or in registers.

Variables in the stack will need to be read and updated through stack dereferences which becomes inefficient for variables that are frequently used throughout a function.

As such, callee-saved registers are sometimes used by the compiler to improve efficiency of the program.

> Sometimes, caller-saved registers are **also called volatile registers** since the value of these registers are volatile and may be modified after a function call.
>
> Conversely, callee-saved registers are **also called non-volatile registers** since the values of these registers are not modified after a function call.

Case Study: Calling constructors in Ubuntu's GLIBC

This idea for obfuscation first came to mind when I was debugging GLIBC internals one day, specifically how constructors are called from `__libc_start_main`. Here's the snippet of code responsible for calling the chain of constructors.

```
1  ; snippet of assembly code taken from __libc_start_main (ubuntu glibc 2.35)
2
3  loc_29EA8:
```

```
4                   mov     rdx, [rsp+48h+var_48]
5                   add     r14, 8 ; get next constructor
6
7   loc_29EB0:
8                   mov     [rsp+48h+var_48], rdx
9                   mov     rsi, r12 ; prepare argv
10                  mov     edi, ebp ; prepare argc
11                  call    qword ptr [rcx] ; call the constructor
12                  mov     rcx, r14
13                  cmp     [rsp+48h+var_40], r14 ; checks if any constructors left
14                  jnz     short loc_29EA8
```

Essentially, there are **3 callee-saved registers** used here.

register	purpose
EBP	`argc` - the number of command line arguments the program is executed with
R12	`argv` - the list of command line arguments the program is executed with
R14	the address where the list of addresses of the constructors are stored

Additionally, `var_40` stores the address of the last constructor in the list of constructors *(defined in .init_array)* and `var_48` stores the pointer to the list of environment variables *(aka `char* envp[]`)*.

The assumption here is that these registers will not be changed by the constructor functions. However, if we are able to modify `r14`, we could possibly **repeat constructor functions** or even **call a different function by writing into R14**!

Endlessly looping our constructor

By analyzing the assembly above, we can see that it is running in a loop where `r14` is the loop index and `r14 != var_40` is the terminating condition.

We can modify `r14 -= 8` to make it call the same constructor over and over again!

```
1   #include <stdio.h>
2
3   // r14 -> pointer to constructor array
4   // r12 -> pointer to argv
5   // ebp -> argc
6
7   int __attribute__((naked)) __attribute__((constructor)) func1() {
8           write(1, "func1 called\n", 14);
9           __asm__(
10                  ".intel_syntax noprefix\n"
11                  "sub r14, 8\n"
12                  "ret\n"
13                  ".att_syntax\n"
14          );
15  }
16  int main() {
17          puts("hello world");
18  }
```

If we try to compile and run the above function, it will re-run the constructor and print `func1 called` forever.

