Enabling coverage-guided binary fuzzing on macOS

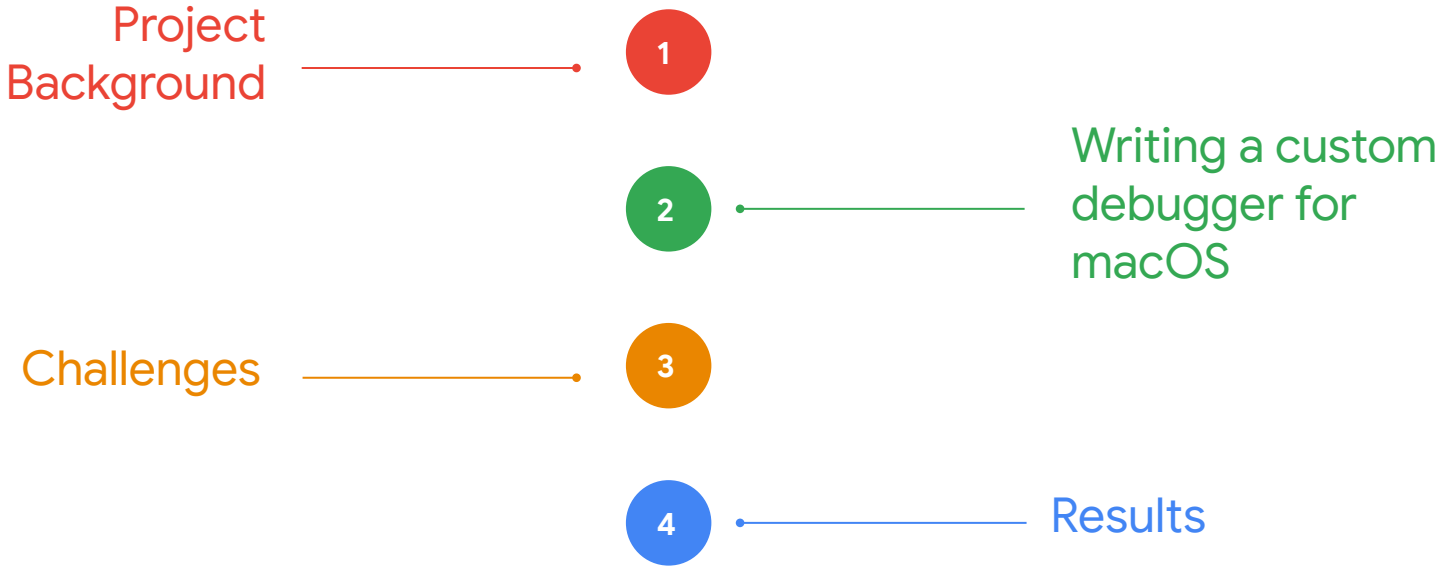
POC x Zer0Con

Alexandru-Vlad Niculae

Alexandru-Vlad Niculae

- Former Google Project Zero Intern
- Computer Science student at University College London
- Passionate about competitive programming

Agenda


Project Background
1
Writing a custom debugger for macOS
2
Challenges
3
4
Results

Project Background

About fuzzing

About fuzzing

What is fuzzing?

“Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks.” - Wikipedia

About fuzzing

What is fuzzing?

“Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks.” - Wikipedia

Fuzzing techniques

“Dumb” fuzzing

- Easy to implement: bit-flipping, inserting/erasing bytes.
- Pretty dumb.

About fuzzing

What is fuzzing?

"Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks." - Wikipedia

Fuzzing techniques

"Dumb" fuzzing

- Easy to implement: bit-flipping, inserting/erasing bytes.
- Pretty dumb.

Grammar-based fuzzing

- Effective when dealing with structured inputs (e.g. Javascript code).
- Requires a lot of non-reusable effort.

About fuzzing

What is fuzzing?

"Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks." - Wikipedia

Fuzzing techniques

"Dumb" fuzzing

- Easy to implement: bit-flipping, inserting/erasing bytes.
- Pretty dumb.

Grammar-based fuzzing

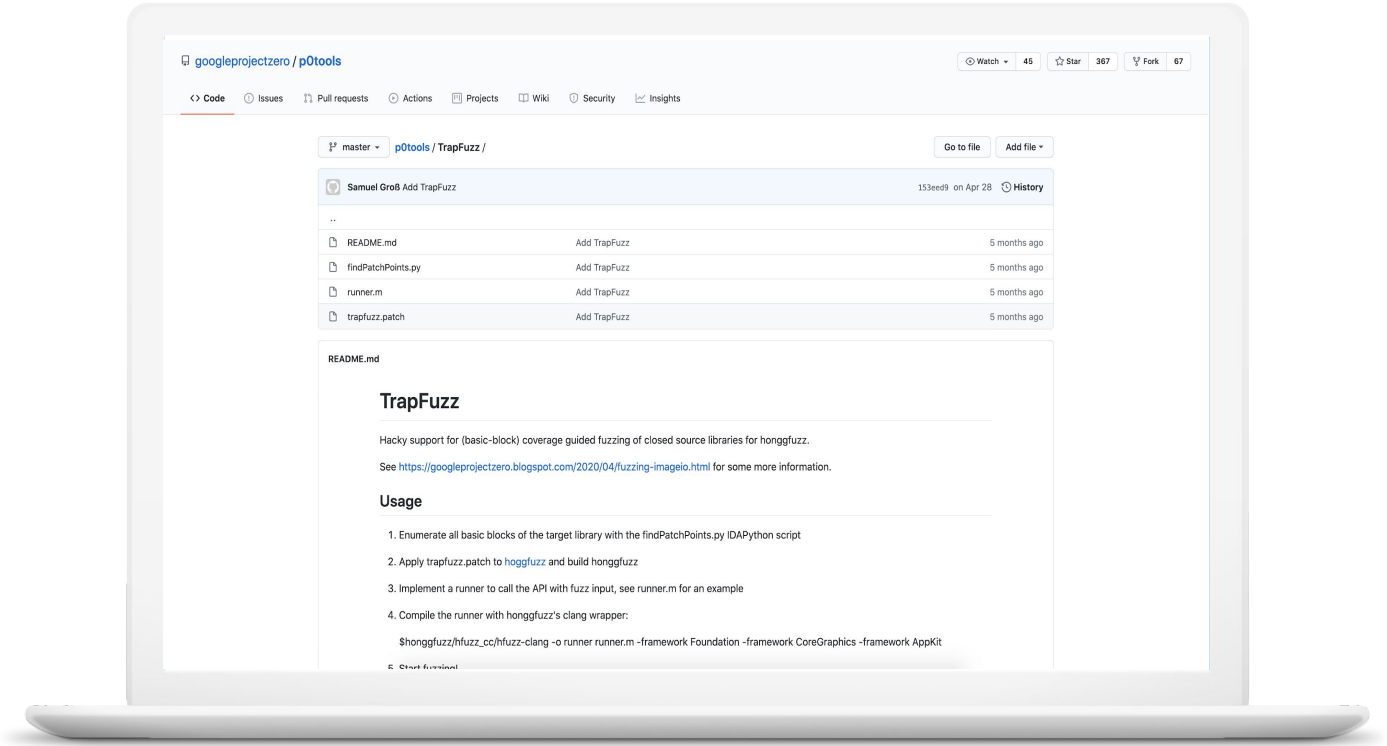
- Effective when dealing with structured inputs (e.g. Javascript code).
- Requires a lot of non-reusable effort.

Coverage-guided fuzzing

- Effective with little effort.
- Needs access to source code.

Coverage-guided fuzzing on binaries?

Coverage-guided fuzzing on binaries?



TinyInst

- Lightweight dynamic binary instrumentation library

“Dynamic binary instrumentation”

Analysing programs at run-time

Analysing programs at machine code level, without having access to source code

The act of adding extra code to a program to measure its performance, diagnose errors and write trace information

TinyInst

- Lightweight dynamic binary instrumentation library

TinyInst

- Lightweight dynamic binary instrumentation library
- Instrument only selected module(s) in the process, while leaving the rest of the process to run natively

TinyInst

- Lightweight dynamic binary instrumentation library
- Instrument only selected module(s) in the process, while leaving the rest of the process to run natively
- Able to extract multiple coverage types on binaries with little overhead
    - Basic block coverage
    - Edge coverage
    - Compare coverage

Code Coverage

“measure used to describe the degree to which the source code of a program is executed when a particular test suite runs” - Wikipedia

Code Coverage

“measure used to describe the degree to which the source code of a program is executed when a particular test suite runs” - Wikipedia

Basic Block Coverage

```
...              (A)
if (...) {
  ...            (B)
}
...              (C)
```

Counts the number of basic blocks reached during runtime.

Code Coverage

“measure used to describe the degree to which the source code of a program is executed when a particular test suite runs” - Wikipedia

Basic Block Coverage

Counts the number of basic blocks reached during runtime.

Edge Coverage

Counts the number of edges hit at runtime.

Code Coverage

“measure used to describe the degree to which the source code of a program is executed when a particular test suite runs” - Wikipedia

Basic Block Coverage

```
...                (A)
if (...) {
  ...              (B)
}
...                (C)
```

Counts the number of basic blocks reached during runtime.

Edge Coverage

Counts the number of edges hit at runtime.

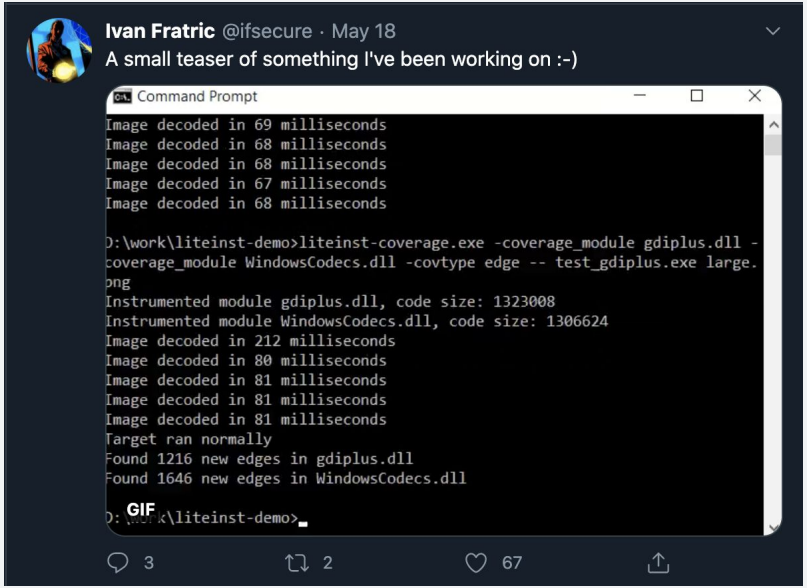
Compare Coverage

```
...                                (A)
if (var == MAGIC_NUMBER) {
  ...                              (B)
}
...                                (C)
```

Counts the number of bytes hit in compare instructions.

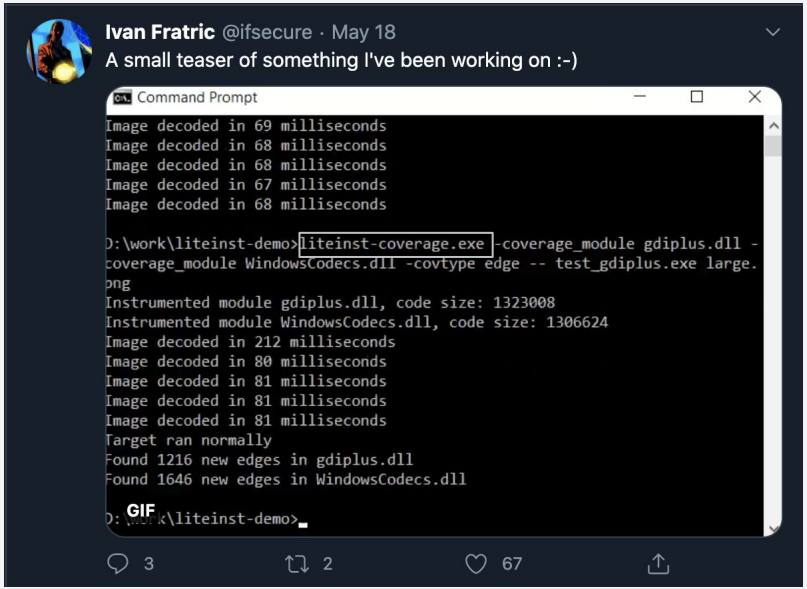
TinyInst

- Lightweight dynamic binary instrumentation library
- Instrument only selected module(s) in the process, while leaving the rest of the process to run natively
- Able to extract coverage on binaries with little overhead
    - Basic block coverage
    - Edge coverage
    - Compare coverage



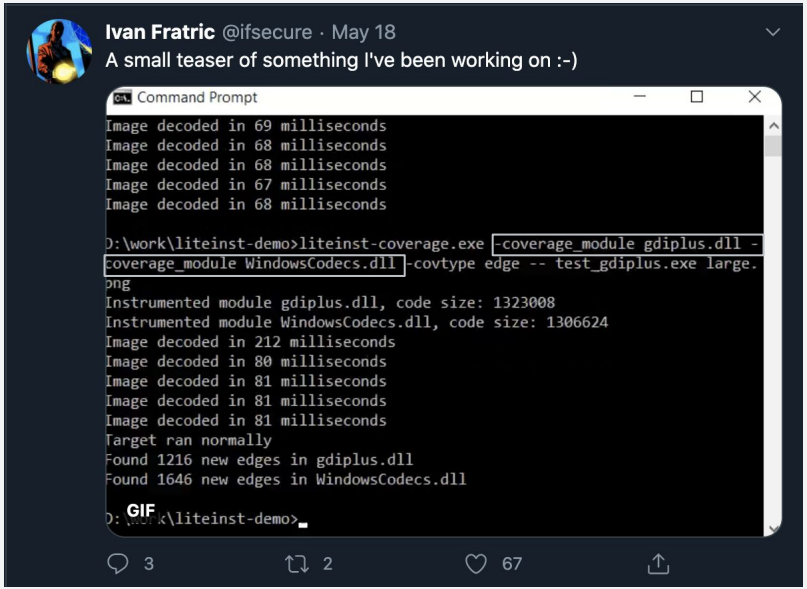
TinyInst

- Lightweight dynamic binary instrumentation library
- Instrument only selected module(s) in the process, while leaving the rest of the process to run natively
- Able to extract coverage on binaries with little overhead
    - Basic block coverage
    - Edge coverage
    - Compare coverage



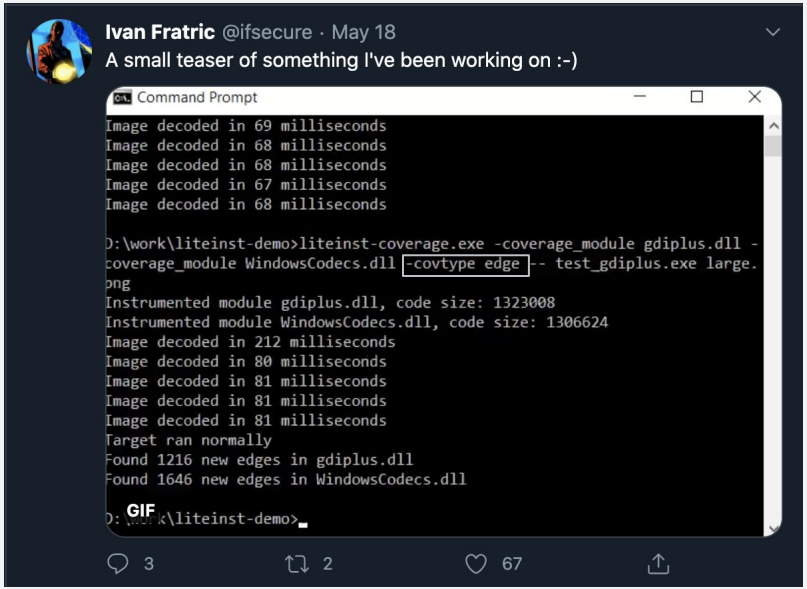
TinyInst

- Lightweight dynamic binary instrumentation library
- Instrument only selected module(s) in the process, while leaving the rest of the process to run natively
- Able to extract coverage on binaries with little overhead
    - Basic block coverage
    - Edge coverage
    - Compare coverage



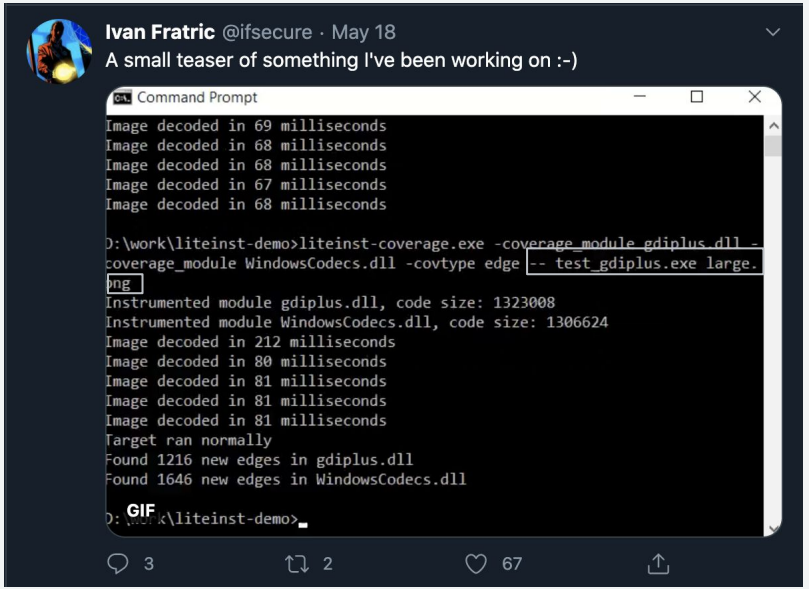
TinyInst

- Lightweight dynamic binary instrumentation library
- Instrument only selected module(s) in the process, while leaving the rest of the process to run natively
- Able to extract coverage on binaries with little overhead
    - Basic block coverage
    - Edge coverage
    - Compare coverage



TinyInst

- Lightweight dynamic binary instrumentation library
- Instrument only selected module(s) in the process, while leaving the rest of the process to run natively
- Able to extract coverage on binaries with little overhead
    - Basic block coverage
    - Edge coverage
    - Compare coverage



How is TinyInst instrumenting binaries?

How is TinyInst instrumenting binaries?

Debugger

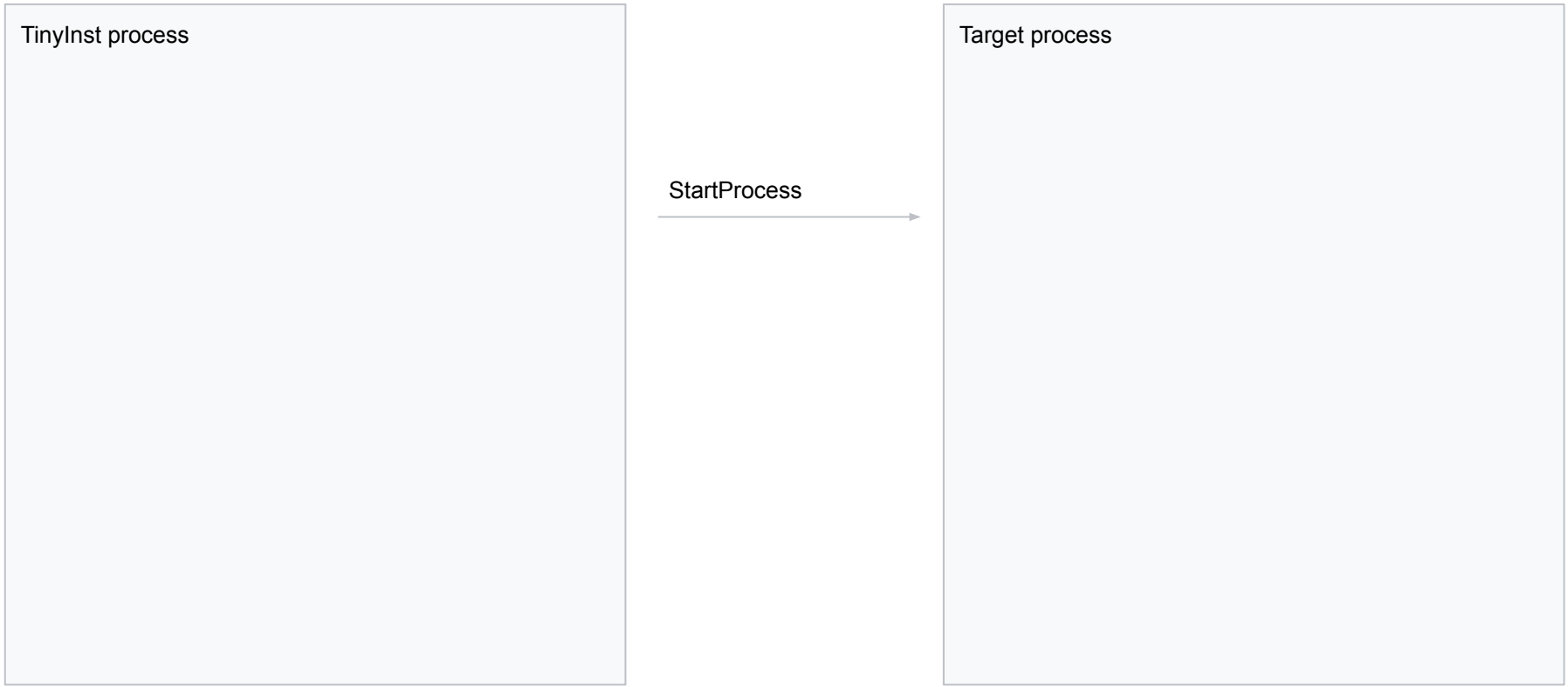
Runs the target process

Monitors exceptions

TinyInst

Instrument selected modules

Target process started



Module loaded into target process (green indicates executable memory)

TinyInst process

Target process

module.dll:

```
00000001800889F4  mov     [rsp+8], rbx
00000001800889F9  push    rdi
00000001800889FA  sub     rsp, 30h
00000001800889FE  mov     rdi, rcx
0000000180088A01  mov     rdx, rcx
0000000180088A04  xor     ecx, ecx
0000000180088A06  mov     r8d, 104h
```

Module is getting instrumented (either when loaded, or when target method is reached or when process entrypoint is reached)

TinyInst process

Target process

module.dll:

```
00000001800889F4  mov     [rsp+8], rbx
00000001800889F9  push    rdi
00000001800889FA  sub     rsp, 30h
00000001800889FE  mov     rdi, rcx
0000000180088A01  mov     rdx, rcx
0000000180088A04  xor     ecx, ecx
0000000180088A06  mov     r8d, 104h
```

Module is getting instrumented: 1. Module is analyzed and executable code copied to the TinyInst process (for speed of later access).

TinyInst process

ModuleInfo.AddressRange.data:

```
00000001800889F4  mov     [rsp+8], rbx
00000001800889F9  push    rdi
00000001800889FA  sub     rsp, 30h
00000001800889FE  mov     rdi, rcx
0000000180088A01  mov     rdx, rcx
0000000180088A04  xor     ecx, ecx
0000000180088A06  mov     r8d, 104h
```

ReadProcessMemory

mach_vm_read

Target process

module.dll:

```
00000001800889F4  mov     [rsp+8], rbx
00000001800889F9  push    rdi
00000001800889FA  sub     rsp, 30h
00000001800889FE  mov     rdi, rcx
0000000180088A01  mov     rdx, rcx
0000000180088A04  xor     ecx, ecx
0000000180088A06  mov     r8d, 104h
```

Module is getting instrumented: 2. Mark executable memory within the instrumented module within the target process as non-executable

TinyInst process

ModuleInfo.AddressRange.data:

```
00000001800889F4  mov     [rsp+8], rbx
00000001800889F9  push    rdi
00000001800889FA  sub     rsp, 30h
00000001800889FE  mov     rdi, rcx
0000000180088A01  mov     rdx, rcx
0000000180088A04  xor     ecx, ecx
0000000180088A06  mov     r8d, 104h
```

VirtualProtectEx

mach_vm_protect

Target process

module.dll:

```
00000001800889F4  mov     [rsp+8], rbx
00000001800889F9  push    rdi
00000001800889FA  sub     rsp, 30h
00000001800889FE  mov     rdi, rcx
0000000180088A01  mov     rdx, rcx
0000000180088A04  xor     ecx, ecx
0000000180088A06  mov     r8d, 104h
```

Module is getting instrumented: 3. Allocate two buffers for instrumented code, one in the TinyInst process, another (as executable) in the target process

TinyInst process

ModuleInfo.AddressRange.data:

```
00000001800889F4  mov     [rsp+8], rbx
00000001800889F9  push    rdi
00000001800889FA  sub     rsp, 30h
00000001800889FE  mov     rdi, rcx
0000000180088A01  mov     rdx, rcx
0000000180088A04  xor     ecx, ecx
0000000180088A06  mov     r8d, 104h
```

instrumented_code_local

VirtualAllocEx

mach_vm_allocate,
mach_vm_protect

Target process

module.dll:

```
00000001800889F4  mov     [rsp+8], rbx
00000001800889F9  push    rdi
00000001800889FA  sub     rsp, 30h
00000001800889FE  mov     rdi, rcx
0000000180088A01  mov     rdx, rcx
0000000180088A04  xor     ecx, ecx
0000000180088A06  mov     r8d, 104h
```

instrumented_code_remote

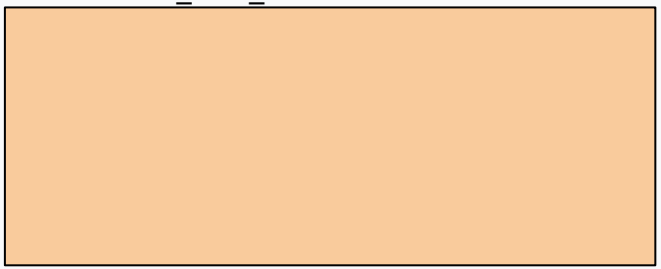
Instrumentation is now "complete", target process is executing and we are waiting for code in the instrumented module to start executing

TinyInst process

ModuleInfo.AddressRange.data:

```
00000001800889F4  mov     [rsp+8], rbx
00000001800889F9  push    rdi
00000001800889FA  sub     rsp, 30h
00000001800889FE  mov     rdi, rcx
0000000180088A01  mov     rdx, rcx
0000000180088A04  xor     ecx, ecx
0000000180088A06  mov     r8d, 104h
```

instrumented_code_local

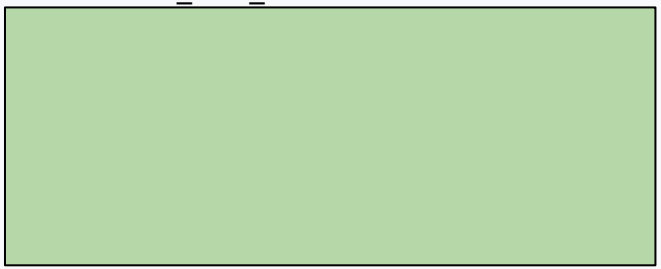


Target process

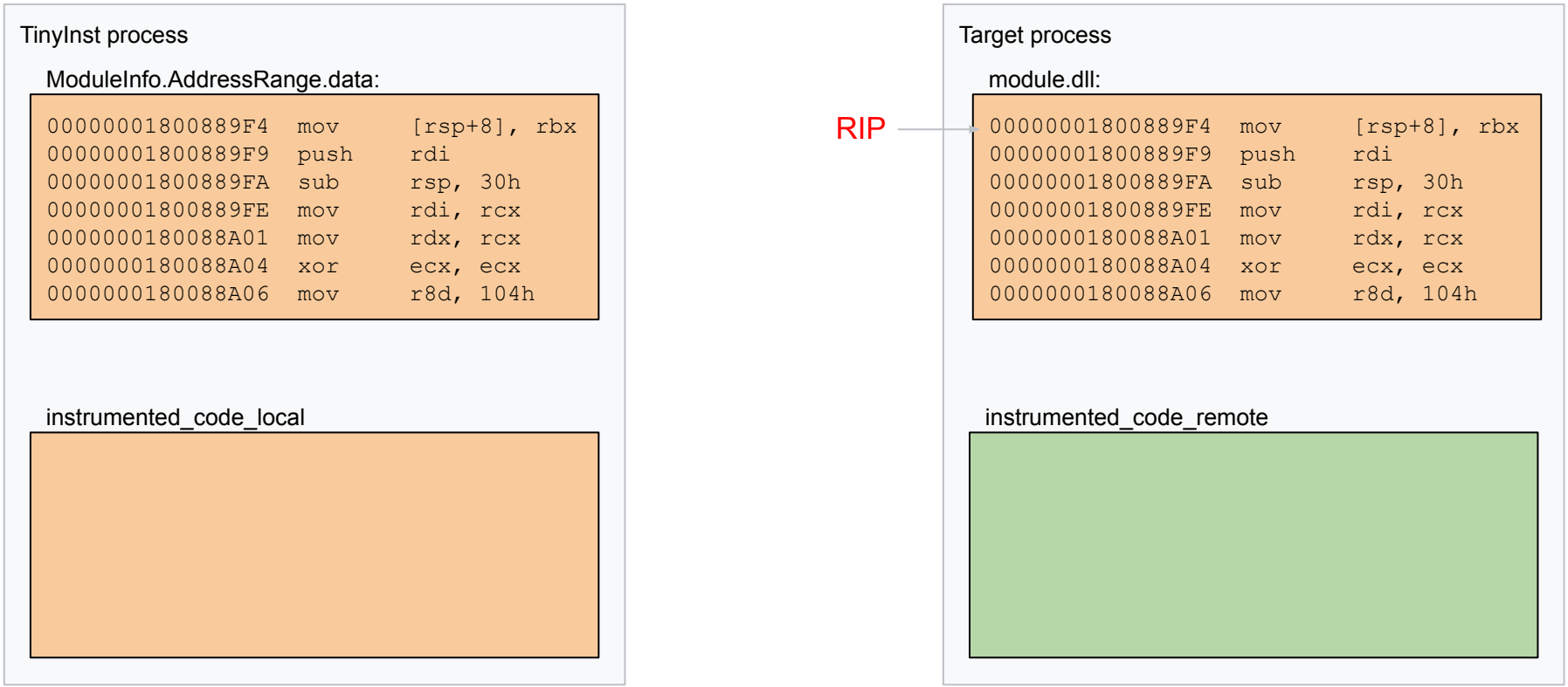
module.dll:

```
00000001800889F4  mov     [rsp+8], rbx
00000001800889F9  push    rdi
00000001800889FA  sub     rsp, 30h
00000001800889FE  mov     rdi, rcx
0000000180088A01  mov     rdx, rcx
0000000180088A04  xor     ecx, ecx
0000000180088A06  mov     r8d, 104h
```

