

# Clang 7 CFI/SafeStack Analysis

Common existing exploit mitigations like ASLR, DEP and Stack Canaries are not enough to mitigate memory corruption vulnerabilities. But, if combined with a forward-edge restriction (Control Flow Integrity, CFI) and backward-edge restriction (SafeStack), are these mitigations finally good enough to prohibit exploitation of most memory corruption vulnerabilities in common programs? (spoiler: pretty much, yes).

While there are papers about CFI and SafeStack, and blog posts about their properties (e.g. trailofbits (3)), I could not find a resource detailing the implementation of these mechanisms. Which I find quite surprising, as it may change the exploitation of memory corruption landscape forever, if the compiler protection would be finally used by default. This paper analyzes the CFI/SafeStack implementation by reverse engineering the generated assembly code in the executables. The same view a exploit writer or reverse engineer has. I focus on C, as protecting C++ is better documented and has more special cases. These mitigations are not introduced specifically in Clang 7, but that's the version I tested.

The content of this paper is organized as follows:

- TL;DR of paper without any technical details
- Summary of the Clang protection mechanisms
- Analysis of Clang CFI, by reversing several C programs
- Analysis of Clang SafeStack, by reversing several C programs

## TL;DR

Clang CFI allows function pointers to only point to a range of function-stubs. This prohibits jumping to arbitrary addresses, which makes it impossible to do ROP. It may however allow calling unintended functions (e.g. something like `CreateBackgroundProcess(char *cmd)`).

Clang SafeStack creates a second stack for local variables of functions. All program-logic relevant metadata (like return addresses) are stored on the original stack. A stack based buffer overflow will only corrupt the second stack, where no return addresses are stored. Therefore it is not possible to change the program logic.

If used together, this seems to be a big step forward to make a large amount of vulnerabilities unexploitable.

*Note: Software which executes arbitrary untrusted code with concurrency (with this I mean primarily browsers with JavaScript, but also Flash and Java) have way other requirements for memory corruption mitigations. Browsers are a complete other beast to tame, and many CFI solutions may be unsuitable for Browsers, but not for normal programs like network servers or image converters. I ignore browsers for sake of this discussion.*

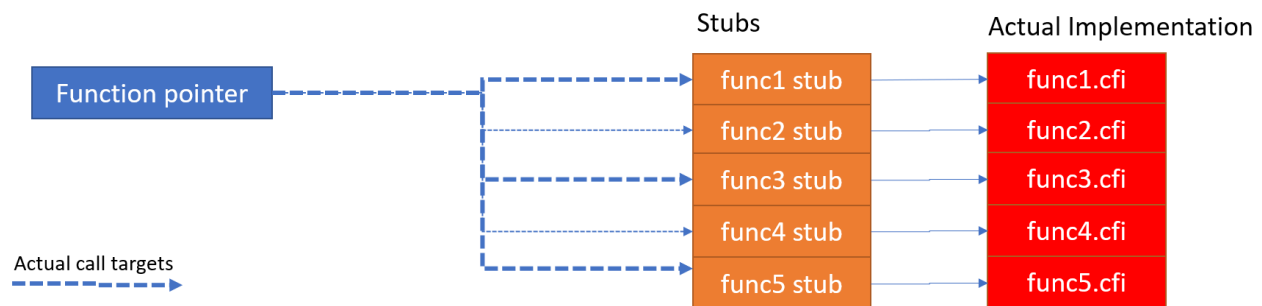
## Summary

### CLANG CFI

CFI implementation of Clang/LLVM for C is based primarily on cfi-icall protection.

- Instead of calling the destination function directly, a stub is called.
  - Function with name "*function*" will generate a stub function called "*function*", which in turn calls "*function.cfi*", where the actual function is stored
- Calling function pointers will include a range check
  - The range check is performed on the stub functions addresses
  - The range check includes all functions with the same function signature

In other words: A C function with the name "addition" will generate a call stub with the symbol "addition" (which we call `addition_stub`), which calls the real function with the symbol "addition.cfi". Clang CFI will perform range checks, to always check if a destination pointer belongs to a certain range of these stub functions. Depending on the complexity of the code, this can lead to additional (non intentional) functions are valid targets for an overwritten function pointer. Although they require to have the same function signature (return value, number of arguments and their type).



Additional possible call targets

`pointer >= &func1_stub && pointer <= func5_stub?`

This can lead to either:

- Pretty fine grained CFI
  - Only actual targets are valid targets.
- Pretty coarse grained CFI (if there are many functions with the same function signature)
  - Some functions can be called, which are not called within the code per se
- ROP is not possible, as stub area does not contain arbitrary code, only calls to the function implementation (and followed by int3 bytes to align to 8 bytes).

Actual implementation, based on *funcptr5.c* and the graphic above. *functionPointer* can either be *&func1*, *&func3*, or *&func5*, but not *&func2* and *&func4*. Note that there are 5 possible functions.

```
0x000000000040120c <+28>:  mov    0x404048,%rax    # rax is the functionPointer we wanna call
0x0000000000401214 <+36>:  movabs $0x401400,%rcx    # rcx is &func1_stub (base pointer)
0x000000000040121e <+46>:  mov     %rax,%rdx        # rdx is the functionPointer we wanna call (copy from
rax)
0x0000000000401221 <+49>:  sub     %rcx,%rdx        # rdx = rdx - rcx.
                                # Or in other words: rdx = functionPointer - &func1_stub
                                # Or in other words: the distance of &func1_stub to

functionPointer in bytes
0x0000000000401224 <+52>:  mov     %rdx,%rcx        # rcx = rdx
                                # rcx has now the distance between the FunctionPointer

we wanna call and the base
0x0000000000401227 <+55>:  shr     $0x3,%rcx        # rcx >> 3: divide memory distance by 8
                                # each stub function is 8 bytes. So we have the number

of "stubs" between these functions in rcx
0x000000000040122b <+59>:  shl     $0x3d,%rdx        # rdx << 0x3d ??? TODO
0x000000000040122f <+63>:  or      %rdx,%rcx        # rcx = rcx ^ rdx
0x0000000000401232 <+66>:  cmp     $0x4,%rcx        # check if rcx is <= 4: max addr of call target is
&func5
0x0000000000401236 <+70>:  jbe     0x40123a <bof+74>
0x0000000000401238 <+72>:  ud2                                # rcx >= 5. Crash here
0x000000000040123a <+74>:  callq   %rax             # rcx <= 4. Call the functionPointer, as it is "safe"
```

Or in pseudocode:

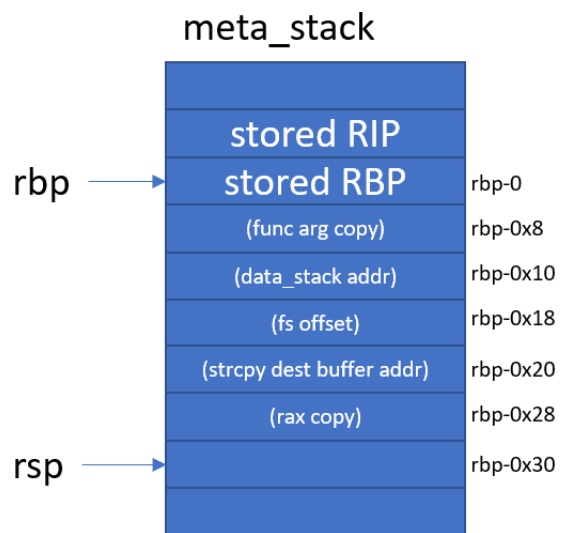
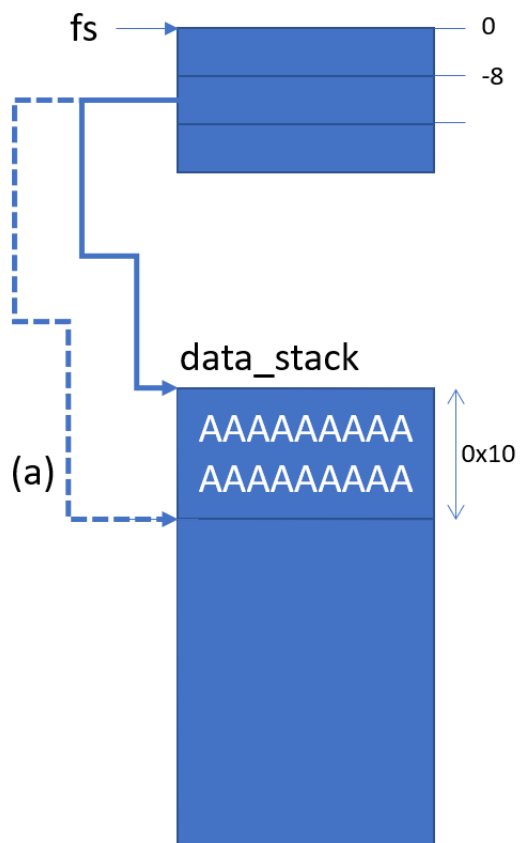
```
distance = (functionPointer - baseStubPointer) / 8
if distance >= 5:
    crash()
else:
    (*functionPointer)()
```

## SafeStack

Clang will continue using the standard stack for all function call convention based things. I call this stack *meta\_stack*, as it contain program logic metadata.

