```
    } while ((uint64_t)uVar15 < CONCAT44(in_register_0000000c, arg4));
  } else {
    iVar13 = CONCAT44(in_register_0000000c, arg4) * uVar12;
    uVar7 = arg1 + iVar13 * 8;
    puVar10 = (uint64_t *)((uVar12 * 8 + 8) * CONCAT44(in_register_0000000c, arg4) + arg1);
    if (puVar5 < puVar10 && uVar7 < CONCAT44(in_register_0000000c, arg4) * 8 * uVar12 + arg2 + 1) {
      uVar15 = 0;
      uVar8 = 0;
      goto code_r0x004015c0;
    }
    uVar15 = 0;
    uVar8 = 0;
    if (uVar7 < (uint64_t)(CONCAT44(in_register_00000014, arg3) + CONCAT44(in_register_0000000c, arg4) * 8)
        && ((uint64_t *)CONCAT44(in_register_00000014, arg3) ≤ puVar10 &&
           puVar10 ≠ (uint64_t *)CONCAT44(in_register_00000014, arg3))) goto code_r0x004015c0;
    if (uVar2 == 0) {
      iVar13 = 0;
    } else {
      iVar16 = arg1 + 0xe0 + iVar13 * 8;
      auVar17 = vbroadcastsd_avx(*puVar5);
      iVar13 = 0;
      uVar7 = uVar14 & 0xfffffffffffffffe;
      do {
        auVar18 = vmulpd_avx(auVar17, *(undefined (*) [32])
                             (CONCAT44(in_register_00000014, arg3) + iVar13 * 8));
        auVar19 = vmulpd_avx(auVar17, *(undefined (*) [32])
                             (CONCAT44(in_register_00000014, arg3) + 0x20 + iVar13 * 8));
        auVar20 = vmulpd_avx(auVar17, *(undefined (*) [32])
                             (CONCAT44(in_register_00000014, arg3) + 0x40 + iVar13 * 8));
auVar21 = vmulpd_avx(auVar17, *(undefined (*) [32])
                             (CONCAT44(in_register_00000014, arg3) + 0x60 + iVar13 * 8));
auVar18 = vmovupd_avx(auVar18);
*(undefined (*) [32])(iVar16 + -0xe0 + iVar13 * 8) = auVar18;
auVar18 = vmovupd_avx(auVar19);
*(undefined (*) [32])(iVar16 + -0xc0 + iVar13 * 8) = auVar18;
auVar18 = vmovupd_avx(auVar20);
*(undefined (*) [32])(iVar16 + -0xa0 + iVar13 * 8) = auVar18;
auVar18 = vmovupd_avx(auVar21);
*(undefined (*) [32])(iVar16 + -0x80 + iVar13 * 8) = auVar18;
auVar18 = vmulpd_avx(auVar17, *(undefined (*) [32])
                             (CONCAT44(in_register_00000014, arg3) + 0x80 + iVar13 * 8));
auVar19 = vmulpd_avx(auVar17, *(undefined (*) [32])
                             (CONCAT44(in_register_00000014, arg3) + 0xa0 + iVar13 * 8));
auVar20 = vmulpd_avx(auVar17, *(undefined (*) [32])
```

Reverse engineering and some hacking

Introduction to executable binaries

Antonin Portelli

ExaLAT

Supported by UK EPSRC grants EP/V001329/1, EP/V001191/1, EP/V001477/1, EP/V001248/1, EP/V001140/1

- Reverse Engineering (RE)
- Introduction to code injection hacks
- Wrap up

Reverse Engineering (RE)

RE overview

- Software reverse engineering consists in **reconstructing a software's source code** from the binary only

- This is **very non-trivial** because close to no high-level structure survives the compilation process

- Some applications:

1. **Offensive security**: understand and circumvent security measures in computer systems
2. **Defensive security**: understand and protect against offensive softwares (malware, viruses, …)
3. **Open-source development**: develop open-source variants of proprietary software (nouveau driver, Samba protocol, …)

Connection to HPC?