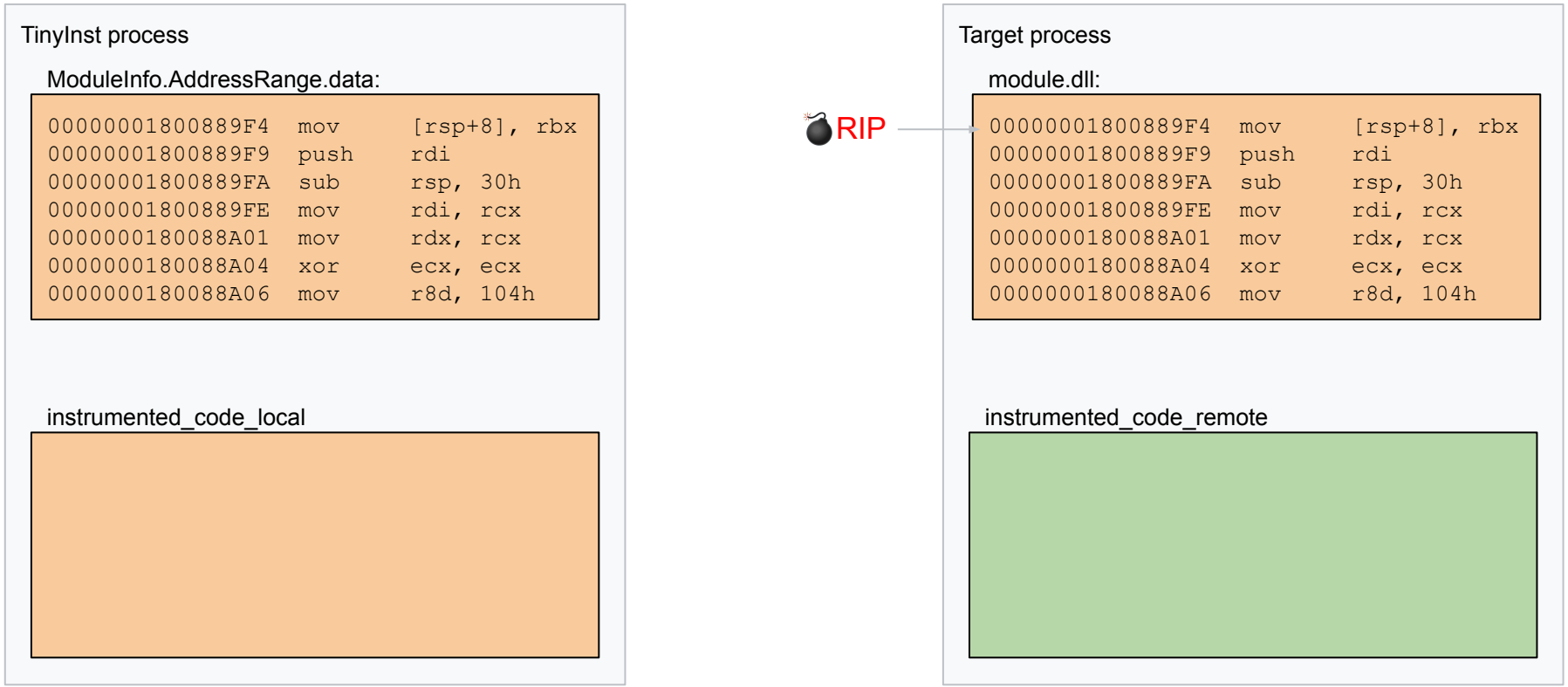
instrumented_code_remote



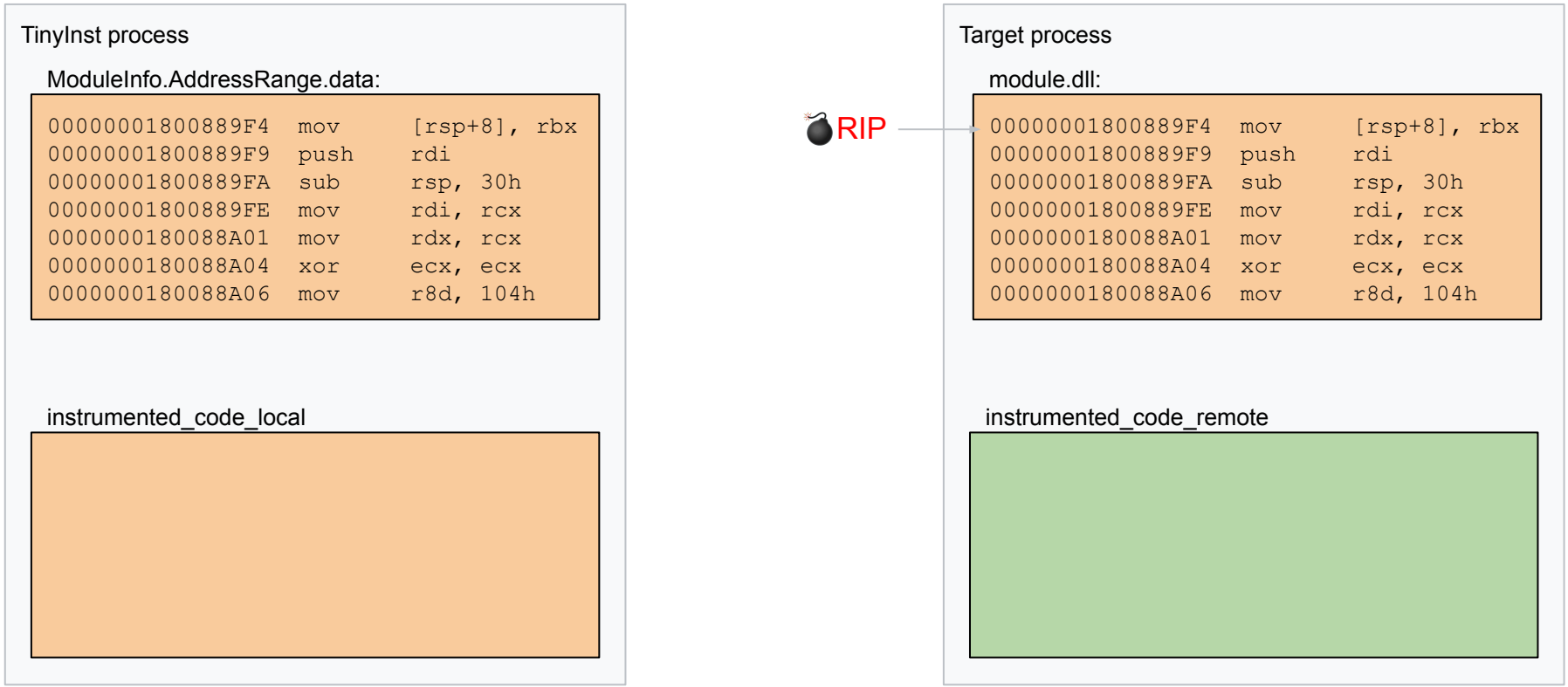
Target wants to execute code in an "instrumented" module



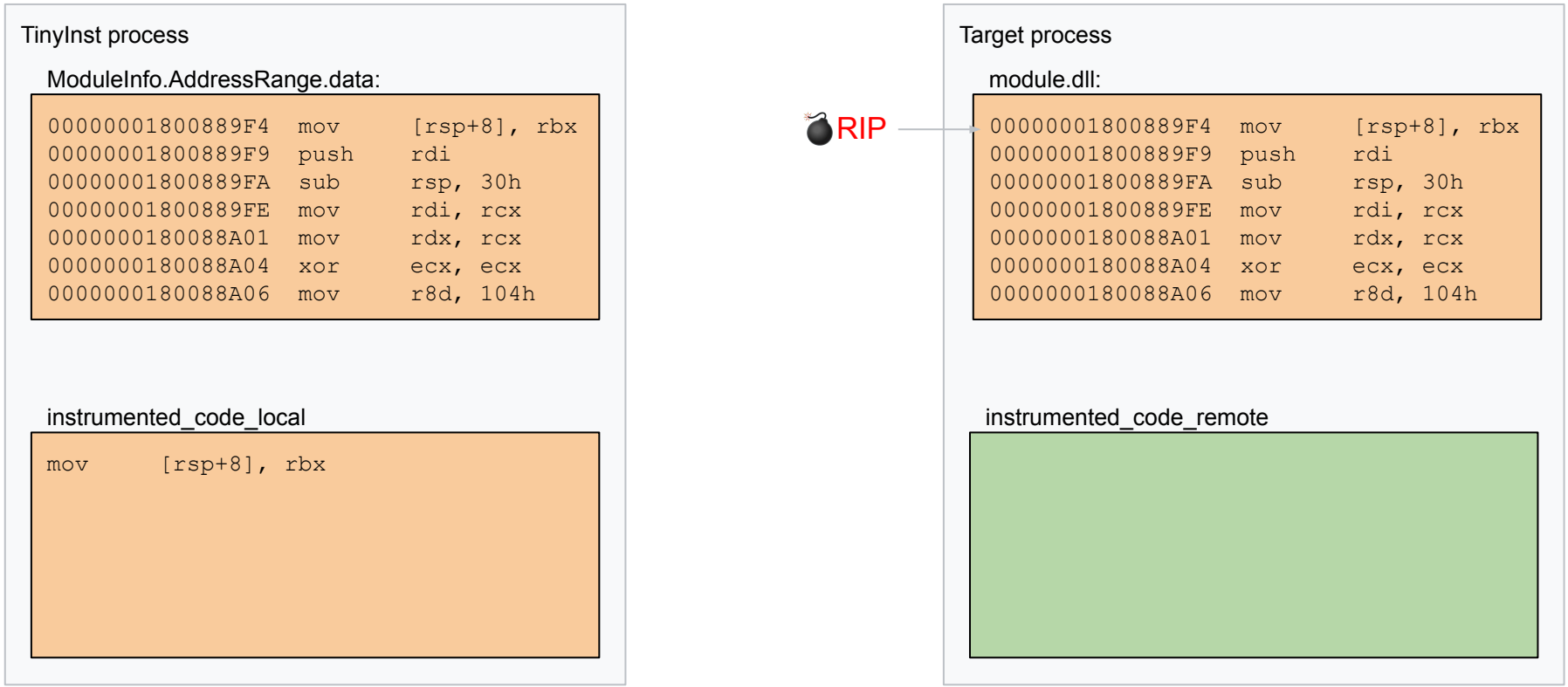
This causes an exception as it's trying to execute memory which is not executable



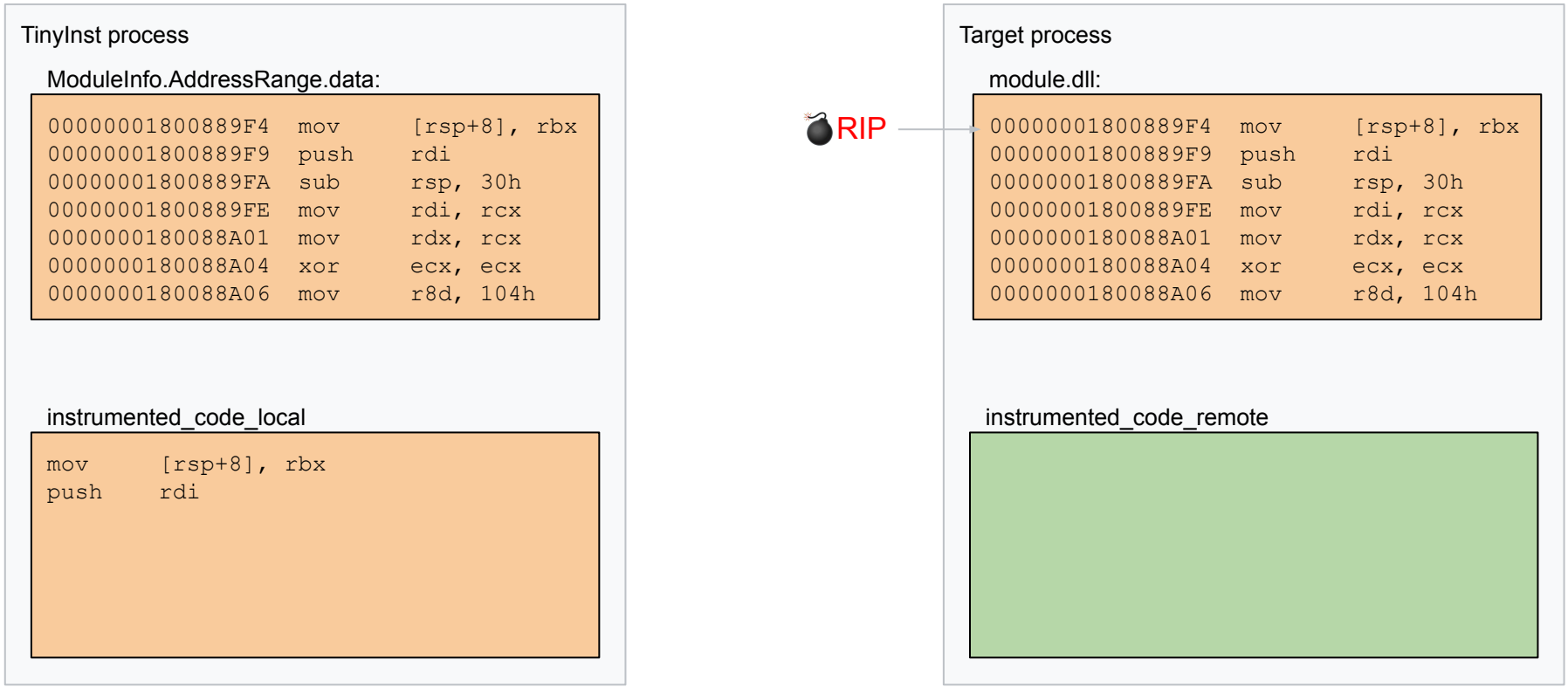
An exception is caught by the TinyInst process. We start rewriting (instrumenting) the code starting at the address that caused the exception.



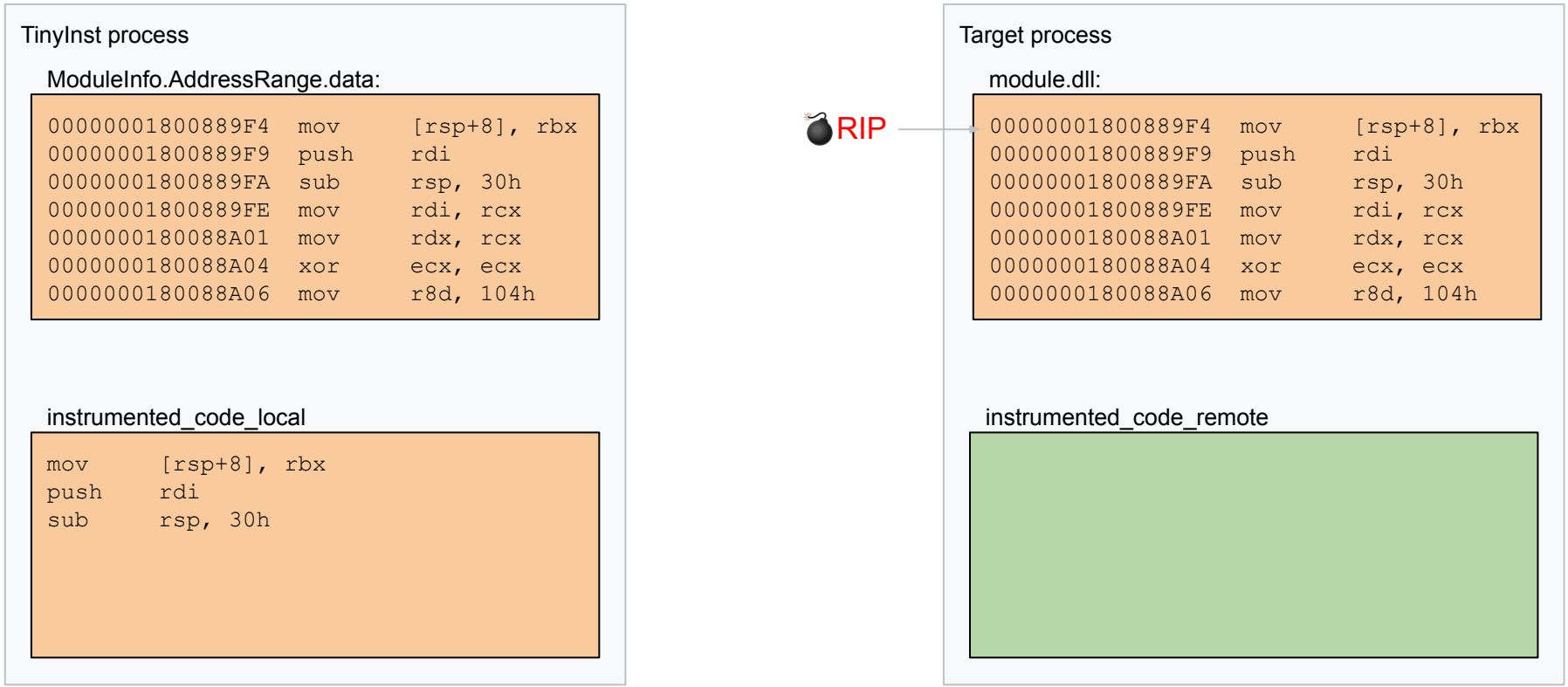
An exception is caught by the TinyInst process. We start rewriting (instrumenting) the code starting with the current basic block. In this example, we are not inserting any extra instructions.



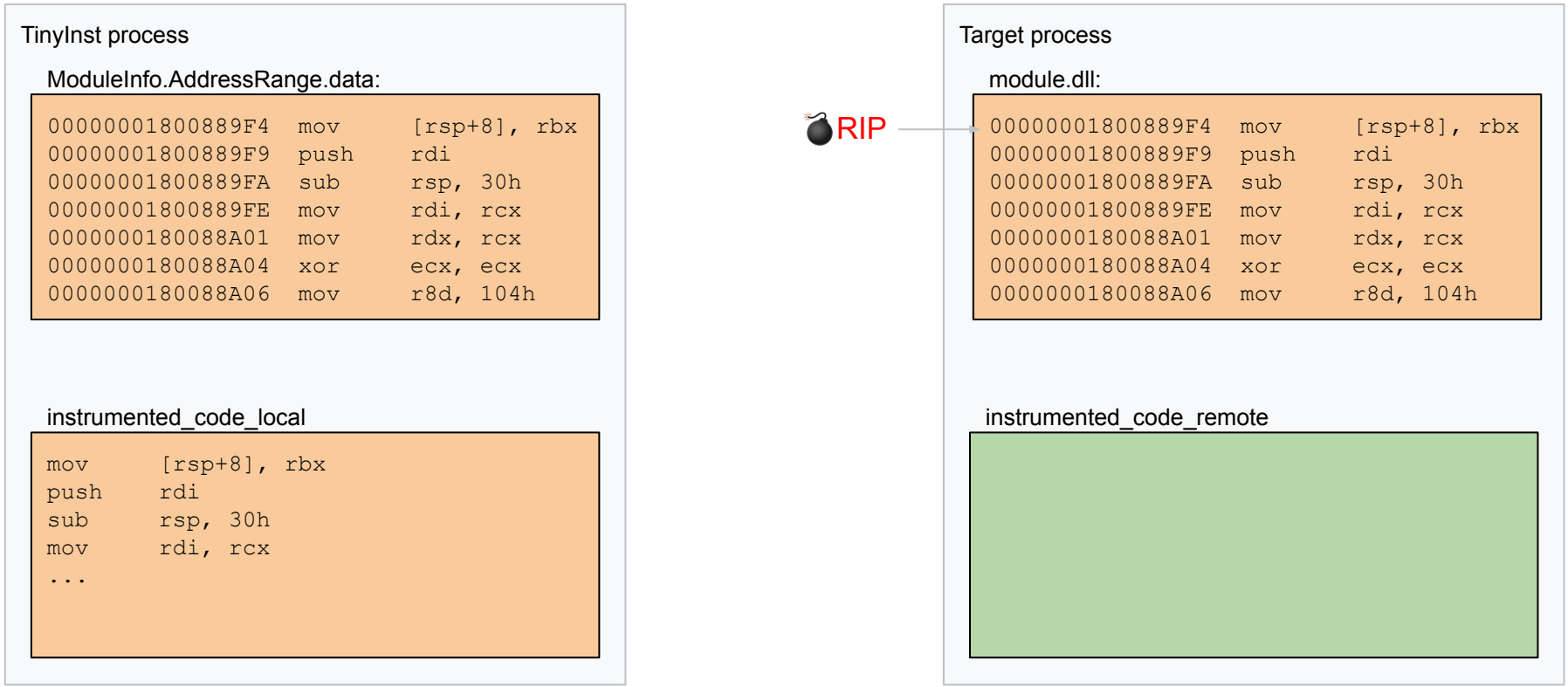
An exception is caught by the TinyInst process. We start rewriting (instrumenting) the code starting with the current basic block. In this example, we are not inserting any extra instructions.



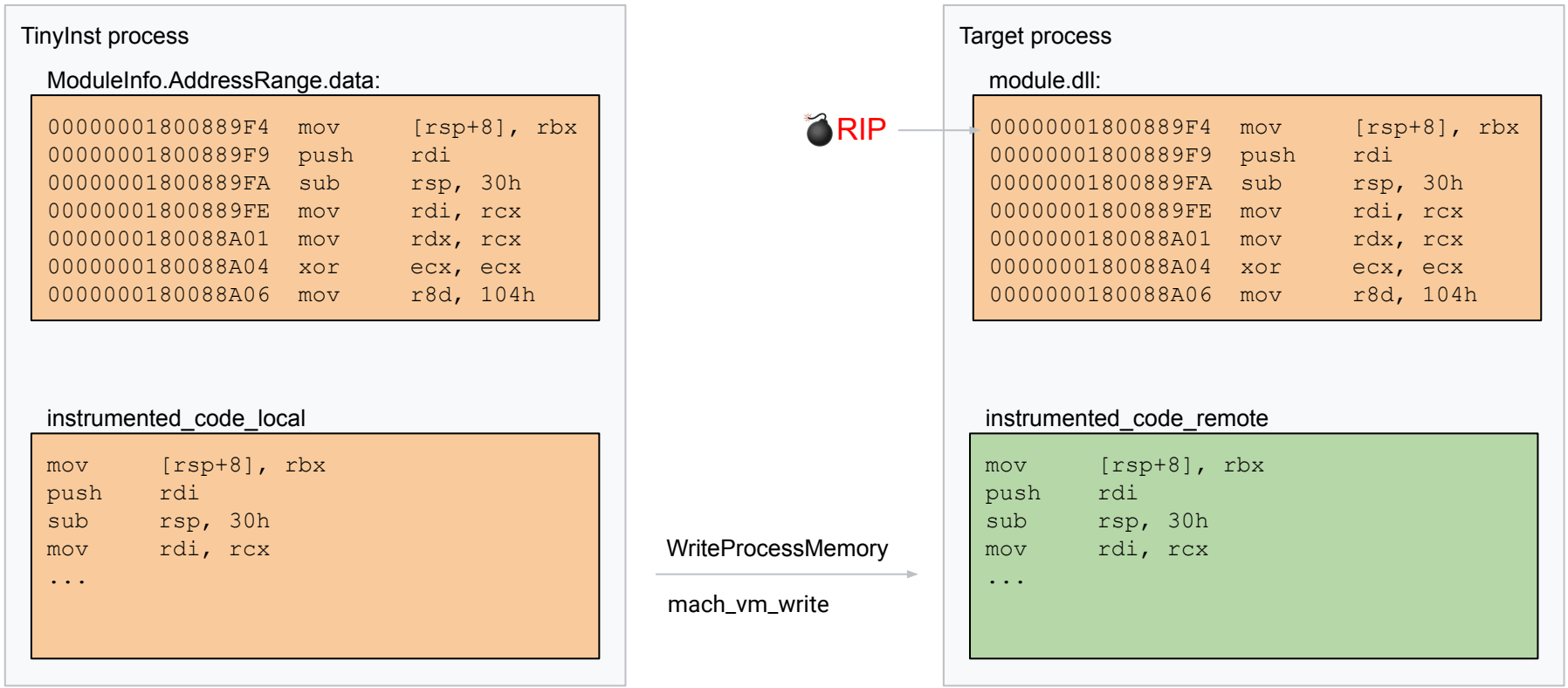
An exception is caught by the TinyInst process. We start rewriting (instrumenting) the code starting with the current basic block. In this example, we are not inserting any extra instructions.



An exception is caught by the TinyInst process. We start rewriting (instrumenting) the code starting with the current basic block. In this example, we are not inserting any extra instructions.



Once we are done, we: 1. Copy the instrumented code to the remote buffer



Once we are done, we: 2. Change the RIP register of the exception thread to point to the instrumented code

TinyInst process

ModuleInfo.AddressRange.data:

```
00000001800889F4  mov     [rsp+8], rbx
00000001800889F9  push    rdi
00000001800889FA  sub     rsp, 30h
00000001800889FE  mov     rdi, rcx
0000000180088A01  mov     rdx, rcx
0000000180088A04  xor     ecx, ecx
0000000180088A06  mov     r8d, 104h
```

instrumented_code_local

```
mov     [rsp+8], rbx
push    rdi
sub     rsp, 30h
mov     rdi, rcx
...
```

✔RIP

Target process

module.dll:

```
00000001800889F4  mov     [rsp+8], rbx
00000001800889F9  push    rdi
00000001800889FA  sub     rsp, 30h
00000001800889FE  mov     rdi, rcx
0000000180088A01  mov     rdx, rcx
0000000180088A04  xor     ecx, ecx
0000000180088A06  mov     r8d, 104h
```

instrumented_code_remote

```
mov     [rsp+8], rbx
push    rdi
sub     rsp, 30h
mov     rdi, rcx
...
```

Target continues executing until some other event (e.g. another exception) occurs

TinyInst process

ModuleInfo.AddressRange.data:

```
00000001800889F4  mov     [rsp+8], rbx
00000001800889F9  push    rdi
00000001800889FA  sub     rsp, 30h
00000001800889FE  mov     rdi, rcx
0000000180088A01  mov     rdx, rcx
0000000180088A04  xor     ecx, ecx
0000000180088A06  mov     r8d, 104h
```

instrumented_code_local

```
mov     [rsp+8], rbx
push    rdi
sub     rsp, 30h
mov     rdi, rcx
...
```

RIP

Target process

module.dll:

```
00000001800889F4  mov     [rsp+8], rbx
00000001800889F9  push    rdi
00000001800889FA  sub     rsp, 30h
00000001800889FE  mov     rdi, rcx
0000000180088A01  mov     rdx, rcx
0000000180088A04  xor     ecx, ecx
0000000180088A06  mov     r8d, 104h
```

instrumented_code_remote

```
mov     [rsp+8], rbx
push    rdi
sub     rsp, 30h
mov     rdi, rcx
...
```

How is TinyInst used to extract code coverage?

How is TinyInst used to extract code coverage?

