THE ENGINEERING BEHIND THE GNIREENIGNE

May contain traces of assembler

Background

What motivated Frida?

- Interoperation
  - connect to black-boxes beyond existing integration points
- Compatibility
  - workarounds for specification vs implementation drift
  - micro-level reverse-engineering
- Design recovery
  - recover specification from implementation
- Lack of dynamic reverse engineering tools

Design goals for Frida

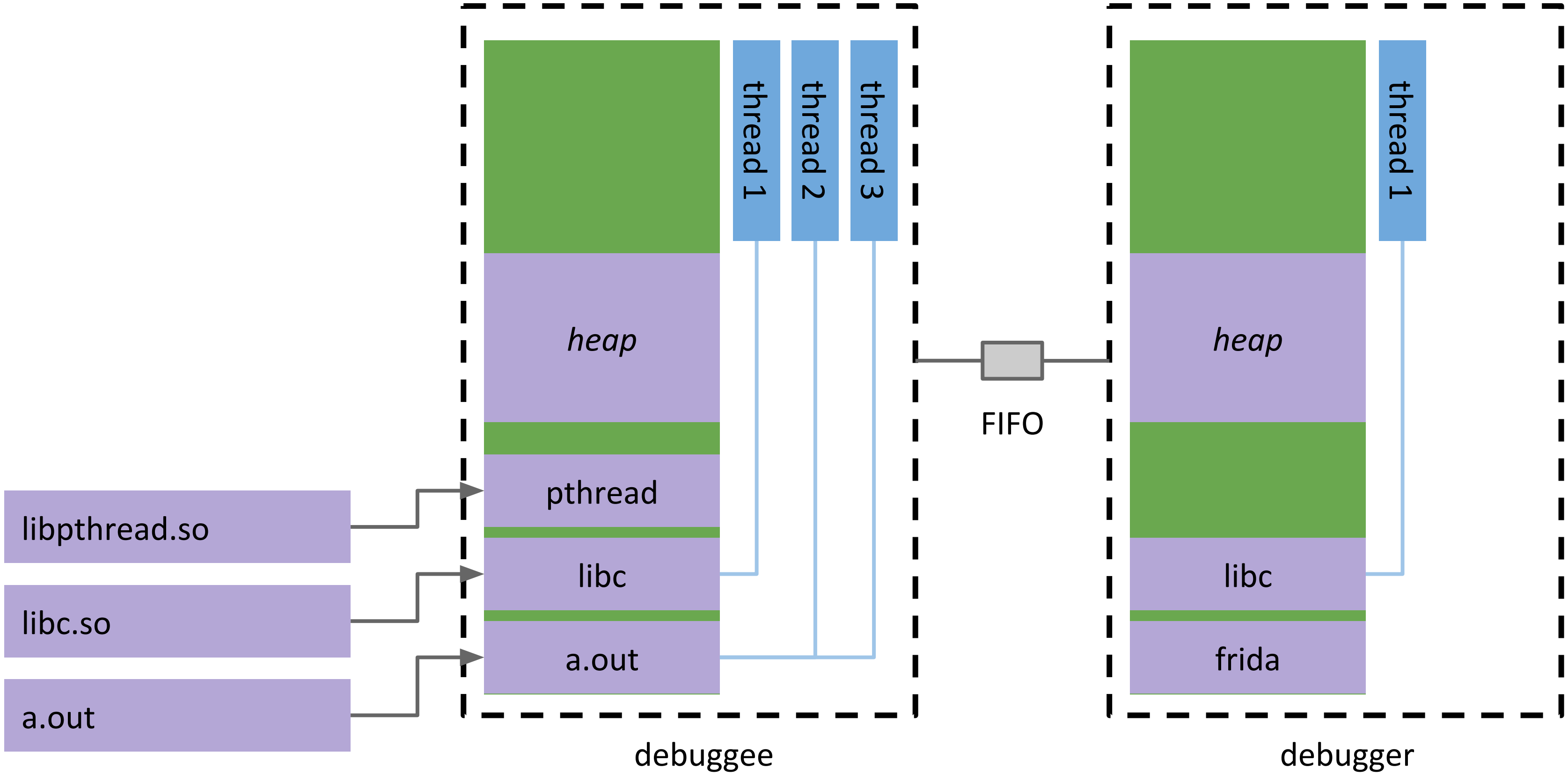
- Live inspection of other processes
    - no source code
    - no debugging symbols
- “Inject” our own agent D into the remote process P without P noticing, and communicate with D from the outside of process P
- Inspect and modify memory, threads, registers
- Avoid anti-debugging defenses

Plan of attack

1. Inject
    a. insert our own custom logic into remote process
2. Intercept
    a. trap function calls in remote process
3. Stalk
    a. instruction-level code tracing in the remote process
    b. avoiding all current anti-debugging products

Injection

Injection - the basics



Injection - the game plan

1. Create `.so` containing our agent
2. Hijack thread in remote process with `ptrace`
3. Allocate memory for bootstrapper in remote process
4. Populate bootstrapper with our own code
5. Execute bootstrapper in remote process, which
    - starts fresh thread, which
        - opens FIFO to debugger process
        - notifies debugger over FIFO
        - loads agent `.so` file
        - executes (long running) agent entry point from `.so` file
        - closes FIFO

Injection - the relevant APIs

- ptrace
  - process trace
- mmap
  - map files or devices into memory
- dlopen
  - loads a dynamic library (`.so` file) into a process
- dlsym
  - finds the address where a function from the `.so` is loaded into memory
- signal
  - set up handlers for UNIX signals (`SIGSTOP`, `SIGCONT`, `...`)

Create `.so` containing our agent

```
0x00000000: 7F454C46 02010100 00000000 00000000      .ELF............
0x00000010: 03003E00 01000000 E0DA0500 00000000      ..>.............
0x00000020: 40000000 00000000 48295C00 00000000      @.......H)\.....
0x00000030: 00000000 40003800 07004000 1D001C00      ....@.8...@.....
0x00000040: 01000000 05000000 00000000 00000000      ................
0x00000050: 00000000 00000000 00000000 00000000      ................
0x00000060: F2E85800 00000000 F2E85800 00000000      ..X.......X.....
0x00000070: 00002000 00000000 01000000 06000000      .. .............
0x00000080: 78E95800 00000000 78E97800 00000000      x.X.....x.x.....
0x00000090: 78E97800 00000000 803E0300 00000000      x.x......>......
0x000000A0: 408F0300 00000000 00002000 00000000      @......... .....
0x000000B0: 02000000 06000000 A8085B00 00000000      ..........[.....
```

Hijack thread in remote process with `ptrace`

```
ptrace (PTRACE_ATTACH, pid, NULL, NULL);
waitpid (pid, &status, 0);
ptrace (PTRACE_GETREGS, pid, NULL, saved_regs);
```

Allocate memory for bootstrapper (1)

```
ptrace (PTRACE_GETREGS, pid, NULL, &regs)
regs.rip = resolve_remote_libc_function (pid, “mmap”);
regs.rdi = 0;
regs.rsi = 8192;
regs.rdx = PROT_READ | PROT_WRITE | PROT_EXEC;
regs.rcx = MAP_PRIVATE | MAP_ANONYMOUS;
regs.r8 = -1;
regs.r9 = 0;
regs.rax = 1337;
regs.rsp -= 8;
```

Allocate memory for bootstrapper (2)

```
ptrace (PTRACE_POKEDATA, pid, regs.rsp, DUMMY_RETURN_ADDRESS)
ptrace (PTRACE_SETREGS, pid, NULL, &regs)
ptrace (PTRACE_CONT, pid, NULL, NULL)
frida_wait_for_child_signal (pid, SIGTRAP)
ptrace (PTRACE_GETREGS, pid, NULL, &regs)
bootstrapper = regs.rax
```

- `bootstrapper` now contains the address of the bootstrapper memory block

Populate bootstrapper with our own code

1. Initialize memory block with generated functions

`create_frida_thread()` [at `bootstrapper + 0`]

```
so = dlopen ("libpthread.so", RTLD_LAZY)
thread_create = dlsym (so, "pthread_create")
thread_create (&worker_thread, NULL, bootstrapper + 128, NULL)
int3()
```

Populate bootstrapper with our own code

1. Initialize memory block with generated functions

`load_and_exec_agent_so()` [at `bootstrapper + 128`]

```
fifo = open(fifo_path, O_WRONLY)
write(fifo, “frida_agent_main”, 1)
so = dlopen(“frida-agent.so”, RTLD_LAZY)
entry = dlsym(so, “frida_agent_main”)
entry(DATA_STRING)
close(fifo)
```

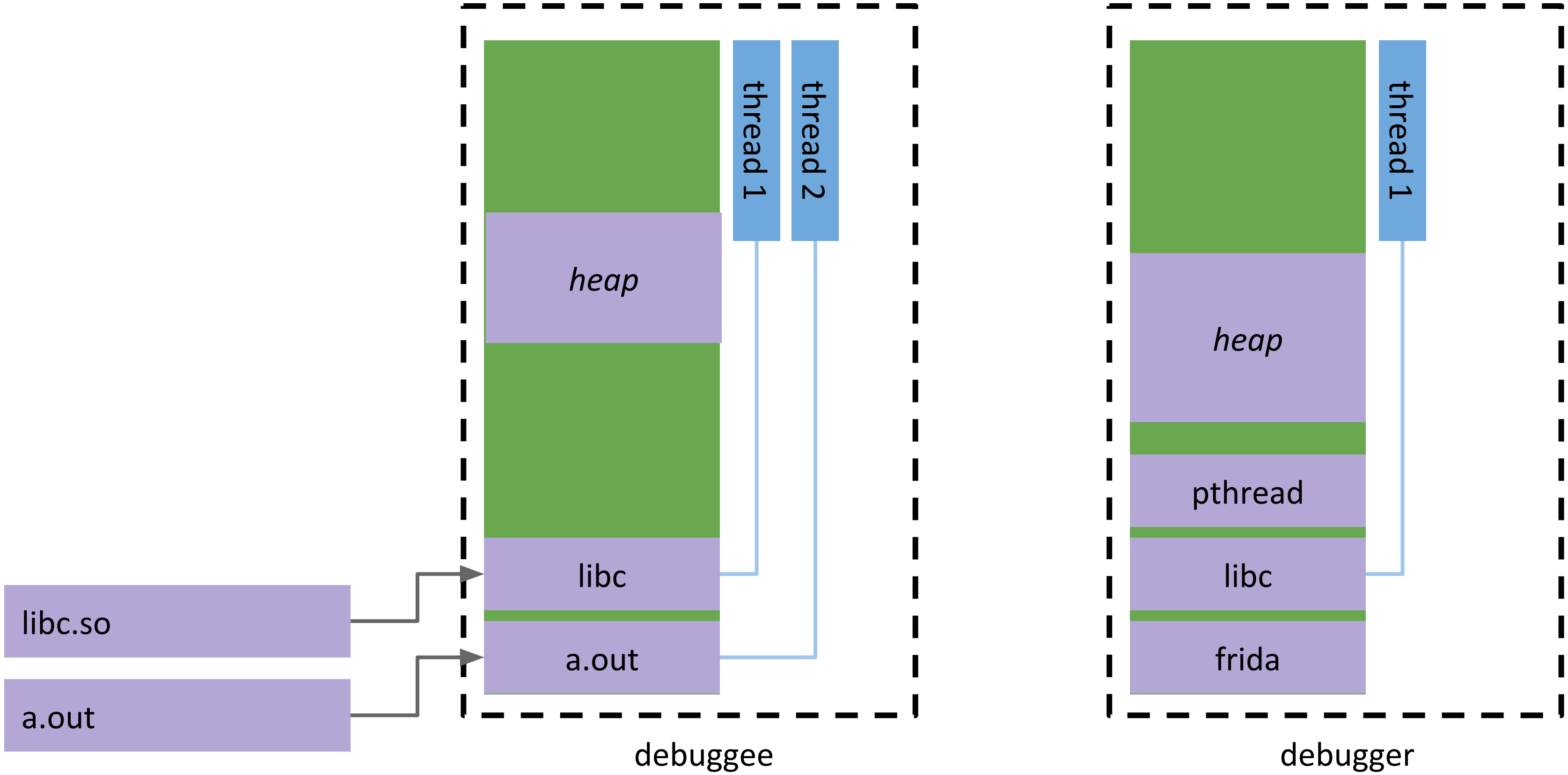
Execute bootstrapper in remote process

```
ptrace (PTRACE_GETREGS, pid, NULL, &regs)
regs.rip = bootstrapper
regs.rsp = bootstrapper + 8192
ptrace (PTRACE_SETREGS, pid, NULL, &regs)
ptrace (PTRACE_CONT, pid, NULL, NULL)
frida_wait_for_child_signal (pid, SIGTRAP)
```