- Not direct, mainly for your own curiosity, however…

- The RE community provides a lot of high-quality materials on binary analysis very useful for low-level development

  cf. for example https://bit.ly/3st1C6Y

- RE tools are quite powerful to just browse and understand an executable binary

- In an HPC context “reverse-engineering” your own programs can be quite instructive regarding the optimisations the compiler made

Radare2/Rizin

- **Open-source**, 15 years old community based **RE toolset**

- Powerful **command-line swiss-army knife** to navigate and understand disassembled binaries

- Radare2 & Rizin are (for the moment) **essentially the same software**…
  … a lot of politics beyond the recent separation

Ghidra

- **NSA RE framework**, first mentioned in a 2017 WikiLeaks publication, then **declassified** in 2019

- Now **open-source** and freely available

- Provide many tools only available in > $100 commercial RE software, in particular a **powerful decompiler**

Cutter

- Best of both worlds: **GUI for Rizin** and packs the **Ghidra decompiler**

- Multi-platform, ambitious with very active development
- Provide a **graph view of assembly code**, very useful to follow jumps, loops, conditionals, …
- Provide **visual representations** of many other aspects of the binary (data, segments, sections, …)
- Let’s try!

Introduction to code injection hacks

What is code injection

- Code injection consists in constraining an executable binary to execute external arbitrary code at run time

- It can be use to “hook” functions, i.e. to wrap-up and existing function in additional code, for example for instrumentation purposes

- It can be used to add features to a closed-source program

- It can be used to build exploits: force a program to reveal sensitive memory content, circumvent a protection…

Hooking dynamically linked functions

- By far the easiest

- If you set the environment variable **`LD_PRELOAD`** to the filename of a shared library, this library will be dynamically linked when a program run before any other

- If this library contains a symbol called in the program, it will override the original dynamical link

- In short you can replace any function that comes from a shared library by whatever you want!

- Of course the system will ignore it if the executable has any form of privilege, that would be a big vulnerability

Basic example: code

- Main program

```c
int main(void)
{
    func(15431);
    return EXIT_SUCCESS;
}
```

- Target function dynamically linked to the main program

```c
void func(const int id)
{
    printf("calling original with id %d\n", id);
}
```

- Replacement function in another library

```c
void func(const int id)
{
    printf("calling variant with id %d\n", id);
}
```

Basic example: execution

- Use `LD_PRELOAD` to replace function

```
$ clang -g -fPIC -shared func1.c -o libfunc1.so
$ clang -g callfunc.c -o callfunc -lfunc1 -L.
$ ./callfunc
./callfunc: error while loading shared libraries: libfunc1.so: cannot open shared object file: No such file or directory
$ export LD_LIBRARY_PATH=$(pwd)
$ ./callfunc
calling original with id 15431
$ LD_PRELOAD=./libfunc2.so ./callfunc
calling variant with id 15431
```

- But what if we want to **still execute the original function**? e.g. just to wrap code around it

Basic example: function wrapper

```c
// RTLD_NEXT is a GNU extension
#define _GNU_SOURCE

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

void func(const int id)
{
    // static pointer to the target
    static void (*target_func)(const int) = NULL;

    if (!target_func)
    {
        // if the target pointer is NULL, load it using dlsym from libdl
        target_func = dlsym(RTLD_NEXT, "func");
        printf("Target loaded at %p\n", target_func);
    }

    // wrapper
    printf("Do some things before func call...\n");
    target_func(id);
    printf("Do some things after func call...\n");
}
```

- libdl (POSIX) allows to get pointer on dynamic symbols

Basic example: function wrapper

- The wrapper in action

```
$ clang -g -fPIC -shared funchook.c -o libfunchook.so -ldl
$ LD_PRELOAD=./libfunchook.so ./callfunc
Target loaded at 0×7f7014cc9110
Do some things before func call...
calling original with id 15431
Do some things after func call...
```

- This trick can be extremely useful to **track, debug or instrument calls to functions**

Example: tracing memory allocations

- Let's track **`malloc`** and **`free`** calls to produce statistics for memory allocations