Debugger

Runs the target process

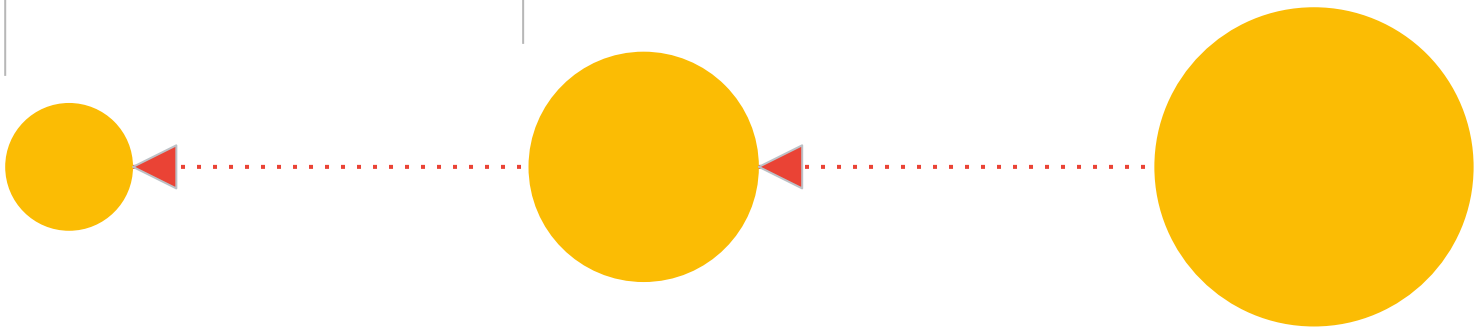
Monitors exceptions

TinyInst

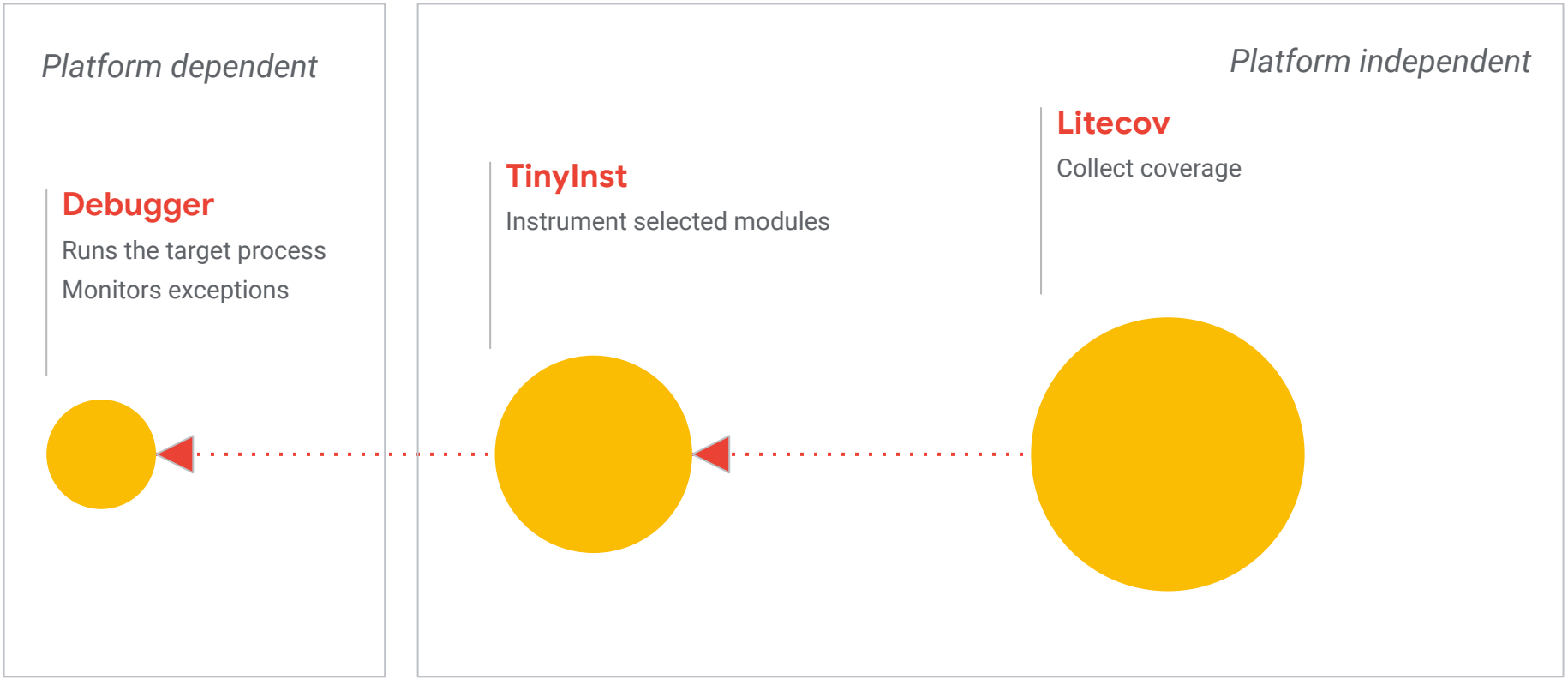
Instrument selected modules

Litecov

Collect coverage



How is TinyInst used to extract code coverage?



Writing a custom debugger for macOS

What should our debugger do?

What should our debugger do?

- Start a new process in a suspended state / Attach to an existing process

What should our debugger do?

- Start a new process in a suspended state / Attach to an existing process
- Handle exceptions in the child process

What should our debugger do?

- Start a new process in a suspended state / Attach to an existing process
- Handle exceptions in the child process
- List libraries loaded in the process, determine the address range at which each library is loaded

What should our debugger do?

- Start a new process in a suspended state / Attach to an existing process
- Handle exceptions in the child process
- List libraries loaded in the process, determine the address range at which each library is loaded
- Get notified when libraries are loaded/unloaded

What should our debugger do?

- Start a new process in a suspended state / Attach to an existing process
- Handle exceptions in the child process
- List libraries loaded in the process, determine the address range at which each library is loaded
- Get notified when libraries are loaded/unloaded
- Examine and manipulate the memory of the target process

Available resources

Available resources

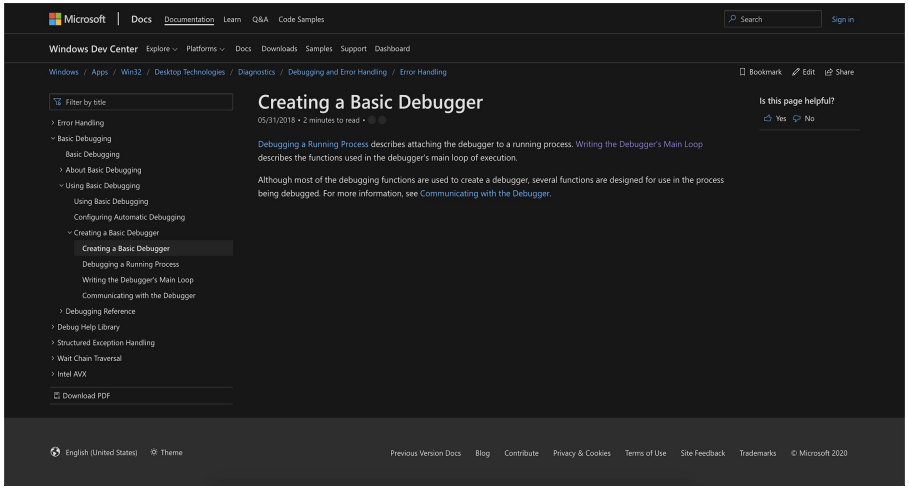
macOS

Windows

Available resources

macOS

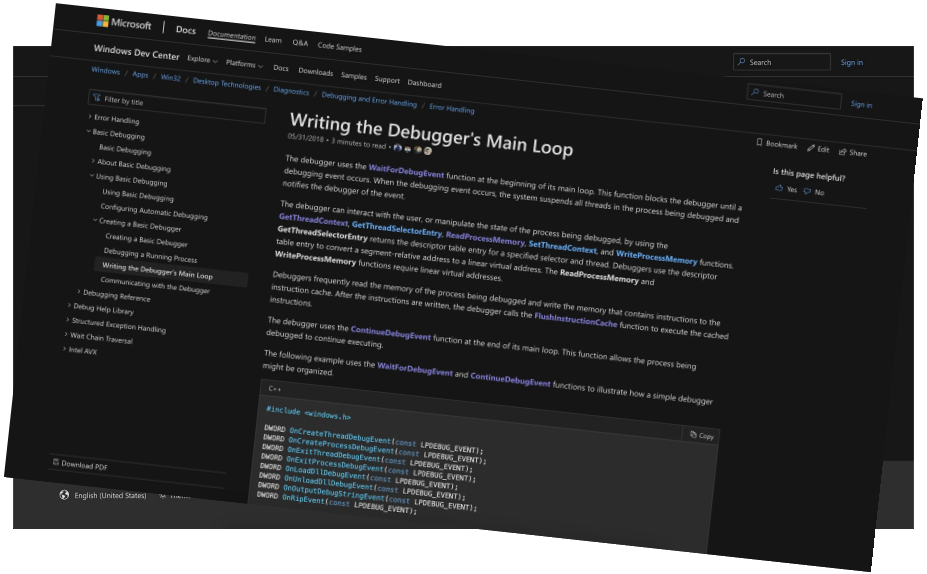
Windows



Available resources

macOS

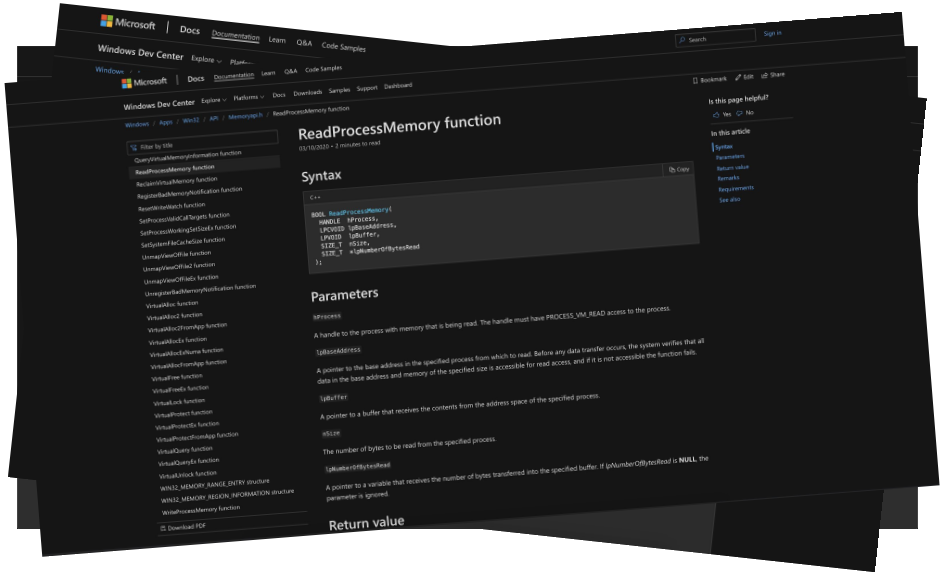
Windows



Available resources

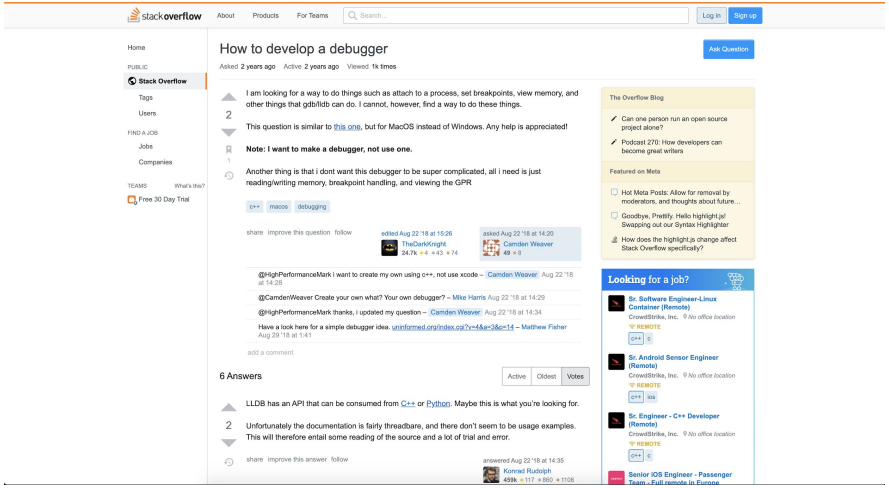
macOS

Windows

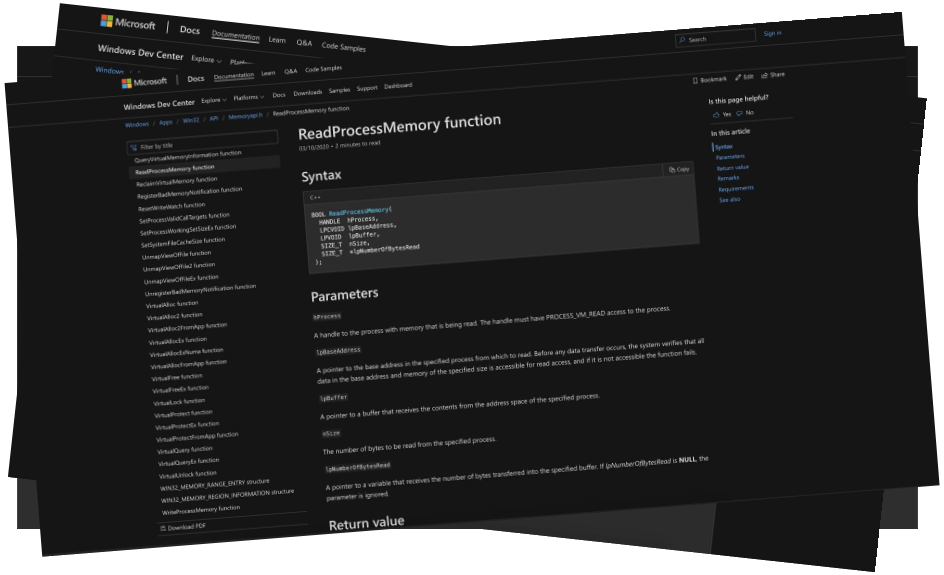


Available resources

macOS

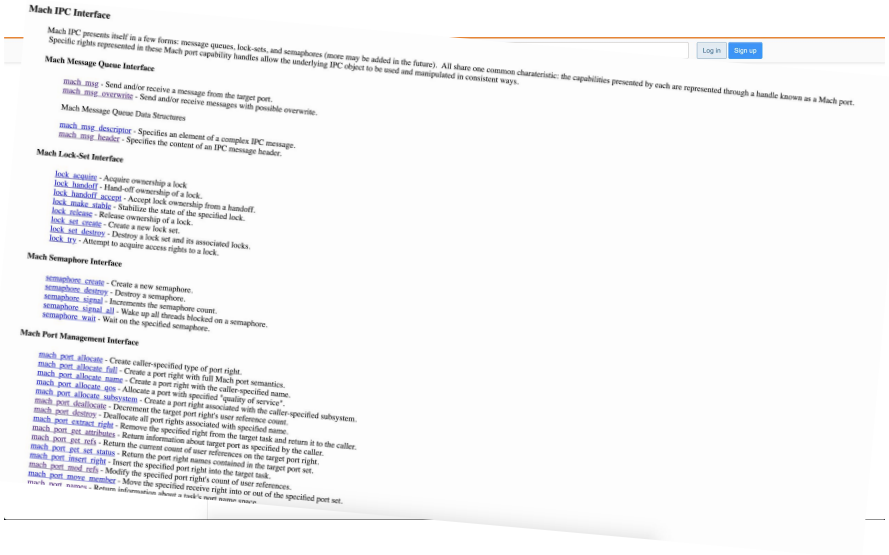


Windows

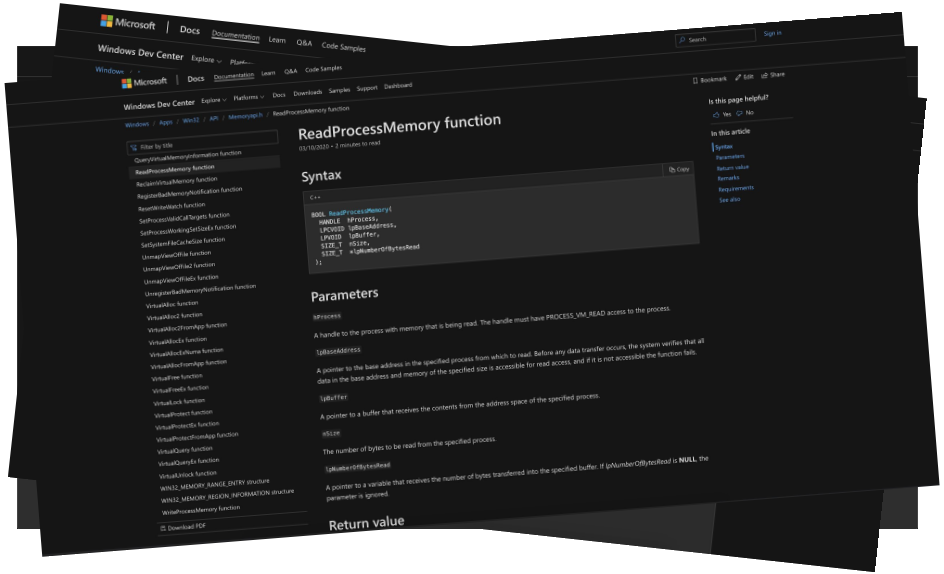


Available resources

macOS

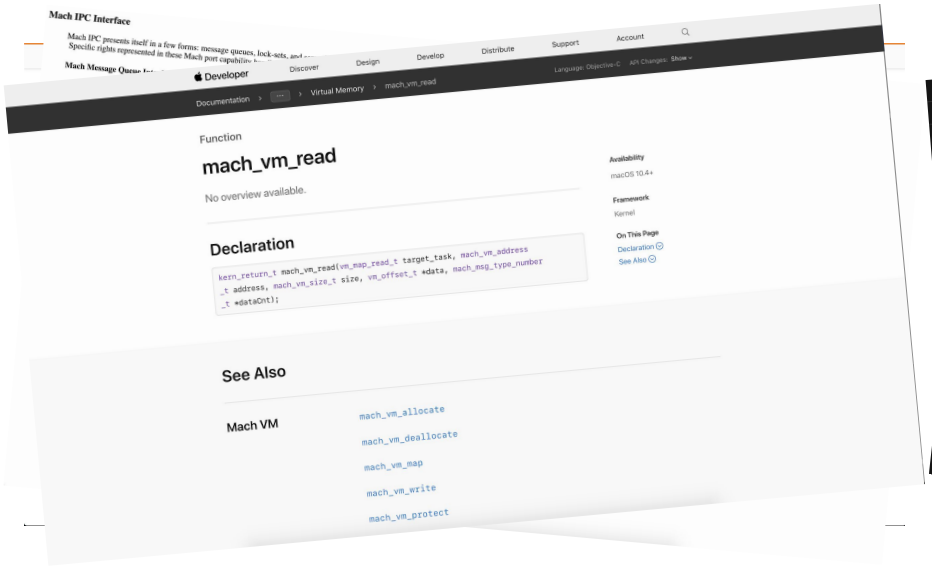


Windows

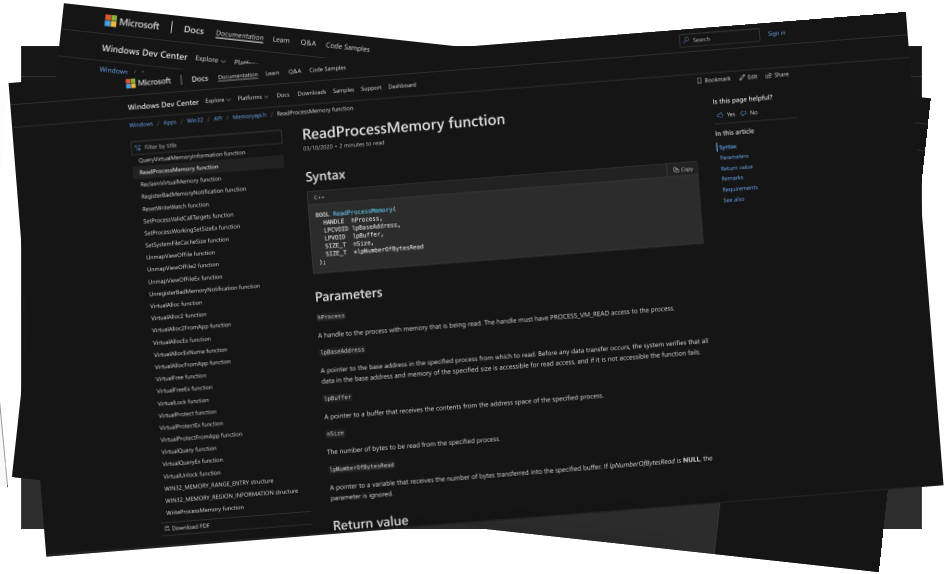


Available resources

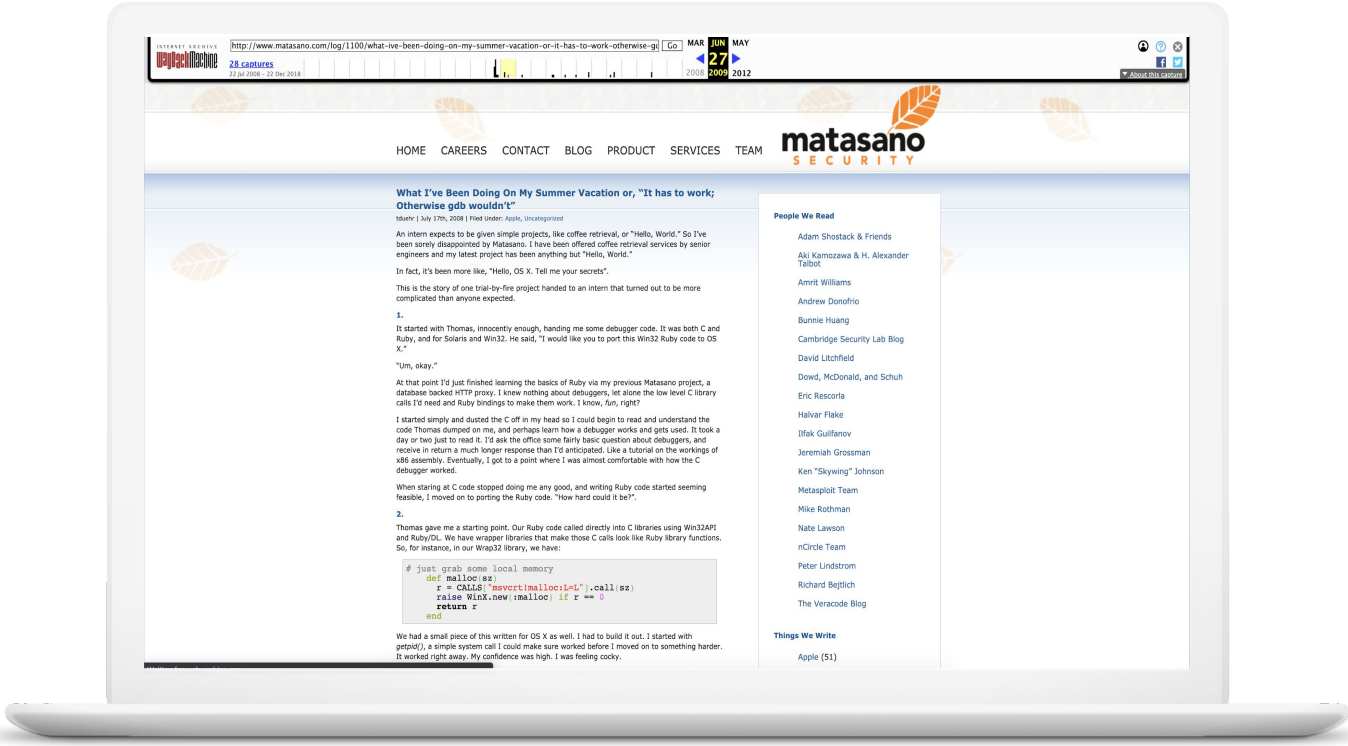
macOS



Windows

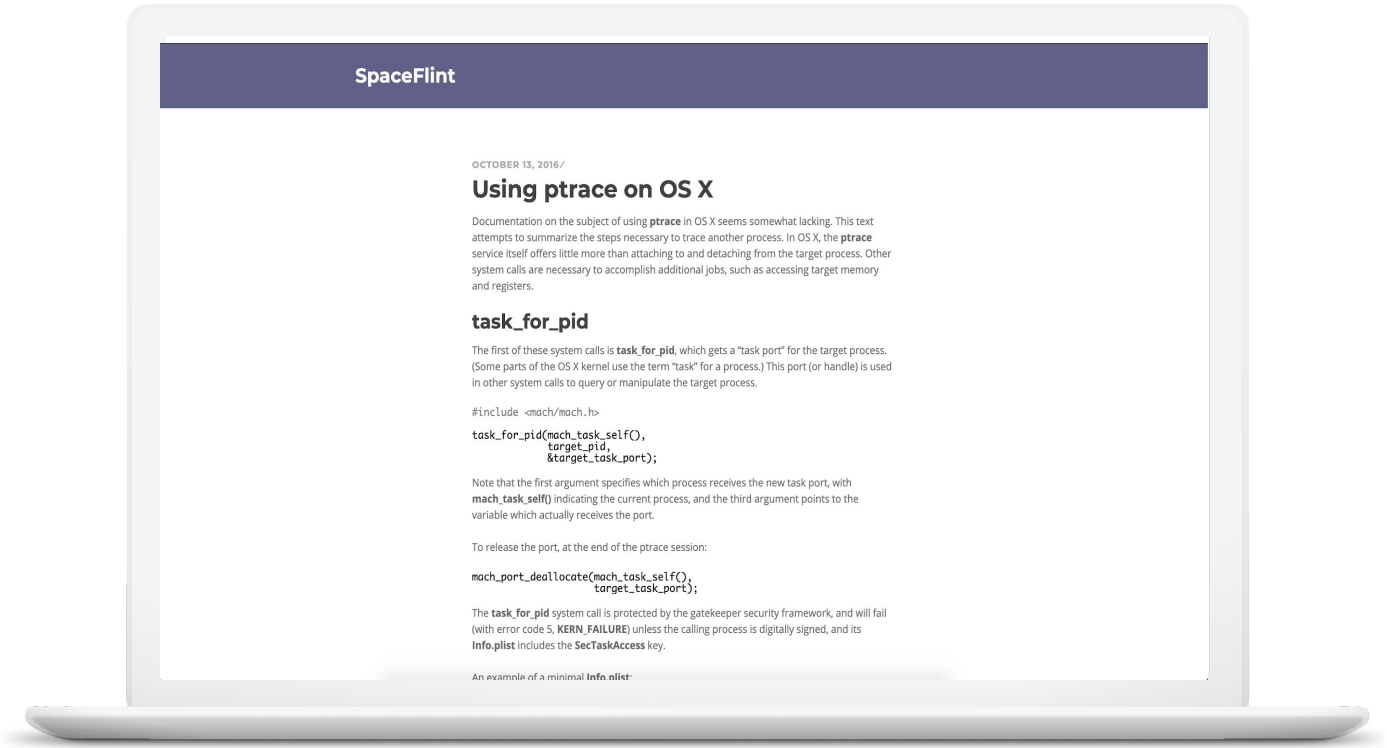


Available resources



http://web.archive.org/web/20090627062246/http://www.matasano.com/log/1100/what-ive-been-doing-on-my-summer-vacation-or-it-has-to-work-otherwise-gdb-wouldnt/

Available resources



https://www.spaceflint.com/?p=150

ptrace on Linux

ptrace on Linux

- Observe and control the execution of another process

ptrace on Linux

- Observe and control the execution of another process
- Examine and change the memory and registers of another process

ptrace on Linux

- Observe and control the execution of another process
- Examine and change the memory and registers of another process
- Used to implement breakpoint debugging and system call tracing

ptrace on macOS

- Observe and control the [illegible] of another process
- Examine and change [illegible] registers of another process
- Used to implement bre[illegible]ugging and system call tracing

ptrace on macOS

- Observe and control the [illegible] of another process
- Examine and change [illegible] registers of another process
- Used to implement bre[illegible]ugging and system call tracing

- Very limited on macOS

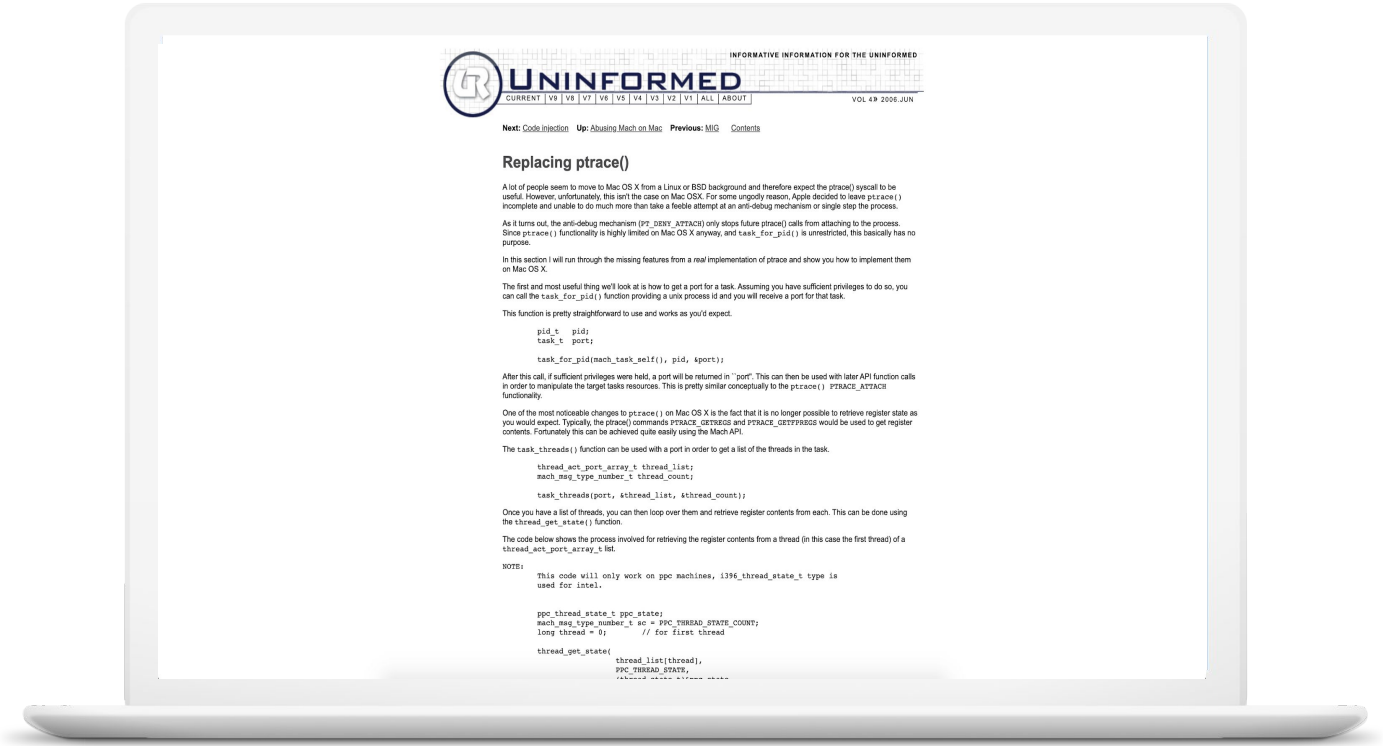
ptrace on macOS

- Observe and control the [illegible] of another process
- Examine and change [illegible] registers of another process
- Used to implement bre[illegible]ugging and system call tracing

- Very limited on macOS
- Used minimally by LLDB and GDB
  - preventing the process from exiting on signals
  - passing signals to the child after it processed them

How to replace the lost ptrace functionality?

How to replace the lost ptrace functionality?