Call a different function

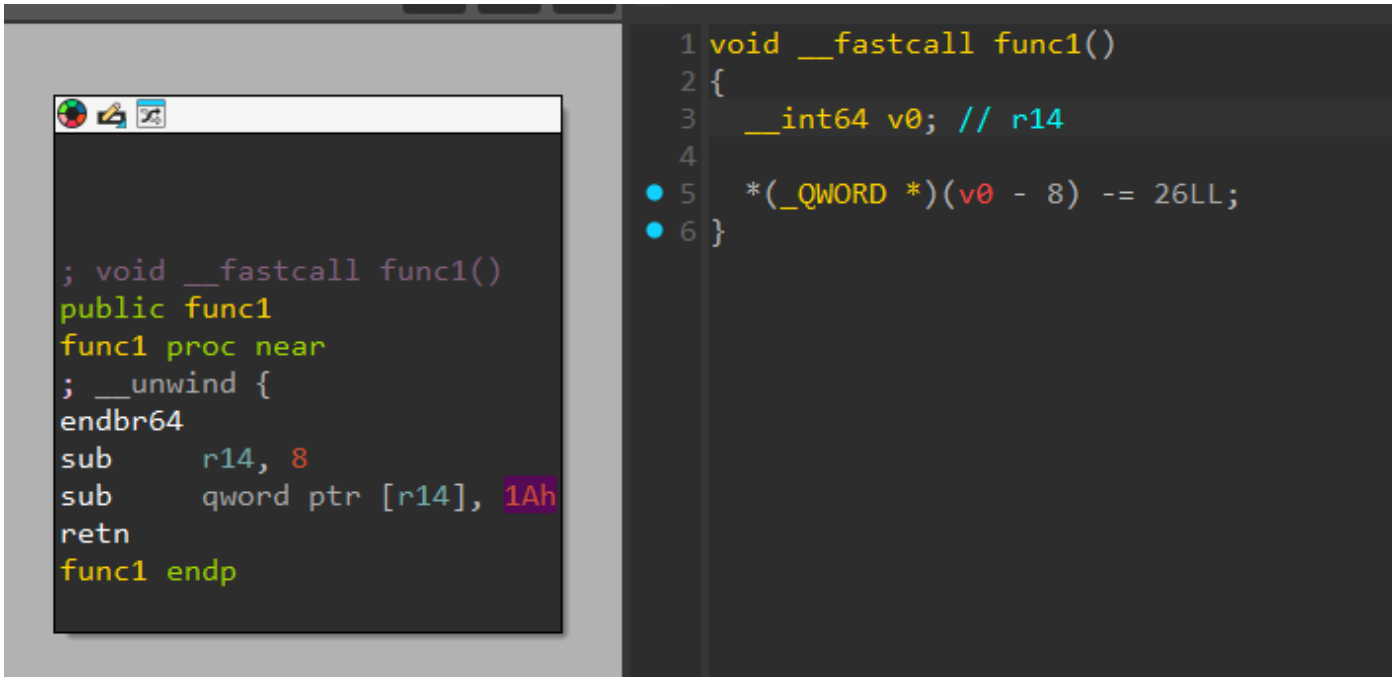
If we go through an extra step to modify the pointer within `r14`, we can achieve calls into other functions.

```
#include <stdio.h>

// r14 -> pointer to constructor array
// r12 -> pointer to argv
// ebp -> argc

void __attribute__((used)) secret() {
        puts("secret");
}

int __attribute__((naked)) __attribute__((constructor)) func1() {
        __asm__(
                ".intel_syntax noprefix\n"
                "sub r14, 8\n"
                "sub qword ptr [r14], func1-secret\n" // we want to find a way to hide
this
                "ret\n"
                ".att_syntax\n"
        );
}
int main() {
        puts("hello world");
}
```

Running the above program will give us this result

```
❯ ./a.out
secret
hello world
```

And let's look at how this looks like in a decompiler!



Although it looks suspicious due to the red font in the decompilation, it is not immediately obvious what this code is doing especially since the logic that executes the `secret` function is hidden within the library code that is not visible in the decompiler.

By building on top of this small poc, we can create larger programs that will be a pain to reverse due to the difficulty in identifying the program flow.

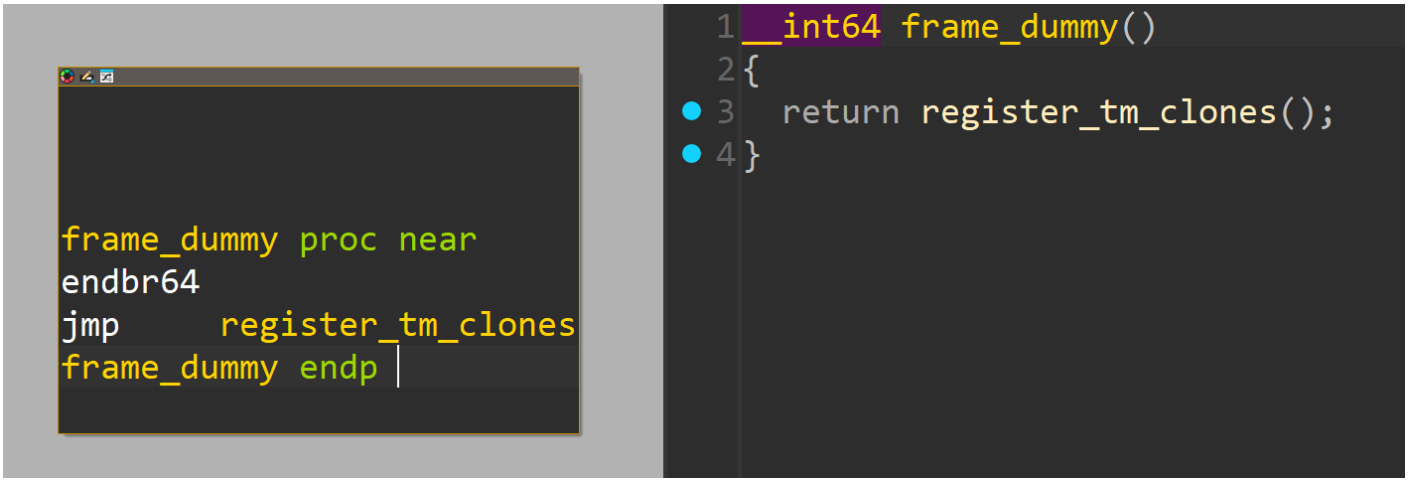
Bonus - How do we further hide our “malicious” code?