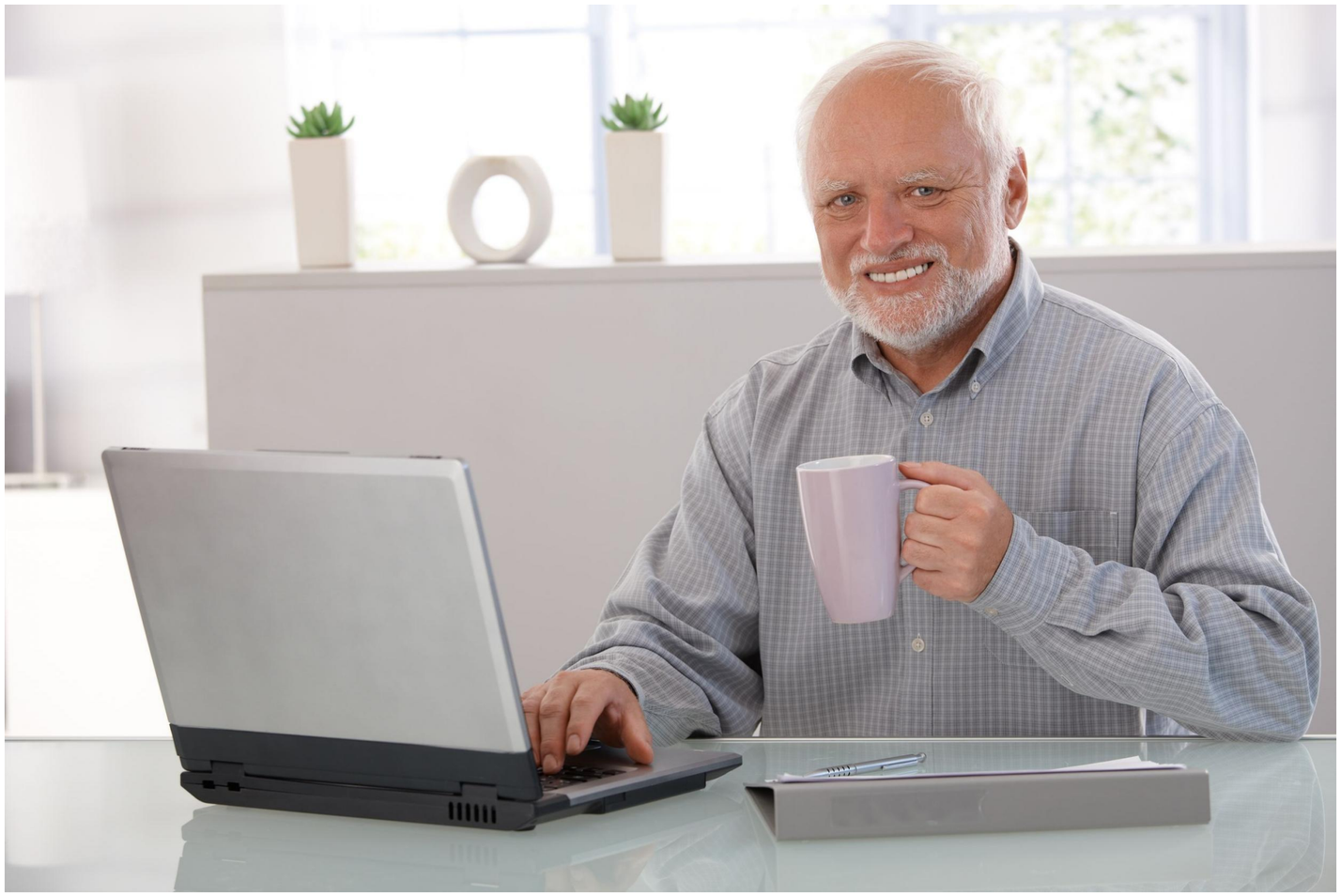
- Wrapper

```c
static size_t peak = 0, current = 0, cumul = 0;

void *malloc(size_t size)
{
    static void *(*real_malloc)(size_t) = NULL;

    if (!real_malloc)
    {
        real_malloc = dlsym(RTLD_NEXT, "malloc");
        printf("Target function malloc loaded at %p\n", real_malloc);
    }

    void   *p         = real_malloc(size);
    size_t real_size = malloc_usable_size(p);

    current += real_size;
    cumul   += real_size;
    peak     = (current > peak) ? current : peak;
    printf("==== MEM DEBUG ==== Allocated %zdB at %p (current %zdB, peak %zdB, cumulative %zdB)\n",
           real_size, p, current, peak, cumul);

    return p;
}

```

Example: tracing memory allocations

- Ok let's see the allocations in the `ls` command…

```
$ LD_PRELOAD=./libmemhookv1.so ls
[1]    55639 segmentation fault (core dumped)  LD_PRELOAD=./libmemhookv1.so ls --color=tty
```



Example: tracing memory allocations

- The issue is that `printf` calls `malloc`, and now we put a `printf` call *inside* `malloc`, so **infinite recursion**