INFORMATIVE INFORMATION FOR THE UNINFORMED

UNINFORMED

CURRENT | V9 | V8 | V7 | V6 | V5 | V4 | V3 | V2 | V1 | ALL | ABOUT

VOL 4 @ 2006.JUN

Next: Code injection **Up:** Abusing Mach on Mac **Previous:** MIG Contents

Replacing ptrace()

A lot of people seem to move to Mac OS X from a Linux or BSD background and therefore expect the ptrace() syscall to be useful. However, unfortunately, this isn't the case on Mac OSX. For some ungodly reason, Apple decided to leave `ptrace()` incomplete and unable to do much more than take a feeble attempt at an anti-debug mechanism or single step the process.

As it turns out, the anti-debug mechanism (`PT_DENY_ATTACH`) only stops future ptrace() calls from attaching to the process. Since `ptrace()` functionality is highly limited on Mac OS X anyway, and `task_for_pid()` is unrestricted, this basically has no purpose.

In this section I will run through the missing features from a *real* implementation of ptrace and show you how to implement them on Mac OS X.

The first and most useful thing we'll look at is how to get a port for a task. Assuming you have sufficient privileges to do so, you can call the `task_for_pid()` function providing a unix process id and you will receive a port for that task.

This function is pretty straightforward to use and works as you'd expect.

```
pid_t   pid;
task_t  port;

task_for_pid(mach_task_self(), pid, &port);
```

After this call, if sufficient privileges were held, a port will be returned in ``port". This can then be used with later API function calls in order to manipulate the target tasks resources. This is pretty similar conceptually to the `ptrace() PTRACE_ATTACH` functionality.

One of the most noticeable changes to `ptrace()` on Mac OS X is the fact that it is no longer possible to retrieve register state as you would expect. Typically, the ptrace() commands `PTRACE_GETREGS` and `PTRACE_GETFPREGS` would be used to get register contents. Fortunately this can be achieved quite easily using the Mach API.

The `task_threads()` function can be used with a port in order to get a list of the threads in the task.

```
thread_act_port_array_t thread_list;
mach_msg_type_number_t thread_count;

task_threads(port, &thread_list, &thread_count);
```

Once you have a list of threads, you can then loop over them and retrieve register contents from each. This can be done using the `thread_get_state()` function.

The code below shows the process involved for retrieving the register contents from a thread (in this case the first thread) of a `thread_act_port_array_t` list.

```
NOTE:
        This code will only work on ppc machines, i396_thread_state_t type is
        used for intel.


        ppc_thread_state_t ppc_state;
        mach_msg_type_number_t sc = PPC_THREAD_STATE_COUNT;
        long thread = 0;        // for first thread

        thread_get_state(
                        thread_list[thread],
                        PPC_THREAD_STATE,
```

http://uninformed.org/index.cgi?v=4&a=3&p=14

Interprocess Communication on macOS

Interprocess Communication on macOS

Mach ports

- Reference-counted message queues
- Send right: ability to send messages to a mach port
- Receive right: ability to receive messages from a mach port

Interprocess Communication on macOS

Mach ports

- Reference-counted message queues
- Send right: ability to send messages to a mach port
- Receive right: ability to receive messages from a mach port

Mach messages

- Data sent to a mach port
- Queued until the owner listens for a message

Interprocess Communication on macOS

Mach ports

- Reference-counted message queues
- Send right: ability to send messages to a mach port
- Receive right: ability to receive messages from a mach port

Mach messages

- Data sent to a mach port
- Queued until the owner listens for a message

Task and Thread ports

- Special types of Mach ports
  - Receive right is owned by the kernel.
- Used to control the target process

Main APIs used

Main APIs used

task_for_pid()

- Gets a Task Mach Port for the given process.

Main APIs used

task_for_pid()

- Gets a Task Mach Port for the given process.

mach_msg()

- Sends and/or receives a Mach message from the target Mach port.

Main APIs used

task_for_pid()

- Gets a Task Mach Port for the given process.

mach_msg()

- Sends and/or receives a Mach message from the target Mach port.

mach_vm_read()

- Reads a specified region of target task's address space.

mach_vm_write()

- Writes data to a specified region of target task's address space.

Main APIs used

task_for_pid()	• Gets a Task Mach Port for the given process.
mach_msg()	• Sends and/or receives a Mach message from the target Mach port.
mach_vm_read()	• Reads a specified region of target task's address space.
mach_vm_write()	• Writes data to a specified region of target task's address space.
mach_vm_allocate()	• Allocates a region of virtual memory in the target task.
mach_vm_deallocate()	• Deallocates a region of virtual memory in the target task.

Main APIs used

task_for_pid()	• Gets a Task Mach Port for the given process.
mach_msg()	• Sends and/or receives a Mach message from the target Mach port.
mach_vm_read()	• Reads a specified region of target task's address space.
mach_vm_write()	• Writes data to a specified region of target task's address space.
mach_vm_allocate()	• Allocates a region of virtual memory in the target task.
mach_vm_deallocate()	• Deallocates a region of virtual memory in the target task.
mach_vm_region_recurse()	• Retrieves information about a virtual memory region

What should our debugger do?

- Start a new process in a suspended state / Attach to an existing process
- Handle exceptions in the child process
- List libraries loaded in the process, determine the address range at which each library is loaded
- Get notified when libraries are loaded/unloaded
- Examine and manipulate the memory of the target process

What should our debugger do?

- **Start a new process in a suspended state / Attach to an existing process**
- Handle exceptions in the child process
- List libraries loaded in the process, determine the address range at which each library is loaded
- Get notified when libraries are loaded/unloaded
- Examine and manipulate the memory of the target process

Start a new process in a suspended state
Attach to an existing process

Start a new process in a suspended state
Attach to an existing process

What should our debugger do?

- Start a new process in a suspended state / Attach to an existing process
- **Handle exceptions in the child process**
- List libraries loaded in the process, determine the address range at which each library is loaded
- Get notified when libraries are loaded/unloaded
- Examine and manipulate the memory of the target process

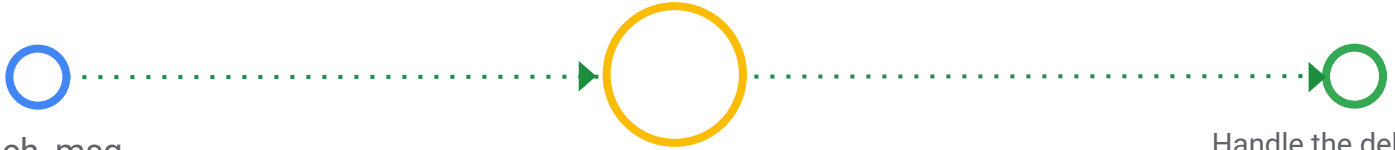
Debugger’s Main Loop

Debugger's Main Loop

Wait for debugging event

Parse the debugging event

Handle the debugging event



mach_msg

MACH_RCV_MSG
MACH_RCV_TIMEOUT
MACH_RCV_INTERRUPTED

```
/**
 * Called by mach_exc_server
 *
 * @param exception_port the exception_port registered in AttachToProcess() method
 * @param task_port the target_task
*/
kern_return_t catch_mach_exception_raise_state_identity(
    mach_port_t exception_port,
    mach_port_t thread_port,
    mach_port_t task_port,
    exception_type_t exception_type,
    mach_exception_data_t code,
    mach_msg_type_number_t code_cnt,
    int *flavor,
    thread_state_t old_state,
    mach_msg_type_number_t old_state_cnt,
    thread_state_t new_state,
    mach_msg_type_number_t *new_state_cnt) {
```

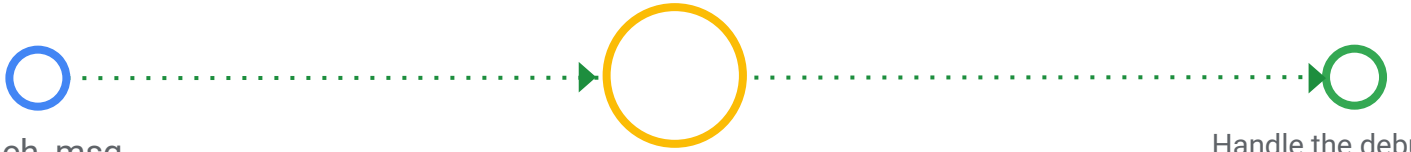
Handle the debugging event accordingly and continue/halt debugger's main loop

Debugger's Main Loop

Wait for debugging event

Parse the debugging event

Handle the debugging event



mach_msg

MACH_RCV_MSG
MACH_RCV_TIMEOUT
MACH_RCV_INTERRUPTED

```
/**
 * Called by mach_exc_server
 *
 * @param exception_port the exception_port registered in AttachToProcess() method
 * @param task_port the target_task
*/
kern_return_t catch_mach_exception_raise_state_identity(
    mach_port_t exception_port,
    mach_port_t thread_port,
    mach_port_t task_port,
    exception_type_t exception_type,
    mach_exception_data_t code,
    mach_msg_type_number_t code_cnt,
    int *flavor,
    thread_state_t old_state,
    mach_msg_type_number_t old_state_cnt,
    thread_state_t new_state,
    mach_msg_type_number_t *new_state_cnt) {
```

Handle the debugging event accordingly and continue/halt debugger's main loop

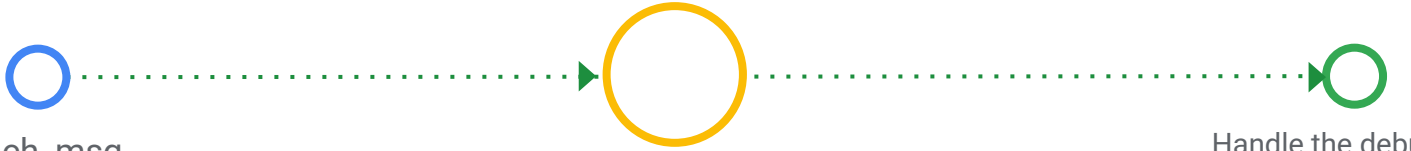
EXC_BAD_ACCESS

Debugger's Main Loop

Wait for debugging event

Parse the debugging event

Handle the debugging event



mach_msg

MACH_RCV_MSG
MACH_RCV_TIMEOUT
MACH_RCV_INTERRUPTED

```
/**
 * Called by mach_exc_server
 *
 * @param exception_port the exception_port registered in AttachToProcess() method
 * @param task_port the target_task
*/
kern_return_t catch_mach_exception_raise_state_identity(
    mach_port_t exception_port,
    mach_port_t thread_port,
    mach_port_t task_port,
    exception_type_t exception_type,
    mach_exception_data_t code,
    mach_msg_type_number_t code_cnt,
    int *flavor,
    thread_state_t old_state,
    mach_msg_type_number_t old_state_cnt,
    thread_state_t new_state,
    mach_msg_type_number_t *new_state_cnt) {
```

Handle the debugging event accordingly and continue/halt debugger's main loop

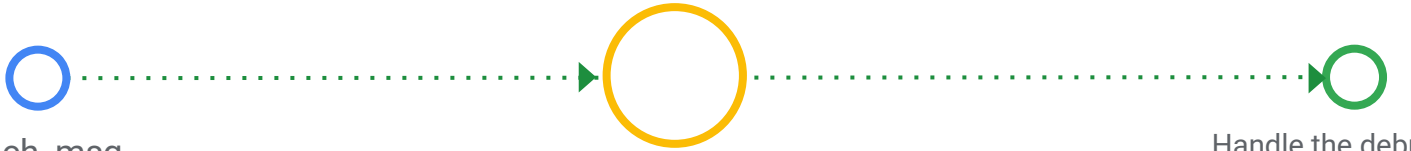
EXC_BAD_INSTRUCTION

Debugger's Main Loop

Wait for debugging event

Parse the debugging event

Handle the debugging event



mach_msg

MACH_RCV_MSG
MACH_RCV_TIMEOUT
MACH_RCV_INTERRUPTED

```
/**
 * Called by mach_exc_server
 *
 * @param exception_port the exception_port registered in AttachToProcess() method
 * @param task_port the target_task
*/
kern_return_t catch_mach_exception_raise_state_identity(
    mach_port_t exception_port,
    mach_port_t thread_port,
    mach_port_t task_port,
    exception_type_t exception_type,
    mach_exception_data_t code,
    mach_msg_type_number_t code_cnt,
    int *flavor,
    thread_state_t old_state,
    mach_msg_type_number_t old_state_cnt,
    thread_state_t new_state,
    mach_msg_type_number_t *new_state_cnt) {
```

Handle the debugging event accordingly and continue/halt debugger's main loop

EXC_BREAKPOINT

What should our debugger do?

- Start a new process in a suspended state / Attach to an existing process
- Handle exceptions in the child process
- **List libraries loaded in the process, determine the address range at which each library is loaded**
- Get notified when libraries are loaded/unloaded
- Examine and manipulate the memory of the target process

Examine loaded libraries in the target

Examine loaded libraries in the target

Examine loaded libraries in the target

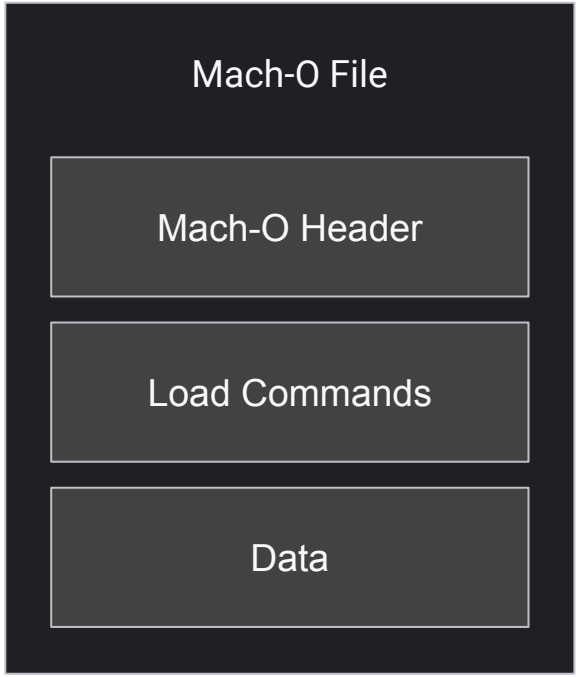
General information about the architecture of the library: byte order, CPU type, number of load commands

Examine loaded libraries in the target

General information about the architecture of the library: byte order, CPU type, number of load commands

Specifies the layout and linkage characteristics of the file (e.g. the location of the symbol table)

Examine loaded libraries in the target



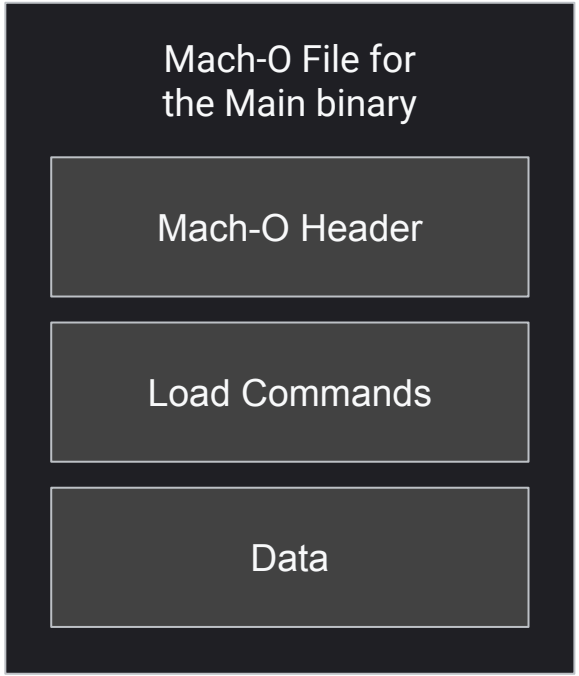
General information about the architecture of the library: byte order, CPU type, number of load commands

Specifies the layout and linkage characteristics of the file (e.g. the location of the symbol table)

Code and data of one or more segments (e.g. symbol tabes, dynamic symbol tables)

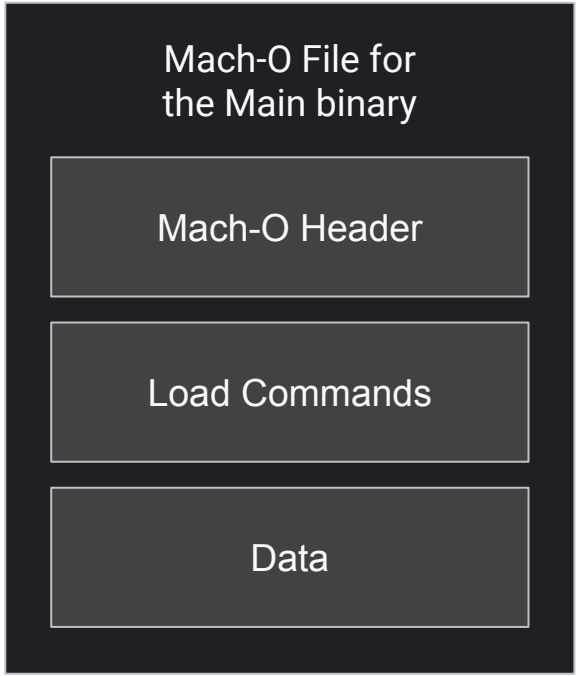
Detect target’s entrypoint

Detect target's entrypoint




```
void *Debugger::GetModuleEntrypoint(void *base_address) {
  mach_header_64 mach_header;
  GetMachHeader(base_address, &mach_header);
  if (mach_header.filetype != MH_EXECUTE) {
    return NULL;
  }

  void *load_commands_buffer = NULL;
  GetLoadCommandsBuffer(base_address, &mach_header, &load_commands_buffer);

  entry_point_command *entry_point_cmd = NULL;
  GetLoadCommand(mach_header, load_commands_buffer, LC_MAIN, NULL, &entry_point_cmd);
  if (entry_point_cmd == NULL) {
    FATAL("Unable to find ENTRY POINT command in GetModuleEntrypoint\n");
  }

  uint64_t entryoff = entry_point_cmd->entryoff;

  free(load_commands_buffer);
  return (void*)((uint64_t)base_address + entryoff);
}
```

Detect target's entrypoint

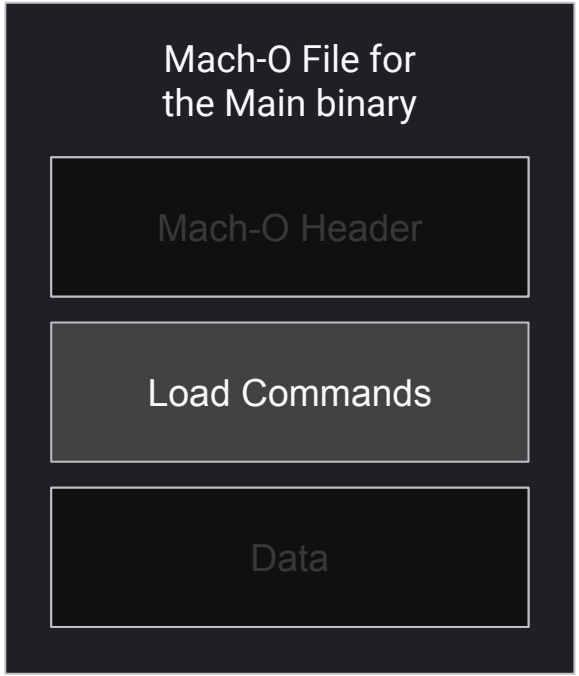



```
void *Debugger::GetModuleEntrypoint(void *base_address) {
  mach_header_64 mach_header;
  GetMachHeader(base_address, &mach_header);
  if (mach_header.filetype != MH_EXECUTE) {
    return NULL;
  }

  void *load_commands_buffer = NULL;
  GetLoadCommandsBuffer(base_address, &mach_header, &load_commands_buffer);

  entry_point_command *entry_point_cmd = NULL;
  GetLoadCommand(mach_header, load_commands_buffer, LC_MAIN, NULL, &entry_point_cmd);
  if (entry_point_cmd == NULL) {
    FATAL("Unable to find ENTRY POINT command in GetModuleEntrypoint\n");
  }

  uint64_t entryoff = entry_point_cmd->entryoff;

  free(load_commands_buffer);
  return (void*)((uint64_t)base_address + entryoff);
}
```

Detect target's entrypoint

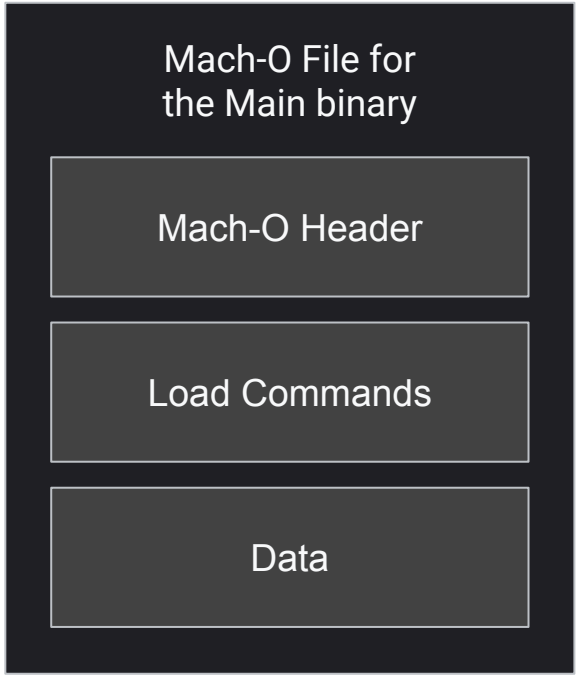

```
void *Debugger::GetModuleEntrypoint(void *base_address) {
  mach_header_64 mach_header;
  GetMachHeader(base_address, &mach_header);
  if (mach_header.filetype != MH_EXECUTE) {
    return NULL;
  }

  void *load_commands_buffer = NULL;
  GetLoadCommandsBuffer(base_address, &mach_header, &load_commands_buffer);

  entry_point_command *entry_point_cmd = NULL;
  GetLoadCommand(mach_header, load_commands_buffer, LC_MAIN, NULL, &entry_point_cmd);
  if (entry_point_cmd == NULL) {
    FATAL("Unable to find ENTRY POINT command in GetModuleEntrypoint\n");
  }

  uint64_t entryoff = entry_point_cmd->entryoff;

  free(load_commands_buffer);
  return (void*)((uint64_t)base_address + entryoff);
}
```

Detect target's entrypoint




```
void *Debugger::GetModuleEntrypoint(void *base_address) {
  mach_header_64 mach_header;
  GetMachHeader(base_address, &mach_header);
  if (mach_header.filetype != MH_EXECUTE) {
    return NULL;
  }

  void *load_commands_buffer = NULL;
  GetLoadCommandsBuffer(base_address, &mach_header, &load_commands_buffer);

  entry_point_command *entry_point_cmd = NULL;
  GetLoadCommand(mach_header, load_commands_buffer, LC_MAIN, NULL, &entry_point_cmd);
  if (entry_point_cmd == NULL) {
    FATAL("Unable to find ENTRY POINT command in GetModuleEntrypoint\n");
  }

  uint64_t entryoff = entry_point_cmd->entryoff;

  free(load_commands_buffer);
  return (void*)((uint64_t)base_address + entryoff);
}
```

Detect target's entrypoint




```
void *Debugger::GetModuleEntrypoint(void *base_address) {
  mach_header_64 mach_header;
  GetMachHeader(base_address, &mach_header);
  if (mach_header.filetype != MH_EXECUTE) {
    return NULL;
  }

  void *load_commands_buffer = NULL;
  GetLoadCommandsBuffer(base_address, &mach_header, &load_commands_buffer);

  entry_point_command *entry_point_cmd = NULL;
  GetLoadCommand(mach_header, load_commands_buffer, LC_MAIN, NULL, &entry_point_cmd);
  if (entry_point_cmd == NULL) {
    FATAL("Unable to find ENTRY POINT command in GetModuleEntrypoint\n");
  }

  uint64_t entryoff = entry_point_cmd->entryoff;

  free(load_commands_buffer);
  return (void*)((uint64_t)base_address + entryoff);
}
```

What should our debugger do?

- Start a new process in a suspended state / Attach to an existing process
- Handle exceptions in the child process
- List libraries loaded in the process, determine the address range at which each library is loaded
- **Get notified when libraries are loaded/unloaded**
- Examine and manipulate the memory of the target process

Get notified when libraries are loaded or unloaded

Get notified when libraries are loaded or unloaded

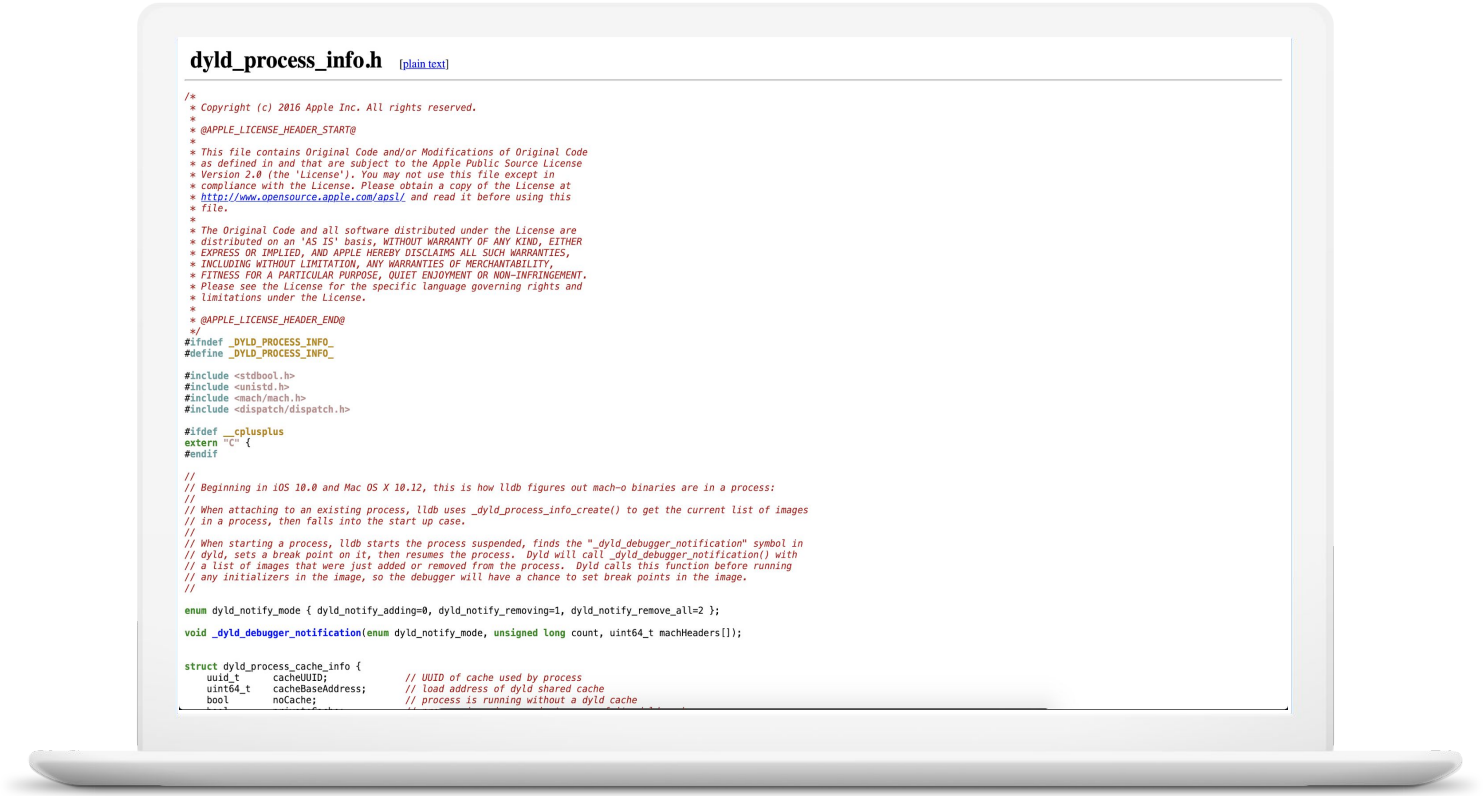
```
enum dyld_notify_mode { dyld_notify_adding=0, dyld_notify_removing=1, dyld_notify_remove_all=2 };

void _dyld_debugger_notification(enum dyld_notify_mode, unsigned long count, uint64_t machHeaders[]);
```

Get notified when libraries are loaded or unloaded

```
enum dyld_notify_mode { dyld_notify_adding=0, dyld_notify_removing=1, dyld_notify_remove_all=2 };

void _dyld_debugger_notification(enum dyld_notify_mode, unsigned long count, uint64_t machHeaders[]);
```

Where do we find this function?