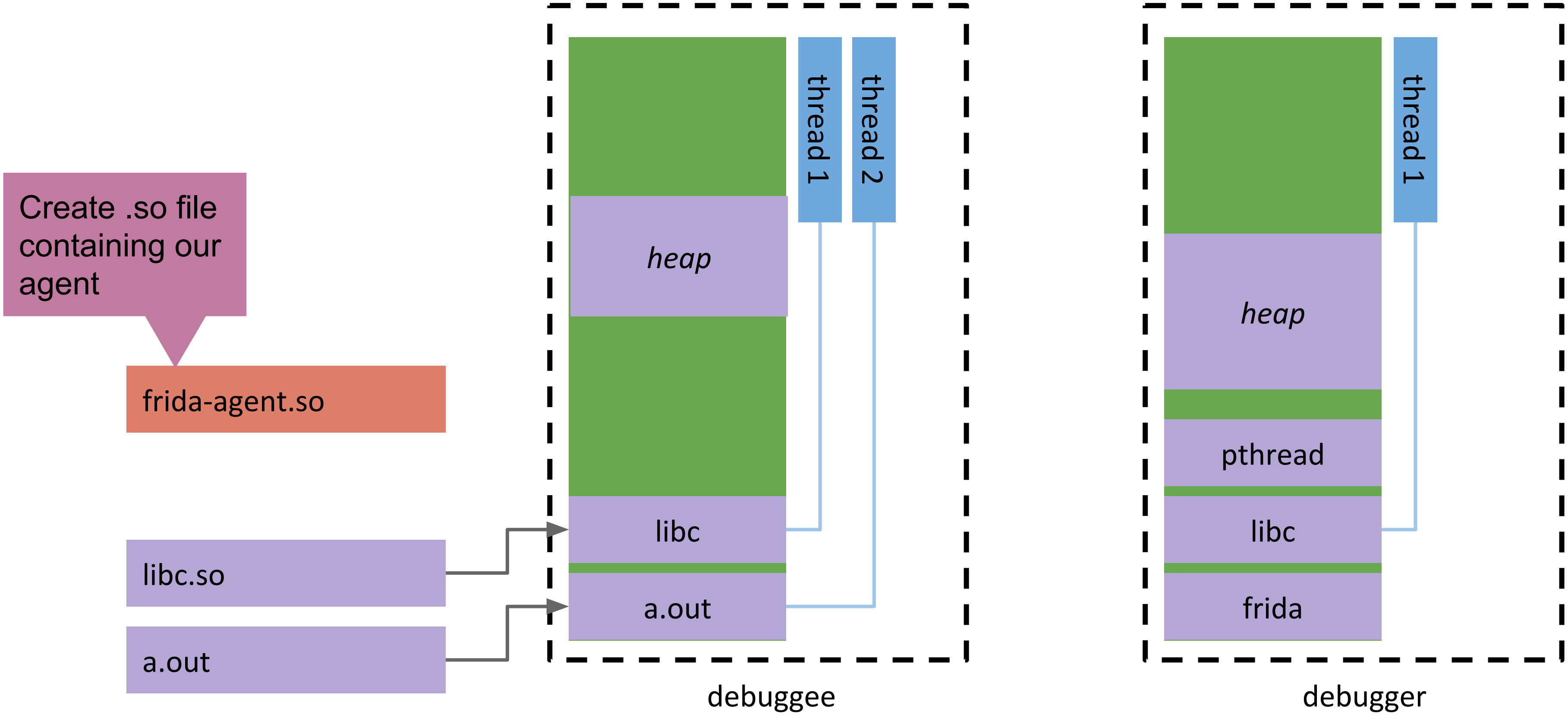
Resume remote thread execution

```
ptrace (PTRACE_SETREGS, pid, NULL, saved_regs)
ptrace (PTRACE_DETACH, pid, NULL, NULL)
```