We’ve managed to obfuscate our control flow by continuously modifying the constructor chain pointer and the constructor address within it.

How can we further make our compiled program more discreet and not draw attention to the constructor modifying assembly snippet?

Hide the initial jump in a ‘default’ function

By default, ELF files compiled in GCC contains the `frame_dummy` function which is executed as a constructor function.



This commonly seen function has become an easily overlooked function by reverse engineers.

As such, if we could fit out callee-saved register modifying piece of assembly code within this function, it would be a perfect hiding place.

To achieve that, I patched the compiled binary to

- Fill up the original `frame_dummy` with a bunch of random bytes
- Add `jmp register_tm_clones` to the end of my assembly snippet
- Patch `SYMTAB` to set the address of `frame_dummy` to my assembly snippet

Hide the initial jump from being decompiled

If we were to write a constructor with the following piece of assembly code,

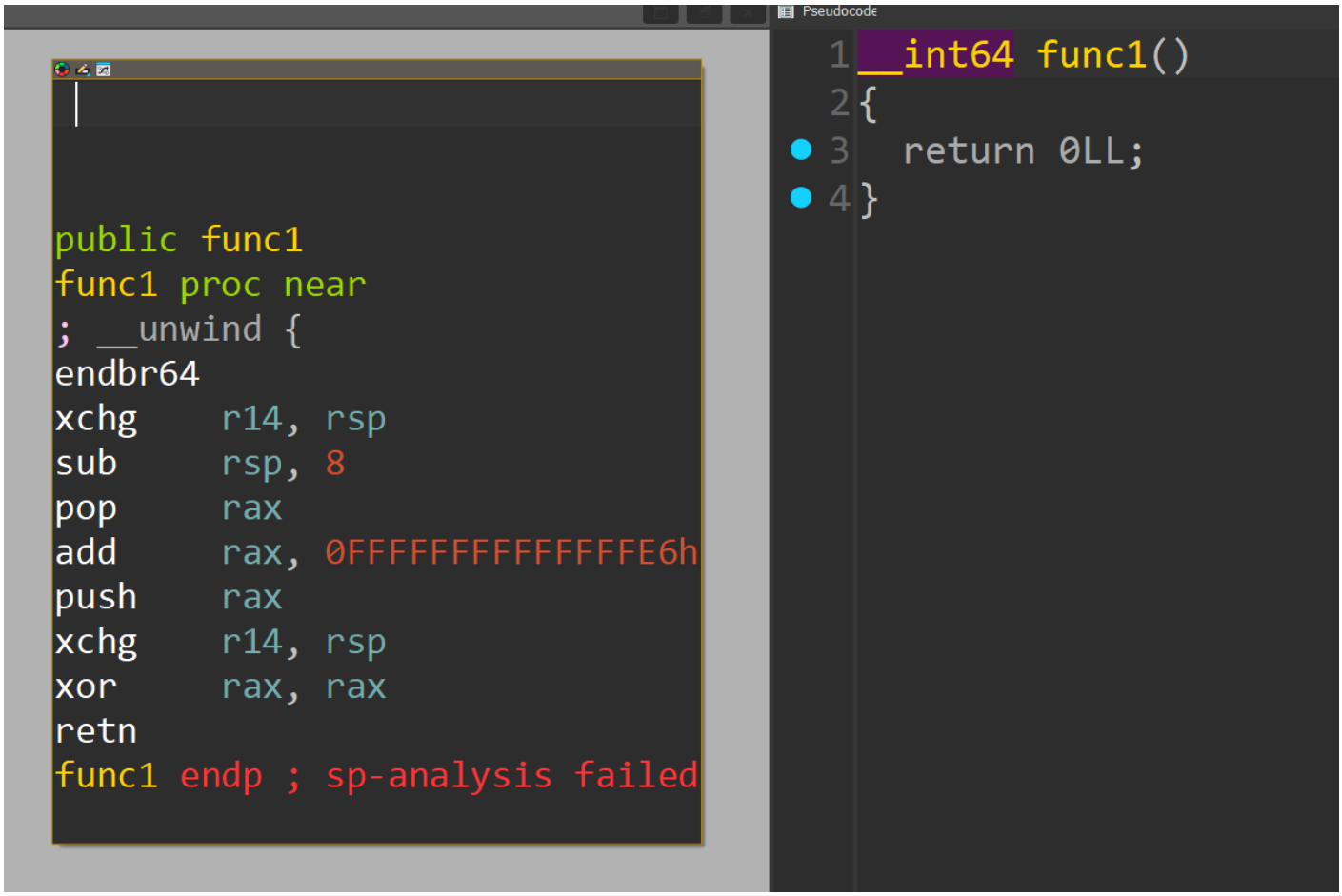
```
1 sub r14, 8
2 add QWORD PTR [r14], 1337
3 ret
```