Additionally, there is a second "fake" stack, which i call *data\_stack* (called Safe Stack by the SafeStack developers). This stack address is retrieved via the *r* segment register, which points to an array of stack addresses. For example at *fs:-8*, there is a pointer to the current end of the *data\_stack* (grows towards lower memory addresses). All local variables of functions are stored in the *data\_stack*.

As no program logic relevant metadata ist stored on the *data\_stack*, a buffer overflow on the *data\_stack* can not corrupt metadata like stored instruction pointers (return addresses). All access to *meta\_stack* is compiler generated, and assumed to be safe.



```

(gdb) disas bof
Dump of assembler code for function bof:
# function prologue
0x000000000040fa20 <+0>:    push    %rbp
0x000000000040fa21 <+1>:    mov     %rsp,%rbp
0x000000000040fa24 <+4>:    sub     $0x40,%rsp

# SafeStack prologue
0x000000000040fa28 <+8>:    mov     0x95b1(%rip),%rax      # 0x418fe0
0x000000000040fa2f <+15>:   mov     %fs:(%rax),%rcx
0x000000000040fa33 <+19>:   mov     %rcx,%rdx
0x000000000040fa36 <+22>:   add     $0xfffffffffffffff0,%rdx
0x000000000040fa3a <+26>:   mov     %rdx,%fs:(%rax)

# rcx is now the base pointer to the base of the relevant data_stack
# rcx = dataStackBaseAddress

# actual function
[...]
0x000000000040fa52 <+50>:   mov     %rcx,%rdx
0x000000000040fa55 <+53>:   add     $0xfffffffffffffff8,%rdx # rdx = rdx - 8 (8 byte array)
[...]
0x000000000040fa5d <+61>:   mov     %rdx,%rdi              # use it as destination for strcpy()
[...]
# copy SafeStack relevant data to meta_stack (via rbp)
0x000000000040fa60 <+64>:   mov     %rcx,-0x18(%rbp)       # rbp-0x18 = dataStackBaseAddress
0x000000000040fa64 <+68>:   mov     %rax,-0x20(%rbp)       # rbp-0x20 = offset
[...]
0x000000000040fa6c <+76>:   callq   0x401040 <strcpy@plt>
[...]

# SafeStack epilogue
# recover SafeStack relevant data from meta_stack (via rbp)
# and use it to write the original dataStackBaseAddress (as seen on entry) via fs
0x000000000040fa90 <+112>:  mov     -0x20(%rbp),%rcx       # offset
0x000000000040fa94 <+116>:  mov     -0x18(%rbp),%rdx       # dataStackBaseAddress
0x000000000040fa98 <+120>:  mov     %rdx,%fs:(%rcx)
[...]

# function epilogue
0x000000000040fa9f <+127>:  add     $0x40,%rsp
0x000000000040faa3 <+131>:  pop     %rbp
0x000000000040faa4 <+132>:  retq
End of assembler dump.

```

Or, in pseudocode:

```

# SafeStack prologue
offset = *(rip + 0x95b1)           // -8
dataStackBaseAddress = fs[offset] // get base of "our" data stack
newDataStackBottom = dataStackBaseAddress - 0x10 // make some space in it (expand stack to lower address)
fs[offset] = newDataStackBottom    // store new base

# actual function
localBufferVar = dataStackBaseAddress - 8 // prepare some space in the data stack. 8 is size of
BufferVar
strcpy(localBufferVar, ...)        // Use data stack for local variable purposes

# SafeStack epilogue
fs[offset] = dataStackBaseAddress   // restore original offset (move stack up to the previous
address)

```

Noteworthy is that the instruction "%fs:(%rax),%rcx" actually accesses the Thread Local Storage (TLS). Therefore, each thread has its own data\_stack.

# Stuff

Newer versions of this document may be available on: <https://github.com/dobin/clang-cfi-safestack-analysis>

There is an experimental checksec update checking for these things: <https://github.com/dobin/checksec.sh>

## Status on Ubuntu

Ubuntu 16.04 uses: Clang 3.8

- CFI requires LLVMgold
- Safestack: Available

Ubuntu 18.04: uses: Clang 7.0.0

- CFI available
- Safestack available

## Status on GCC

GCC has neither CFI or SafeStack.

## Nginx Calltargets Analysis

Compiling Nginx 1.15.9 with CFI, I asked myself how many call targets a function pointer usually has. I objdump'd the code, grep'd for the "cmp" instruction (range check of allowed functions) and plotted its immediate value:

```
$ objdump -d objs/nginx > nginx.objdump
$ egrep "callq.*\*\%" nginx.objdump -B 7 | grep cmp | awk '{print $7}' | cut -d"," -f1 | sed 's/\$/ /' > nginx.cmpnum

$ python hexcount.py nginx.cmpnum
Count:    404
Sum:      5495
Average:  13

# cat nginx.cmpnum.decimal | sort -V | uniq -c | sort -nr
128 13
56 3
49 2
24 46
21 9
17 5
17 1
15 4
11 7
11 25
10 26
8 29
6 8
6 45
5 36
4 35
3 30
3 0
2 83
2 111
1 82
1 6
1 28
1 24
1 10

# count|number of valid targets
```

While the most common, and also the mean value is 13 functions, some pointer can reach up to 111 functions (only two instances though). There are 24 instances of range checks which cover 46 functions though.

# Clang CFI Analysis

To reverse engineer the CFI implementation of Clang, I created several example programs and analyzed them.

Note: In GDB/ASM source, reversing comments are indicated by "#". In C source code by "//".

I created and analyzed several programs, to analyze various aspects of the CFI implementation:

- Funcptr2: stack based buffer overflow into a function pointer (which has only one target)
- Funcptr3: Array of function pointers, with out of bound dereferencing
- Funcptr4: A function pointer which can have 5 targets
- Funcptr5: Two function pointers, mixing 3 of 5 possible target functions
- Funcptr7: Two function pointers, with different function signature (void vs. int)

Note that while we enable all CFI functions with "-fsanitize=cfi", we focus on ical-protection with "-fsanitize=cfi-icall", which protects indirect calls.

## Funcptr2 - BoF into Pointer

A simple stack based buffer overflow into a function pointer, which gets called later.

Lets get familiar with the source:

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

void func(void) {
    printf("Yay\n");
}

void bof(char *a) {
    void (*f)(void) = &func; // This pointer can be overwritten
    char buffer[8];

    strcpy(buffer, a);
    (*f)();
}

int main(int argc, char **argv) {
    printf("A: %s\n", argv[1]);
    bof(argv[1]);

    return 0;
}
```

Lets compile it without CFI and have a look. With this, we are able to spot the changes to the code CFI makes.

```
(gdb) disas bof
Dump of assembler code for function bof:
0x000000000401170 <+0>:      push    %rbp
0x000000000401171 <+1>:      mov     %rsp,%rbp
0x000000000401174 <+4>:      sub     $0x20,%rsp
0x000000000401178 <+8>:      lea     -0x18(%rbp),%rax
0x00000000040117c <+12>:     mov     %rdi,-0x8(%rbp)
0x000000000401180 <+16>:     movabs  $0x401140,%rdi
0x00000000040118a <+26>:     mov     %rdi,-0x10(%rbp)
0x00000000040118e <+30>:     mov     -0x8(%rbp),%rsi
0x000000000401192 <+34>:     mov     %rax,%rdi
0x000000000401195 <+37>:     callq  0x401030 <strcpy@plt>

0x00000000040119a <+42>:     mov     %rax,-0x20(%rbp)
0x00000000040119e <+46>:     callq  *-0x10(%rbp)

0x0000000004011a1 <+49>:     add     $0x20,%rsp
0x0000000004011a5 <+53>:     pop     %rbp
0x0000000004011a6 <+54>:     retq

End of assembler dump.
```

And now lets compile it with CFI, and analyze the code. I stopped execution on bof+61:

```
$ clang funcptr2.c -fsanitize=cfi -flto -fvisibility=hidden -o funcptr2 && ./funcptr2
```

```
(gdb) r AAAAAAAAAAAAAAAAAAAAAA
(gdb) disas bof
Dump of assembler code for function bof:
0x000000000401160 <+0>:      push    %rbp
0x000000000401161 <+1>:      mov     %rsp,%rbp
0x000000000401164 <+4>:      sub     $0x20,%rsp
0x000000000401168 <+8>:      lea     -0x18(%rbp),%rax
0x00000000040116c <+12>:     mov     %rdi,-0x10(%rbp)
0x000000000401170 <+16>:     movabs  $0x401200,%rcx
0x00000000040117a <+26>:     mov     %rcx,-0x8(%rbp)
0x00000000040117e <+30>:     mov     -0x10(%rbp),%rsi
0x000000000401182 <+34>:     mov     %rax,%rdi
0x000000000401185 <+37>:     callq  0x401030 <strcpy@plt>

# check if destination addr is 0x401200
0x00000000040118a <+42>:     mov     -0x8(%rbp),%rax
0x00000000040118e <+46>:     movabs  $0x401200,%rcx
0x000000000401198 <+56>:     cmp     %rcx,%rax
0x00000000040119b <+59>:     je      0x40119f <bof+63>
=> 0x00000000040119d <+61>:     ud2
0x00000000040119f <+63>:     callq  %rax

0x0000000004011a1 <+65>:     add     $0x20,%rsp
0x0000000004011a5 <+69>:     pop     %rbp
0x0000000004011a6 <+70>:     retq

End of assembler dump.