- We need to protect the `printf` call

```
static int in_hook = 0;
if (!in_hook)
{
    size_t real_size = malloc_usable_size(p);

    in_hook  = 1;
    current += real_size;
    cumul   += real_size;
    peak     = (current > peak) ? current : peak;
    printf("==== MEM DEBUG ==== Allocated %zdB at %p (current %zdB, peak %zdB, cumulative %zdB)\n",
           real_size, p, current, peak, cumul);
    in_hook  = 0;
}
```

Example: tracing memory allocations

- It works!

```
$ LD_PRELOAD=./libmemhookv2.so ls
==== MEM DEBUG ==== Allocated 1032B at 0×55da077342d0 (current 1032B, peak 1032B, cumulative 1032B)
Target function free loaded at 0×7fa93569e850
==== MEM DEBUG ==== Freed 1032B at 0×55da077346e0 (current 0B, peak 1032B, cumulative 1032B)
Target function malloc loaded at 0×7fa93569e260
==== MEM DEBUG ==== Allocated 472B at 0×55da07734af0 (current 472B, peak 1032B, cumulative 1504B)
==== MEM DEBUG ==== Allocated 120B at 0×55da07734cd0 (current 592B, peak 1032B, cumulative 1624B)
==== MEM DEBUG ==== Allocated 1032B at 0×55da077346e0 (current 1624B, peak 1624B, cumulative 2656B)
==== MEM DEBUG ==== Freed 120B at 0×55da07734cd0 (current 1504B, peak 1624B, cumulative 2656B)
==== MEM DEBUG ==== Freed 1032B at 0×55da077346e0 (current 472B, peak 1624B, cumulative 2656B)
==== MEM DEBUG ==== Freed 472B at 0×55da07734af0 (current 0B, peak 1624B, cumulative 2656B)
...
```

- You can iterate on that to make it a **full memory profiler**
- For example give a **backtrace** for every call
- For C++ you can also change `new` and `delete`

Other applications

- You can hook performance critical routines to make performance measurements

- You can hook MPI functions in an HPC code to see all network transfers and measure their speed

- You can hook CUDA library functions to trace the interaction with a GPU

- You can hook any buggy function if you can’t run your program interactively in a debugger

- etc…

Warnings about potential issues

- In C++ be careful about name mangling when using `libdl`
- HPC applications are often multi-threaded, be careful about the reentrancy of the wrapper
- The memory tracker would not work for a multi-threaded application. Do you see why? Can you improve it?
- What about statically linked functions?

Hooking statically linked functions

- This is much harder: the target function code is part of the main program binary and its calls are **hardcoded jumps**

- You can `LD_PRELOAD` a replacement, but it will just be ignored as calls don't go through the dynamic linker

- You could imagine "de-linking" the binary, and then statically link it again with the replacement function

- De-linking is non-trivial because of relocations, but it is not impossible and practised in RE

- But here we'll do something else…

Static function hack: strategy

- Remember: when a program run **its code is loaded somewhere in memory** (in the text segment)

- We can do the following

    1. **Load a replacement** with `LD_PRELOAD`
    2. Find the address of the target function, and **overwrite its first instructions with a jump** to the replacement

- Step 2. will need to be executed **before `main`** starts, and the jump will have to be **coded directly in binary**

Static function hack: strategy

Normal execution

```
main:

    ; some code

    call func

    ; more code


func:
    push rbp
    mov  rbp, rsp
    sub  rsp, 0x10

    ; function code

    ret
```

```
mov r10, new_func
jmp r10
```

```
ctor:

    ; hack the function
    ; in main program


new_func:

    ; replacement code

    ret
```

Static function hack: strategy

Normal execution

```
main:

    ; some code

    call func

    ; more code


func:
    push rbp
    mov  rbp, rsp
    sub  rsp, 0x10

    ; function code

    ret
```

```
mov r10, new_func
jmp r10
```

```
ctor:

    ; hack the function
    ; in main program


new_func:

    ; replacement code

    ret
```

Static function hack: strategy

Normal execution

```
main:

    ; some code

    call func

    ; more code


func:
    push rbp
    mov  rbp, rsp
    sub  rsp, 0x10

    ; function code

    ret
```

```
mov r10, new_func
jmp r10
```

```
ctor:

    ; hack the function
    ; in main program


new_func:

    ; replacement code

    ret
```