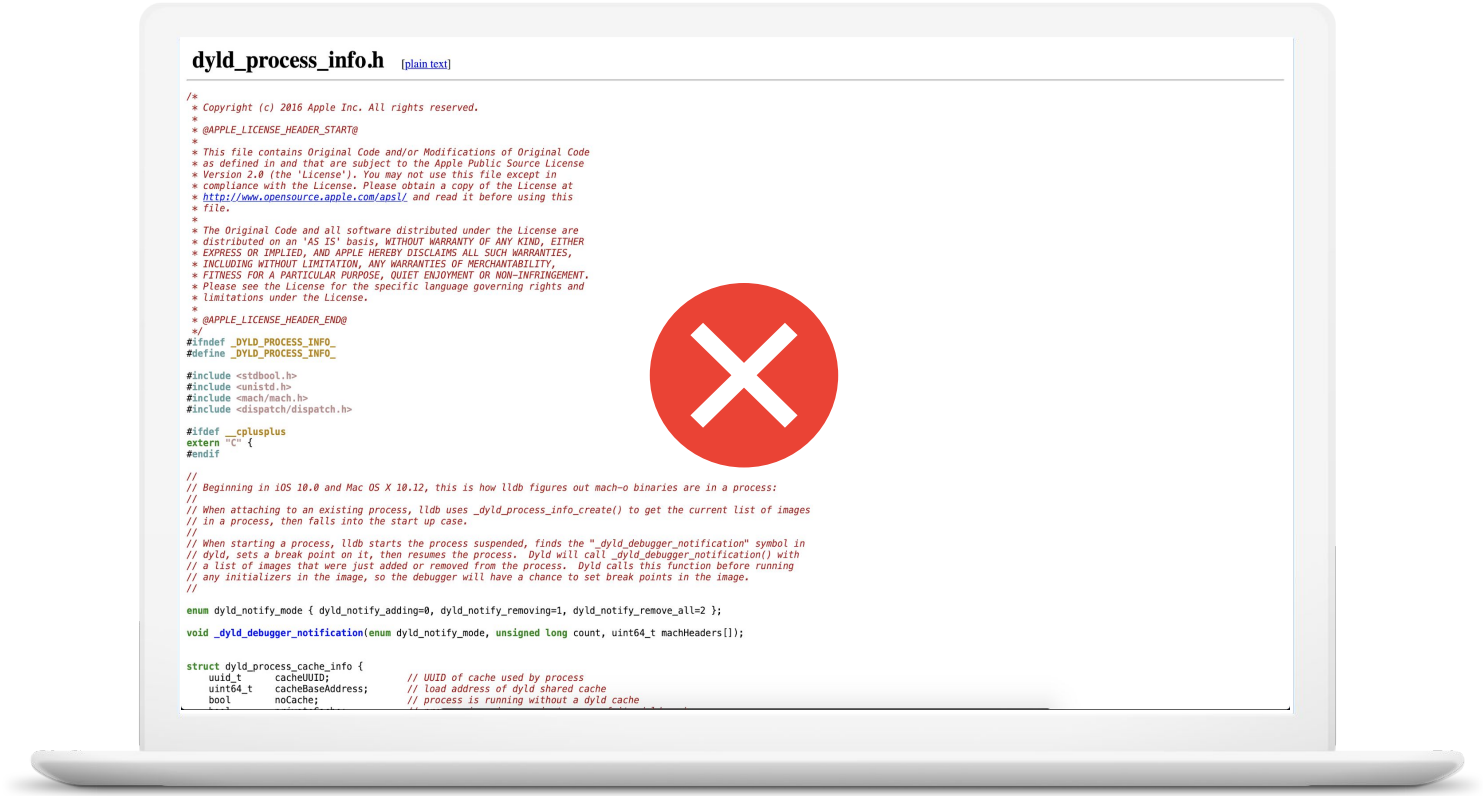
dyld_process_info.h?



dyld_process_info.h [plain text]

```
/*
 * Copyright (c) 2016 Apple Inc. All rights reserved.
 *
 * @APPLE_LICENSE_HEADER_START@
 *
 * This file contains Original Code and/or Modifications of Original Code
 * as defined in and that are subject to the Apple Public Source License
 * Version 2.0 (the 'License'). You may not use this file except in
 * compliance with the License. Please obtain a copy of the License at
 * http://www.opensource.apple.com/apsl/ and read it before using this
 * file.
 *
 * The Original Code and all software distributed under the License are
 * distributed on an 'AS IS' basis, WITHOUT WARRANTY OF ANY KIND, EITHER
 * EXPRESS OR IMPLIED, AND APPLE HEREBY DISCLAIMS ALL SUCH WARRANTIES,
 * INCLUDING WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE, QUIET ENJOYMENT OR NON-INFRINGEMENT.
 * Please see the License for the specific language governing rights and
 * limitations under the License.
 *
 * @APPLE_LICENSE_HEADER_END@
 */
#ifndef _DYLD_PROCESS_INFO_
#define _DYLD_PROCESS_INFO_

#include <stdbool.h>
#include <unistd.h>
#include <mach/mach.h>
#include <dispatch/dispatch.h>

#ifdef __cplusplus
extern "C" {
#endif

//
// Beginning in iOS 10.0 and Mac OS X 10.12, this is how lldb figures out mach-o binaries are in a process:
//
// When attaching to an existing process, lldb uses _dyld_process_info_create() to get the current list of images
// in a process, then falls into the start up case.
//
// When starting a process, lldb starts the process suspended, finds the "_dyld_debugger_notification" symbol in
// dyld, sets a break point on it, then resumes the process.  Dyld will call _dyld_debugger_notification() with
// a list of images that were just added or removed from the process.  Dyld calls this function before running
// any initializers in the image, so the debugger will have a chance to set break points in the image.
//

enum dyld_notify_mode { dyld_notify_adding=0, dyld_notify_removing=1, dyld_notify_remove_all=2 };

void _dyld_debugger_notification(enum dyld_notify_mode, unsigned long count, uint64_t machHeaders[]);


struct dyld_process_cache_info {
    uuid_t      cacheUUID;              // UUID of cache used by process
    uint64_t    cacheBaseAddress;       // load address of dyld shared cache
    bool        noCache;                // process is running without a dyld cache
```

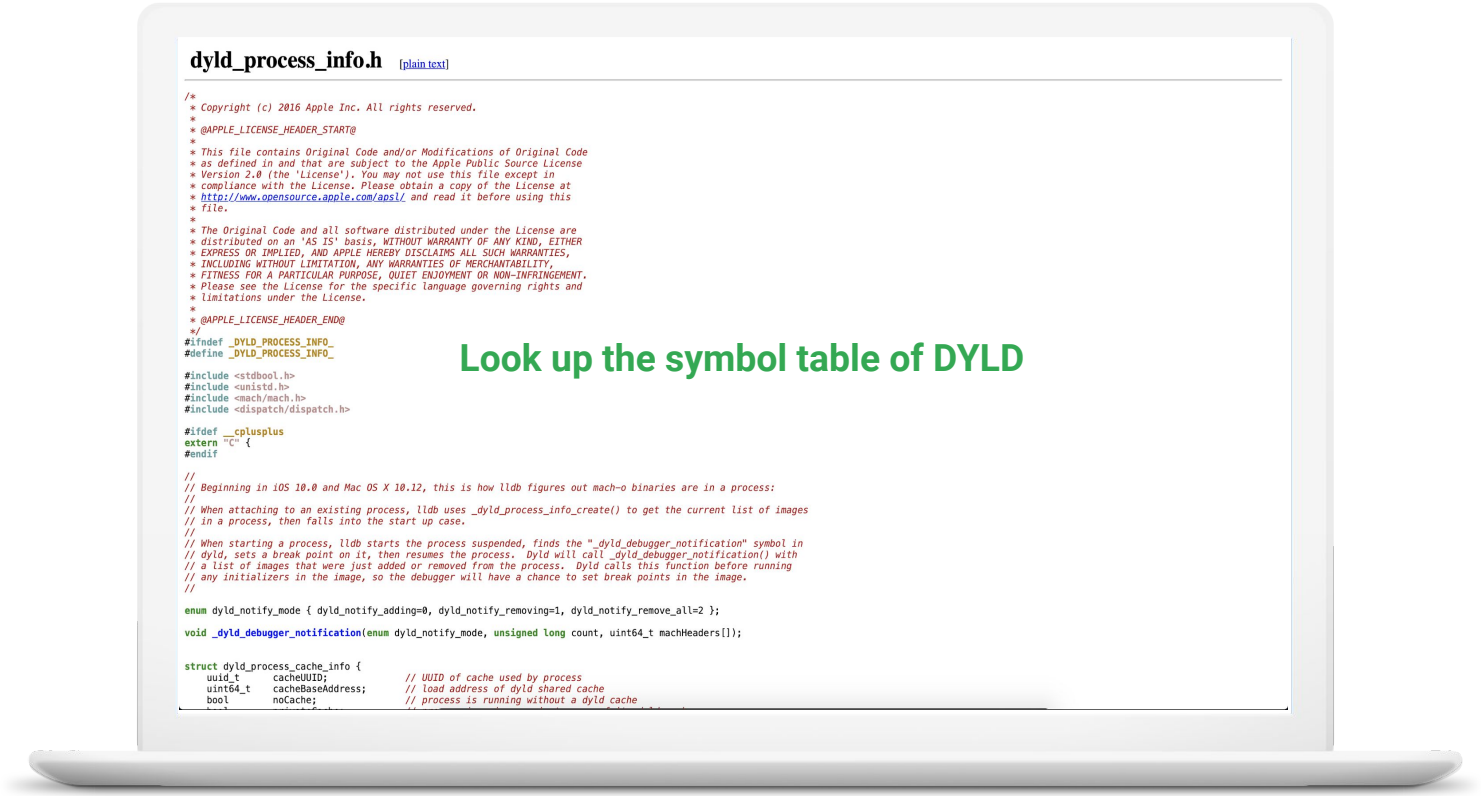
dyld_process_info.h?



dyld_process_info.h [plain text]

```
/*
 * Copyright (c) 2016 Apple Inc. All rights reserved.
 *
 * @APPLE_LICENSE_HEADER_START@
 *
 * This file contains Original Code and/or Modifications of Original Code
 * as defined in and that are subject to the Apple Public Source License
 * Version 2.0 (the 'License'). You may not use this file except in
 * compliance with the License. Please obtain a copy of the License at
 * http://www.opensource.apple.com/apsl/ and read it before using this
 * file.
 *
 * The Original Code and all software distributed under the License are
 * distributed on an 'AS IS' basis, WITHOUT WARRANTY OF ANY KIND, EITHER
 * EXPRESS OR IMPLIED, AND APPLE HEREBY DISCLAIMS ALL SUCH WARRANTIES,
 * INCLUDING WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE, QUIET ENJOYMENT OR NON-INFRINGEMENT.
 * Please see the License for the specific language governing rights and
 * limitations under the License.
 *
 * @APPLE_LICENSE_HEADER_END@
 */
#ifndef _DYLD_PROCESS_INFO_
#define _DYLD_PROCESS_INFO_

#include <stdbool.h>
#include <unistd.h>
#include <mach/mach.h>
#include <dispatch/dispatch.h>

#ifdef __cplusplus
extern "C" {
#endif

//
// Beginning in iOS 10.0 and Mac OS X 10.12, this is how lldb figures out mach-o binaries are in a process:
//
// When attaching to an existing process, lldb uses _dyld_process_info_create() to get the current list of images
// in a process, then falls into the start up case.
//
// When starting a process, lldb starts the process suspended, finds the "_dyld_debugger_notification" symbol in
// dyld, sets a break point on it, then resumes the process.  Dyld will call _dyld_debugger_notification() with
// a list of images that were just added or removed from the process.  Dyld calls this function before running
// any initializers in the image, so the debugger will have a chance to set break points in the image.
//

enum dyld_notify_mode { dyld_notify_adding=0, dyld_notify_removing=1, dyld_notify_remove_all=2 };

void _dyld_debugger_notification(enum dyld_notify_mode, unsigned long count, uint64_t machHeaders[]);


struct dyld_process_cache_info {
    uuid_t      cacheUUID;              // UUID of cache used by process
    uint64_t    cacheBaseAddress;       // load address of dyld shared cache
    bool        noCache;                // process is running without a dyld cache
```

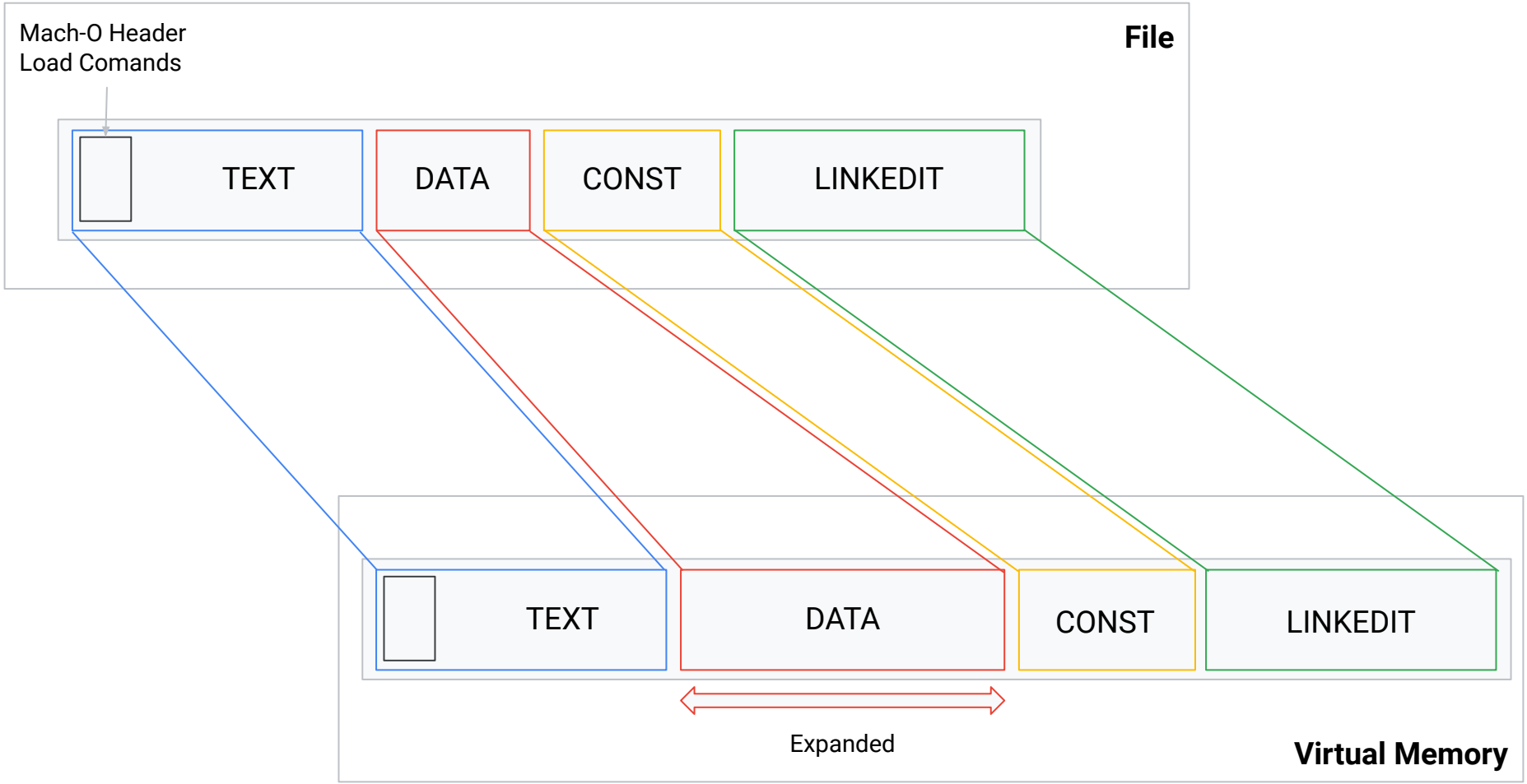
dyld_process_info.h?

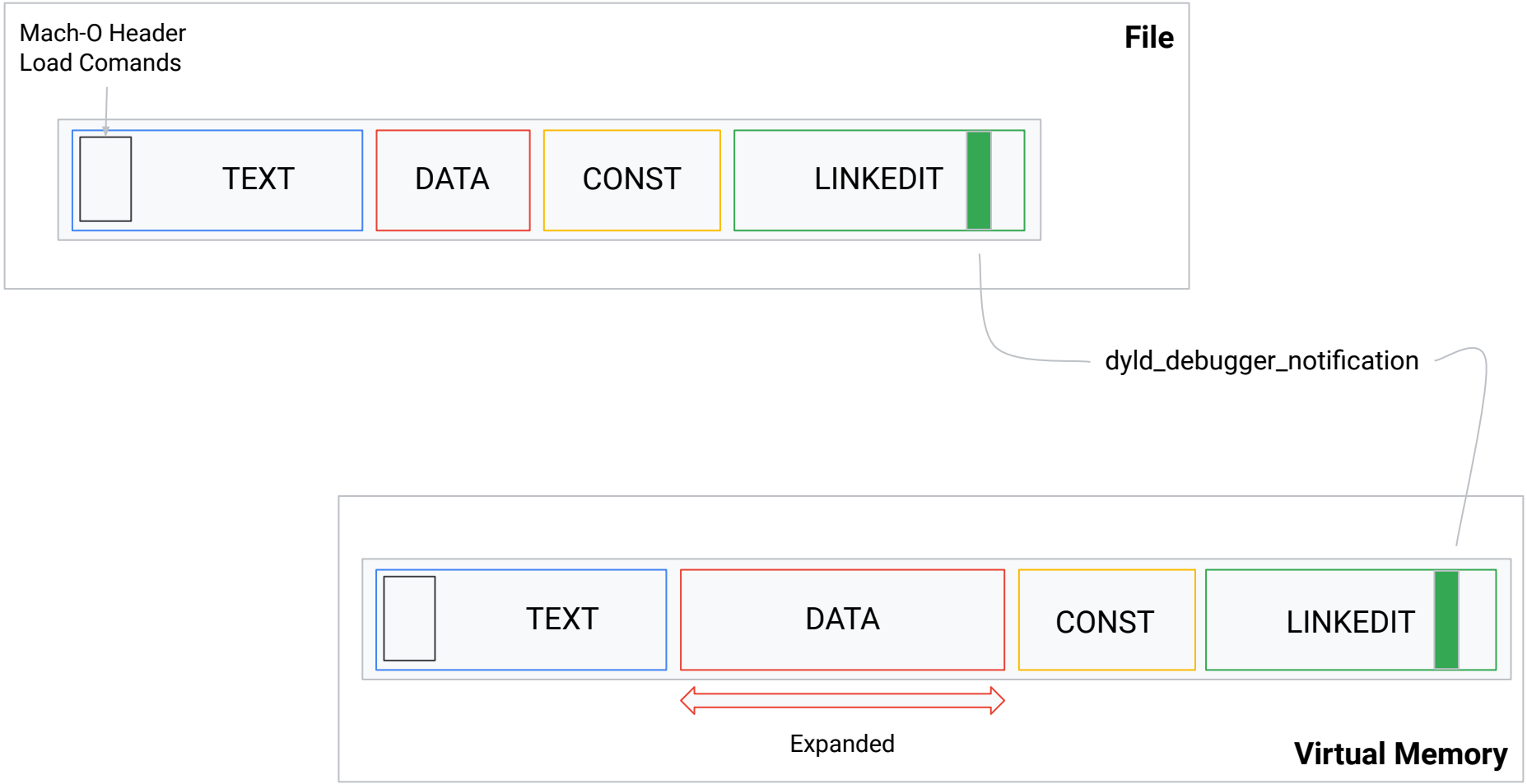


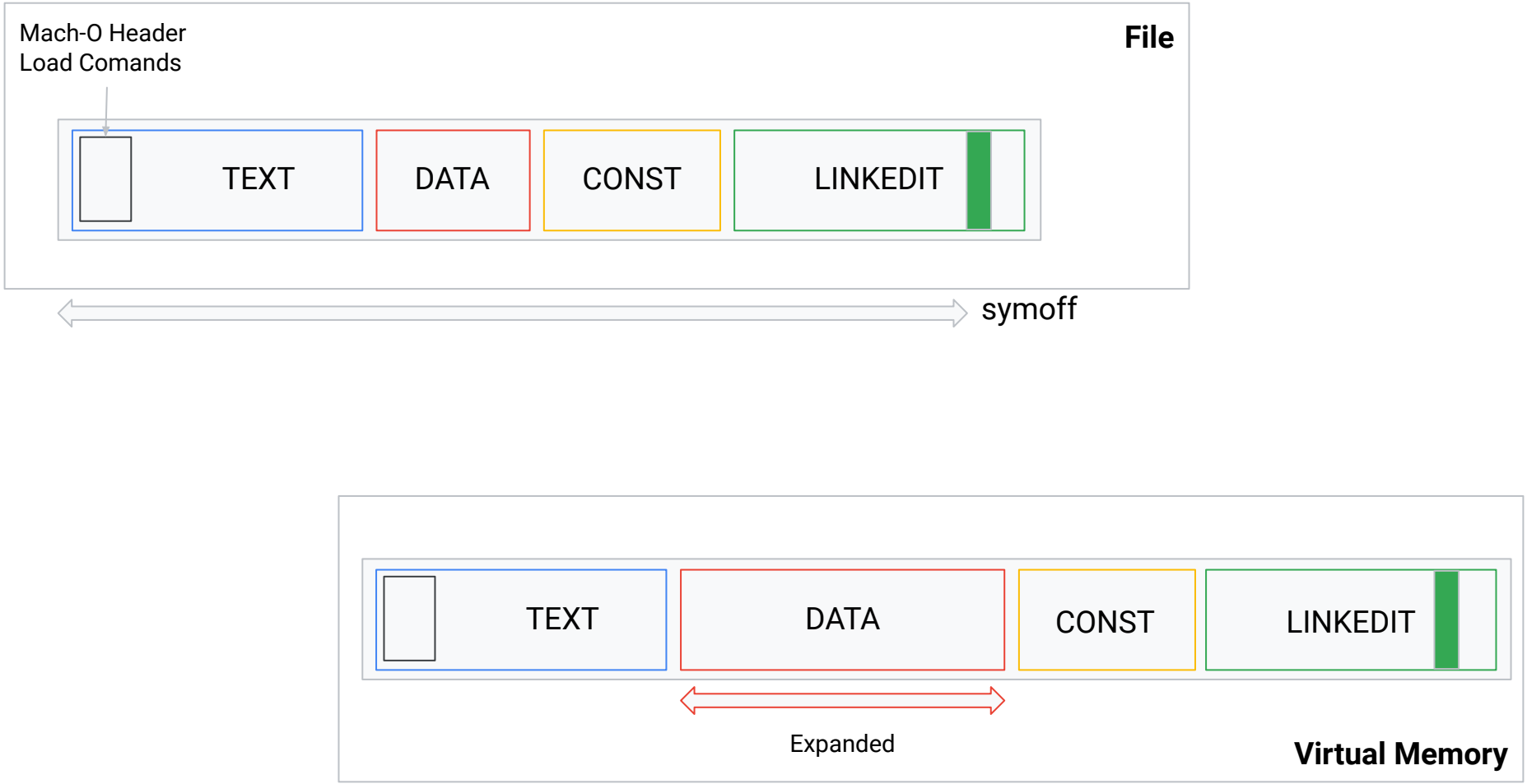
dyld_process_info.h [plain text]

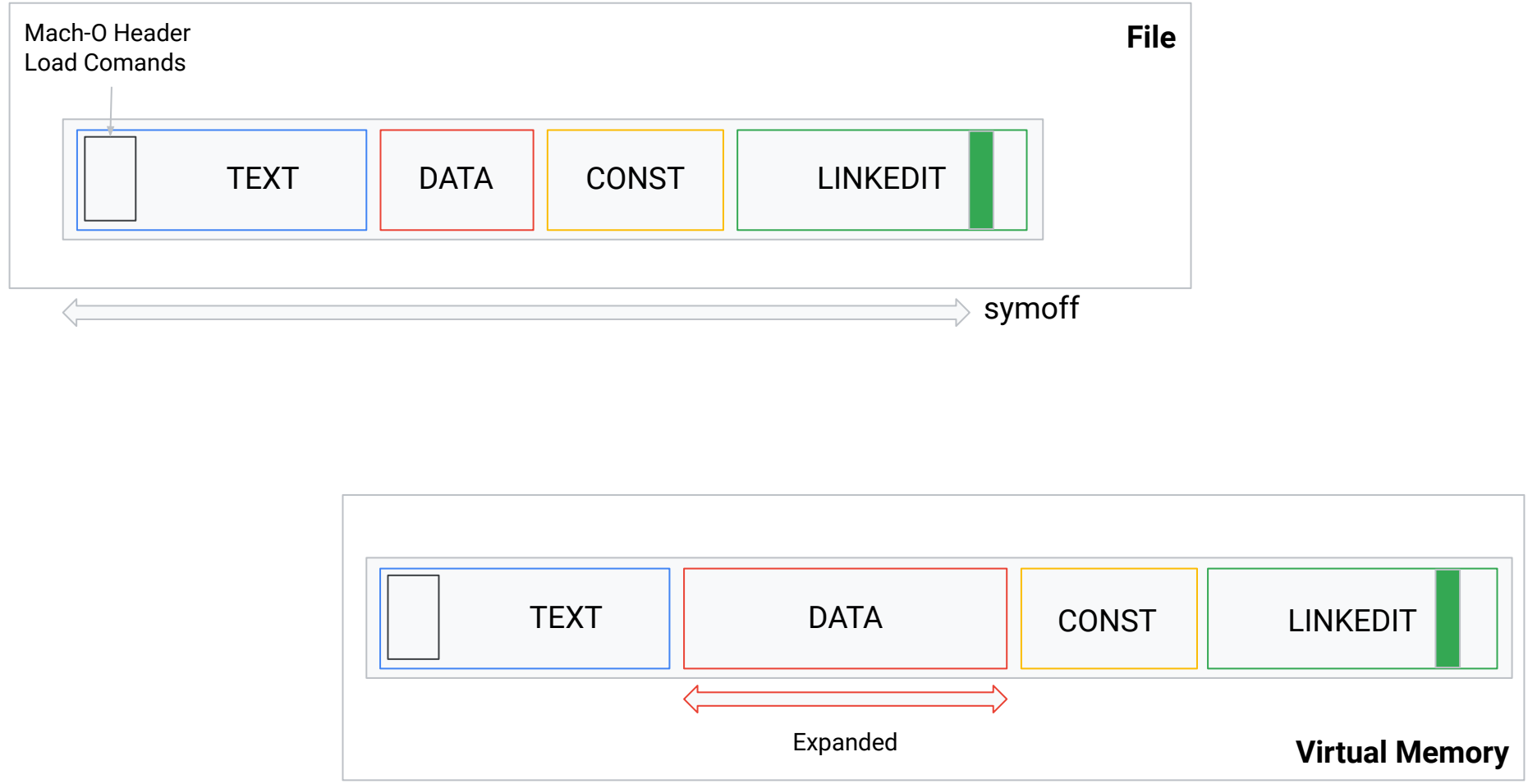
```
/*
 * Copyright (c) 2016 Apple Inc. All rights reserved.
 *
 * @APPLE_LICENSE_HEADER_START@
 *
 * This file contains Original Code and/or Modifications of Original Code
 * as defined in and that are subject to the Apple Public Source License
 * Version 2.0 (the 'License'). You may not use this file except in
 * compliance with the License. Please obtain a copy of the License at
 * http://www.opensource.apple.com/apsl/ and read it before using this
 * file.
 *
 * The Original Code and all software distributed under the License are
 * distributed on an 'AS IS' basis, WITHOUT WARRANTY OF ANY KIND, EITHER
 * EXPRESS OR IMPLIED, AND APPLE HEREBY DISCLAIMS ALL SUCH WARRANTIES,
 * INCLUDING WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE, QUIET ENJOYMENT OR NON-INFRINGEMENT.
 * Please see the License for the specific language governing rights and
 * limitations under the License.
 *
 * @APPLE_LICENSE_HEADER_END@
 */
#ifndef _DYLD_PROCESS_INFO_
#define _DYLD_PROCESS_INFO_

#include <stdbool.h>
#include <unistd.h>
#include <mach/mach.h>
#include <dispatch/dispatch.h>

#ifdef __cplusplus
extern "C" {
#endif

//
// Beginning in iOS 10.0 and Mac OS X 10.12, this is how lldb figures out mach-o binaries are in a process:
//
// When attaching to an existing process, lldb uses _dyld_process_info_create() to get the current list of images
// in a process, then falls into the start up case.
//
// When starting a process, lldb starts the process suspended, finds the "_dyld_debugger_notification" symbol in
// dyld, sets a break point on it, then resumes the process.  Dyld will call _dyld_debugger_notification() with
// a list of images that were just added or removed from the process.  Dyld calls this function before running
// any initializers in the image, so the debugger will have a chance to set break points in the image.
//

enum dyld_notify_mode { dyld_notify_adding=0, dyld_notify_removing=1, dyld_notify_remove_all=2 };

void _dyld_debugger_notification(enum dyld_notify_mode, unsigned long count, uint64_t machHeaders[]);


struct dyld_process_cache_info {
    uuid_t      cacheUUID;              // UUID of cache used by process
    uint64_t    cacheBaseAddress;       // load address of dyld shared cache
    bool        noCache;                // process is running without a dyld cache
```

Look up the symbol table of DYLD

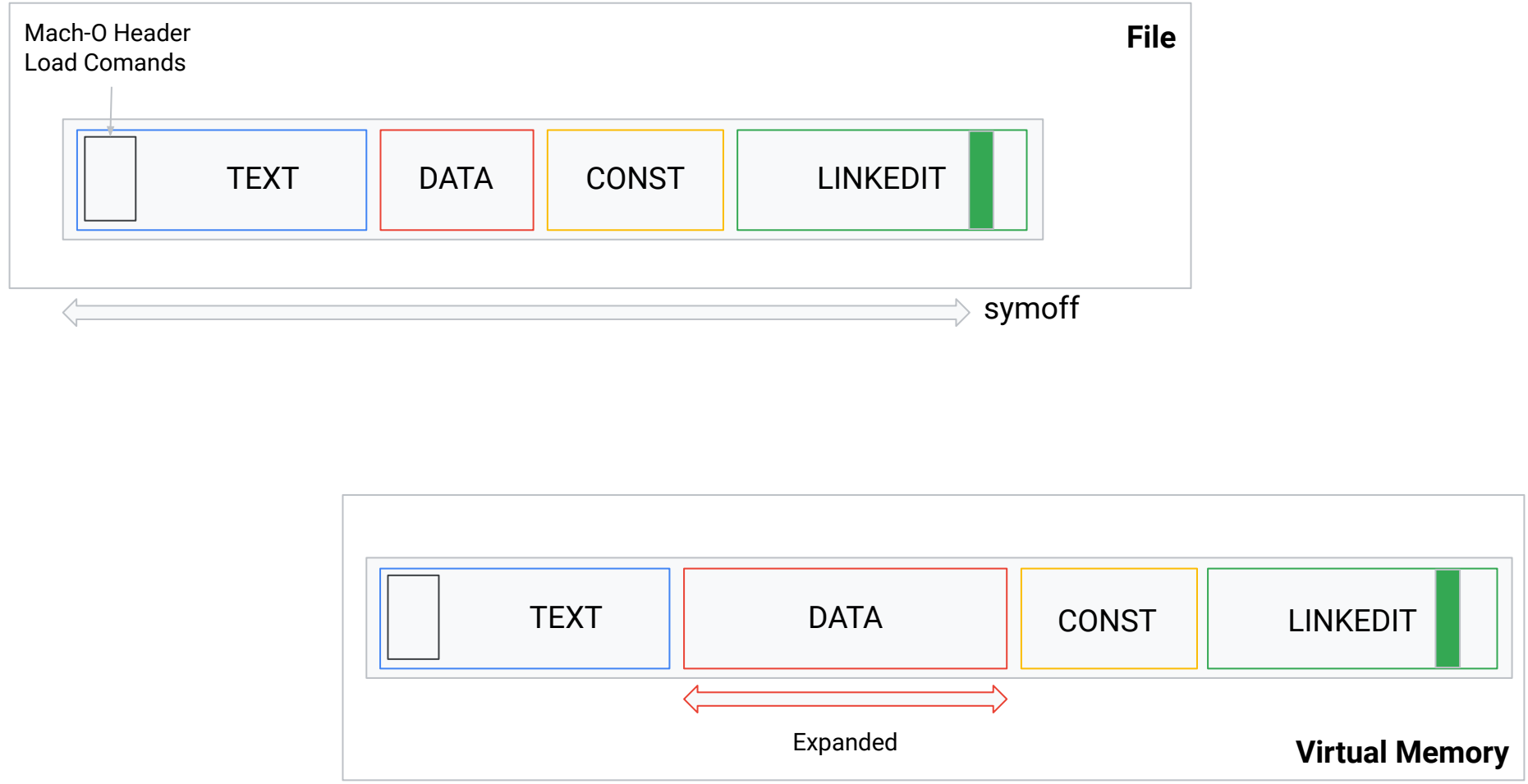

Mach-O Header
Load Comands
File
TEXT
DATA
CONST
LINKEDIT
TEXT
DATA
CONST
LINKEDIT
Expanded
Virtual Memory