(gdb) i r
rax            0x4141414141414141  4702111234474983745    # this
rbx            0x0                0
rcx            0x401200           4198912                 # this
rdx            0x41              65
rsi            0x7fffffff800      140737488349184
rdi            0x7fffffff490      140737488348304
rbp            0x7fffffff490      0x7fffffff490
rsp            0x7fffffff470      0x7fffffff470
r8             0x0                0
r9             0xffffffff         4294967295
r10            0x3                3
r11            0x7ffff7f80a60     140737353615968
r12            0x401050           4198480
```

```

r13      0x7fffffff590      140737488348560
r14      0x0                0
r15      0x0                0
rip      0x40119d          0x40119d <bof+61>
eflags   0x10206           [ PF IF RF ]
cs       0x33              51
ss       0x2b              43
ds       0x0               0
es       0x0               0
fs       0x0               0
gs       0x0               0

(gdb) disas 0x401200
Dump of assembler code for function func:
   0x0000000000401200 <+0>:    jmpq    0x401140 <func.cfi>
   0x0000000000401205 <+5>:    int3
   0x0000000000401206 <+6>:    int3
   0x0000000000401207 <+7>:    int3

(gdb) disas 0x401140
Dump of assembler code for function func.cfi:
   0x0000000000401140 <+0>:    push    %rbp
   0x0000000000401141 <+1>:    mov     %rsp,%rbp
   0x0000000000401144 <+4>:    movabs  $0x402004,%rdi
   0x000000000040114e <+14>:   mov     $0x0,%al
   0x0000000000401150 <+16>:   callq   0x401040 <printf@plt>
   0x0000000000401155 <+21>:   pop     %rbp
   0x0000000000401156 <+22>:   retq
End of assembler dump.

```

The CFI code at bof+42 checks if the destination address of the function pointer *void (\*)(void)* is 0x401200. The code at 0x401200 is just a call to a stub, which will then call the real *void func1(void)*.

Relevant code logic walkthrough:

ASM	valid ("BBBB")	invalid ("AAAAAAAAAAAAAAAA")
mov -0x8(%rbp),%rax	rax = 0x401200	rax = 0x4141414141414141
movabs \$0x401200,%rcx	rcx = 0x401200	rcx = 0x401200
cmp %rcx,%rax	rcx = rax? Yes	rcx = rax? No



## Funcptr3 - Function Pointer Array Out Of Bounds

We have an array of function pointers, indexed by a command line argument. We can try to access a function pointer at an index which is greater than array size (e.g. 2). There will be "random" data at this location for now, but this is not relevant for the reversing effort.

Source:



```

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

void func1(void) {
    printf("Yay1\n");
}

void func2(void) {
    printf("Yay2\n");
}

void bof(int idx) {
    void (*fa[2])(void);
    void (*f)(void);
    char buffer[8];

    fa[0] = &func1;
    fa[1] = &func2;

    f = fa[idx]; // idx can be >= 2, but array has only 2 elements

    (*f)();
}

int main(int argc, char **argv) {
    printf("A: %s\n", argv[1]);
    bof(atoi(argv[1]));

    return 0;
}

```

Without CFI:

```

(gdb) disas bof
Dump of assembler code for function bof:
0x00000000004011a0 <+0>:      push    %rbp
0x00000000004011a1 <+1>:      mov     %rsp,%rbp
0x00000000004011a4 <+4>:      sub     $0x30,%rsp

# populate array
0x00000000004011a8 <+8>:      mov     %edi,-0x4(%rbp)
0x00000000004011ab <+11>:     movabs  $0x401140,%rax
0x00000000004011b5 <+21>:     mov     %rax,-0x20(%rbp)      # rbp-0x20 = 0x401140 -> fa[0] = &func1;
0x00000000004011b9 <+25>:     movabs  $0x401170,%rax
0x00000000004011c3 <+35>:     mov     %rax,-0x18(%rbp)     # rbp-0x18 = 0x401170 -> fa[1] = &func2;

# load pointer to call from array
0x00000000004011c7 <+39>:     movslq  -0x4(%rbp),%rax      # argument in rax (e.g. int 0, 1, 2..) (via
arg, ebp-0x4, 32 bit)
0x00000000004011cb <+43>:     mov     -0x20(%rbp,%rax,8),%rax # load "rbp-0x20 + (rax * 8)" to rax =
destination addr
0x00000000004011d0 <+48>:     mov     %rax,-0x28(%rbp)     # mov destination addr to stack rbp-0x28
0x00000000004011d4 <+52>:     callq  *-0x28(%rbp)         # call destination addr on stack rbp-0x28

# rax:          0x1
# (gdb) x/1xg $rbp-0x20
# 0x7fffffffef470: 0x0000000000401140
#
# Memory layout of array: two 64 bit pointers
# 0x7fffffffef470: 0x00401140      0x00000000      0x00401170      0x00000000
#                  func1          func2

0x00000000004011d7 <+55>:     add     $0x30,%rsp
0x00000000004011db <+59>:     pop     %rbp
0x00000000004011dc <+60>:     retq

End of assembler dump.
(gdb)

```

With CFI:

```

(gdb) disas bof
Dump of assembler code for function bof:
0x0000000000401180 <+0>:      push    %rbp
0x0000000000401181 <+1>:      mov     %rsp,%rbp
0x0000000000401184 <+4>:      sub     $0x30,%rsp

# load ptr
0x0000000000401188 <+8>:      mov     %edi,-0x4(%rbp)
0x000000000040118b <+11>:     movabs  $0x401240,%rax
0x0000000000401195 <+21>:     mov     %rax,-0x20(%rbp)
0x0000000000401199 <+25>:     movabs  $0x401248,%rax
0x00000000004011a3 <+35>:     mov     %rax,-0x18(%rbp)
0x00000000004011a7 <+39>:     movslq  -0x4(%rbp),%rax
0x00000000004011ab <+43>:     mov     -0x20(%rbp,%rax,8),%rax
0x00000000004011b0 <+48>:     mov     %rax,-0x10(%rbp) # store pointer to func.cfi stub in $rbp-0x10

# (gdb) x/1x $rbp-0x10
# 0x7fffffff490: 0x00401248

# 0x0000000000401240 <+0>:      jmpq    0x401140 <func1.cfi>
# 0x0000000000401245 <+5>:      int3
# 0x0000000000401246 <+6>:      int3
# 0x0000000000401247 <+7>:      int3
# 0x0000000000401248 <+0>:      jmpq    0x401160 <func2.cfi>
# 0x000000000040124d <+5>:      int3
# 0x000000000040124e <+6>:      int3
# 0x000000000040124f <+7>:      int3

# CFI
# Load base pointer of array, and destination pointer
0x00000000004011b4 <+52>:     mov     -0x10(%rbp),%rax # rax = 0x401248 (func2.cfi)
0x00000000004011b8 <+56>:     movabs  $0x401240,%rcx # rcx = 0x401240 (func1.cfi)
0x00000000004011c2 <+66>:     mov     %rax,%rdx # rdx = 0x401248 (func2.cfi)
0x00000000004011c5 <+69>:     sub     %rcx,%rdx # rdx = rdx - rcx // rdx = 8 -> distance

# check if distance is valid (less than 1)
0x00000000004011c8 <+72>:     mov     %rdx,%rcx # rcx = rdx // rdx = 8
0x00000000004011cb <+75>:     shr     $0x3,%rcx # rcx >> 3 // rcx = 1
0x00000000004011cf <+79>:     shl     $0x3d,%rdx # rdx << 0x3d = 61 // rdx = 0
0x00000000004011d3 <+83>:     or      %rdx,%rcx # rcx = rcx OR rdx // rcx = 1
0x00000000004011d6 <+86>:     cmp     $0x1,%rcx # rcx <= 1?
0x00000000004011da <+90>:     jbe     0x4011de <bof+94>
0x00000000004011dc <+92>:     ud2 # rcx > 1!
0x00000000004011de <+94>:     callq   *%rax # rcx <= 1, call ptr in rax

0x00000000004011e0 <+96>:     add     $0x30,%rsp
0x00000000004011e4 <+100>:    pop     %rbp
0x00000000004011e5 <+101>:    retq

End of assembler dump.