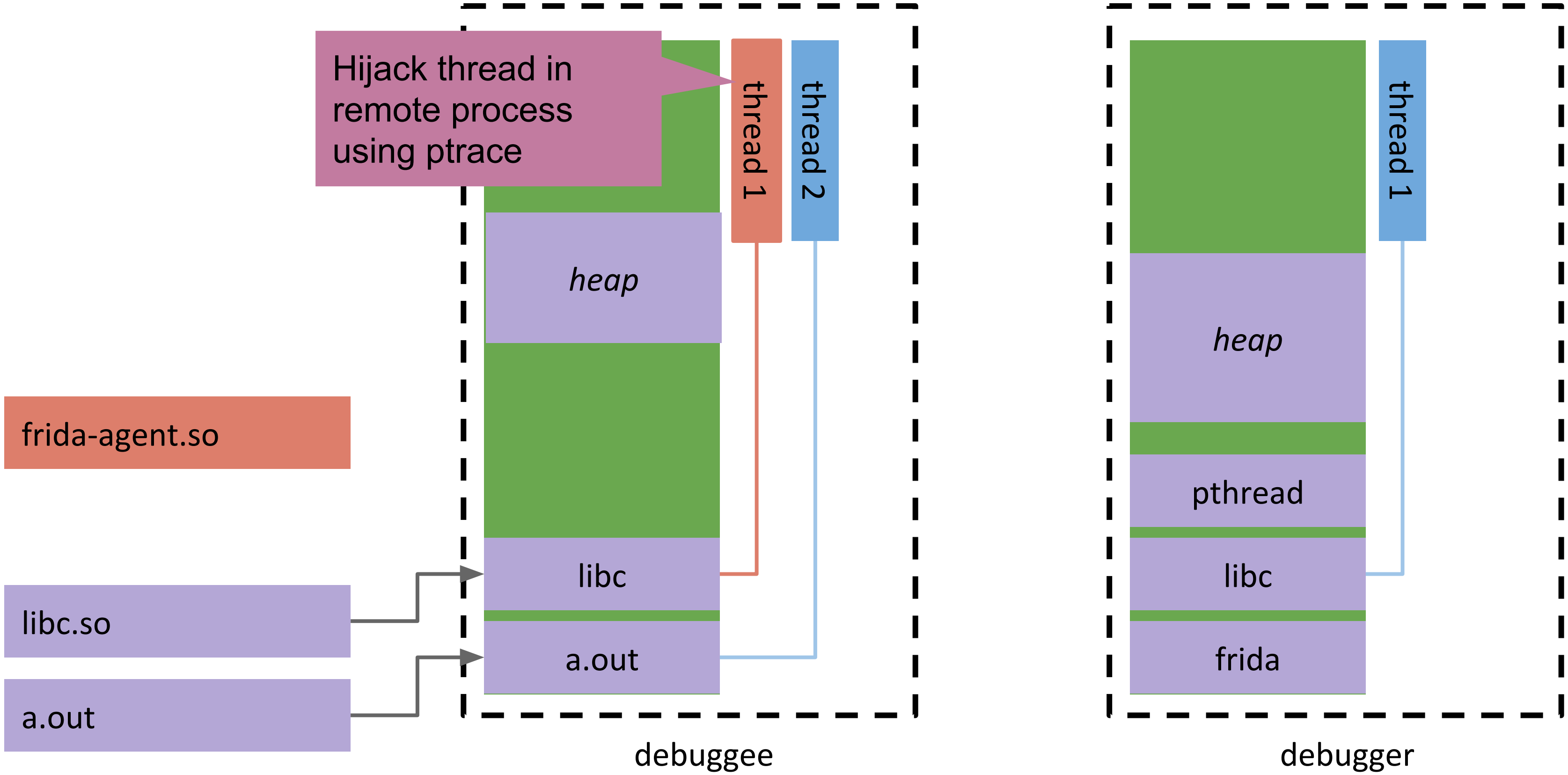
Injection - the summary



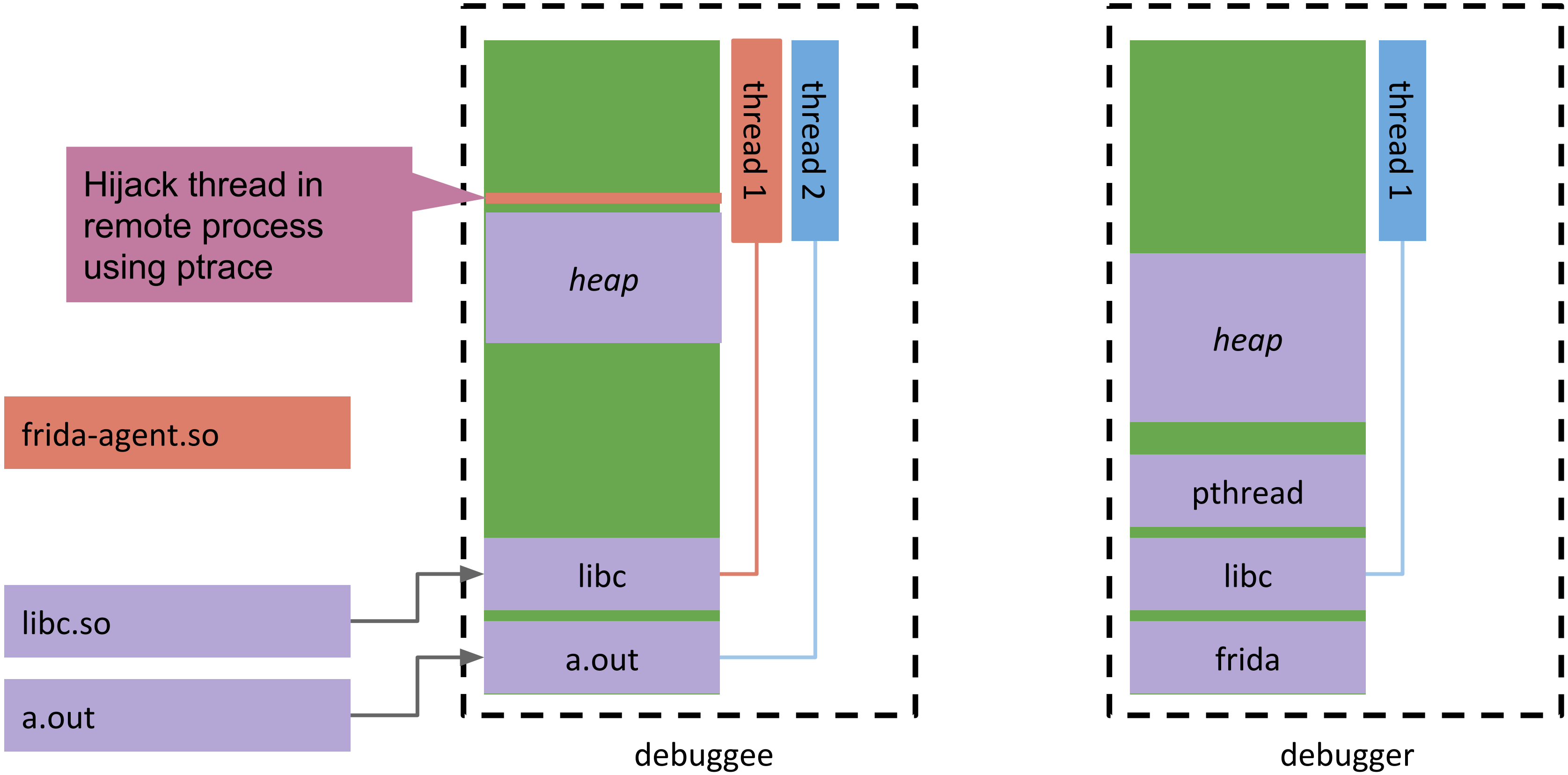
Injection - the summary



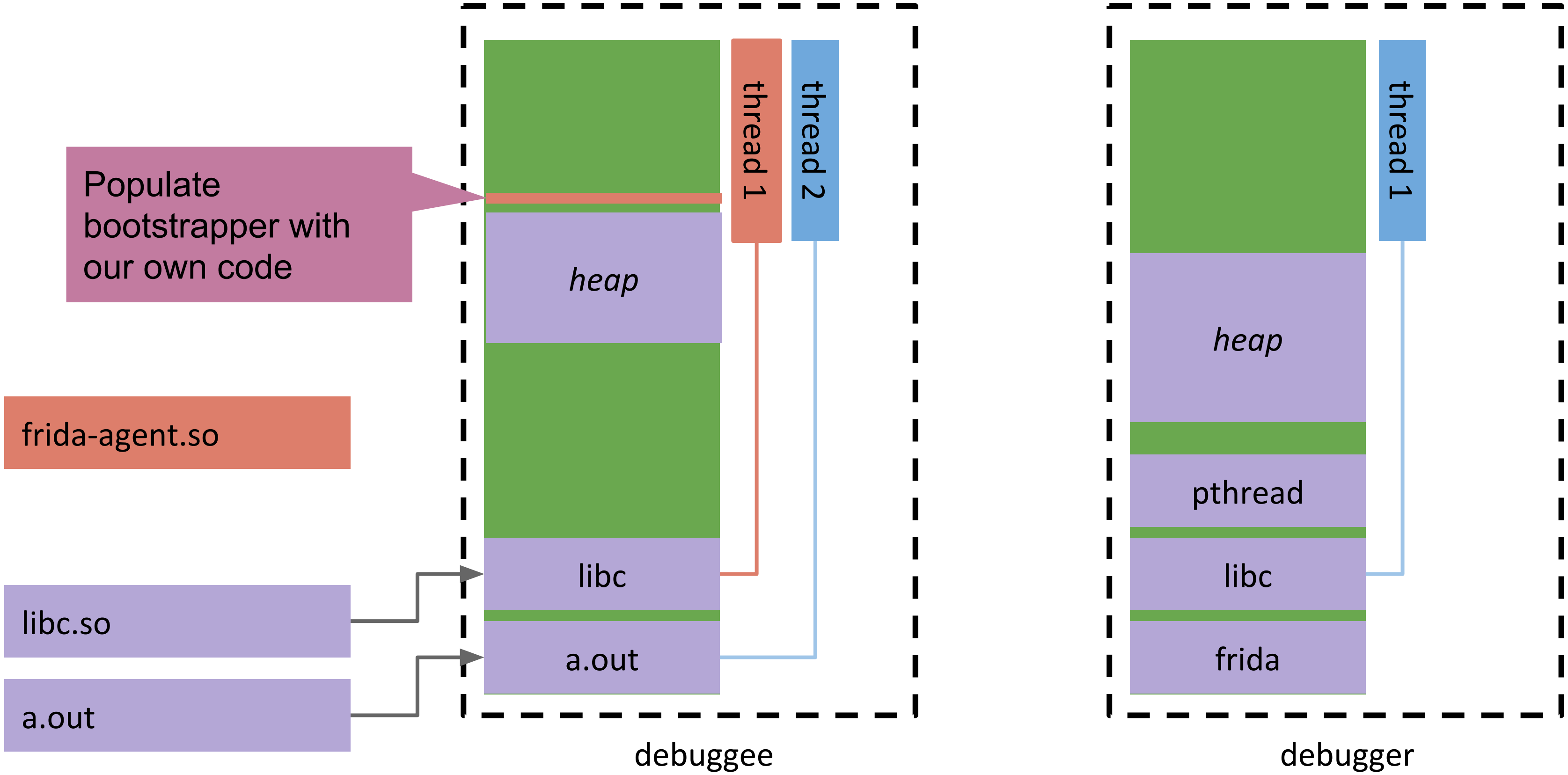
Injection - the summary



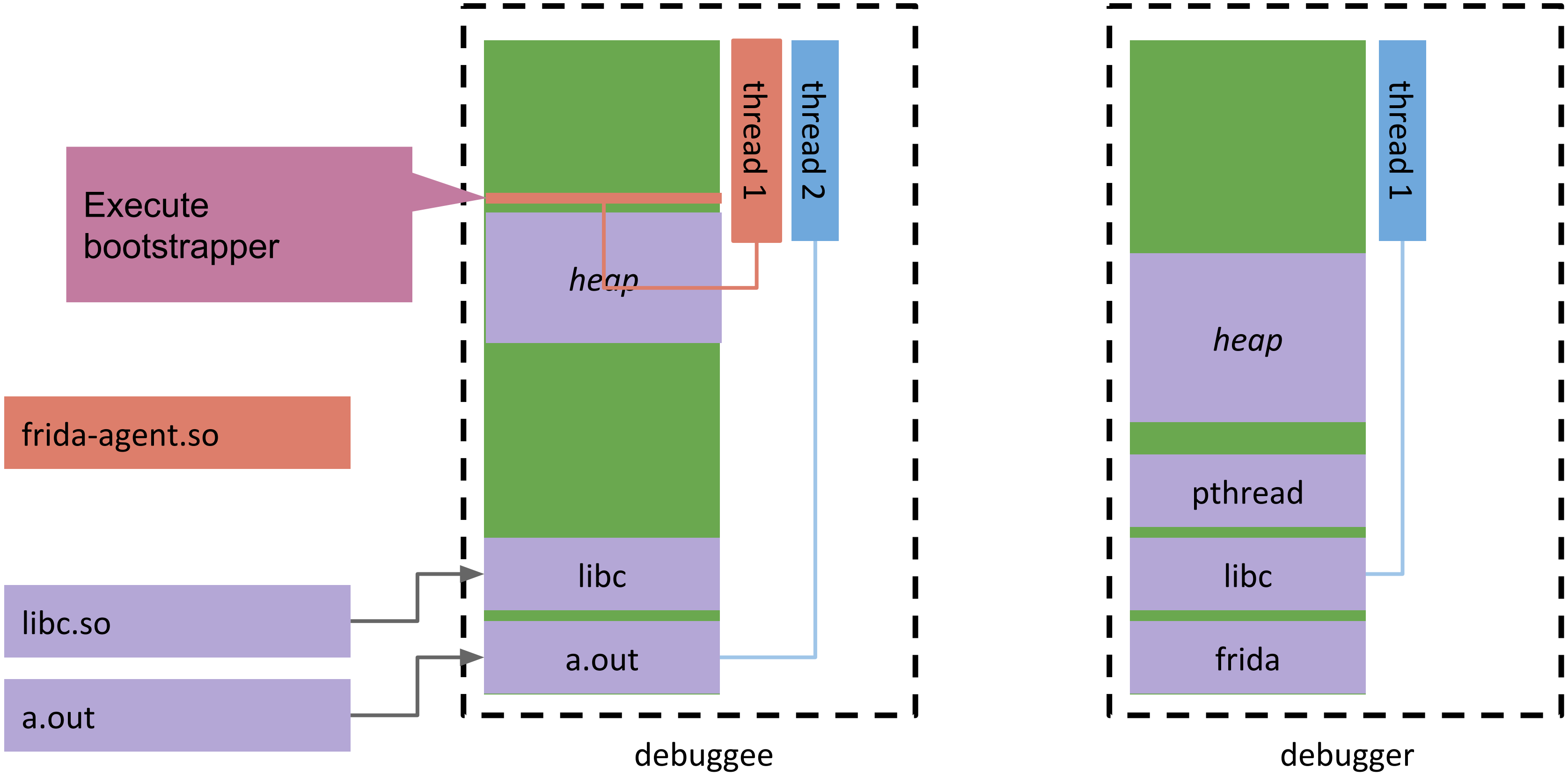
Injection - the summary



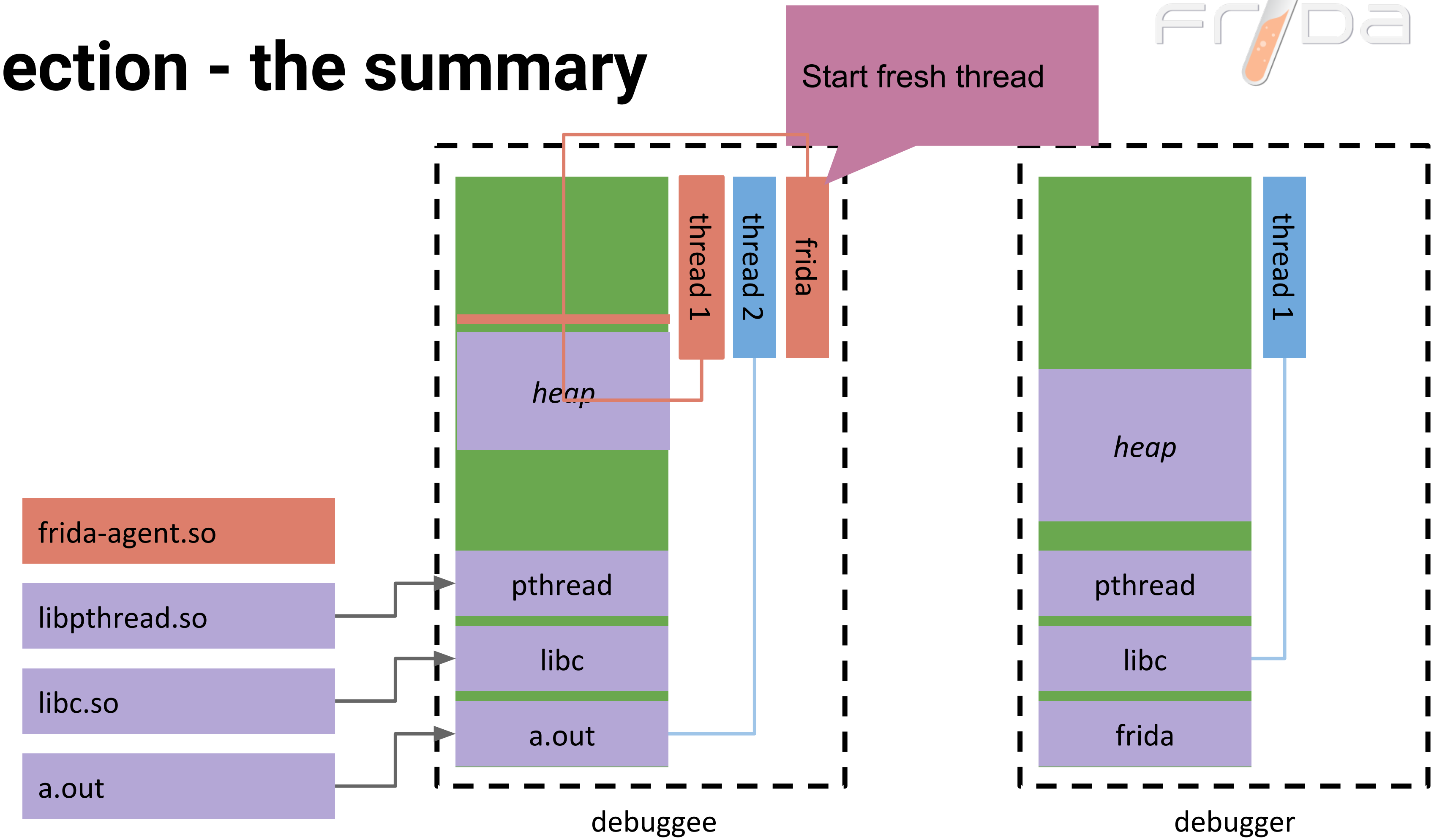
Injection - the summary



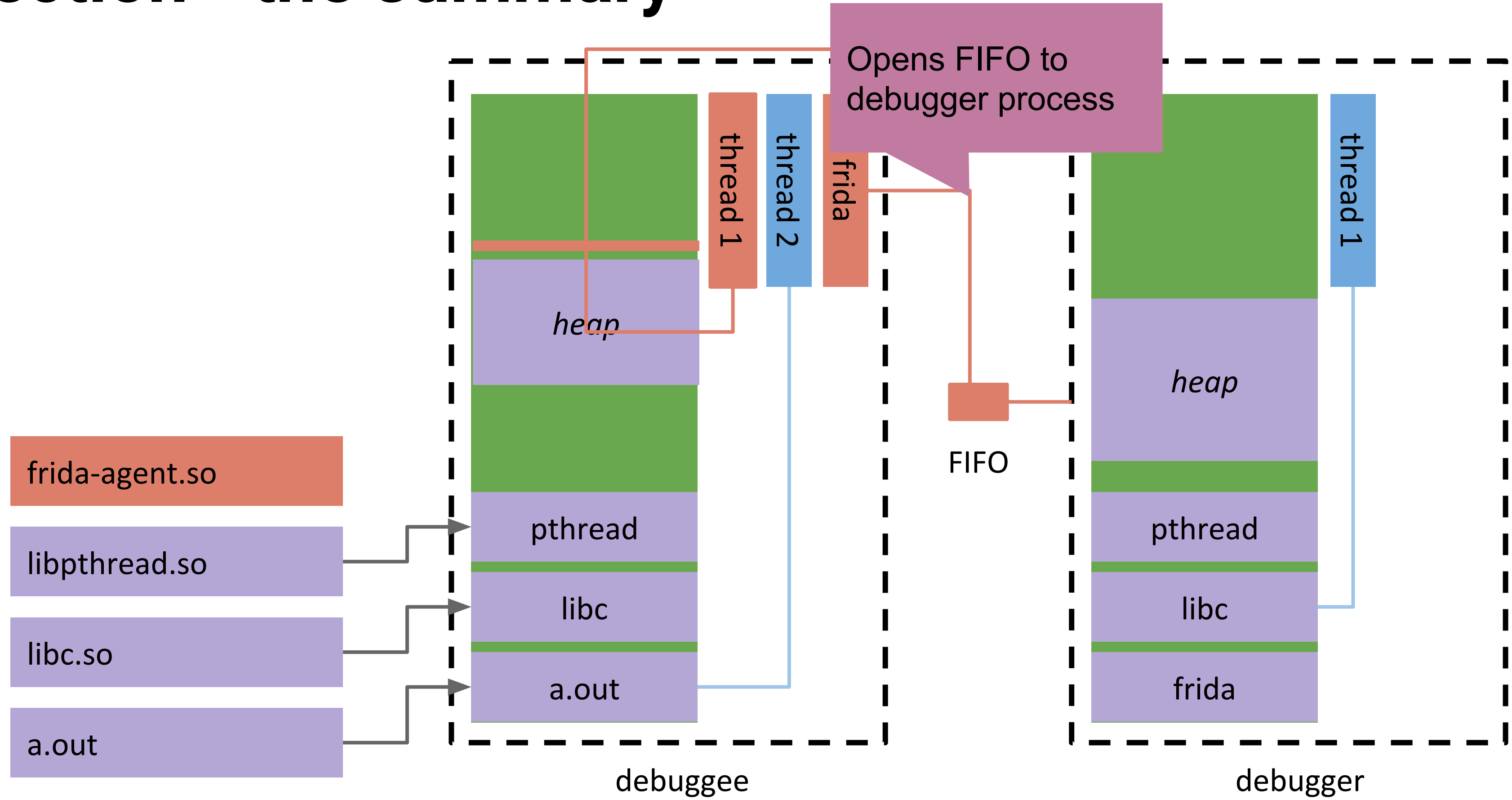
Injection - the summary



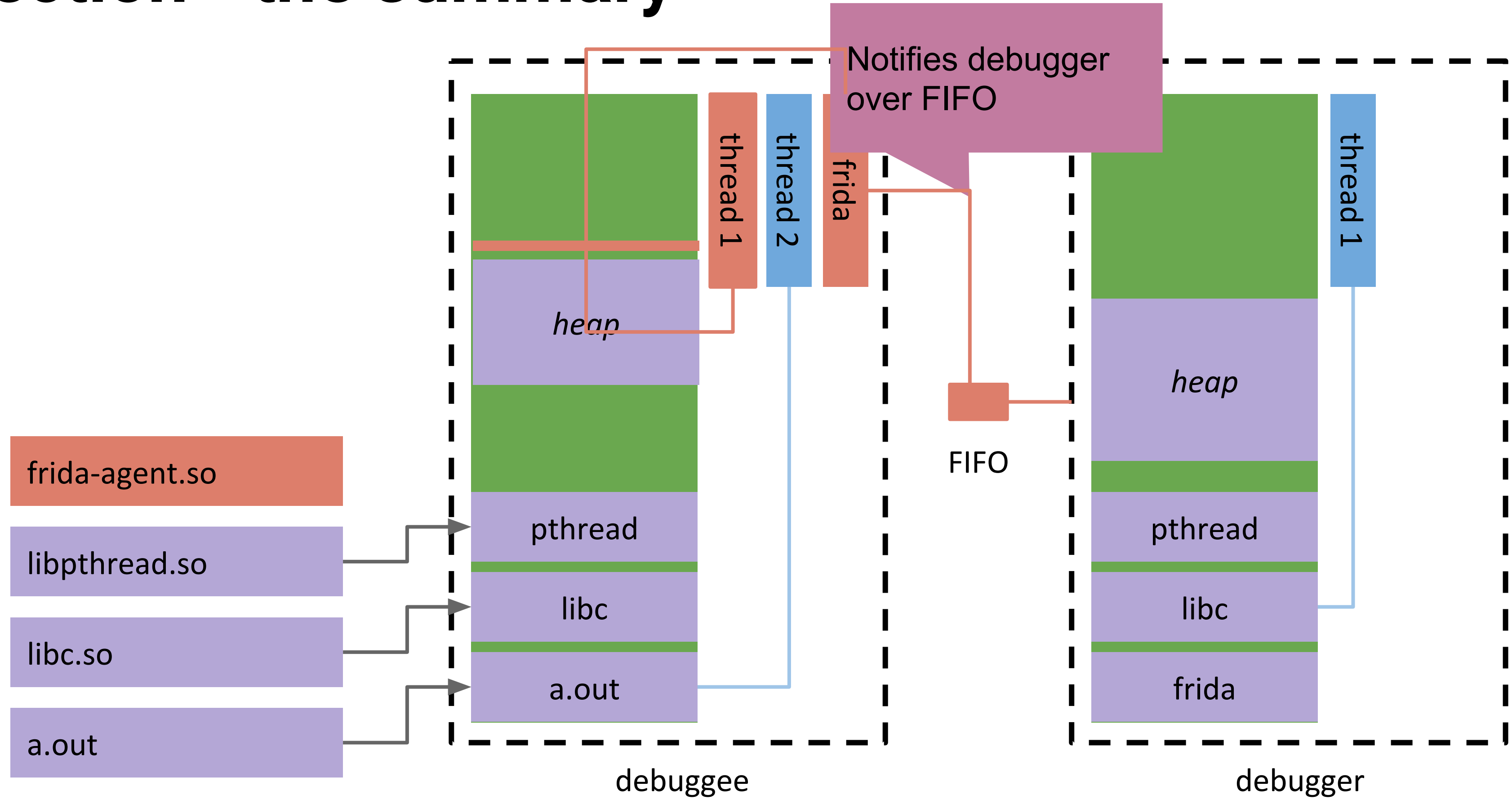
Injection - the summary



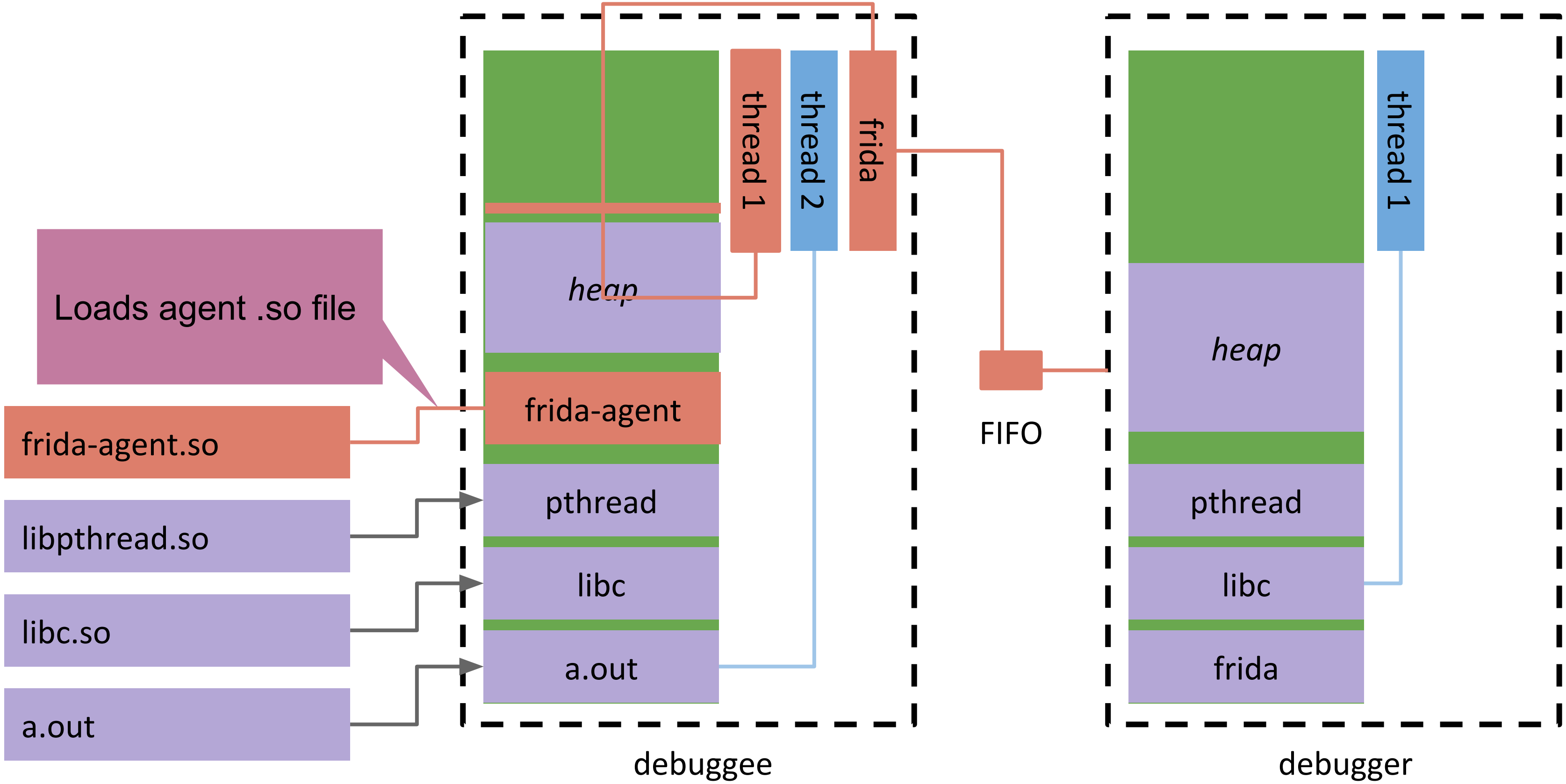
Injection - the summary



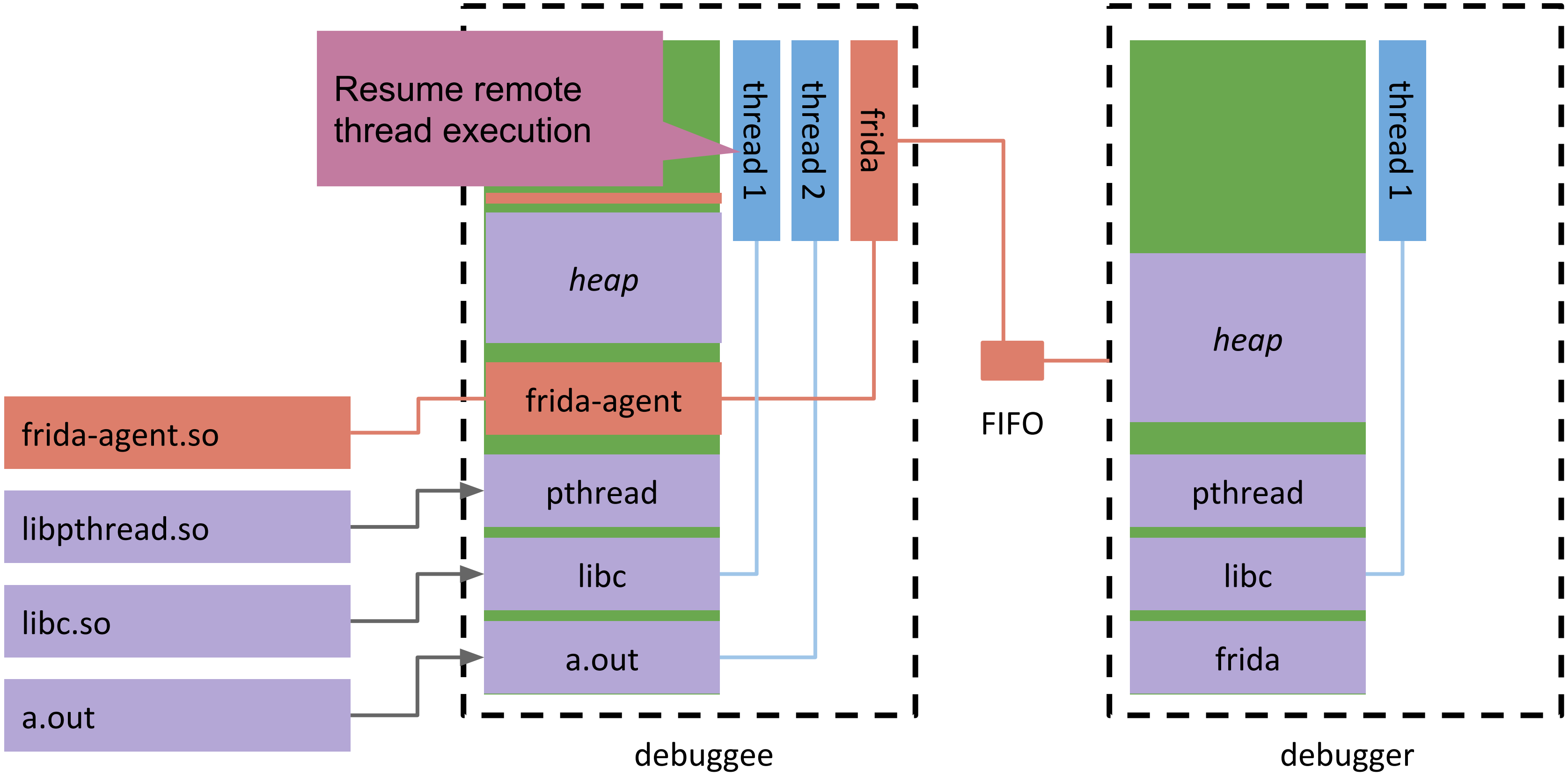
Injection - the summary



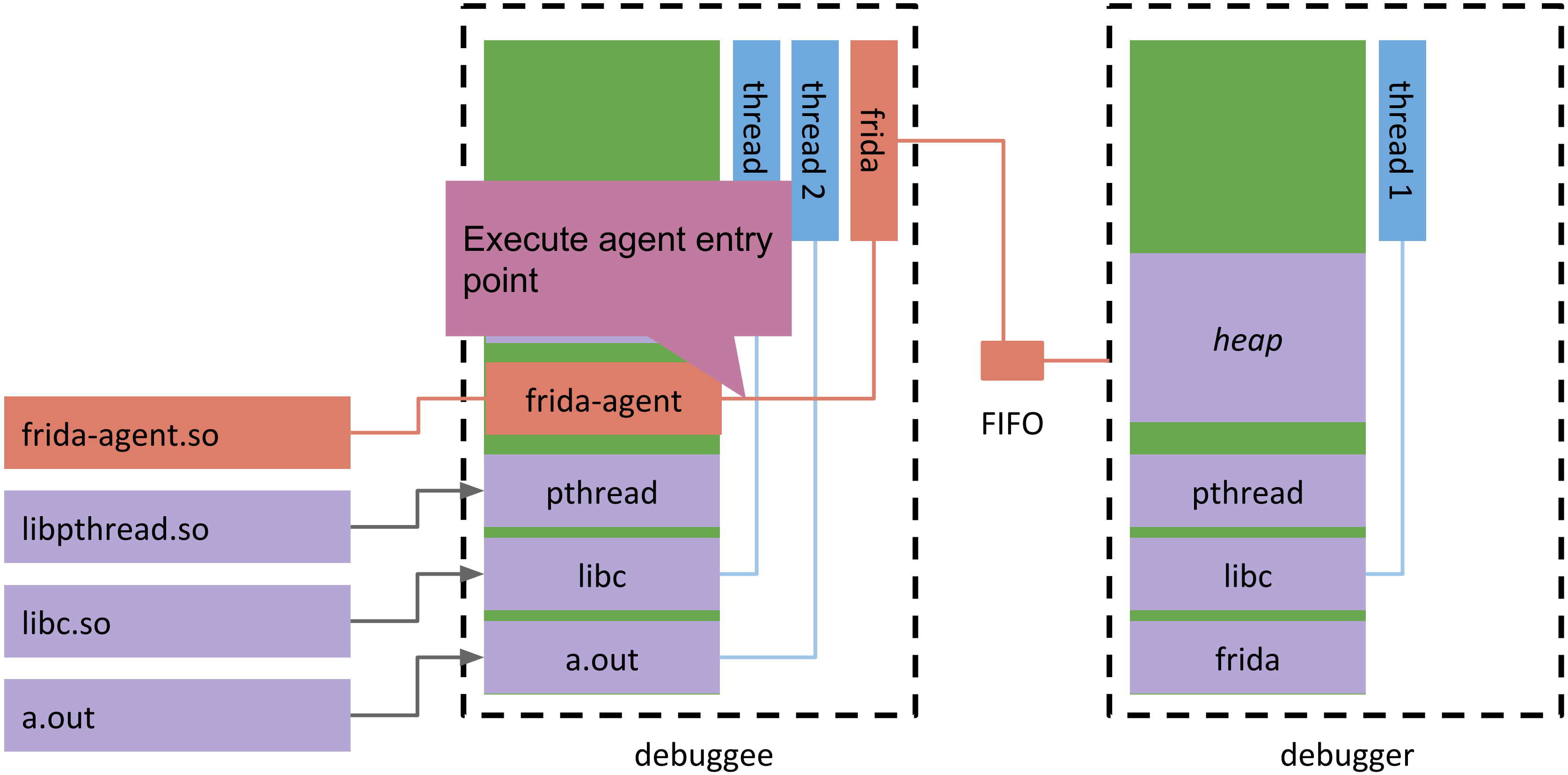
Injection - the summary



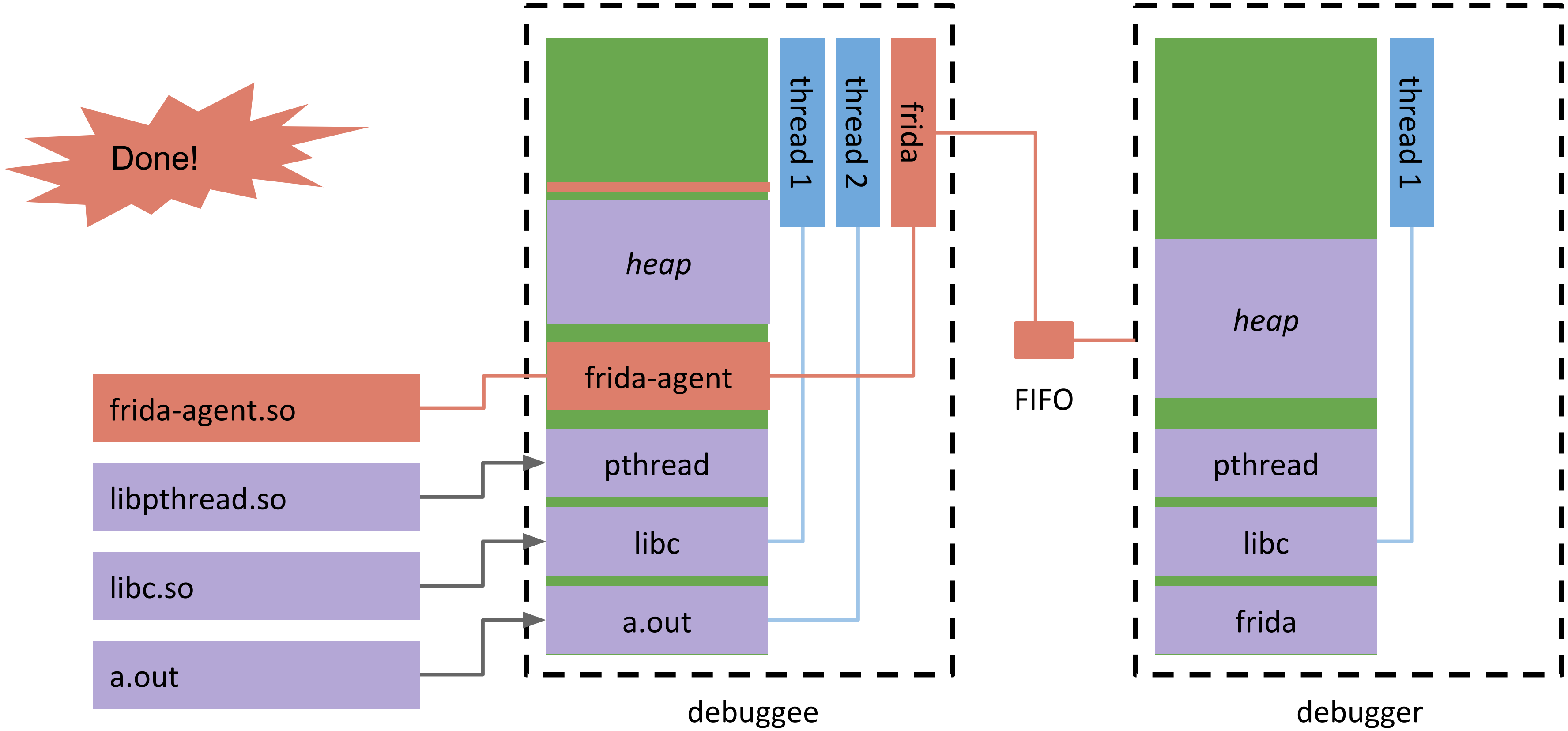
Injection - the summary