Static function hack: strategy

Normal execution

```
main:

    ; some code

    call func

    ; more code


func:
    push rbp
    mov  rbp, rsp
    sub  rsp, 0x10

    ; function code

    ret
```

```
mov r10, new_func
jmp r10
```

```
ctor:

    ; hack the function
    ; in main program


new_func:

    ; replacement code

    ret
```

Static function hack: strategy

Modified execution

```
main:

    ; some code

    call func

    ; more code

func:
    push rbp
    mov  rbp, rsp
    sub  rsp, 0x10

    ; function code

    ret
```

```
mov r10, new_func
jmp r10
```

```
ctor:

    ; hack the function
    ; in main program


new_func:

    ; replacement code

    ret
```

Static function hack: strategy

```
main:

    ; some code

    call func

    ; more code


func:
    push rbp
    mov  rbp, rsp
    sub  rsp, 0x10

    ; function code


    ret
```

```
mov r10, new_func
jmp r10
```

Modified execution

Before `main`

```
ctor:

    ; hack the function
    ; in main program


new_func:

    ; replacement code

    ret
```

Static function hack: strategy

Modified execution

```
main:

    ; some code

    call func

    ; more code

func:
    push rbp
    mov  rbp, rsp
    sub  rsp, 0x10

    ; function code

    ret
```

```
mov r10, new_func
jmp r10
```

```
ctor:

    ; hack the function
    ; in main program


new_func:

    ; replacement code

    ret
```

Static function hack: strategy

Modified execution

```
main:

    ; some code

    call func

    ; more code


func:
    push rbp
    mov  rbp, rsp
    sub  rsp, 0x10

    ; function code


    ret
```

```
mov r10, new_func
jmp r10
```

```
ctor:

    ; hack the function
    ; in main program


new_func:

    ; replacement code

    ret
```

Static function hack: strategy

Modified execution

```
main:

    ; some code

    call func

    ; more code


func:
    push rbp
    mov  rbp, rsp
    sub  rsp, 0x10

    ; function code


    ret
```

```
mov r10, new_func
jmp r10
```

```
ctor:

    ; hack the function
    ; in main program


new_func:

    ; replacement code


    ret
```

Static function hack: strategy

Modified execution

```
main:

    ; some code

    call func

    ; more code

func:
    push rbp
    mov  rbp, rsp
    sub  rsp, 0x10

    ; function code

    ret
```

```
mov r10, new_func
jmp r10
```

```
ctor:

    ; hack the function
    ; in main program

new_func:

    ; replacement code

    ret
```

Static function hack: making the shellcode

- Assembly:

```
1 mov r10, 0xaaaaaaaaaaaaaaaa ; address is just a 64 bits placeholder
2 jmp r10
```

  Address is a placeholder for the target function address, it will need to be changed at runtime

- Shellcode:

```
$ nasm -f elf64 jump.asm -o jump.o
$ objcopy -O binary jump.o jump.bin
$ hexdump -v -e '"\\""x" 1/1 "%02x" ""' jump.bin
\x49\xba\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\x41\xff\xe2
```

  13B long

Static function hack: code modification

```c
__attribute__((constructor)) static void ctor(void)
{
    // get pointers to the original and new functions and calculate the jump offset
    // only works if the main program exports its symbols! (cf. v2 for an alternative)
    void *main_program_handle = dlopen(NULL, RTLD_NOW);
    int64_t *target_func      = dlsym(main_program_handle, "func");
    if (!target_func)
    {
        fprintf(stderr, "%s\n", dlerror());
        exit(EXIT_FAILURE);
    }
    int64_t *new_func = (int64_t*)&replacement_func;

    // skeleton for shellcode
    unsigned char shellcode[] = "\x49\xba\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\x41\xff\xe2";
    size_t code_size = sizeof(shellcode) - 1;

    // copy address into shellcode and inject
    memcpy(shellcode + 2, &new_func, sizeof(uint64_t));
    memcpy(target_func, shellcode, code_size);
}
```

- The `constructor` attribute allows execution before `main`

Static function hack: first attempt

- Uffffff.....

```
$ clang -g callfunc.c -o callfunc -ldl -Wl,--export-dynamic
$ clang -g -fPIC -shared funchookv0.c -o libfunchookv0.so -ldl
$ LD_PRELOAD=./libfunchookv0.so ./callfunc
[1]    97703 segmentation fault (core dumped)  LD_PRELOAD=./libfunchookv0.so ./callfunc
```