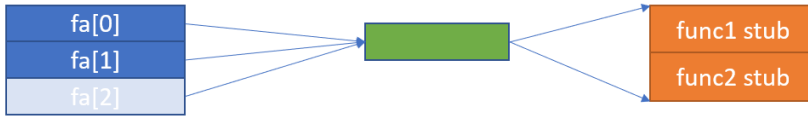
```

Relevant code logic walkthrough:

ASM	valid (1)	invalid (2)
mov -0x10(%rbp),%rax	rax = 0x401248	rax = 0
movabs \$0x401240,%rcx	rcx = 0x401240	rcx = 0x401240
mov %rax,%rdx	rdx = 0x401248	rdx = 0
sub %rcx,%rdx	rdx = 8	rdx = 0xffffffffbfedc0
mov %rdx,%rcx	rcx = 8	rcx = 0xffffffffbfedc0
shr \$0x3,%rcx	rcx = 1	rcx = 0xffffffff7fdb8
shl \$0x3d,%rdx	rdx = 0	rdx = 0

or %rdx,%rcx	rcx = 1	rcx = 0x1ffffffff7fdb8
--------------	---------	------------------------

Checks if addresses is not more than 1 element away from base of array (array bound check)



$ptr > \&func1\_stub \ \&\& \ ptr \leq func2\_stub?$

## Optimization

With CFI and -O2, lets see if the code is equivalent, or if we can spot implementation mistakes.

```
(gdb) disas main
Dump of assembler code for function main:
0x0000000000401170 <+0>:      push    %rbx
0x0000000000401171 <+1>:      sub     $0x10,%rsp
0x0000000000401175 <+5>:      mov     %rsi,%rbx
0x0000000000401178 <+8>:      mov     0x8(%rsi),%rsi
0x000000000040117c <+12>:     mov     $0x40200e,%edi
0x0000000000401181 <+17>:     xor     %eax,%eax
0x0000000000401183 <+19>:     callq   0x401040 <printf@plt>
0x0000000000401188 <+24>:     mov     0x8(%rbx),%rdi
0x000000000040118c <+28>:     xor     %esi,%esi
0x000000000040118e <+30>:     mov     $0xa,%edx
0x0000000000401193 <+35>:     callq   0x401050 <strtol@plt>
0x0000000000401198 <+40>:     mov     $0x4011e8,%ecx
0x000000000040119d <+45>:     movq    %rcx,%xmm0
0x00000000004011a2 <+50>:     mov     $0x4011e0,%ecx
0x00000000004011a7 <+55>:     movq    %rcx,%xmm1
0x00000000004011ac <+60>:     punpcklqdq %xmm0,%xmm1
0x00000000004011b0 <+64>:     movdqa  %xmm1, (%rsp)
0x00000000004011b5 <+69>:     cltq

# CFI
0x00000000004011b7 <+71>:     mov     (%rsp,%rax,8),%rax    # load ptr into rax
0x00000000004011bb <+75>:     mov     $0x4011e0,%ecx      # base
0x00000000004011c0 <+80>:     mov     %rax,%rdx
0x00000000004011c3 <+83>:     sub     %rcx,%rdx
0x00000000004011c6 <+86>:     rol     $0x3d,%rdx
0x00000000004011ca <+90>:     cmp     $0x2,%rdx           # compare length here?
0x00000000004011ce <+94>:     jae     0x4011da <main+106>
0x00000000004011d0 <+96>:     callq   *%rax

0x00000000004011d2 <+98>:     xor     %eax,%eax
0x00000000004011d4 <+100>:    add     $0x10,%rsp
0x00000000004011d8 <+104>:    pop     %rbx
0x00000000004011d9 <+105>:    retq
0x00000000004011da <+106>:    ud2
```

Relevant code logic walkthrough:

	valid (1)	invalid (2)
mov (%rsp,%rax,8),%rax	rax = 0x4011e8 (func2.cfi)	rax = 0
mov \$0x4011e0,%ecx	ecx = 0x4011e0	ecx = 0x4011e0
mov %rax,%rdx	rdx = 0x4011e8	rdx = 0
sub %rcx,%rdx	rdx = 8	rdx = 0xffffffffbfee20
rol \$0x3d,%rdx	rdx = 1	rdx = 0x1ffffffff7fdc4
cmp \$0x2,%rdx	rdx >= 2? No	rdx >= 2? Yes

The code is basically equivalent to the nonoptimized version.

## About shl 0x3d

A piece of code we didnt touch until now is:

```
0x00000000004011cf <+79>:    shl     $0x3d,%rdx          # rdx << 0x3d      // rdx = 0,
```

the complete code:

```
# CFI
# Load base pointer of array, and destination pointer
0x00000000004011b4 <+52>:    mov     -0x10(%rbp),%rax      # rax = 0x401248 (func2.cfi)
0x00000000004011b8 <+56>:    movabs $0x401240,%rcx      # rcx = 0x401240 (func1.cfi)
0x00000000004011c2 <+66>:    mov     %rax,%rdx          # rdx = 0x401248 (func2.cfi)
0x00000000004011c5 <+69>:    sub     %rcx,%rdx          # rdx = rdx - rcx    // rdx = 8 -> distance

# check if distance is valid (less than 1)
0x00000000004011c8 <+72>:    mov     %rdx,%rcx          # rcx = rdx          // rdx = 8
0x00000000004011cb <+75>:    shr     $0x3,%rcx          # rcx >> 3          // rcx = 1
0x00000000004011cf <+79>:    shl     $0x3d,%rdx          # rdx << 0x3d      // rdx = 0,
                                # because 8 << 61 = 0x10000000000000000 in 65 bit
                                # = 0x00000000000000000 in 64 bit

0x00000000004011d3 <+83>:    or      %rdx,%rcx          # rcx = rcx OR rdx  // rcx = 1
0x00000000004011d6 <+86>:    cmp     $0x1,%rcx          # rcx <= 1?
0x00000000004011da <+90>:    jbe     0x4011de <bof+94>
0x00000000004011dc <+92>:    ud2                          # rcx > 1!
0x00000000004011de <+94>:    callq   *%rax              # rcx <= 1, call ptr in rax
```

If the distance is not a multiple of 8 = 1000b, the comparison (*o*) at bof+83 and bof+86 will fail.

```
7 << 61 = 0xE000000000000000 in 64 bit
```

The CFI function stubs are 8 bytes apart, so this prohibits jumping to the int 3's.

```
# 0x0000000000401240 <+0>:    jmpq    0x401140 <func1.cfi>
# 0x0000000000401245 <+5>:    int3
# 0x0000000000401246 <+6>:    int3
# 0x0000000000401247 <+7>:    int3
# 0x0000000000401248 <+0>:    jmpq    0x401160 <func2.cfi>
# 0x000000000040124d <+5>:    int3
# 0x000000000040124e <+6>:    int3
# 0x000000000040124f <+7>:    int3
```

## Funcptr 4 - Many call targets

Create code where a function pointer can have 5 different call targets.