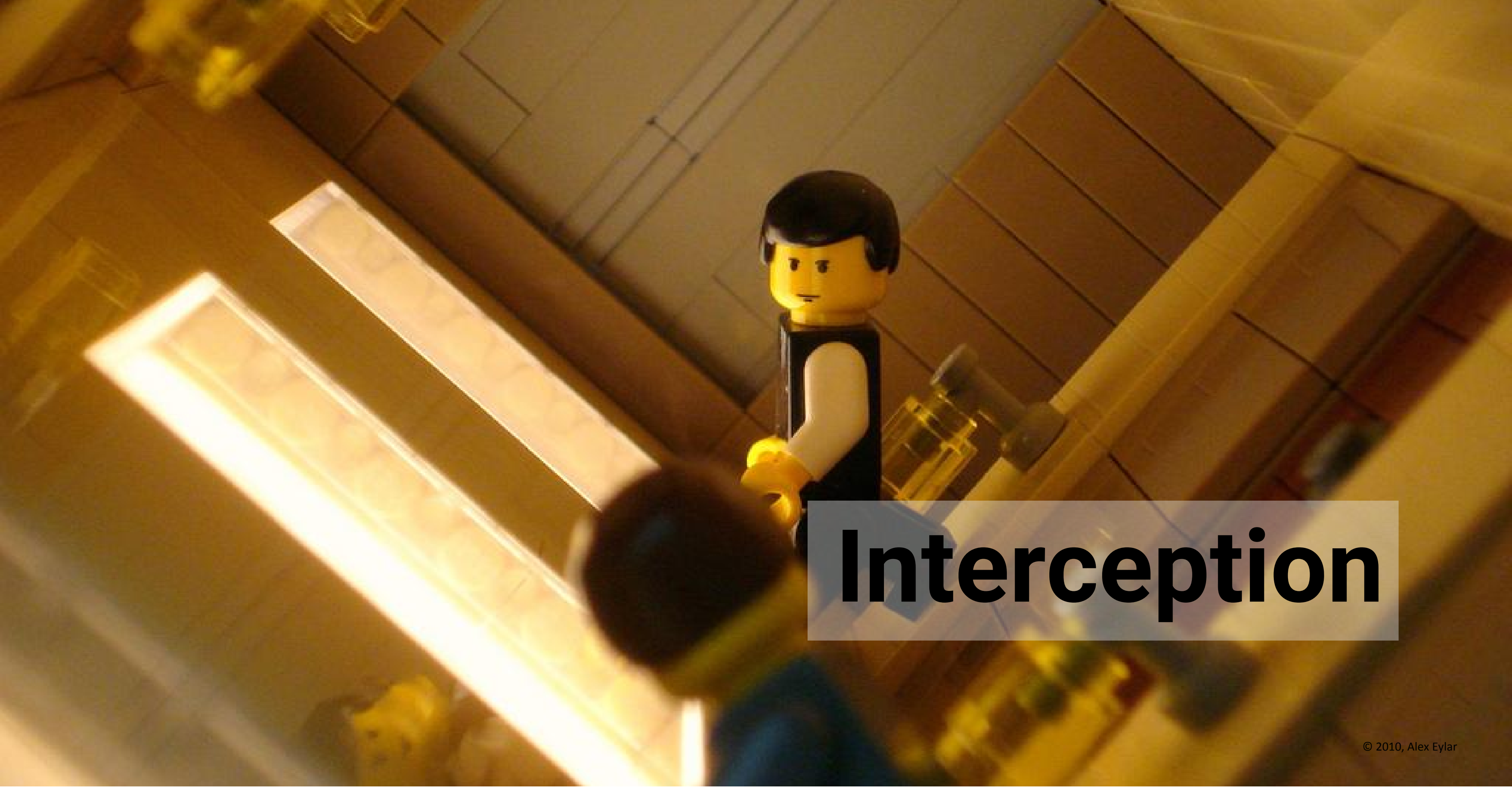


Injection - the summary



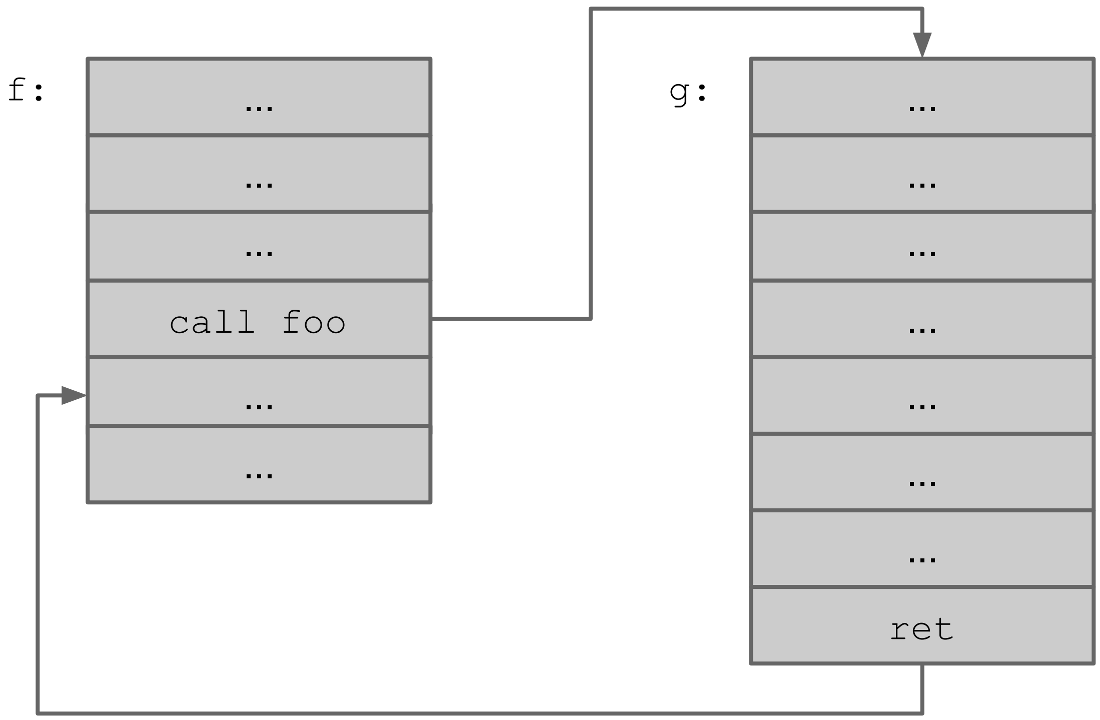
Injection - the summary



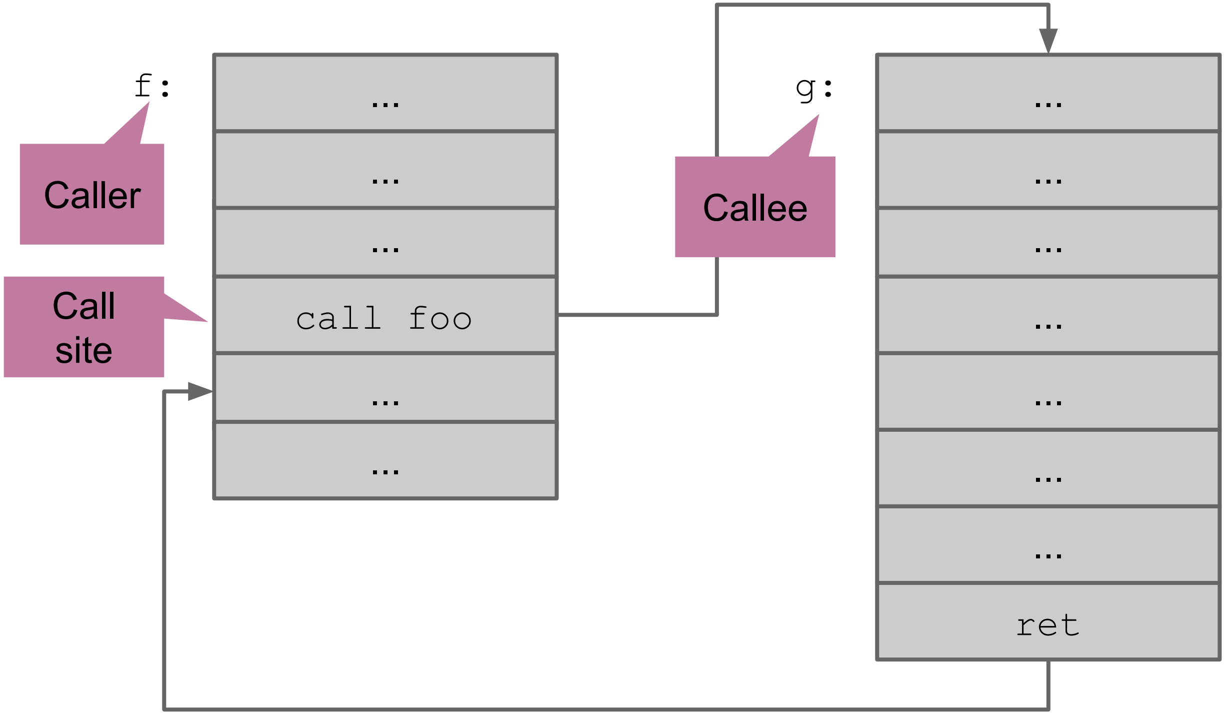


Interception

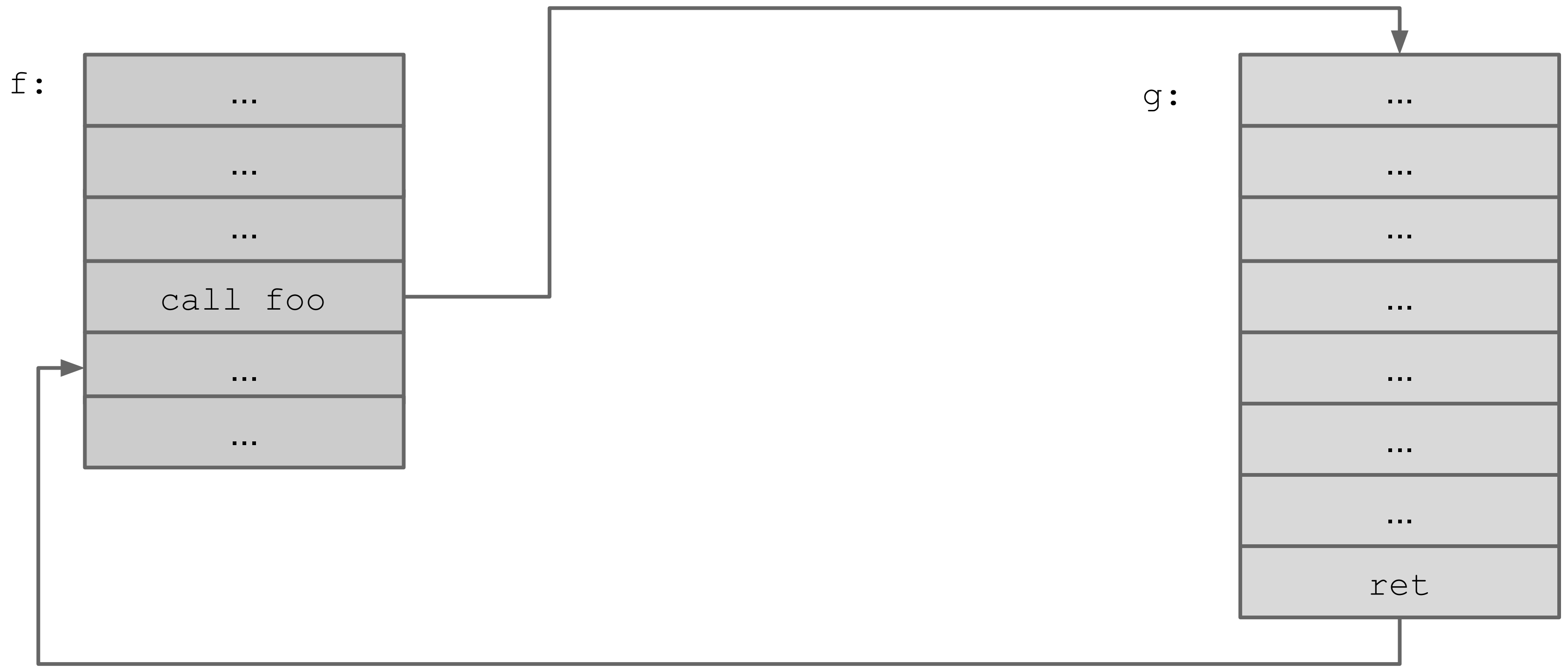
Interception - the basics



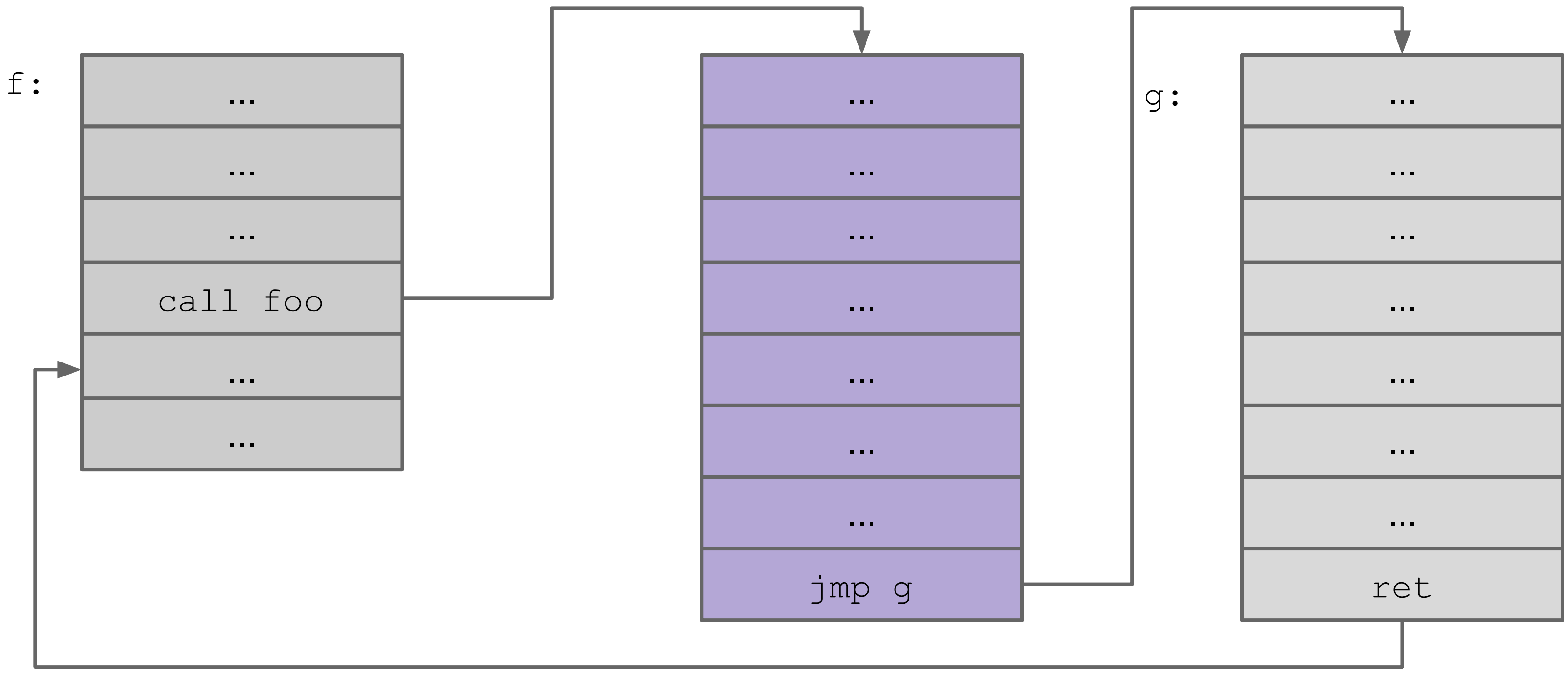
Interception - the basics



Interception - the basics



Interception - the basics



Interception - the game plan

- find address of function of interest
- generate trampoline for calling our interceptor function
- replace first instruction(s) with call to our own trampoline
- trampoline calls interceptor function
- trampoline hides all stack/register modifications

Find address of function

1. Enumerate all modules for current process