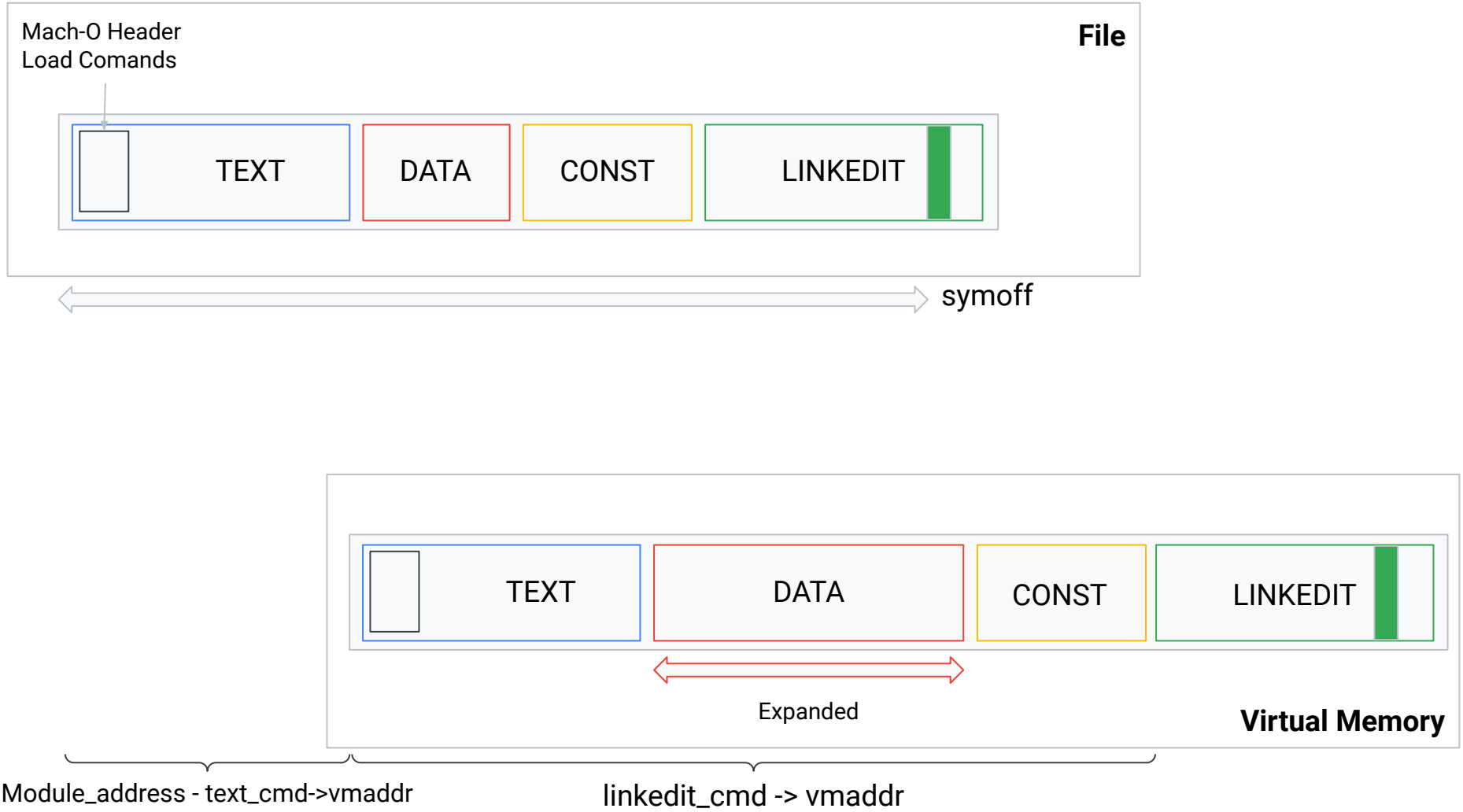

Mach-O Header
Load Comands
File
TEXT
DATA
CONST
LINKEDIT
dyld_debugger_notification
TEXT
DATA
CONST
LINKEDIT
Expanded
Virtual Memory

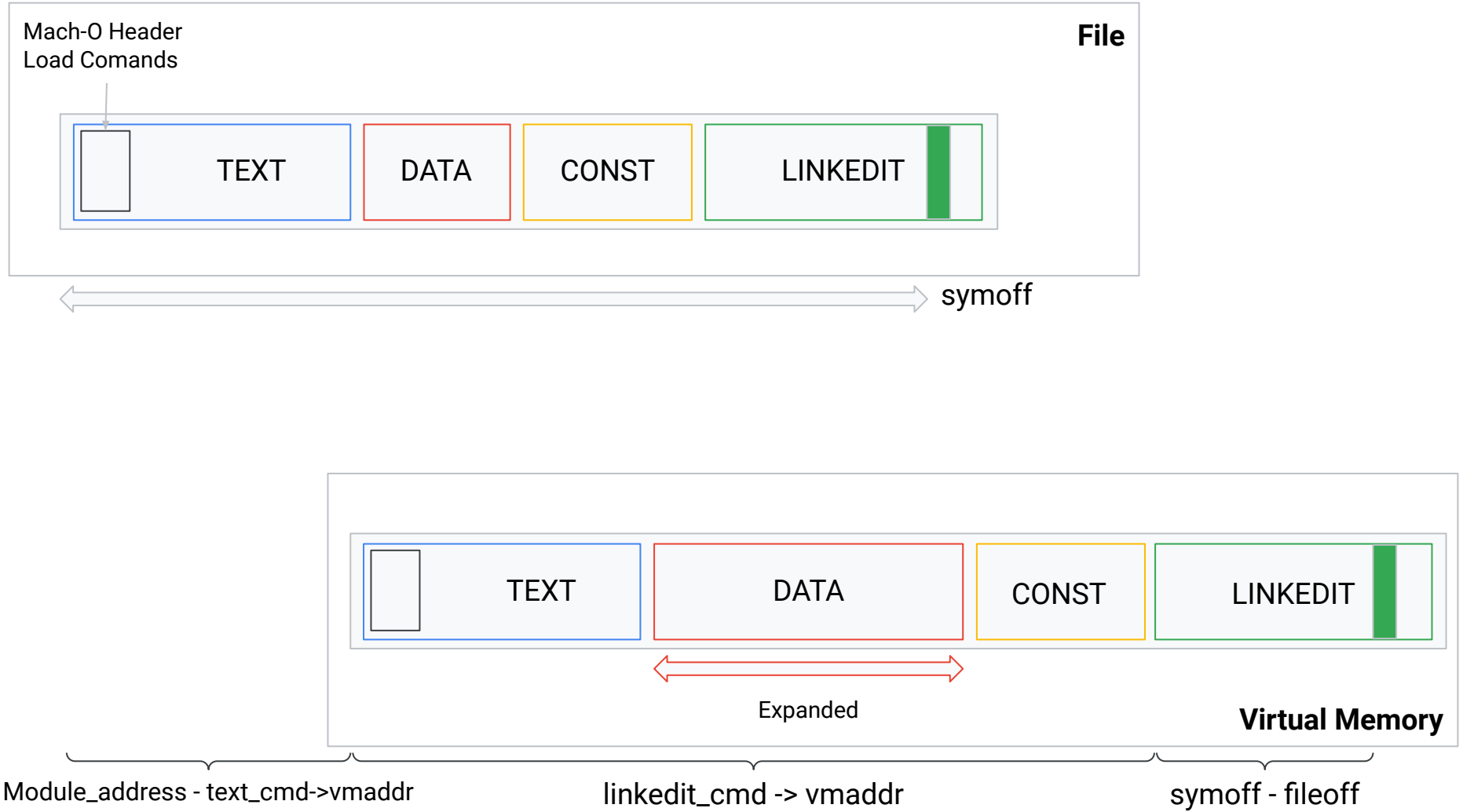

Mach-O Header
Load Comands
File
TEXT
DATA
CONST
LINKEDIT
symoff
TEXT
DATA
CONST
LINKEDIT
Expanded
Virtual Memory

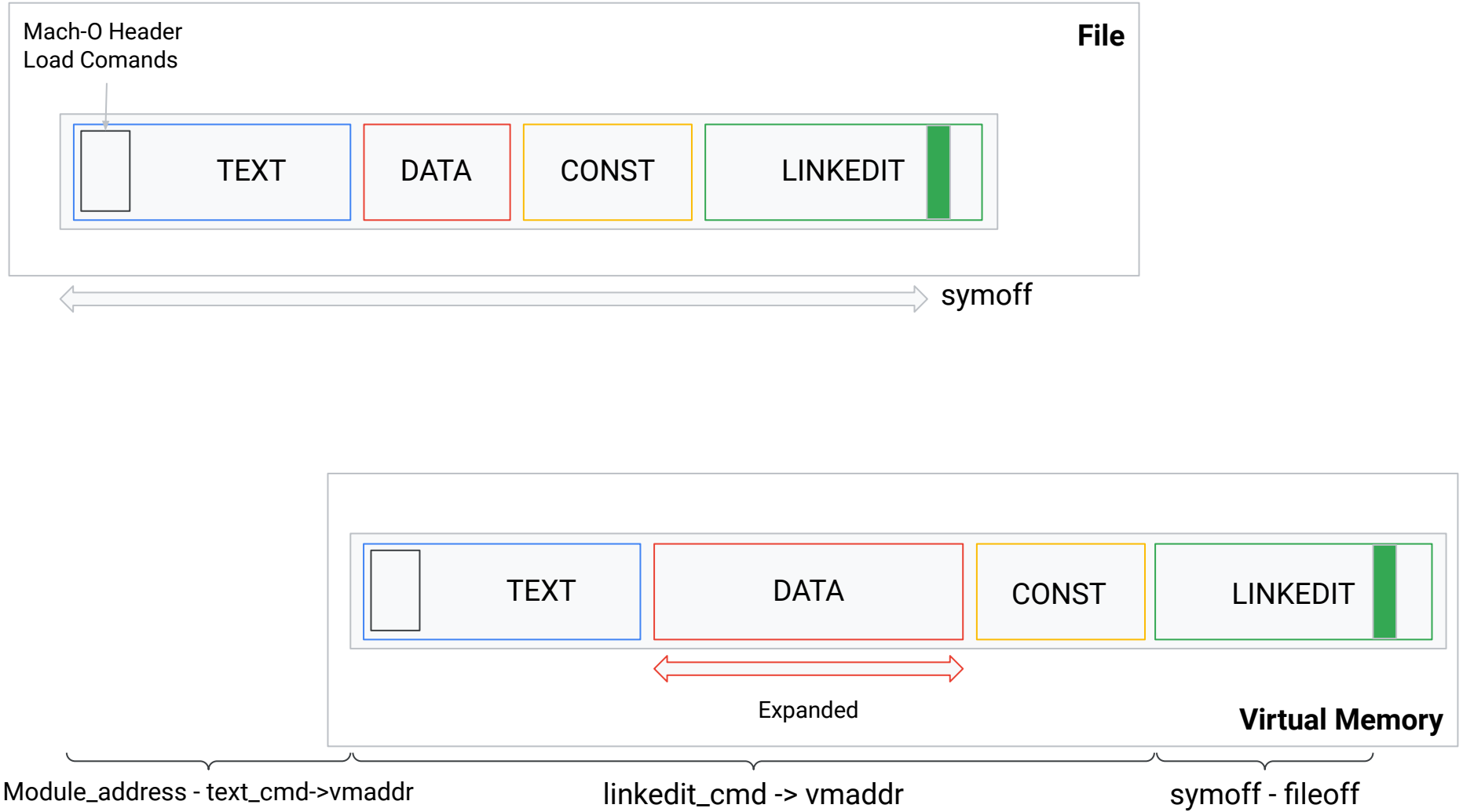

Mach-O Header
Load Comands
File
TEXT
DATA
CONST
LINKEDIT
symoff
TEXT
DATA
CONST
LINKEDIT
Expanded
Virtual Memory
Module_address - text_cmd->vmaddr

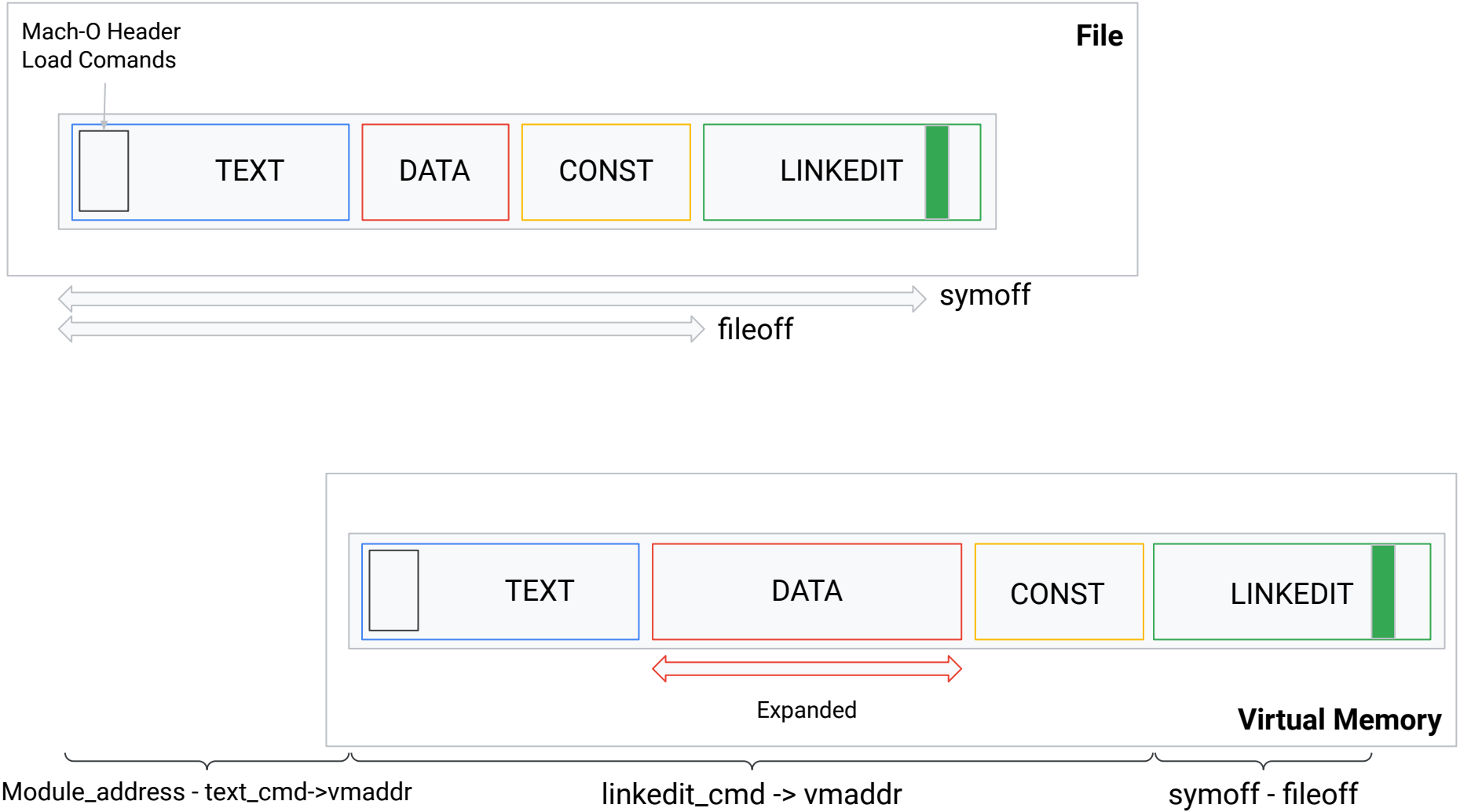

Mach-O Header
Load Comands
File
/*
* The 64-bit segment load command indicates that a part of this file is to be
* mapped into a 64-bit task's address space. If the 64-bit segment has
* sections then section_64 structures directly follow the 64-bit segment
* command and their size is reflected in cmdsize.
*/
struct segment_command_64 { /* for 64-bit architectures */
uint32_t cmd; /* LC_SEGMENT_64 */
uint32_t cmdsize; /* includes sizeof section_64 structs */
char segname[16]; /* segment name */
uint64_t vmaddr; /* memory address of this segment */
uint64_t vmsize; /* memory size of this segment */
uint64_t fileoff; /* file offset of this segment */
uint64_t filesize; /* amount to map from the file */
vm_prot_t maxprot; /* maximum VM protection */
vm_prot_t initprot; /* initial VM protection */
uint32_t nsects; /* number of sections in segment */
uint32_t flags; /* flags */
};
Expanded
Virtual Memory
Module_address - text_cmd->vmaddr


Mach-O Header
Load Comands
File
TEXT
DATA
CONST
LINKEDIT
symoff
TEXT
DATA
CONST
LINKEDIT
Expanded
Virtual Memory
Module_address - text_cmd->vmaddr


Mach-O Header
Load Comands
File
TEXT
DATA
CONST
LINKEDIT
symoff
TEXT
DATA
CONST
LINKEDIT
Expanded
Virtual Memory
Module_address - text_cmd->vmaddr
linkedit_cmd -> vmaddr


Mach-O Header
Load Comands
File
TEXT
DATA
CONST
LINKEDIT
symoff
TEXT
DATA
CONST
LINKEDIT
Expanded
Virtual Memory
Module_address - text_cmd->vmaddr
linkedit_cmd -> vmaddr
symoff - fileoff


Mach-O Header
Load Comands
File
TEXT
DATA
CONST
LINKEDIT
symoff
TEXT
DATA
CONST
LINKEDIT
Expanded
Virtual Memory
Module_address - text_cmd->vmaddr
linkedit_cmd -> vmaddr
symoff - fileoff


Mach-O Header
Load Comands
File
TEXT
DATA
CONST
LINKEDIT
symoff
fileoff
TEXT
DATA
CONST
LINKEDIT
Expanded
Virtual Memory
Module_address - text_cmd->vmaddr
linkedit_cmd -> vmaddr
symoff - fileoff

Challenges

Get notified when libraries are loaded or unloaded

```
enum dyld_notify_mode { dyld_notify_adding=0, dyld_notify_removing=1, dyld_notify_remove_all=2 };

void _dyld_debugger_notification(enum dyld_notify_mode, unsigned long count, uint64_t machHeaders[]);
```

Get notified when libraries are loaded or unloaded

```
__text:000000000001201F
__text:000000000001201F ; =============== S U B R O U T I N E =======================================
__text:000000000001201F
__text:000000000001201F ; Attributes: bp-based frame
__text:000000000001201F
__text:000000000001201F                 public __dyld_debugger_notification
__text:000000000001201F __dyld_debugger_notification proc near  ; CODE XREF: gdb_image_notifier(dy
__text:000000000001201F                 push    rbp
__text:0000000000012020                 mov     rbp, rsp
__text:0000000000012023                 pop     rbp
__text:0000000000012024                 retn
__text:0000000000012024 __dyld_debugger_notification endp
__text:0000000000012024
```

Thanks to @bazad for this screenshot

Get notified when libraries are loaded or unloaded

```
__text:000000000001201F
__text:000000000001201F ; =============== S U B R O U T I N E =======================================
__text:000000000001201F
__text:000000000001201F ; Attributes: bp-based frame
__text:000000000001201F
__text:000000000001201F                 public __dyld_debugger_notification
__text:000000000001201F __dyld_debugger_notification proc near  ; CODE XREF: gdb_image_notifier(dy
__text:000000000001201F
__text:0000000000012020                 mov     rbp, rsp
__text:0000000000012023                 pop     rbp
__text:0000000000012024                 retn
__text:0000000000012024 __dyld_debugger_notification endp
__text:0000000000012024
```

Thanks to @bazad for this screenshot

Get notified when libraries are loaded or unloaded

```
__text:000000000001201F
__text:000000000001201F ; =============== S U B R O U T I N E =======================================
__text:000000000001201F
__text:000000000001201F ; Attributes: bp-based frame
__text:000000000001201F
__text:000000000001201F                         public __dyld_debugger_notification
__text:000000000001201F __dyld_debugger_notification proc near  ; CODE XREF: gdb_image_notifier(dy
__text:000000000001201F                         INT3 (0xCC)
__text:0000000000012020                         mov     rbp, rsp
__text:0000000000012023                         pop     rbp
__text:0000000000012024                         retn
__text:0000000000012024 __dyld_debugger_notification endp
__text:0000000000012024
```

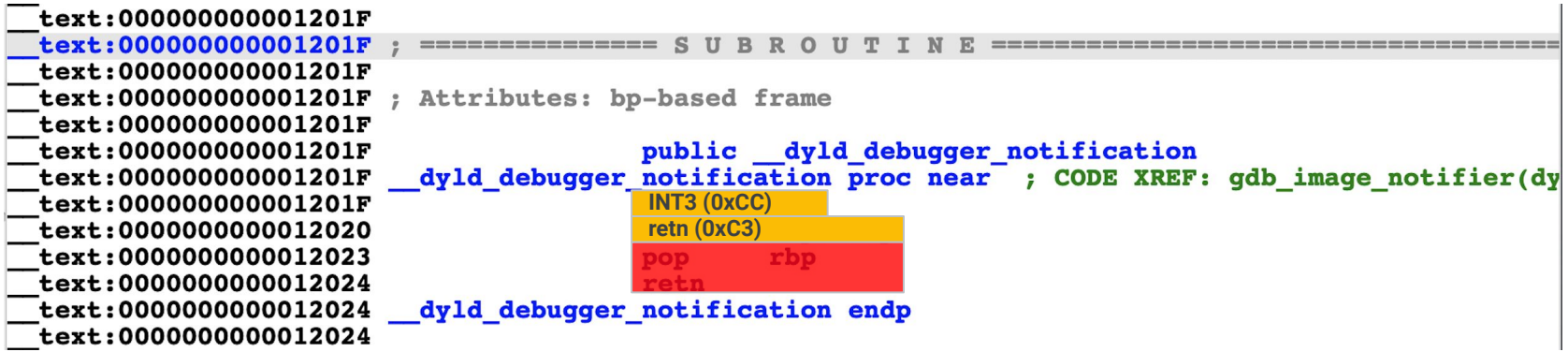
Thanks to @bazad for this screenshot

Get notified when libraries are loaded or unloaded

```
__text:000000000001201F
__text:000000000001201F ; =============== S U B R O U T I N E =======================================
__text:000000000001201F
__text:000000000001201F ; Attributes: bp-based frame
__text:000000000001201F
__text:000000000001201F                         public __dyld_debugger_notification
__text:000000000001201F __dyld_debugger_notification proc near  ; CODE XREF: gdb_image_notifier(dy
__text:000000000001201F                         INT3 (0xCC)
__text:0000000000012020                         retn (0xC3)
__text:0000000000012023                         pop     rbp
__text:0000000000012024                         retn
__text:0000000000012024 __dyld_debugger_notification endp
__text:0000000000012024
```

Thanks to @bazad for this screenshot

Get notified when libraries are loaded or unloaded



Thanks to @bazad for this screenshot

Other challenges?

How does TinyInst work?

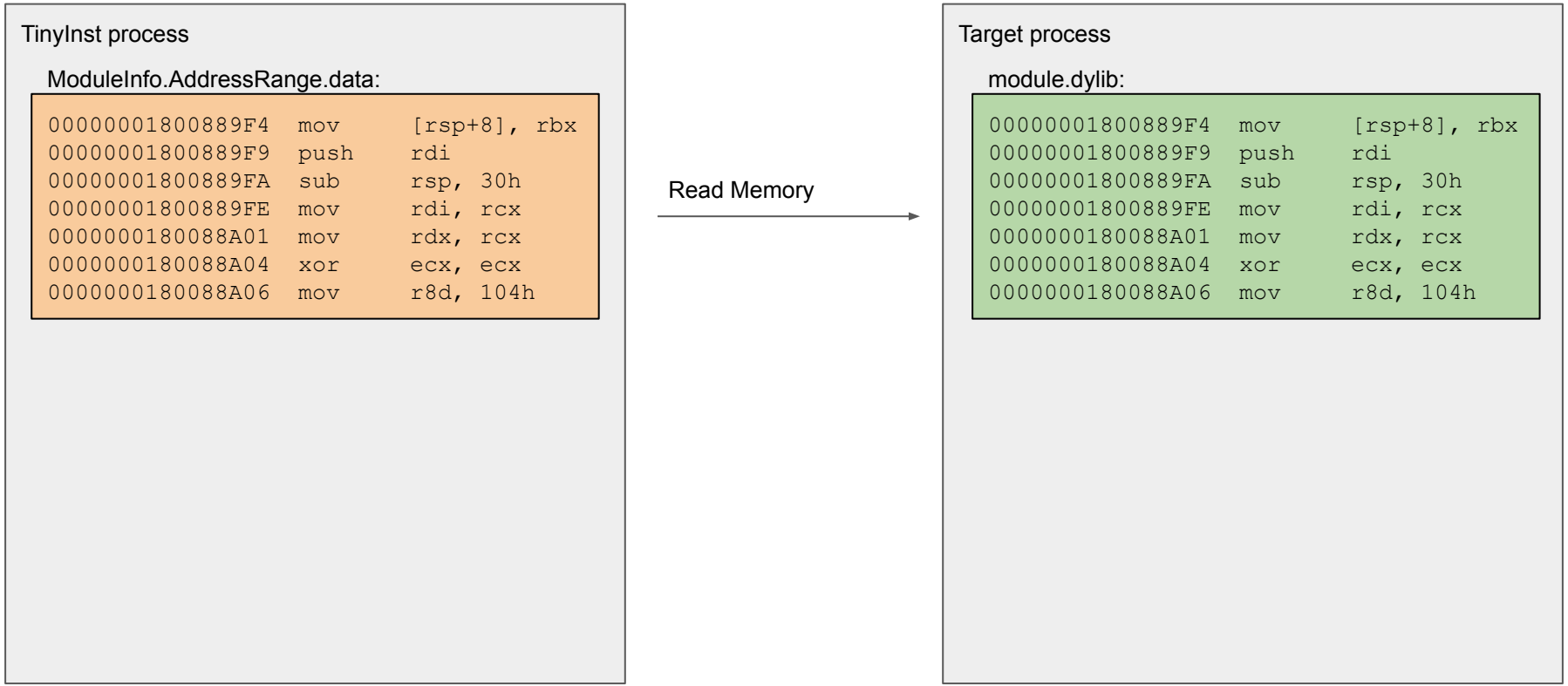
TinyInst process

Target process

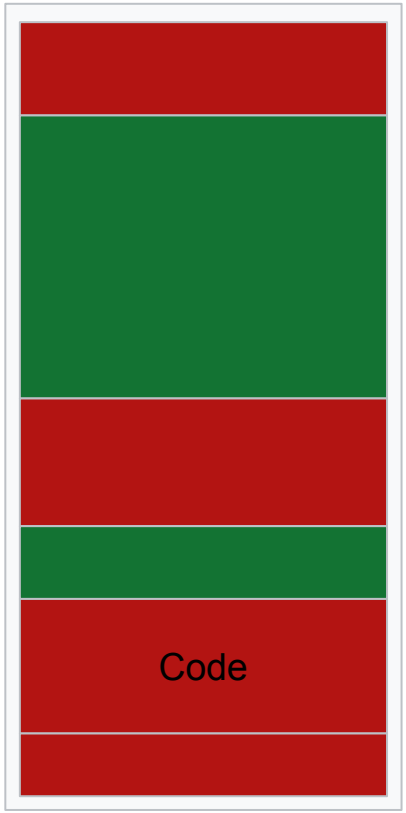
module.dylib:

```
00000001800889F4  mov     [rsp+8], rbx
00000001800889F9  push    rdi
00000001800889FA  sub     rsp, 30h
00000001800889FE  mov     rdi, rcx
0000000180088A01  mov     rdx, rcx
0000000180088A04  xor     ecx, ecx
0000000180088A06  mov     r8d, 104h
```

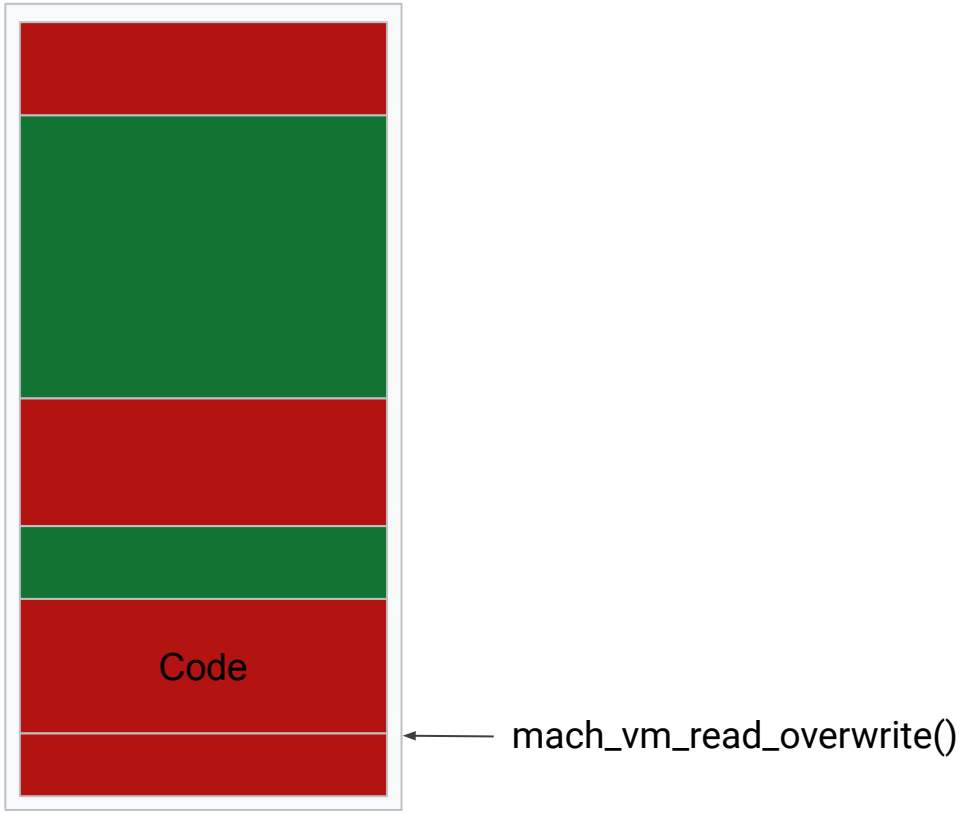
How does TinyInst work?