```

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

void func1(void) {
    printf("Yay1\n");
}
void func2(void) {
    printf("Yay2\n");
}
void func3(void) {
    printf("Yay4\n");
}
void func4(void) {
    printf("Yay4\n");
}
void func5(void) {
    printf("Yay5\n");
}

void (*f)(void);

void bof(char *a) {
    char buffer[8];

    strcpy(buffer, a); // Note: No relevant buffer overflow here, just an artefact
    (*f)();
}

void init(int x) {
    switch(x) {
        case 1: f = &func1;
                break;
        case 2: f = &func2;
                break;
        case 3: f = &func3;
                break;
        case 4: f = &func4;
                break;
        case 5: f = &func5;
                break;
        default: f = NULL;
    }
}

int main(int argc, char **argv) {
    if (argc != 3) {
        printf("Usage: %s <integer> <string>", argv[0]);
        return 1;
    }

    printf("A: %s\n", argv[2]);
    init(atoi(argv[1]));
    bof(argv[2]);

    return 0;
}

```

Analysis of the CFI:

```

(gdb) r 3 asdf
...
(gdb) disas bof
Dump of assembler code for function bof:
0x0000000004011f0 <+0>:      push    %rbp
0x0000000004011f1 <+1>:      mov     %rsp,%rbp
0x0000000004011f4 <+4>:      sub     $0x10,%rsp
0x0000000004011f8 <+8>:      lea     -0x10(%rbp),%rax
0x0000000004011fc <+12>:     mov     %rdi,-0x8(%rbp)
0x000000000401200 <+16>:     mov     -0x8(%rbp),%rsi
0x000000000401204 <+20>:     mov     %rax,%rdi
0x000000000401207 <+23>:     callq  0x401030 <strcpy@plt>

# CFI check
0x00000000040120c <+28>:     mov     0x404048,%rax    # rax = *0x404048 = 0x401380 (global f = &func3)
0x000000000401214 <+36>:     movabs  $0x401370,%rcx  # rcx = 0x401370 (base, &func1)
0x00000000040121e <+46>:     mov     %rax,%rdx      # rdx = 0x401380 (function pointer in global f)
0x000000000401221 <+49>:     sub     %rcx,%rdx      # rdx = rdx - rcx
0x000000000401224 <+52>:     mov     %rdx,%rcx      # rcx = rdx
0x000000000401227 <+55>:     shr     $0x3,%rcx
0x00000000040122b <+59>:     shl     $0x3d,%rdx
0x00000000040122f <+63>:     or      %rdx,%rcx
0x000000000401232 <+66>:     cmp     $0x4,%rcx      # array of size 5, max element 4
0x000000000401236 <+70>:     jbe     0x40123a <bof+74>
0x000000000401238 <+72>:     ud2
0x00000000040123a <+74>:     callq  *%rax

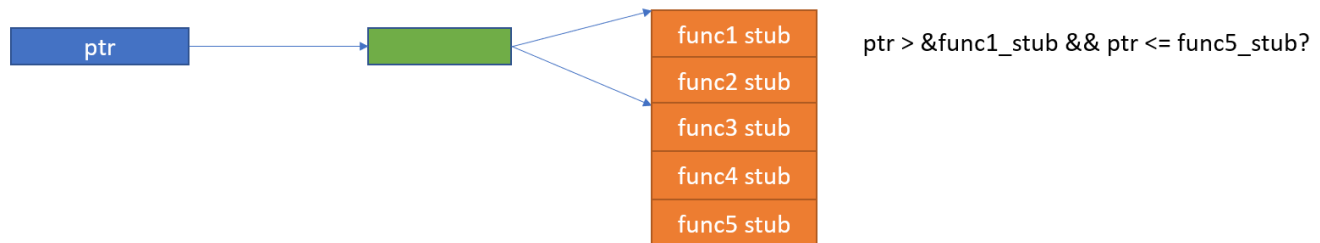
0x00000000040123c <+76>:     add     $0x10,%rsp
0x000000000401240 <+80>:     pop     %rbp
0x000000000401241 <+81>:     retq

End of assembler dump.

(gdb) x/lxg 0x404048
0x404048 <f>:      0x000000000401380

```

Basically, checks if function pointer is between some valid memory addresses (between &func1 and &func5)



## Funcptr 5 - Intermixed call targets

Create code where a function pointer can have 3 out of 5 functions as targets, and another one which can have the other 2 functions as targets, and also one which is callable by the first function pointer.

Relevant code:

```

void (*fa)(void);
void (*fb)(void);

void bof(char *a) {
    char buffer[8];

    strcpy(buffer, a);
    (*fa)();
    (*fb)();
}

void init(int x, int y) {
    switch(x) {
        case 1: fa = &func1;
                break;
        case 2: fa = &func3;
                break;
        case 3: fa = &func5;
                break;
        default: fa = NULL;
    }
    switch(y) {
        case 1: fb = &func2;
                break;
        case 2: fb = &func4;
                break;
        case 3: fb = &func5;
                break;
        default: fb = NULL;
    }
}

```

Note:

- *fa* can point to func1, func3, func5
- *fb* can point to func2, func4, func5

Disassembly:



```

(gdb) disas bof
Dump of assembler code for function bof:
0x00000000004011f0 <+0>:      push    %rbp
0x00000000004011f1 <+1>:      mov     %rsp,%rbp
0x00000000004011f4 <+4>:      sub     $0x10,%rsp
0x00000000004011f8 <+8>:      lea     -0x10(%rbp),%rax
0x00000000004011fc <+12>:     mov     %rdi,-0x8(%rbp)
0x0000000000401200 <+16>:     mov     -0x8(%rbp),%rsi
0x0000000000401204 <+20>:     mov     %rax,%rdi
0x0000000000401207 <+23>:     callq  0x401030 <strcpy@plt>

# CFI for fa
0x000000000040120c <+28>:     mov     0x404048,%rax
0x0000000000401214 <+36>:     movabs  $0x401400,%rcx    # &func1
0x000000000040121e <+46>:     mov     %rax,%rdx
0x0000000000401221 <+49>:     sub     %rcx,%rdx
0x0000000000401224 <+52>:     mov     %rdx,%rcx
0x0000000000401227 <+55>:     shr     $0x3,%rcx
0x000000000040122b <+59>:     shl     $0x3d,%rdx
0x000000000040122f <+63>:     or      %rdx,%rcx
0x0000000000401232 <+66>:     cmp     $0x4,%rcx        # 4+1=5 elements
0x0000000000401236 <+70>:     jbe     0x40123a <bof+74>
0x0000000000401238 <+72>:     ud2
0x000000000040123a <+74>:     callq  *%rax

# CFI for fb
0x000000000040123c <+76>:     mov     0x404050,%rax
0x0000000000401244 <+84>:     movabs  $0x401400,%rcx    # &func1
0x000000000040124e <+94>:     mov     %rax,%rdx
0x0000000000401251 <+97>:     sub     %rcx,%rdx
0x0000000000401254 <+100>:    mov     %rdx,%rcx
0x0000000000401257 <+103>:    shr     $0x3,%rcx
0x000000000040125b <+107>:    shl     $0x3d,%rdx
0x000000000040125f <+111>:    or      %rdx,%rcx
0x0000000000401262 <+114>:    cmp     $0x4,%rcx        # 4+1=5 elements
0x0000000000401266 <+118>:    jbe     0x40126a <bof+122>
0x0000000000401268 <+120>:    ud2
0x000000000040126a <+122>:    callq  *%rax

0x000000000040126c <+124>:    add     $0x10,%rsp
0x0000000000401270 <+128>:    pop     %rbp
0x0000000000401271 <+129>:    retq

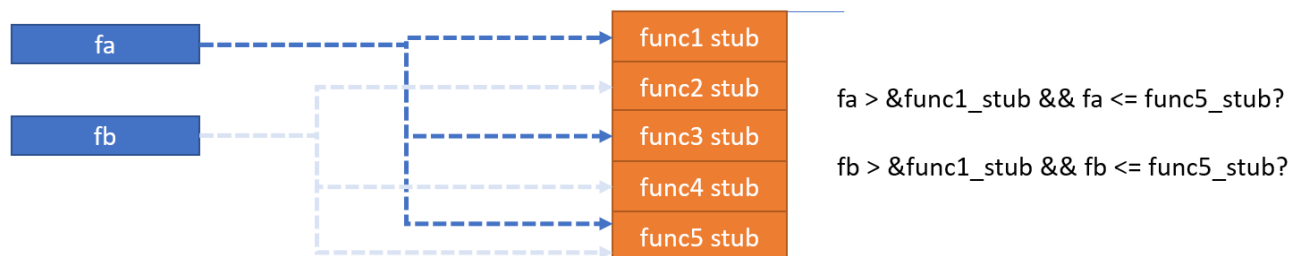
End of assembler dump.