   a. Look up in `/proc/self/maps`
2. For each module (= each .so or executable)
   a. Parse ELF format
   b. Find all symbols (= function names)
   c. Find base address for code segment
3. If relevant symbol found
   a. Compute location of symbol relative to base address
   b. Find base address of module in current process

Interception - initial conditions

```
…
e8 04 03 01 01  call __decrypt_frame
…
```

```
__decrypt_frame:
55        push ebp
8b ec     mov ebp, esp
8b 45 08 mov eax, [rbp + 8]
8b 4d 0c mov ecx, [rbp + 12]
…
```

Interception - desired flow

```
…
e8 04 03 01 01  call __decrypt_frame
…
```

```
__decrypt_frame:
55       push ebp
8b ec    mov ebp, esp
8b 45 08 mov eax, [rbp + 8]
8b 4d 0c mov ecx, [rbp + 12]
…
```

```
trampoline:
<save registers>
call js_on_enter_callback
<restore registers>
```

Generate trampoline

```
…
e8 04 03 01 01  call __decrypt_frame
…
```

```
__decrypt_frame:
55        push ebp
8b ec     mov ebp, esp
8b 45 08 mov eax, [rbp + 8]
8b 4d 0c mov ecx, [rbp + 12]
…
```

```
trampoline:
<save registers>
call js_on_enter_callback
<restore registers>
```

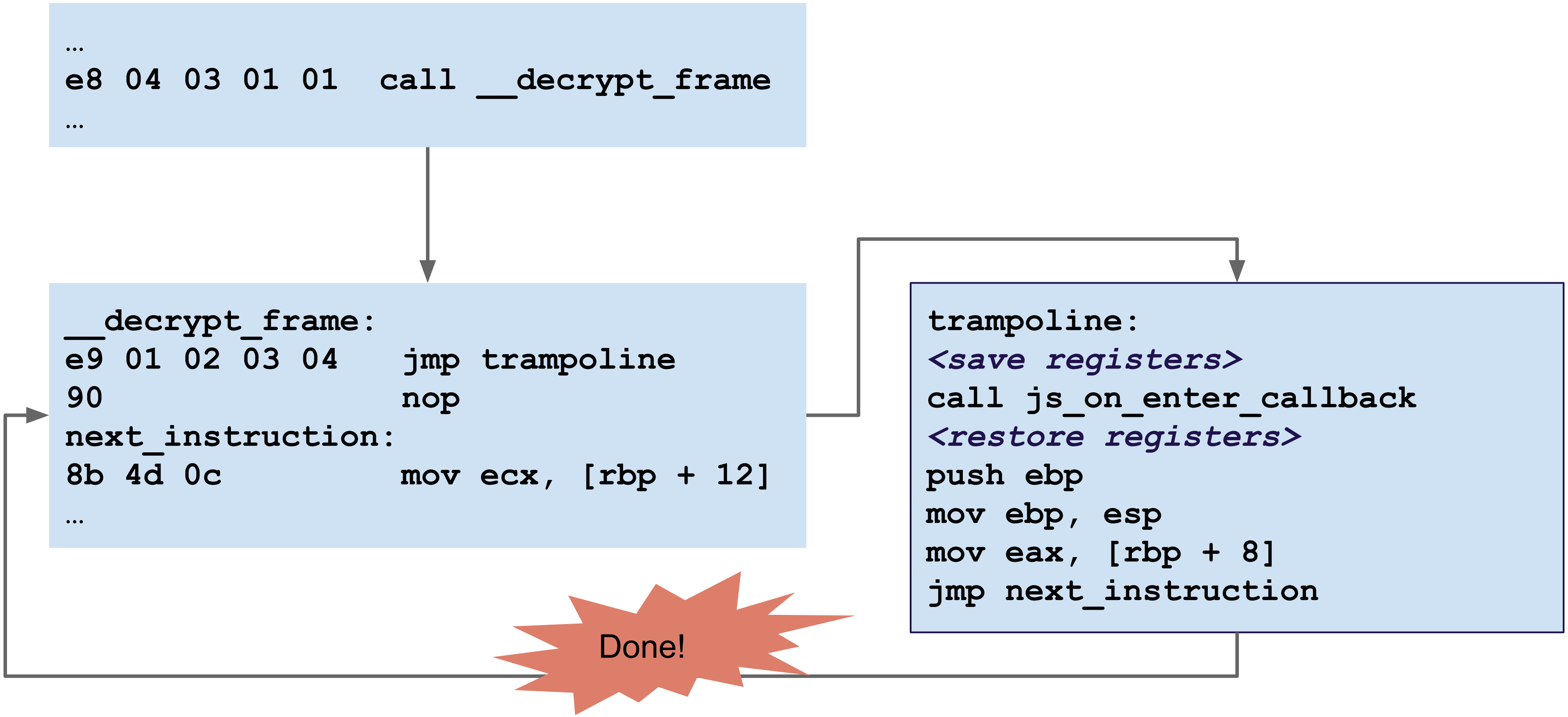
Save initial instructions at trampoline end

```
…
e8 04 03 01 01  call __decrypt_frame
…
```

```
__decrypt_frame:
55        push ebp
8b ec     mov ebp, esp
8b 45 08 mov eax, [rbp + 8]
8b 4d 0c mov ecx, [rbp + 12]
…
```

```
trampoline:
<save registers>
call js_on_enter_callback
<restore registers>
push ebp
mov ebp, esp
mov eax, [rbp + 8]
jmp next_instruction
```

Replace first instructions → desired flow



Stalking

Stalking - the game plan

- intercept (trap) function call
- apply instruction-based binary code rewriting to first basic block of function
- wrap each instruction with a prologue and epilogue
- rewrite every branch instruction to call into stalker
- place resulting basic block in a new memory page
- mark page executable
- replace first instruction in original function with branch to new basic block

Stalking - the basics

max:

```
1   cmp eax, ebx
2   jg 6
3   push eax
4   mov eax, ebx
5   pop ebx
6   ret
```

Stalking - the basics

```
max:  1  cmp eax, ebx
      2  jg 6

      3  push eax
      4  mov eax, ebx
      5  pop ebx

      6  ret
```