This would be easily recognized and decompiled by tools like IDA to display the following, which might raise suspicion.

```
1 void __fastcall func1()
2 {
3   __int64 v0; // r14
4
5   *(_QWORD *)(v0 - 8) -= 26LL;
6 }
```

However, if we messed up the assembly to look weird enough *(and still accomplish the same effect)*, IDA will actually ignore it and not decompile anything! :)

In this specific case, we can use `push` and `pop` to effectively do any read/write.



Sick!

> Also check out this challenge where I obfuscated a program by replacing all the `mov` instructions with `push` and `pop`.
>
> That totally broke the decompilation xd

Hide the main shellcode from disassembly

By default, IDA will sweep the bytes in the `.text` section to try to identify and disassembly any assembly code.

This makes it difficult to discreetly weave in shellcode that we can jump to.

To solve this, we can actually simply add a bunch of random bytes before the shellcode so that IDA will give up on disassembling the entire chunk of bytes.

```
# add some junk bytes before
strip_dumb:
        .string "\x0e\xe9\x86\xf1/l\xdc\xaa/OZ>\xf9\xbd\xff\x10y\xc6\xf9\xe4"

# make RWX memory so we can decrypt our memory inplace
strip_make_rwx_memory:
        lea rsi, [rip]
        mov edi, 0xfff
        not rdi
        and rdi, rsi
        lea rbx, [rip+frame_dummy]
        xor rdx, rdx
        xor rsi, rsi
        xor rax, rax
        mov si, 0x1000
        mov dl, 0x7
        mov al, 0xa
        syscall

        # jmp to next part of shellcode
        xchg rsp, r14
        sub rsp, 8
        pop rax
        sub rax, strip_make_rwx_memory-strip_decrypt_memory_init
        push rax
        xchg rsp, r14
        ret

# add some junk bytes after
strip_dump2:
        .string "\x9f\xb6LAJVH#T\xc1\x14-\x81v\xcc\xe9\x8dPP\x8a"
```

Stripping unwanted symbols

When writing my shellcode, I made use of many symbols that would make reversing much more trivial if they are left within the program.

However, I did not want to entirely strip the program as that would remove symbols like `main` and `frame_dummy` which would prompt the reverse engineer to manually reverse every function.

As such, I prefixed all my symbols with `strip_` and manually parsed the symbol table to remove entries that begin with `strip_`.