(gdb) print &func1
$1 = (<text variable, no debug info> *) 0x401400 <func1>

```

CFI restricts *fa* and *fb* to all 5 functions (func1, func2, func3, func4, func5). Even though they only point to a subset of it (coarse grained CFI).

Square dot lines indicate call targets:



Note that the range check is quite broad. Changing fb to only calls either func2 or func3 in the code (and no call to func4), we get the following:

```
switch(y) {
    case 1: fb = &func2;
            break;
    case 2: fb = &func3;
            break;
    default: fb = NULL;
}
```

And the disassembly shows:

```
0x00000000004011ec <+28>:    mov     0x404048,%rax
0x00000000004011f4 <+36>:    movabs $0x4013c0,%rcx    # func1
0x00000000004011fe <+46>:    mov     %rax,%rdx
0x0000000000401201 <+49>:    sub     %rcx,%rdx
0x0000000000401204 <+52>:    mov     %rdx,%rcx
0x0000000000401207 <+55>:    shr     $0x3,%rcx
0x000000000040120b <+59>:    shl     $0x3d,%rdx      # changed from 4+1 to 3+1 elements
0x000000000040120f <+63>:    or      %rdx,%rcx
0x0000000000401212 <+66>:    cmp     $0x3,%rcx
0x0000000000401216 <+70>:    jbe     0x40121a <bof+74>
0x0000000000401218 <+72>:    ud2
0x000000000040121a <+74>:    callq   *%rax
0x000000000040121c <+76>:    mov     0x404050,%rax
0x0000000000401224 <+84>:    movabs $0x4013c0,%rcx    # same base, func1!
0x000000000040122e <+94>:    mov     %rax,%rdx
0x0000000000401231 <+97>:    sub     %rcx,%rdx
0x0000000000401234 <+100>:   mov     %rdx,%rcx
0x0000000000401237 <+103>:   shr     $0x3,%rcx
0x000000000040123b <+107>:   shl     $0x3d,%rdx
0x000000000040123f <+111>:   or      %rdx,%rcx
0x0000000000401242 <+114>:   cmp     $0x3,%rcx      # changed from 4+1 to 3+1 elements
0x0000000000401246 <+118>:   jbe     0x40124a <bof+122>
0x0000000000401248 <+120>:   ud2
0x000000000040124a <+122>:   callq   *%rax
```

The check got a bit more restrictive, but just because the amount of functions was reduced by one (no call to func4).

## Funcptr7 - Function Signature Bucketing

It appears that the range checks always cover all functions. I assume it is only for the functions with the same function signature (return value, argument count and type). Lets change the function type of the functions called by fb to int. So there are three functions "func\_v1-3" (v for void), called by fa, which take no arguments. And two functions "func\_i1-2" (i for int), called by fb, which take one int argument:

```

void func_v1(void) {
    printf("Yay v1\n");
}
void func_v2(void) {
    printf("Yay v2\n");
}
void func_v3(void) {
    printf("Yay v4\n");
}
void func_i1(int a) { // int argument
    printf("Yay i1 %i\n", a);
}
void func_i2(int a) { // int argument
    printf("Yay i2 %i\n", a);
}

void (*fa)(void);
void (*fb)(int); // int argument

void bof(void) {
    (*fa)();
    (*fb)(2);
}

void init(int x, int y) {
    switch(x) {
        case 1: fa = &func_v1;
                break;
        case 2: fa = &func_v2;
                break;
        case 3: fa = &func_v3;
                break;
        default: fa = NULL;
    }
    switch(y) {
        case 1: fb = &func_i1;
                break;
        case 2: fb = &func_i2;
                break;
        default: fb = NULL;
    }
}

int main(int argc, char **argv) {
    if (argc != 3) {
        printf("Usage: %s <integer> <integer>", argv[0]);
        return 1;
    }

    init(atoi(argv[1]), atoi(argv[2]));
    bof();

    return 0;
}

```

Lets have a look at the CFI check:

```

(gdb) disas bof
Dump of assembler code for function bof:
   0x0000000000401200 <+0>:      push    %rbp
   0x0000000000401201 <+1>:      mov     %rsp,%rbp

   # CFI for fa (void)
   0x0000000000401204 <+4>:      mov     0x404040,%rax
   0x000000000040120c <+12>:     movabs  $0x4013b0,%rcx    # &func_v1
   0x0000000000401216 <+22>:     mov     %rax,%rdx
   0x0000000000401219 <+25>:     sub     %rcx,%rdx
   0x000000000040121c <+28>:     mov     %rdx,%rcx
   0x000000000040121f <+31>:     shr     $0x3,%rcx          # 3 functions
   0x0000000000401223 <+35>:     shl     $0x3d,%rdx
   0x0000000000401227 <+39>:     or      %rdx,%rcx
   0x000000000040122a <+42>:     cmp     $0x2,%rcx
   0x000000000040122e <+46>:     jbe     0x401232 <bof+50>
   0x0000000000401230 <+48>:     ud2
   0x0000000000401232 <+50>:     callq   *%rax

   # CFI for fb (int)
   0x0000000000401234 <+52>:     mov     0x404048,%rax
   0x000000000040123c <+60>:     movabs  $0x4013d0,%rcx    # &func_i1
   0x0000000000401246 <+70>:     mov     %rax,%rdx
   0x0000000000401249 <+73>:     sub     %rcx,%rdx
   0x000000000040124c <+76>:     mov     %rdx,%rcx
   0x000000000040124f <+79>:     shr     $0x3,%rcx
   0x0000000000401253 <+83>:     shl     $0x3d,%rdx
   0x0000000000401257 <+87>:     or      %rdx,%rcx
   0x000000000040125a <+90>:     cmp     $0x1,%rcx
   0x000000000040125e <+94>:     jbe     0x401262 <bof+98>
   0x0000000000401260 <+96>:     ud2
   0x0000000000401262 <+98>:     mov     $0x2,%edi        # 2 functions
   0x0000000000401267 <+103>:    callq   *%rax

   0x0000000000401269 <+105>:    pop     %rbp
   0x000000000040126a <+106>:    retq

End of assembler dump.

```

```

(gdb) disas 0x4013b0
Dump of assembler code for function func_v1:
   0x00000000004013b0 <+0>:      jmpq     0x401140 <func_v1.cfi>
   0x00000000004013b5 <+5>:      int3
   0x00000000004013b6 <+6>:      int3
   0x00000000004013b7 <+7>:      int3
   0x00000000004013b8 <+0>:      jmpq     0x401160 <func_v2.cfi>
   0x00000000004013bd <+5>:      int3
   0x00000000004013be <+6>:      int3
   0x00000000004013bf <+7>:      int3
   0x00000000004013c0 <+0>:      jmpq     0x401180 <func_v3.cfi>
   0x00000000004013c5 <+5>:      int3
   0x00000000004013c6 <+6>:      int3
   0x00000000004013c7 <+7>:      int3

```

End of assembler dump.

```

(gdb) disas 0x4013d0
Dump of assembler code for function func_i1:
   0x00000000004013d0 <+0>:      jmpq     0x4011a0 <func_i1.cfi>
   0x00000000004013d5 <+5>:      int3
   0x00000000004013d6 <+6>:      int3
   0x00000000004013d7 <+7>:      int3
   0x00000000004013d8 <+0>:      jmpq     0x4011d0 <func_i2.cfi>
   0x00000000004013dd <+5>:      int3
   0x00000000004013de <+6>:      int3
   0x00000000004013df <+7>:      int3

```

End of assembler dump.

The conclusion is, that Clang CFI checks if the destination function pointer belongs to a function with the same function signature. All functions with the same signature are valid targets.

# Safestack

## retbof.c

Source:

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

void bof(char *a) {
    char buf[8];

    strcpy(buf, a);
    printf("buf: %s\n", buf);
}

int main(int argc, char **argv) {
    bof(argv[1]);

    return 0;
}
```

Without SafeStack:

```
root@ubuntu-1804:~/cfi# clang retbof.c -o retbof-plain && ./retbof-plain AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
buf: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
```

With SafeStack:

```
root@ubuntu-1804:~/cfi# clang retbof.c -fsanitize=safe-stack -o retbof && ./retbof
AAAA
buf: AAAA
root@ubuntu-1804:~/cfi# clang retbof.c -fsanitize=safe-stack -o retbof && ./retbof
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
buf: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
root@ubuntu-1804:~/cfi#
```

There is no crash, even though it appears that the complete buffer has been written to the stack. But it appears that the saved instruction pointer was not overwritten.