Static function hack: first attempt

- The debugger is, as usual, more helpful

```
$ lldb ./callfunc
(lldb) target create "./callfunc"
Current executable set to '/mnt/hgfs/binary-hack/lectures/4-hack/statichook/callfunc' (x86_64).
(lldb) env LD_PRELOAD=./libfunchookv0.so
(lldb) r
Process 94869 launched: '/mnt/hgfs/binary-hack/lectures/4-hack/statichook/callfunc' (x86_64)
Process 94869 stopped
* thread #1, name = 'callfunc', stop reason = signal SIGSEGV: address access protected (fault address: 0x401135)
    frame #0: 0x00007ffff7f426ff libc.so.6`___lldb_unnamed_symbol1092$$libc.so.6 + 143
libc.so.6`___lldb_unnamed_symbol1092$$libc.so.6:
→  0x7ffff7f426ff <+143>: mov    qword ptr [rdi + rdx - 0x8], rcx
   0x7ffff7f42704 <+148>: mov    qword ptr [rdi], rsi
   0x7ffff7f42707 <+151>: ret
   0x7ffff7f42708 <+152>: mov    ecx, dword ptr [rsi + rdx - 0x4]
(lldb) p/a 0x401135
(int) $0 = 0x00401135 callfunc`func + 5 at callfunc.c:5
```

- It actually makes sense, nowadays in essentially all Unix OSs the **text segment is read-only**
- But as the user running the program **you own the memory** and have the right to change that!

Static function hack: writing on text segment

- Find the first page of the target's code and make it writable using **`mprotect`**

```
// make the memory containing the original function writable
// the address passed to mprotect must be aligned on page boundary
size_t    page_size  = sysconf(_SC_PAGESIZE);
uintptr_t start      = (uintptr_t)target_func;
uintptr_t end        = start + 1;
uintptr_t page_start = start & -page_size;
int status = mprotect((void *)page_start, end - page_start, PROT_READ | PROT_WRITE | PROT_EXEC);
if (status)
{
    fprintf(stderr, "Error unprotecting page\n");
    fprintf(stderr, "Value of errno    : %d\n", errno);
    fprintf(stderr, "Error opening file: %s\n", strerror(errno));
    exit(EXIT_FAILURE);
}
```

- Pages are at generally at least 4kB so one page is more than enough for our 13B shellcode

Static function hack

- It works!

```
$ clang -g callfunc.c -o callfunc -ldl -Wl,--export-dynamic
$ clang -g -fPIC -shared funchookv1.c -o libfunchookv1.so -ldl
$ LD_PRELOAD=./libfunchookv1.so ./callfunc
====================
INJECTION SUCCESSFUL
Injected code   : \x49\xba\xb0\xe1\x78\x61\x62\x7f\x0\x0\x41\xff\xe2
Target function : 0×401130
New function    : 0×7f626178e1b0
====================
calling variant with id 15431
```

- … but there is a catch: did you notice the linker option **`-Wl,--export-dynamic`**?

- It exports the symbols of the main program for linking, this is the only way to get the address of the target with **`dlsym`**

Static function hack: for real, i.e. without `dlsym`

- If the export option is dropped

```
$ clang -g callfunc.c -o callfunc -ldl
$ LD_PRELOAD=./libfunchookv1.so ./callfunc
./callfunc: undefined symbol: func
```

- But the symbol is of course here, it is just not visible to the dynamic linker anymore.

- The BFD library, part of GNU binutils, can browse and ELF image and give a symbol's address

- Have a look at the v2!

Limitations and extensions

- *What if the target is less than 13B?*

  The the injected code must be written in a new “relay” function, this can be done by adding executable memory with `mmap` (cf. “Hello world” shellcode example)

- *What if we just want to wrap around the target?*

  The issue is that the jump overwrites code in the target. This can be solved by creating a “trampoline” function, which execute the replaced code and resume the execution of the target right after the injected jump.

Is that safe?

- **Very**, all code injections example here rely on the fact that the program is unprivileged and belongs to the user
- Everything was done through calls to system functions
- Attack vectors using code injection are generally designed around **buffer overflow exploits**
- Remember the Fibonacci sequence going crazy through buffer overflow?
- Think about it, if you keep overflowing a stack buffer you will eventually **corrupt the return address** of the function…