The final script can be found later

Encrypt our shellcode

To further discourage IDA to disassemble our shellcode, I also encrypted most of my shellcode at runtime.

I used a script to encrypt the bytes between the symbols `ENCRYPT_BEGIN` and `ENCRYPT_END` and included a decrypting routine at the start of the shellcode.

```
lea r8, [rip+ENCRYPT_BEGIN] # begin of encryption area
lea r9, [rip+ENCRYPT_END] # end of encryption area
lea rbx, [rip+frame_dummy] # xor key

# we implement our memory decryption here
strip_decrypt_memory_init:
        mov al, byte ptr [rbx]
        xor [r8], al
```

```
9           inc rbx
10          inc r8
11
12          xchg rsp, r14
13          sub rsp, 8
14          pop rax
15          cmp r8, r9 # if r8 == r9
16          jnz strip_l20
17          add rax, strip_check_flag_init-strip_decrypt_memory_init
18  strip_l20:
19          push rax
20          xchg rsp, r14
21          ret
22
```

The loop is also implemented by deciding using a comparison to decide whether to modify the constructor pointer.

The ugly patching code

Finally, here's the super super ugly code that I used to patch the ELF to look as "Hello World" as possible :)

```
1   from elftools.elf.elffile import ELFFile
2   from capstone import *
3   from keystone import *
4   from pwn import xor
5   import struct
6
7   cs = Cs(CS_ARCH_X86, CS_MODE_64)
8   cs.detail = True
9
10  ks = Ks(KS_ARCH_X86, KS_MODE_64)
11
12  def get_symtab_sh_info_offset(filename):
13      with open(filename, 'rb') as f:
14          elf = ELFFile(f)
15          symtab = elf.get_section_by_name('.symtab')
16          if symtab is None:
17              return None, "No .symtab section found"
18          section_index = elf.get_section_index(symtab.name)
19          section_header_offset = elf['e_shoff'] + section_index * elf['e_shentsize']
20          sh_info_offset = section_header_offset + 44
21          return sh_info_offset
22
23  def get_symtab_sh_size_offset(filename):
24      with open(filename, 'rb') as f:
25          elf = ELFFile(f)
26          symtab = elf.get_section_by_name('.symtab')
27          if symtab is None:
28              return None, "No .symtab section found"
29          section_index = elf.get_section_index(symtab.name)
30          section_header_offset = elf['e_shoff'] + section_index * elf['e_shentsize']
31          sh_size_offset = section_header_offset + 32
32          return sh_size_offset
33
34  filename = "poc"
35
36  with open(filename, 'rb') as f:
37      elffile = ELFFile(f)
38
39      symtab = elffile.get_section_by_name('.symtab')
40      offset = symtab['sh_offset']
41      size = symtab['sh_size']
42      symtab = symtab.data()
43
```

```
    strtab = elffile.get_section_by_name('.strtab')
    strtab_offset = strtab['sh_offset']
    strtab = strtab.data()

symtab = [symtab[i:i+0x18] for i in range(0, len(symtab), 0x18)]
final_symtab = []
strtab_strip = []
# strip symbols that we should strip
# we should also strip strtab names
for entry in symtab:
    sym_name_offs = struct.unpack("<I", entry[:4])[0]
    if sym_name_offs:
        sym_name = ""
        i = 0
        while True:
            if strtab[sym_name_offs+i] == 0:
                break
            sym_name += chr(strtab[sym_name_offs+i])
            i += 1
        if sym_name.startswith("strip") or sym_name.startswith(".strip"):
            strtab_strip.append((sym_name_offs, i))
            continue
    final_symtab.append(entry)

# now we also want to remove frame_dummy
frame_dummy_entry = False
for entry in final_symtab:
    sym_name_offs = struct.unpack("<I", entry[:4])[0]
    if sym_name_offs:
        sym_name = ""
        i = 0
        while True:
            if strtab[sym_name_offs+i] == 0:
                break
            sym_name += chr(strtab[sym_name_offs+i])
            i += 1
        if sym_name == "frame_dummy" and not frame_dummy_entry:
            frame_dummy_entry = entry
            frame_dummy_addr = struct.unpack("<Q", entry[8:16])[0]
            # strtab_strip.append((sym_name_offs, i))
            # break
        elif sym_name.endswith("ENCRYPT_BEGIN"):
            encrypt_start = entry
            encrypt_start_addr = struct.unpack("<Q", entry[8:16])[0]
            print(encrypt_start_addr)
        elif sym_name.endswith("ENCRYPT_END"):
            encrypt_end = entry
            encrypt_end_addr = struct.unpack("<Q", entry[8:16])[0]
            print(encrypt_end_addr)
final_symtab.remove(frame_dummy_entry)
final_symtab.remove(encrypt_start)
final_symtab.remove(encrypt_end)


# find fake frame_dummy
for entry in final_symtab:
    sym_name_offs = struct.unpack("<I", entry[:4])[0]
    if sym_name_offs:
        sym_name = ""
        i = 0
        while True:
            if strtab[sym_name_offs+i] == 0:
                break
            sym_name += chr(strtab[sym_name_offs+i])
            i += 1
        if sym_name == "frame_dummy":
            fake_frame_dummy_addr = struct.unpack("<Q", entry[8:16])[0]