Assembly source, without SafeStack:

```

(gdb) disas main
Dump of assembler code for function main:
0x000000000401190 <+0>:      push    %rbp
0x000000000401191 <+1>:      mov     %rsp,%rbp
0x000000000401194 <+4>:      sub     $0x10,%rsp
0x000000000401198 <+8>:      movl    $0x0,-0x4(%rbp)
0x00000000040119f <+15>:     mov     %edi,-0x8(%rbp)
0x0000000004011a2 <+18>:     mov     %rsi,-0x10(%rbp)
0x0000000004011a6 <+22>:     mov     -0x10(%rbp),%rsi
0x0000000004011aa <+26>:     mov     0x8(%rsi),%rdi
0x0000000004011ae <+30>:     callq   0x401140 <bof>
0x0000000004011b3 <+35>:     xor     %eax,%eax
0x0000000004011b5 <+37>:     add     $0x10,%rsp
0x0000000004011b9 <+41>:     pop     %rbp
0x0000000004011ba <+42>:     retq
End of assembler dump.

(gdb) disas bof
Dump of assembler code for function bof:
0x000000000401140 <+0>:      push    %rbp
0x000000000401141 <+1>:      mov     %rsp,%rbp
0x000000000401144 <+4>:      sub     $0x30,%rsp

0x000000000401148 <+8>:      lea     -0x10(%rbp),%rax
0x00000000040114c <+12>:     mov     %rdi,-0x8(%rbp)
0x000000000401150 <+16>:     mov     -0x8(%rbp),%rsi      # rsi = source (char *a)
0x000000000401154 <+20>:     mov     %rax,%rdi           # rdi = destination (stack)
0x000000000401157 <+23>:     mov     %rax,-0x18(%rbp)
=> 0x00000000040115b <+27>:     callq   0x401030 <strcpy@plt>

0x000000000401160 <+32>:     movabs  $0x402004,%rdi
0x00000000040116a <+42>:     mov     -0x18(%rbp),%rsi
0x00000000040116e <+46>:     mov     %rax,-0x20(%rbp)
0x000000000401172 <+50>:     mov     $0x0,%al
0x000000000401174 <+52>:     callq   0x401040 <printf@plt>

0x000000000401179 <+57>:     mov     %eax,-0x24(%rbp)
0x00000000040117c <+60>:     add     $0x30,%rsp
0x000000000401180 <+64>:     pop     %rbp
0x000000000401181 <+65>:     retq
End of assembler dump.

(gdb) i r rdi rsi
rdi             0x7fffffff470      140737488348272
rsi             0x7fffffff7e6      140737488349158
(gdb) x/1s $rsi
0x7fffffff7e6: "AAAA"
(gdb) x/1xg $rbp-0x8
0x7fffffff478: 0x00007fffffff7e6
(gdb) x/1s 0x00007fffffff7e6
0x7fffffff7e6: "AAAA"

```

Assembly source, with SafeStack:

# main() with safestack is identical to main() without safestack.

# No changes in caller required

(gdb) disas main

Dump of assembler code for function main:

```
0x000000000040faa0 <+0>:    push    %rbp
0x000000000040faa1 <+1>:    mov     %rsp,%rbp
0x000000000040faa4 <+4>:    sub     $0x10,%rsp
0x000000000040faa8 <+8>:    movl    $0x0,-0x4(%rbp)
0x000000000040faaf <+15>:   mov     %edi,-0x8(%rbp)
0x000000000040fab2 <+18>:   mov     %rsi,-0x10(%rbp)
0x000000000040fab6 <+22>:   mov     -0x10(%rbp),%rsi
0x000000000040faba <+26>:   mov     0x8(%rsi),%rdi
0x000000000040fabe <+30>:   callq   0x40fa20 <bof>
0x000000000040fac3 <+35>:   xor     %eax,%eax
0x000000000040fac5 <+37>:   add     $0x10,%rsp
0x000000000040fac9 <+41>:   pop     %rbp
0x000000000040faca <+42>:   retq
```

(gdb) disas

Dump of assembler code for function bof:

# standard prologue

```
0x000000000040fa20 <+0>:    push    %rbp
0x000000000040fa21 <+1>:    mov     %rsp,%rbp
0x000000000040fa24 <+4>:    sub     $0x30,%rsp
```

# safestack: prologue

# get value from fs segment, decrement by 0x10, and store it again

```
0x000000000040fa28 <+8>:    mov     0x95b1(%rip),%rax    # rax = -8 = 0xfffffffffffffffff8
0x000000000040fa2f <+15>:   mov     %fs:(%rax),%rcx      # rcx = 0x7ffff7c1a000 = *fs:ra
0x000000000040fa33 <+19>:   mov     %rcx,%rdx           # rdx = 0x7ffff7c1a000
0x000000000040fa36 <+22>:   add     $0xfffffffffffffff0,%rdx # rdx = 0x7ffff7c19ff0 // rdx -= 0x10
0x000000000040fa3a <+26>:   mov     %rdx,%fs:(%rax)      # fs:ra = 0x7ffff7c19ff0
```

# rcx is now base pointer to data\_stack

# strcpy() part

```
0x000000000040fa3e <+30>:   mov     %rdi,-0x8(%rbp)      # rdi is argument of this function, char *a
0x000000000040fa42 <+34>:   mov     %rcx,%rdx           # rdx = rcx // rdx =
&data_stack
0x000000000040fa45 <+37>:   add     $0xfffffffffffffff8,%rdx # rdx -= 8 // rdx -= 8
0x000000000040fa49 <+41>:   mov     -0x8(%rbp),%rsi      # rsi = source // argument argv[1]
0x000000000040fa4d <+45>:   mov     %rdx,%rdi           # rdi = rdx = destination // &data_stack-8
0x000000000040fa50 <+48>:   mov     %rcx,-0x10(%rbp)
0x000000000040fa54 <+52>:   mov     %rax,-0x18(%rbp)
0x000000000040fa58 <+56>:   mov     %rdx,-0x20(%rbp)
0x000000000040fa5c <+60>:   callq   0x401040 <strcpy@plt> # rdi = destination = 0x7ffff7c19ff8
```

# printf() part

```
0x000000000040fa61 <+65>:   mov     $0x4142e0,%r8d
0x000000000040fa67 <+71>:   mov     %r8d,%edi
0x000000000040fa6a <+74>:   xor     %r8d,%r8d
0x000000000040fa6d <+77>:   mov     %r8b,%r9b
0x000000000040fa70 <+80>:   mov     -0x20(%rbp),%rsi
0x000000000040fa74 <+84>:   mov     %rax,-0x28(%rbp)
0x000000000040fa78 <+88>:   mov     %r9b,%al
0x000000000040fa7b <+91>:   callq   0x401050 <printf@plt>
```

# safestack: epilogue

```
0x000000000040fa80 <+96>:   mov     -0x18(%rbp),%rcx    # rcx = -8
0x000000000040fa84 <+100>:  mov     -0x10(%rbp),%rdx    # rdx = 0x7ffff7c1a000
0x000000000040fa88 <+104>:  mov     %rdx,%fs:(%rcx)     # fs:rcx = 0x7ffff7c1a000
```

# standard return value

```
0x000000000040fa8c <+108>:  mov     %eax,-0x2c(%rbp)    # eax = 0xa == 10
```

# standard epilogue

```
0x000000000040fa8f <+111>:  add     $0x30,%rsp
0x000000000040fa93 <+115>:  pop     %rbp
0x000000000040fa94 <+116>:  retq                                     # take return address from stack, eip = *rsp
```

End of assembler dump.