Read target's memory

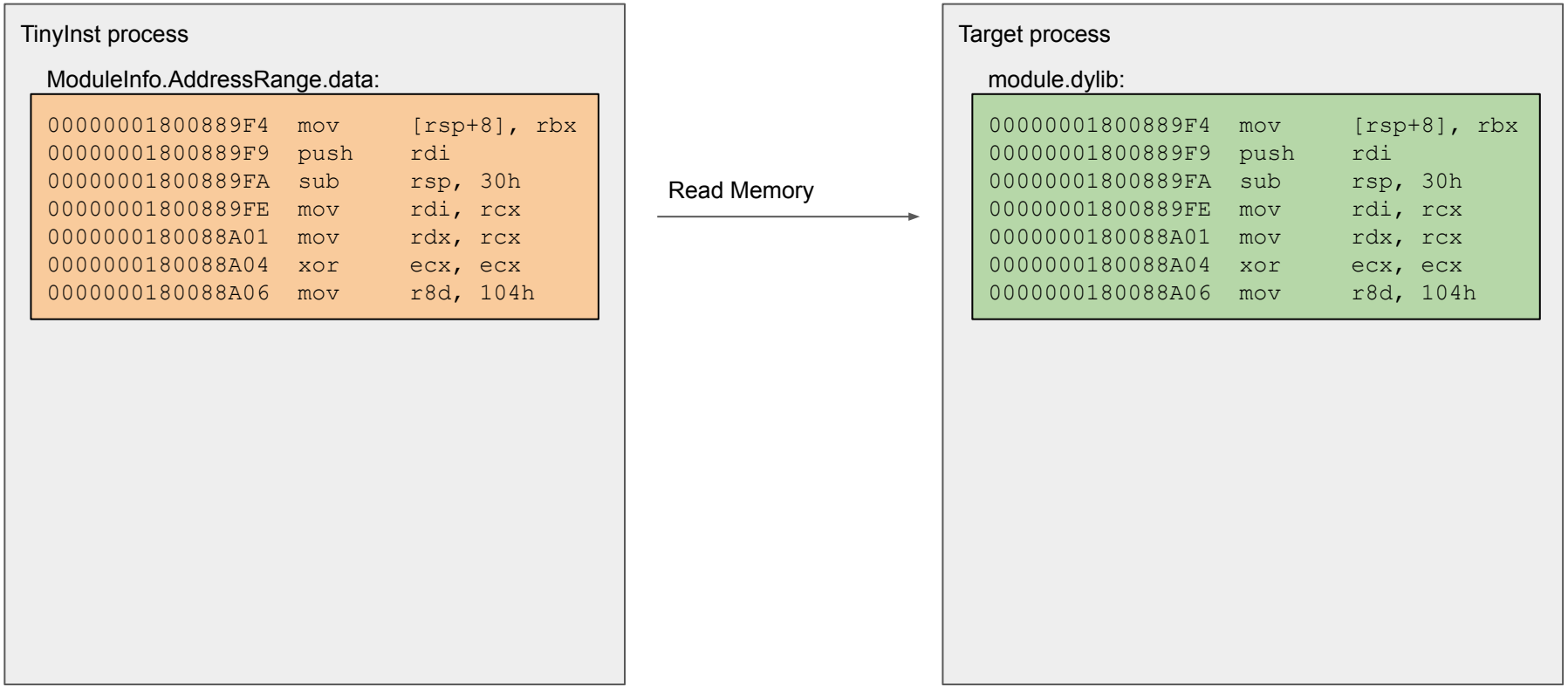


Read target's memory

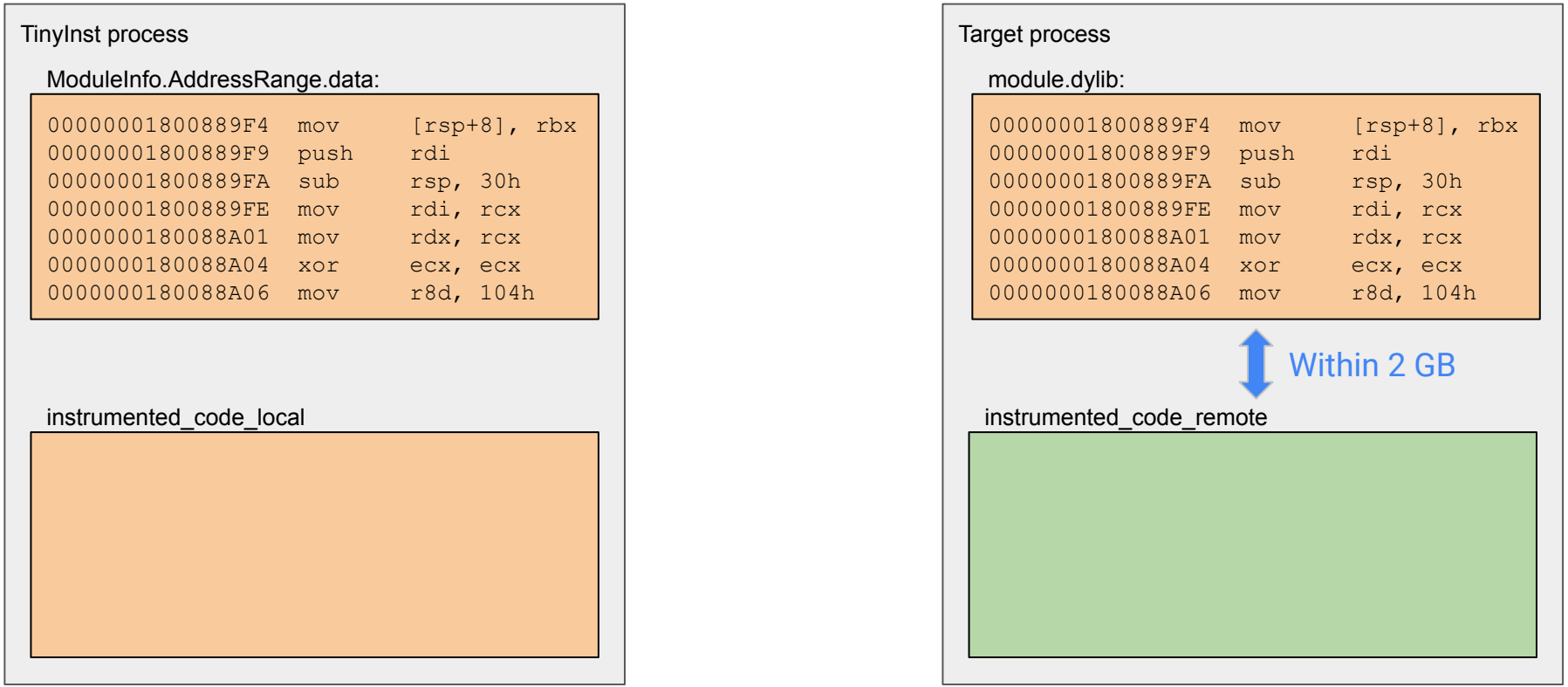


Demo

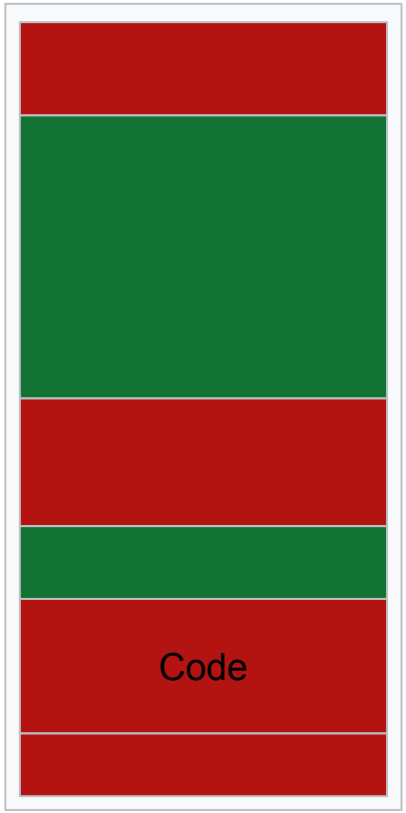
How does TinyInst work?



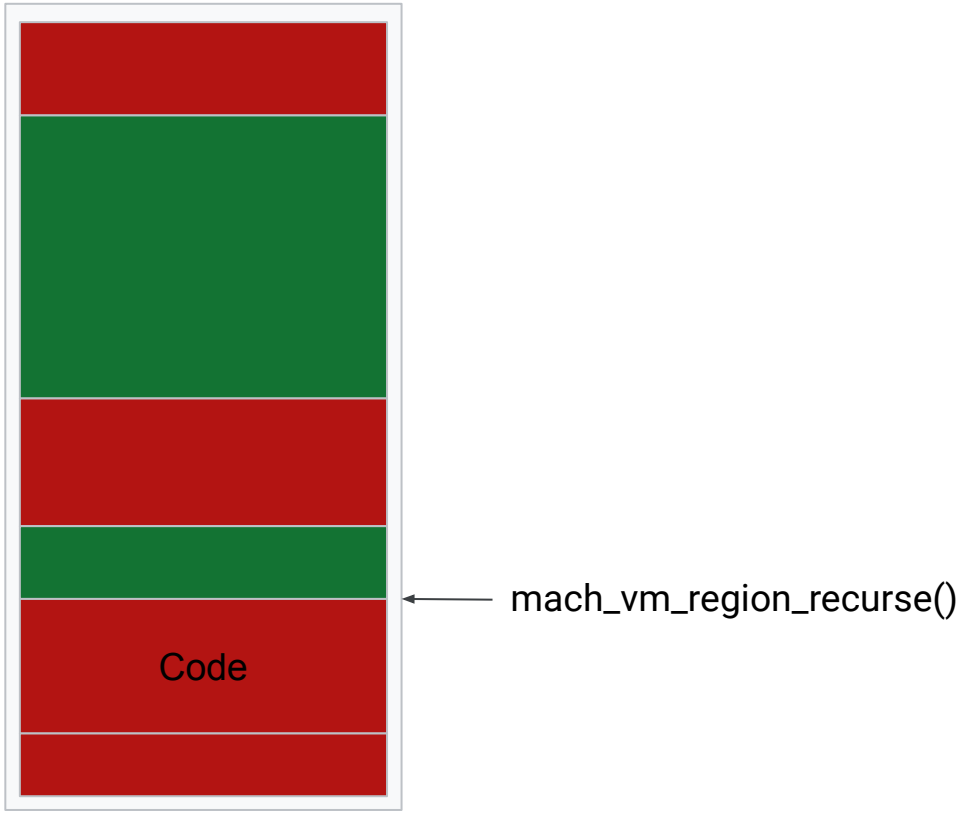
How does TinyInst work?



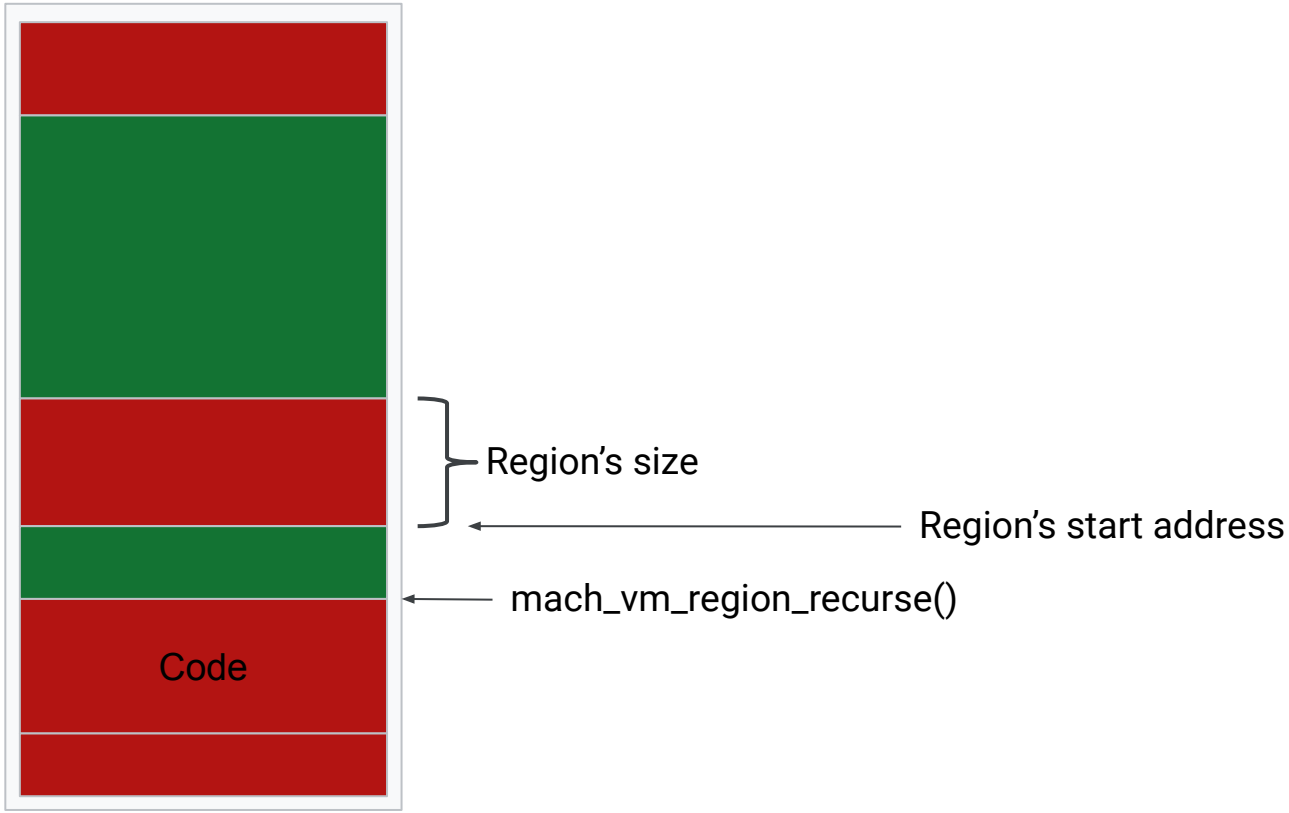
Allocate memory region in the target process



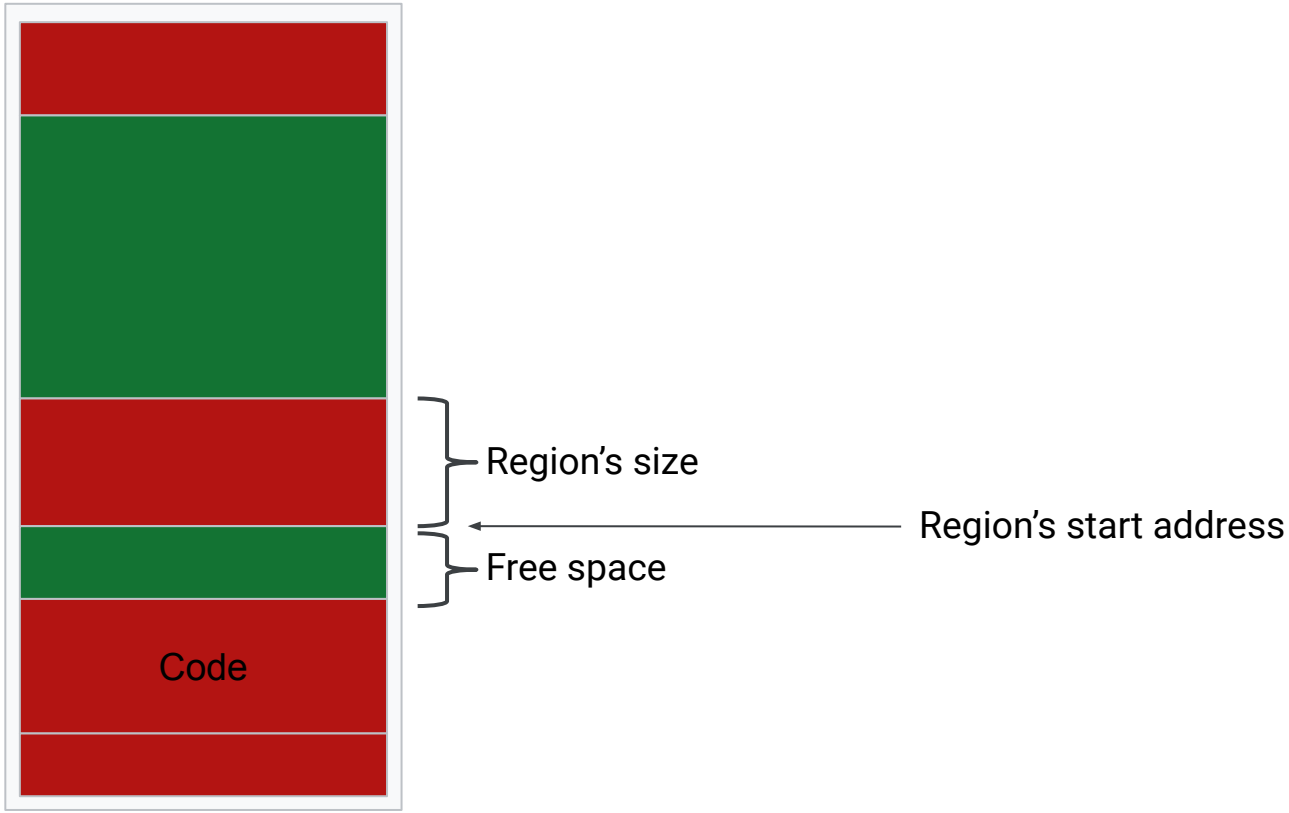
Allocate memory region in the target process



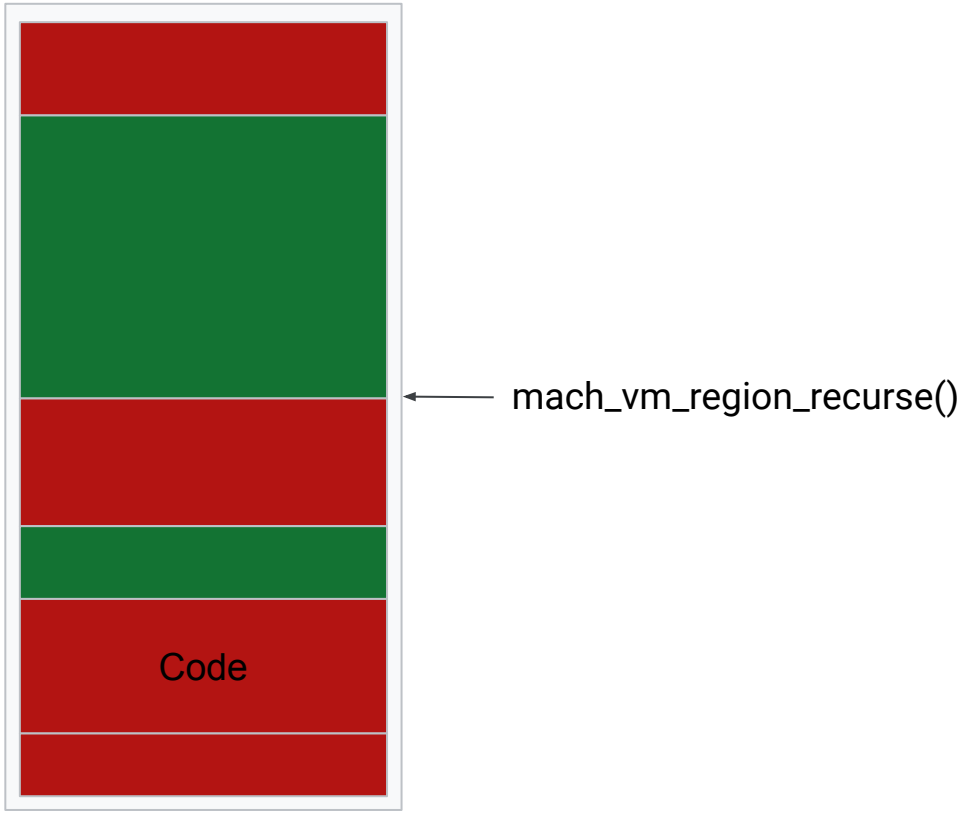
Allocate memory region in the target process



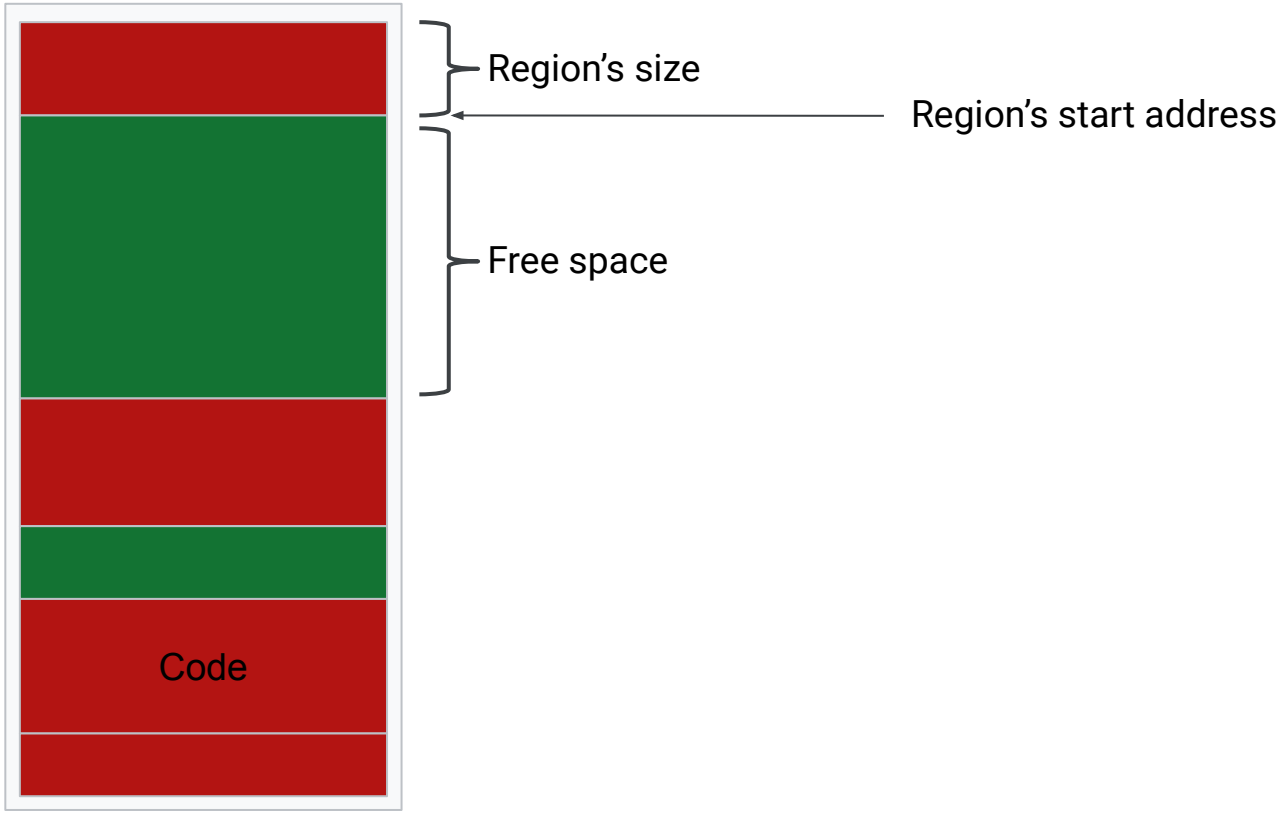
Allocate memory region in the target process



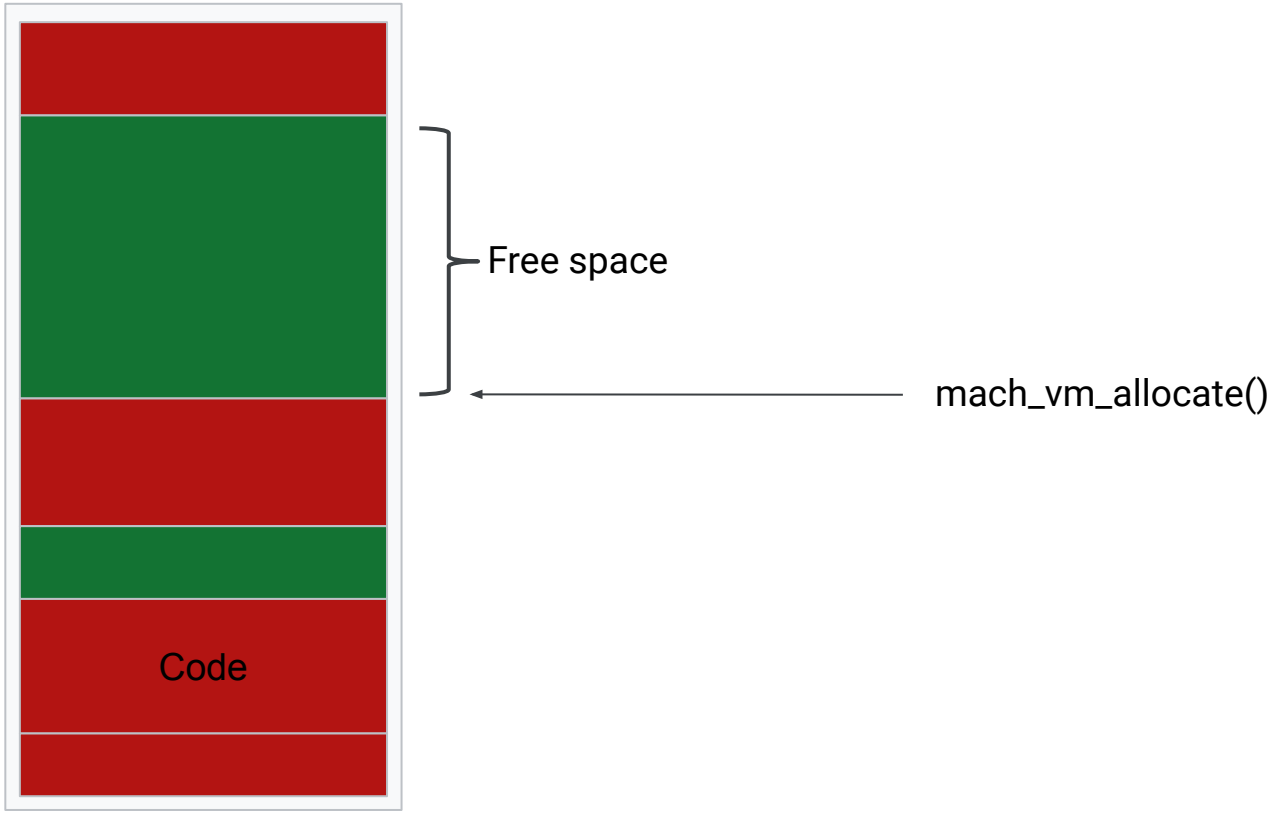
Allocate memory region in the target process