```

```
112 new_symtab = b"".join(final_symtab) + b"\x00"*(0x18*(len(symtab)-len(final_symtab)))
113 assert len(new_symtab) == size
114
115 with open(filename, 'rb') as f:
116     file_contents = bytearray(f.read())
117
118 # we create our new strtab
119 file_contents[offset:offset+size] = new_symtab
120
121 # we NULL all the strtab entries that are useless now
122 # we don't want to give readable strings ;)
123 for entry in strtab_strip:
124     file_contents[strtab_offset+entry[0]:strtab_offset+entry[0]+entry[1]] =
125 b"\x00"*entry[1]
126
127 # we need to update symtab section size
128 file_contents[get_symtab_sh_size_offset(filename):get_symtab_sh_size_offset(filename)+8]
129 = struct.pack("<Q", 0x18*(len(final_symtab))) # why do we need to include X blank
130 entries??
131
132 # we need to update symtab sh_info to 0
133 file_contents[get_symtab_sh_info_offset(filename):get_symtab_sh_info_offset(filename)+4]
134 = struct.pack("<I", 0) # why do we need to include X blank entries??
135
136 # we null the old constructor frame_dummy
137 register_tm_clones_addr =
138 next(cs.disasm(file_contents[frame_dummy_addr+4:frame_dummy_addr+9],
139 frame_dummy_addr+4)).op_str
140 file_contents[frame_dummy_addr:frame_dummy_addr+9] = b"\x00"*9
141
142 # we add jmp register_tm_clones back into our fake frame dummy to make it seem legit
143 replace_offs = file_contents.index(struct.pack(">I", 0xcafebabe))
144 file_contents[replace_offs:replace_offs+5] = ks.asm(f"jmp {register_tm_clones_addr}",
145 replace_offs)[0]
146
147 # we replace the old constructor with the new constructor
148 file_contents = file_contents.replace(struct.pack("<Q", frame_dummy_addr),
    struct.pack("<Q", fake_frame_dummy_addr))

    # we ENCRYPT the code between ENCRYPT_START and ENCRYPT_END
    # ENCRYPT ^ frame_dummy
    file_contents[encrypt_start_addr:encrypt_end_addr] =
    xor(file_contents[encrypt_start_addr:encrypt_end_addr],
    file_contents[fake_frame_dummy_addr:fake_frame_dummy_addr+encrypt_end_addr-
    encrypt_start_addr])

    with open(filename, 'wb') as f:
        f.write(file_contents)
```

Conclusion

By utilizing callee-saved registers from library functions, we were able to create an **invisible** jump that executes some shellcode discreetly.

While this is not any ground-breaking research, such research is fun and interesting as it gives us insights into how compilers and our reverse-engineering tools work and might even be useful by helping us understand and prepare for more complicated low-level assembly tricks that might be used in malware to achieve code obfuscation.

In my next obfuscation adventure, I hope to be able to do less manual work and directly instrument the assembly code on the compiler level, possibly through writing LLVM passes.

Hopefully this was a good read and see you in the next one!