Additional information, from runtime:

```
# 0x95b1(%rip)
(gdb) x/lgx 0x418fe0
0x418fe0:      0xfffffffffffffffff8

(gdb) info proc mapping
process 54086
Mapped address spaces:

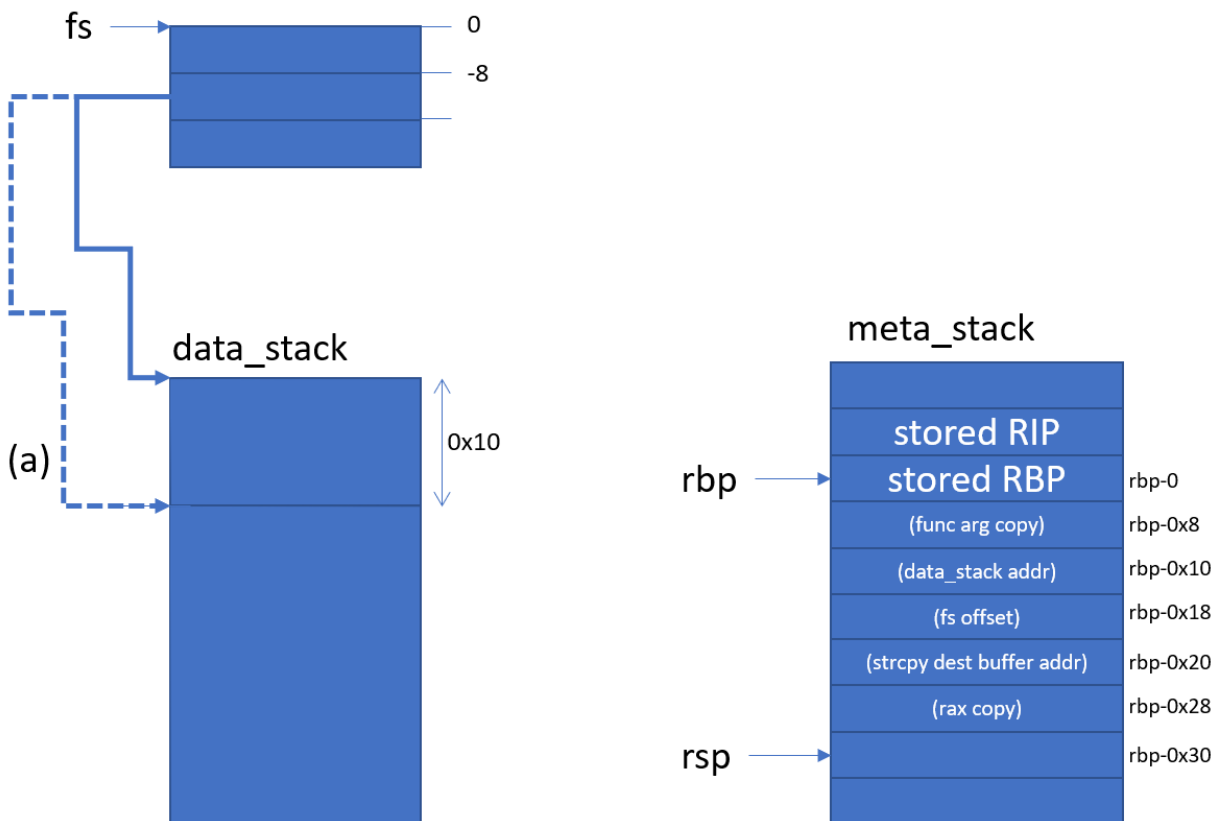
      Start Addr      End Addr      Size      Offset objfile
      0x400000      0x401000      0x1000      0x0 /root/cfi/retbof
      0x401000      0x410000      0xf000      0x1000 /root/cfi/retbof
      0x410000      0x418000      0x8000      0x10000 /root/cfi/retbof
      0x418000      0x419000      0x1000      0x17000 /root/cfi/retbof
$rip+0x95b1 = 0x418fe0 is here
      0x419000      0x41c000      0x3000      0x18000 /root/cfi/retbof
      0x41c000      0x4ac000      0x90000      0x0 [heap]
0x7ffff7419000      0x7ffff741a000      0x1000      0x0
0x7ffff741a000      0x7ffff7c1c000      0x802000      0x0 # data stack
0x7ffff7c1c000      0x7ffff7c3e000      0x22000      0x0 /lib/x86_64-linux-gnu/libc-2.28.so
0x7ffff7c3e000      0x7ffff7daf000      0x171000      0x22000 /lib/x86_64-linux-gnu/libc-2.28.so
0x7ffff7daf000      0x7ffff7dfb000      0x4c000      0x193000 /lib/x86_64-linux-gnu/libc-2.28.so
0x7ffff7dfb000      0x7ffff7dfc000      0x1000      0x1df000 /lib/x86_64-linux-gnu/libc-2.28.so
0x7ffff7dfc000      0x7ffff7e00000      0x4000      0x1df000 /lib/x86_64-linux-gnu/libc-2.28.so
0x7ffff7e00000      0x7ffff7e02000      0x2000      0x1e3000 /lib/x86_64-linux-gnu/libc-2.28.so
0x7ffff7e02000      0x7ffff7e06000      0x4000      0x0
0x7ffff7e06000      0x7ffff7e07000      0x1000      0x0 /lib/x86_64-linux-gnu/libdl-2.28.so
0x7ffff7e07000      0x7ffff7e09000      0x2000      0x1000 /lib/x86_64-linux-gnu/libdl-2.28.so
0x7ffff7e09000      0x7ffff7e0a000      0x1000      0x3000 /lib/x86_64-linux-gnu/libdl-2.28.so
0x7ffff7e0a000      0x7ffff7e0b000      0x1000      0x3000 /lib/x86_64-linux-gnu/libdl-2.28.so
0x7ffff7e0b000      0x7ffff7e0c000      0x1000      0x4000 /lib/x86_64-linux-gnu/libdl-2.28.so
0x7ffff7e0c000      0x7ffff7e19000      0xd000      0x0 /lib/x86_64-linux-gnu/libm-2.28.so
0x7ffff7e19000      0x7ffff7ec4000      0xab000      0xd000 /lib/x86_64-linux-gnu/libm-2.28.so
0x7ffff7ec4000      0x7ffff7f97000      0xd3000      0xb8000 /lib/x86_64-linux-gnu/libm-2.28.so
0x7ffff7f97000      0x7ffff7f98000      0x1000      0x18a000 /lib/x86_64-linux-gnu/libm-2.28.so
0x7ffff7f98000      0x7ffff7f99000      0x1000      0x18b000 /lib/x86_64-linux-gnu/libm-2.28.so
0x7ffff7f99000      0x7ffff7f9b000      0x2000      0x0 /lib/x86_64-linux-gnu/librt-2.28.so
0x7ffff7f9b000      0x7ffff7f9f000      0x4000      0x2000 /lib/x86_64-linux-gnu/librt-2.28.so
0x7ffff7f9f000      0x7ffff7fa1000      0x2000      0x6000 /lib/x86_64-linux-gnu/librt-2.28.so
0x7ffff7fa1000      0x7ffff7fa2000      0x1000      0x7000 /lib/x86_64-linux-gnu/librt-2.28.so
0x7ffff7fa2000      0x7ffff7fa3000      0x1000      0x8000 /lib/x86_64-linux-gnu/librt-2.28.so
0x7ffff7fa3000      0x7ffff7fa9000      0x6000      0x0 /lib/x86_64-linux-gnu/libpthread-2.28.so
0x7ffff7fa9000      0x7ffff7fb8000      0xf000      0x6000 /lib/x86_64-linux-gnu/libpthread-2.28.so
0x7ffff7fb8000      0x7ffff7fbe000      0x6000      0x15000 /lib/x86_64-linux-gnu/libpthread-2.28.so
0x7ffff7fbe000      0x7ffff7fbf000      0x1000      0x1a000 /lib/x86_64-linux-gnu/libpthread-2.28.so
0x7ffff7fbf000      0x7ffff7fc0000      0x1000      0x1b000 /lib/x86_64-linux-gnu/libpthread-2.28.so
0x7ffff7fc0000      0x7ffff7fc6000      0x6000      0x0
0x7ffff7fc6000      0x7ffff7fd1000      0x3000      0x0 [vvar]
0x7ffff7fd1000      0x7ffff7fd3000      0x2000      0x0 [vdso]
0x7ffff7fd3000      0x7ffff7fd4000      0x1000      0x0 /lib/x86_64-linux-gnu/ld-2.28.so
0x7ffff7fd4000      0x7ffff7ff4000      0x20000      0x1000 /lib/x86_64-linux-gnu/ld-2.28.so
0x7ffff7ff4000      0x7ffff7ffc000      0x8000      0x21000 /lib/x86_64-linux-gnu/ld-2.28.so
0x7ffff7ffc000      0x7ffff7ffd000      0x1000      0x28000 /lib/x86_64-linux-gnu/ld-2.28.so
0x7ffff7ffd000      0x7ffff7ffe000      0x1000      0x29000 /lib/x86_64-linux-gnu/ld-2.28.so
0x7ffff7ffe000      0x7ffff7fff000      0x1000      0x0
0x7ffff7fff000      0x7ffff7fff000      0x21000      0x0 [stack]
normal, "metadata" stack
0xfffffffff600000 0xfffffffff601000      0x1000      0x0 [vsyscall]
```

Step by step:

- Function prologue is standard (like without SafeStack)
- SafeStack prologue:
  - Will get a pointer from fs:-8. This pointer will be called data\_stack\_funcbase.
  - Decrease the pointer by 0x10. This is similar as allocating 0x10 bytes.
  - Store that decremented pointer (a) again at fs:-8.



- Presumably the value at fs:-8 is also used for other functions. It always points to the top of the data\_stack (lower addresses), with usable space
- strcpy() part:
  - It will use data\_stack\_funcbase as argument to strcpy()
    - strcpy() is therefore not able to modify data in the meta\_stack
  - It will use meta\_stack to temporarily store registers on the stack
    - Note that this are mov's. No adjecant data can be overwritten
    - Especially data\_stack\_funcbase and the fs offset "-8" is stored at location rbp-0x10 and 0x18 respectively
- SafeStack epilogue:
  - Restore the pointer at fs:-8 from (a) to its initial value. This basically "free's" the allocated memory of the stack.
    - The values (offset -8, and data\_stack\_funcbase) are retrieved from meta\_stack
- Function epilogue is standard (like without SafeStack)



## References

CFI:

- <https://clang.llvm.org/docs/ControlFlowIntegrity.html>
- <https://blog.trailofbits.com/2016/10/17/lets-talk-about-cfi-clang-edition/>
- [https://patrickfunke.de/wp-content/uploads/CFI\\_2016.pdf](https://patrickfunke.de/wp-content/uploads/CFI_2016.pdf) (Chapter 4, IFCC: Indirect Function-Call Checks)
  - With is mostly identical to <https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-tice.pdf>

SafeStack:

- <https://clang.llvm.org/docs/SafeStack.html>
- <https://www.blackhat.com/docs/eu-16/Goktas-Bypassing-Clangs-SafeStack.pdf>
- <http://blog.includesecurity.com/2015/11/LLVM-SafeStack-buffer-overflowprotection.html>