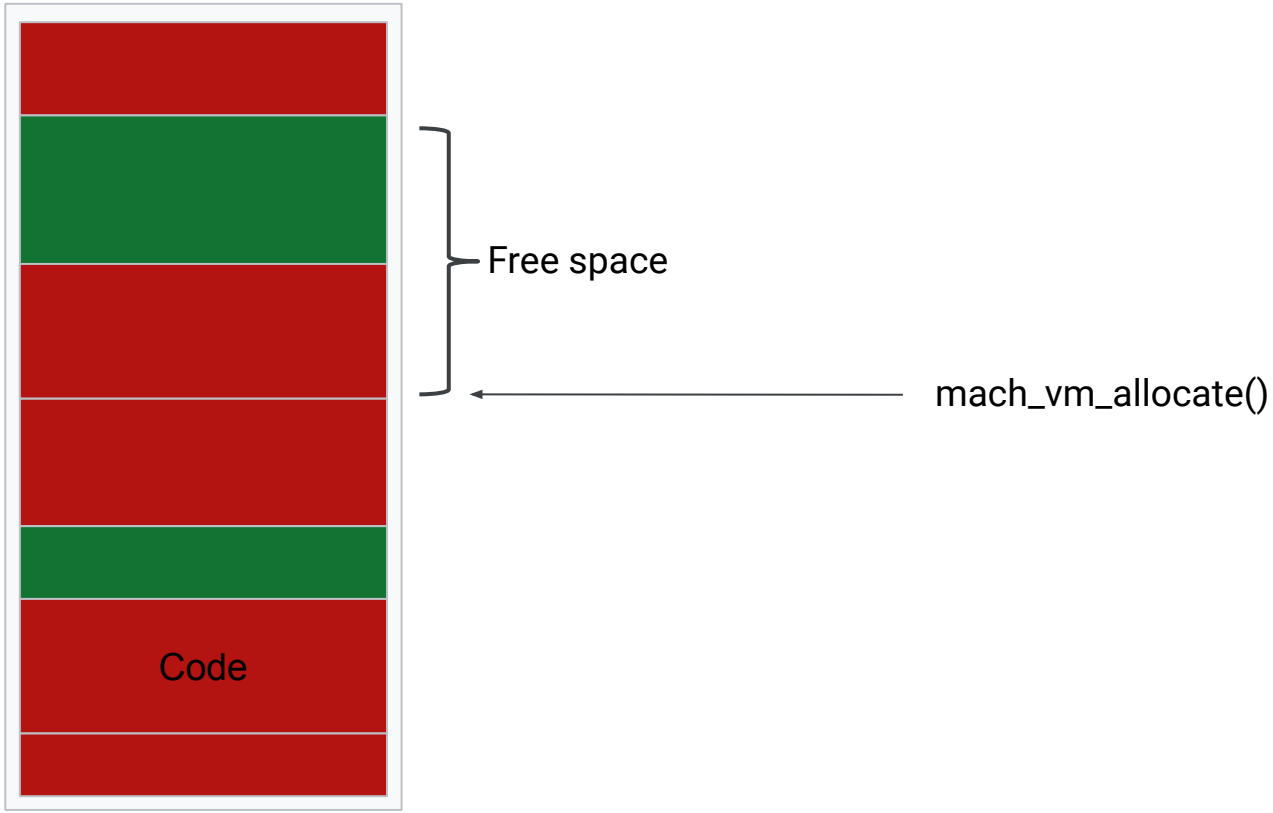
Allocate memory region in the target process



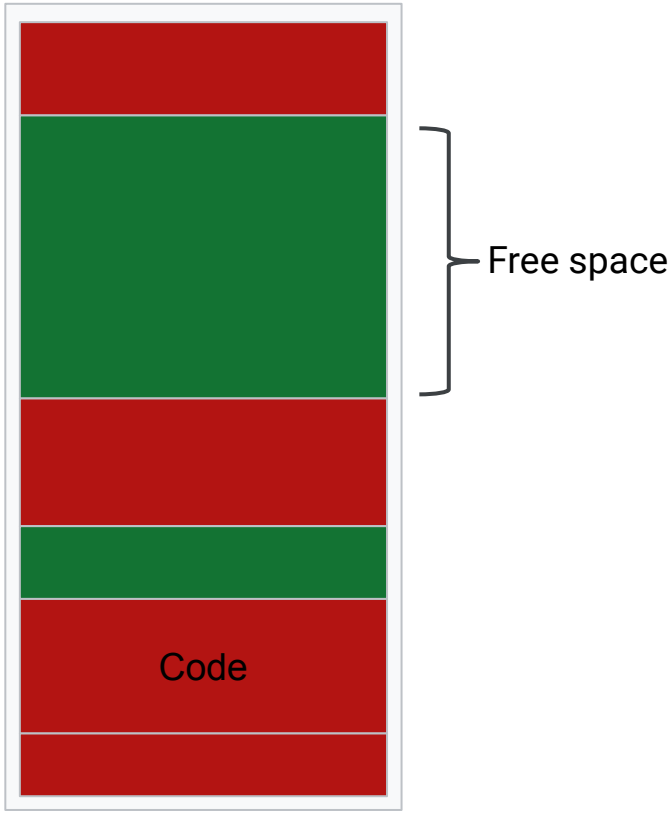
Allocate memory region in the target process



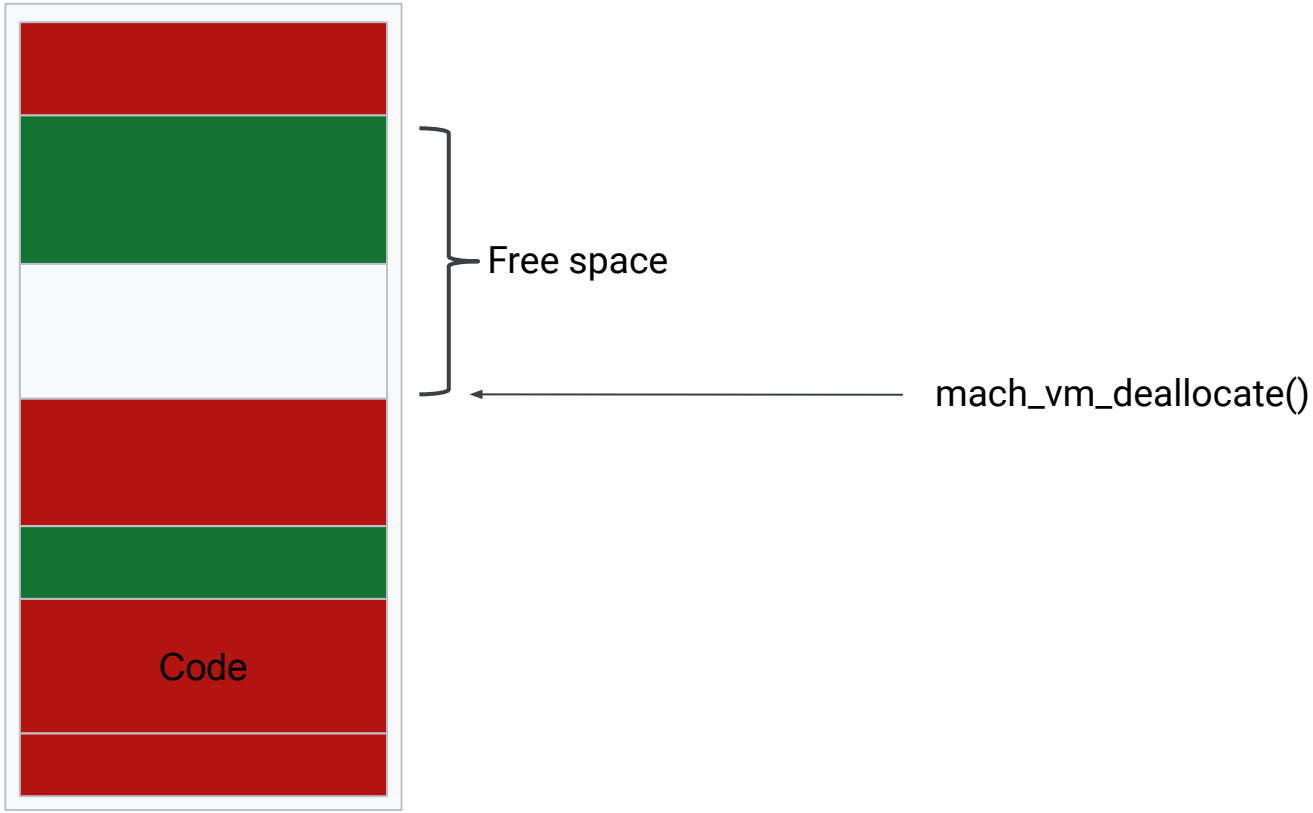
Allocate memory region in the target process



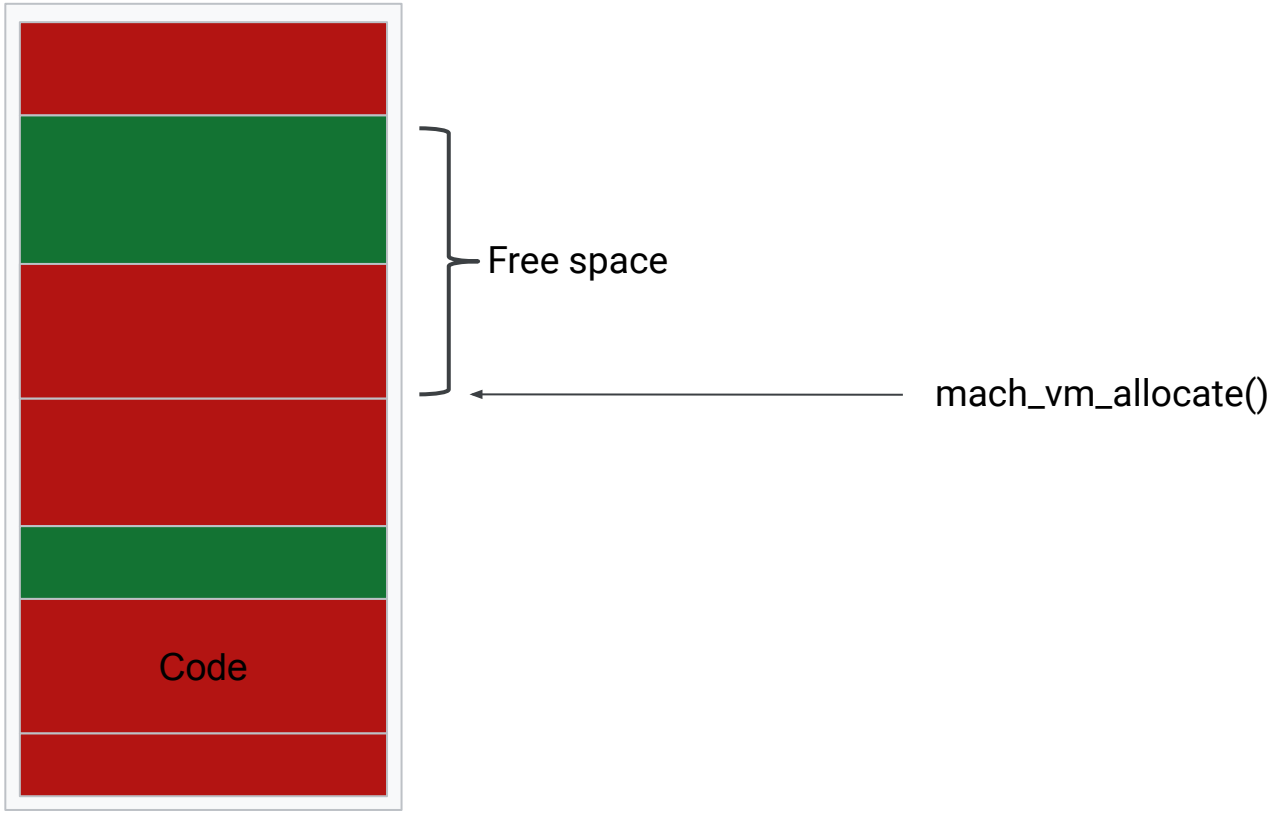
Allocate memory region in the target process



Allocate memory region in the target process

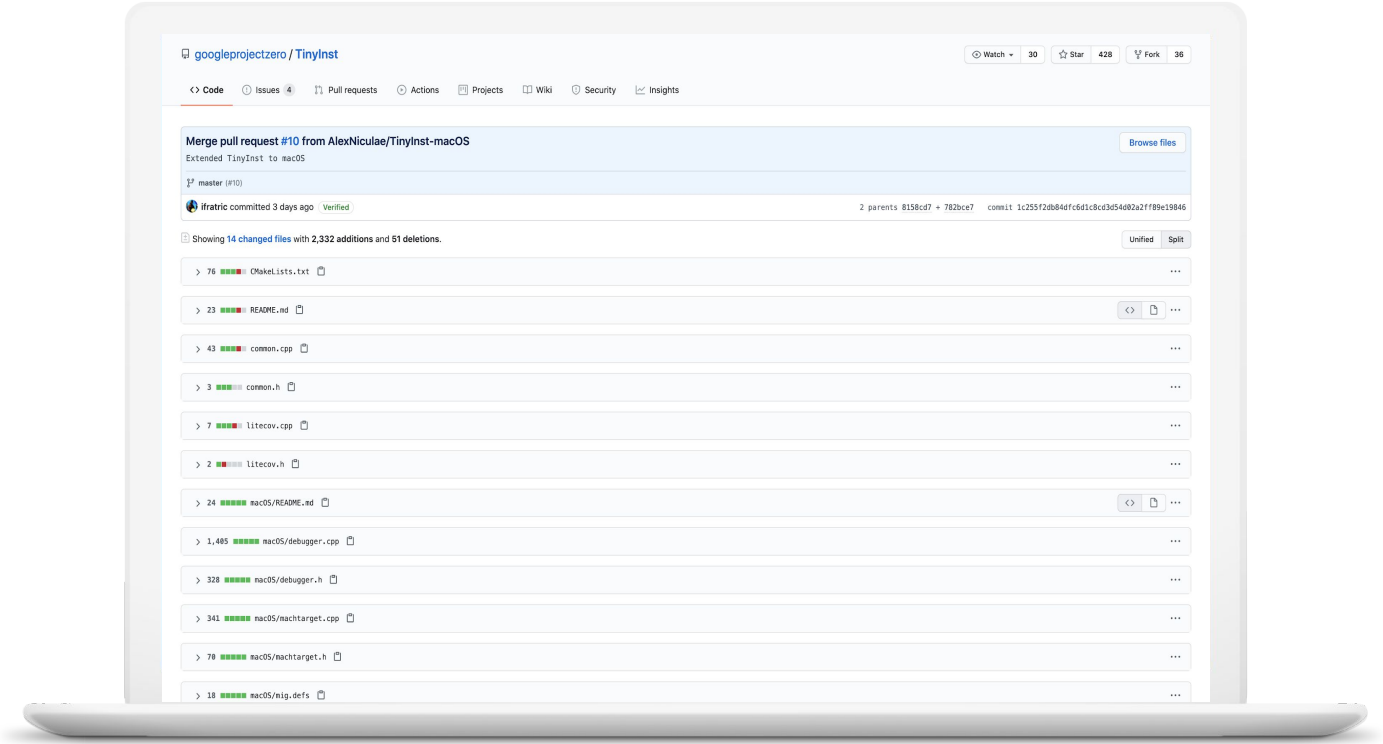


Allocate memory region in the target process



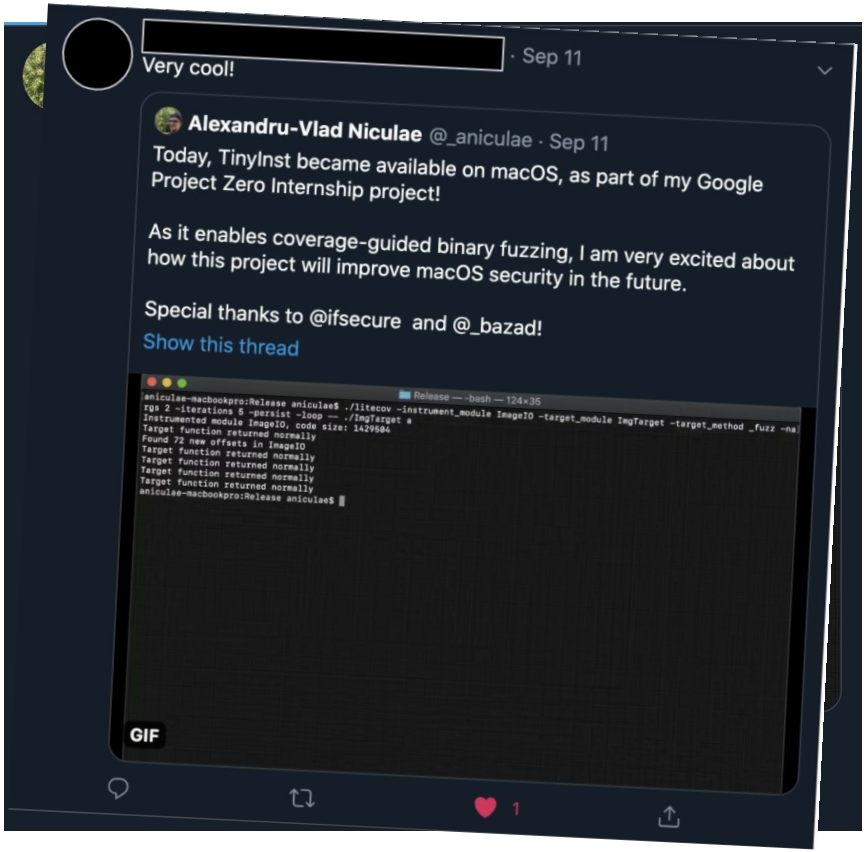
Results

TinyInst on macOS is public



TinyInst on macOS is public

TinyInst on macOS is public



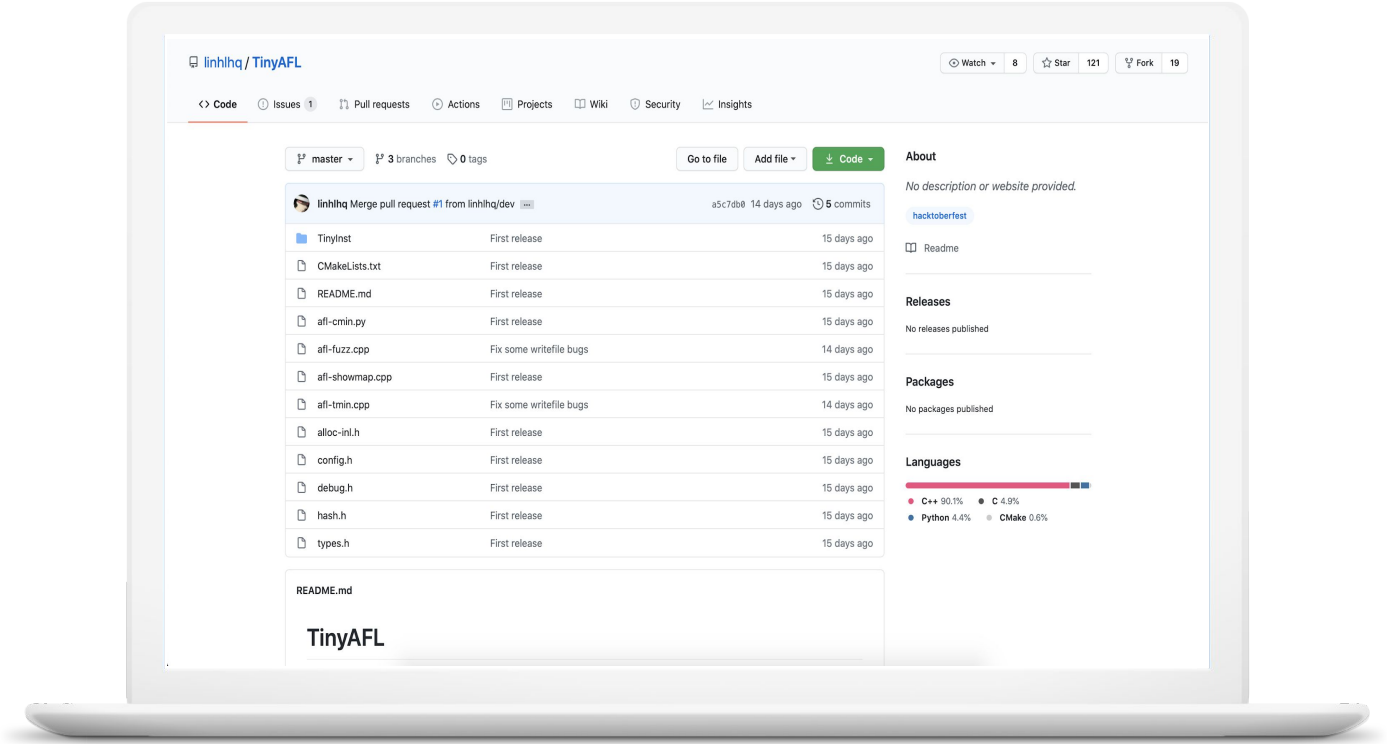
TinyInst on macOS is public

TinyInst on macOS is public

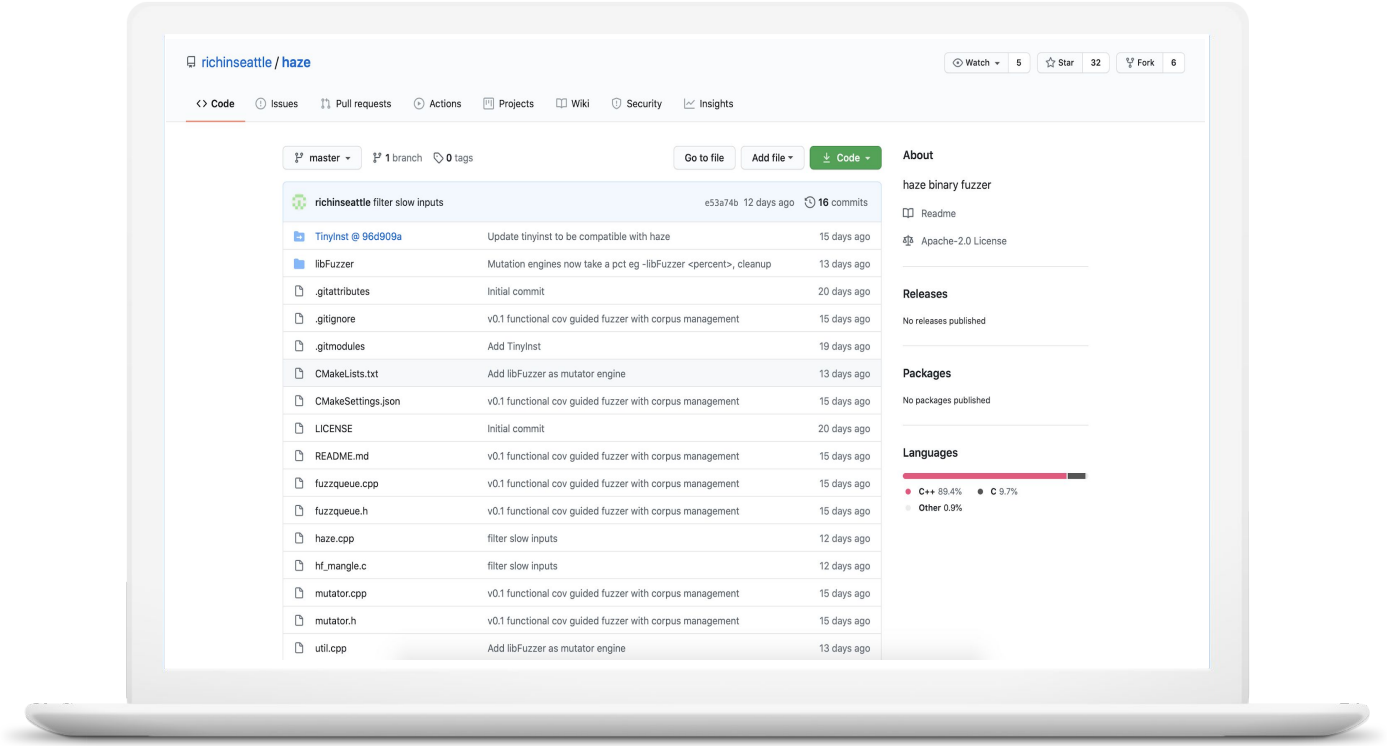
TinyInst on macOS is public

TinyInst Fuzzers

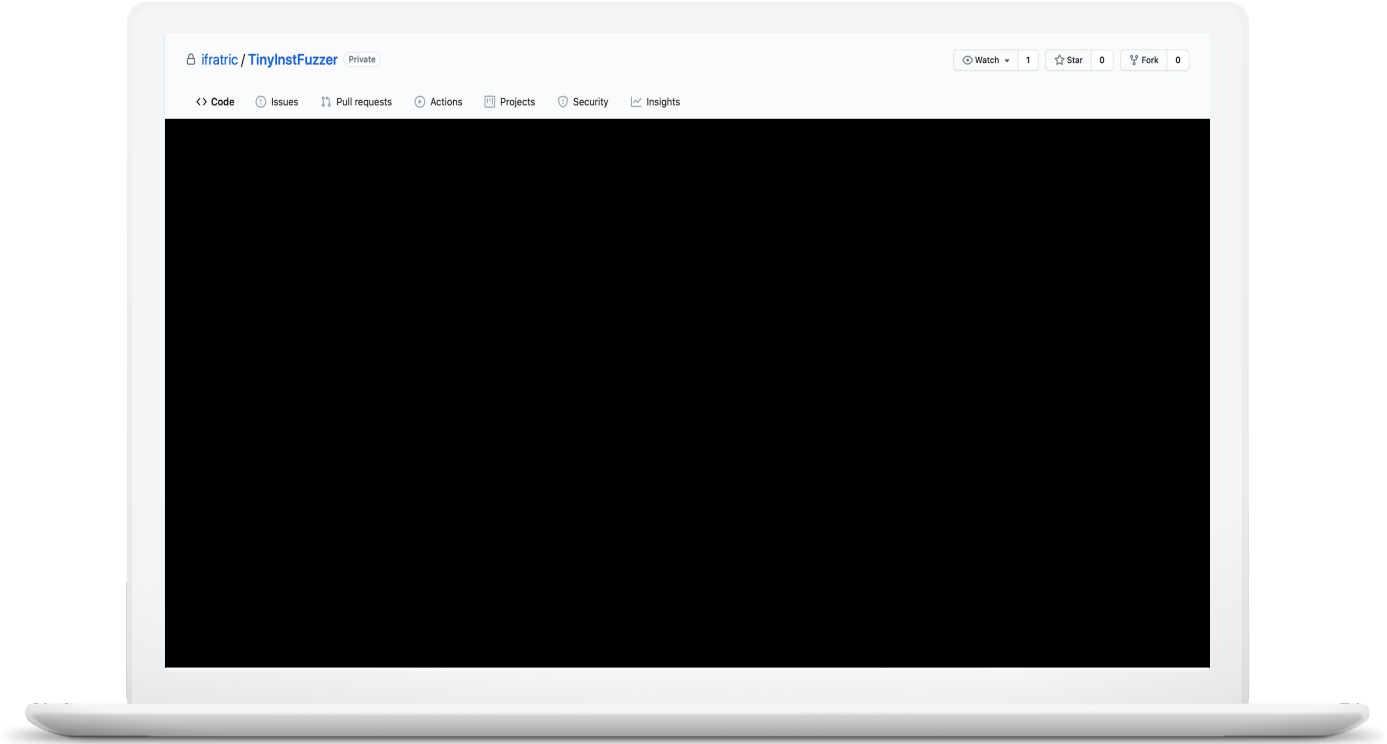
TinyInst Fuzzers



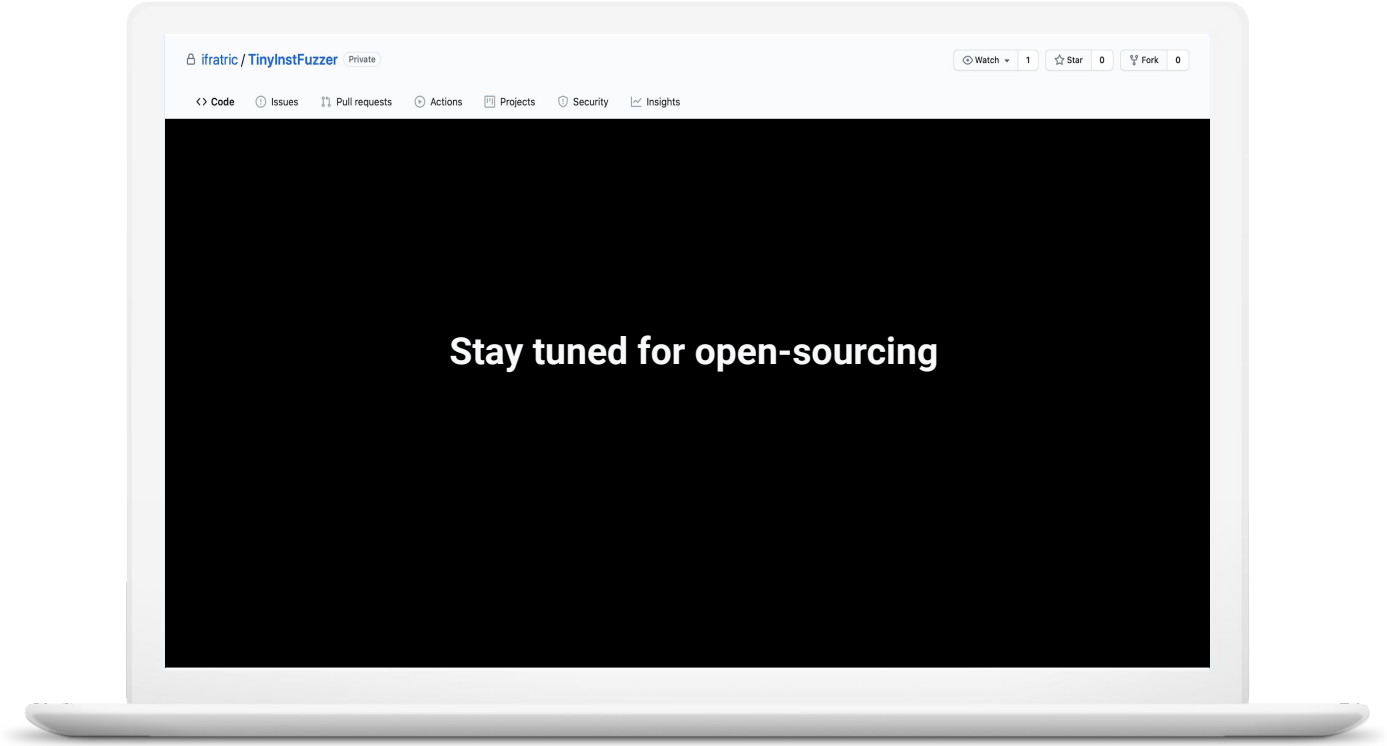
TinyInst Fuzzers



TinyInst Fuzzers



TinyInst Fuzzers



My fuzzing

- Found an Out-of-Bounds Write vulnerability into a closed-source image parsing library.


That's all Folks!

Questions?