Split into basic blocks

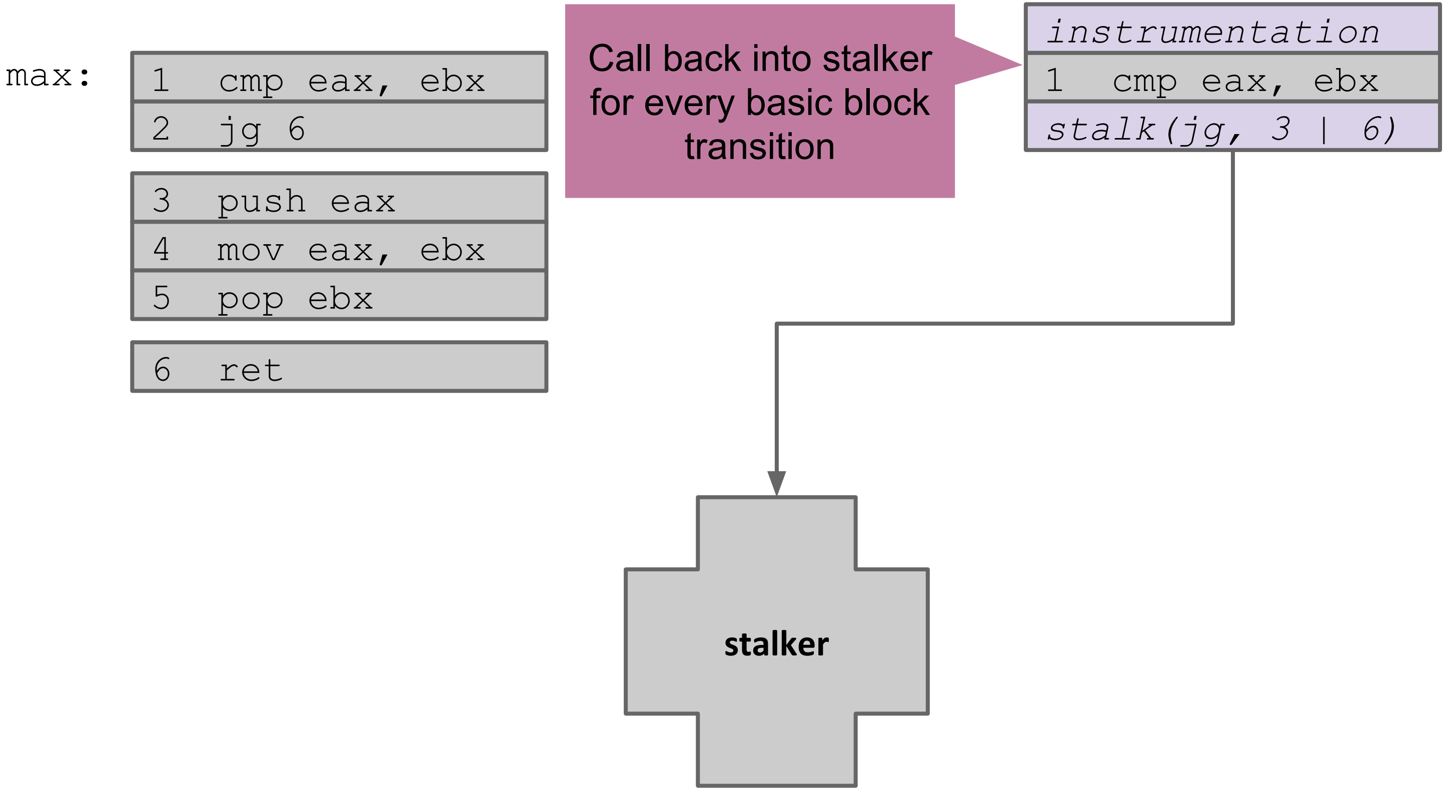
Stalking - the basics

```
max:  1   cmp eax, ebx
      2   jg 6

      3   push eax
      4   mov eax, ebx
      5   pop ebx

      6   ret
```

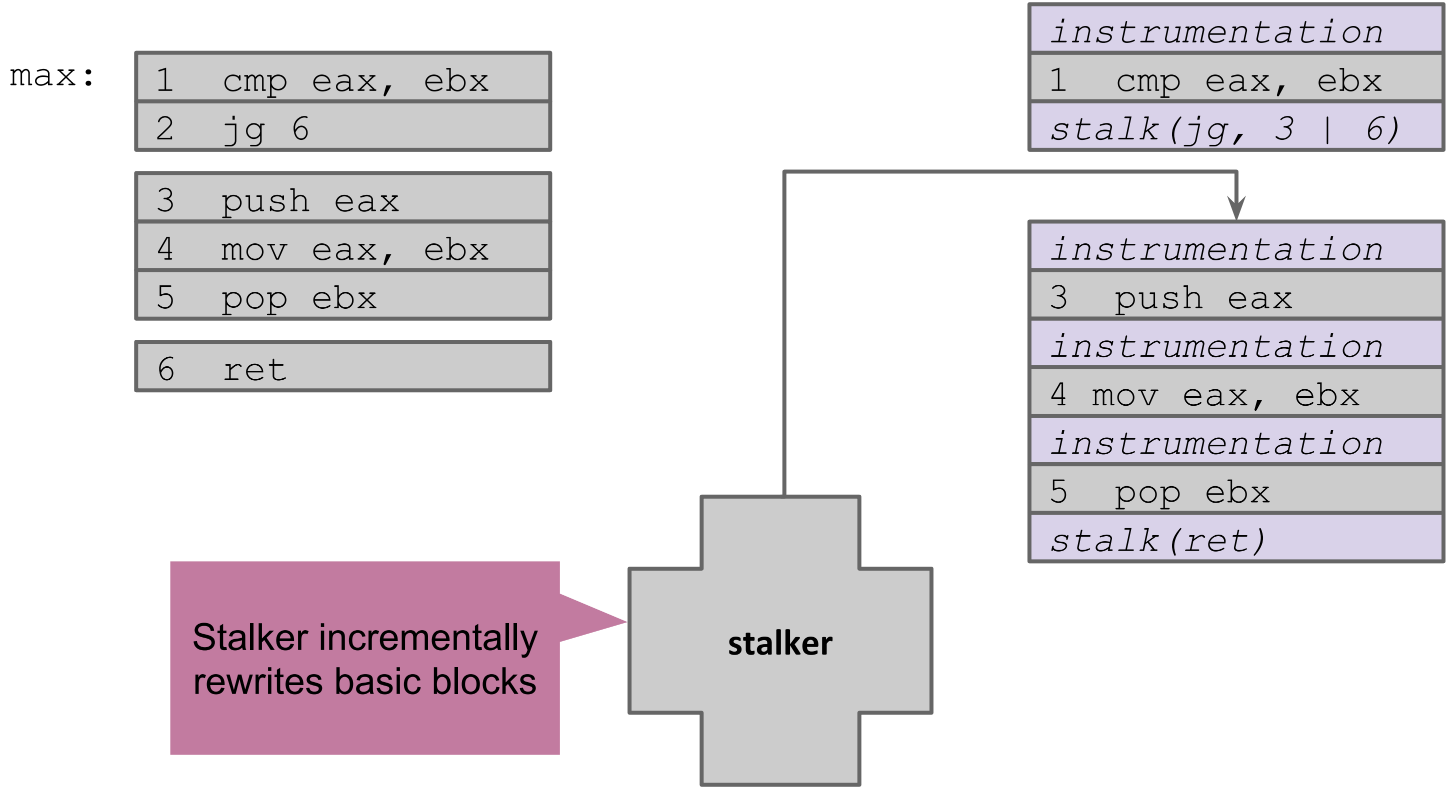
Wrap each instruction with instrumentation

```
instrumentation
1   cmp eax, ebx
instrumentation
```

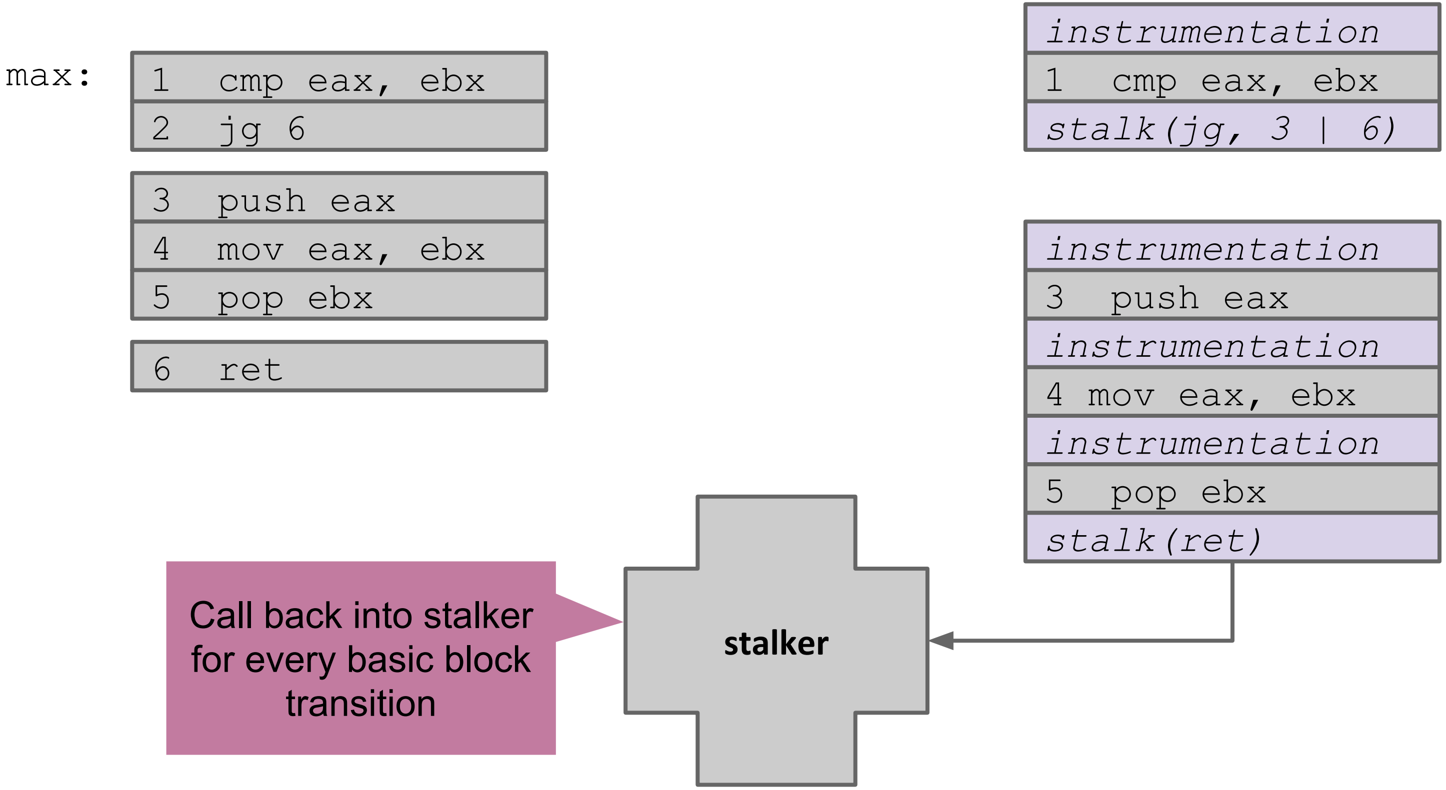
Stalking - the basics



Stalking - the basics



Stalking - the basics



Stalking - challenges

- must decode every instruction
- prologue and epilogue must be “invisible”
- no flags modification
- no stack modification
- no register modification
- self-modifying code
- self-checking code (checksums)
- code that accesses instruction pointer

Stalking - challenges

- must decode every instruction
- prologue and epilogue must be “invisible”
- no flags modification
- no stack m dificatio
- no register m
- self-mod
- self-checking
- code that access ter

Use the source, Luke


Chuck Norris