

Enabling coverage-guided binary fuzzing on macOS

POC x Zer0Con



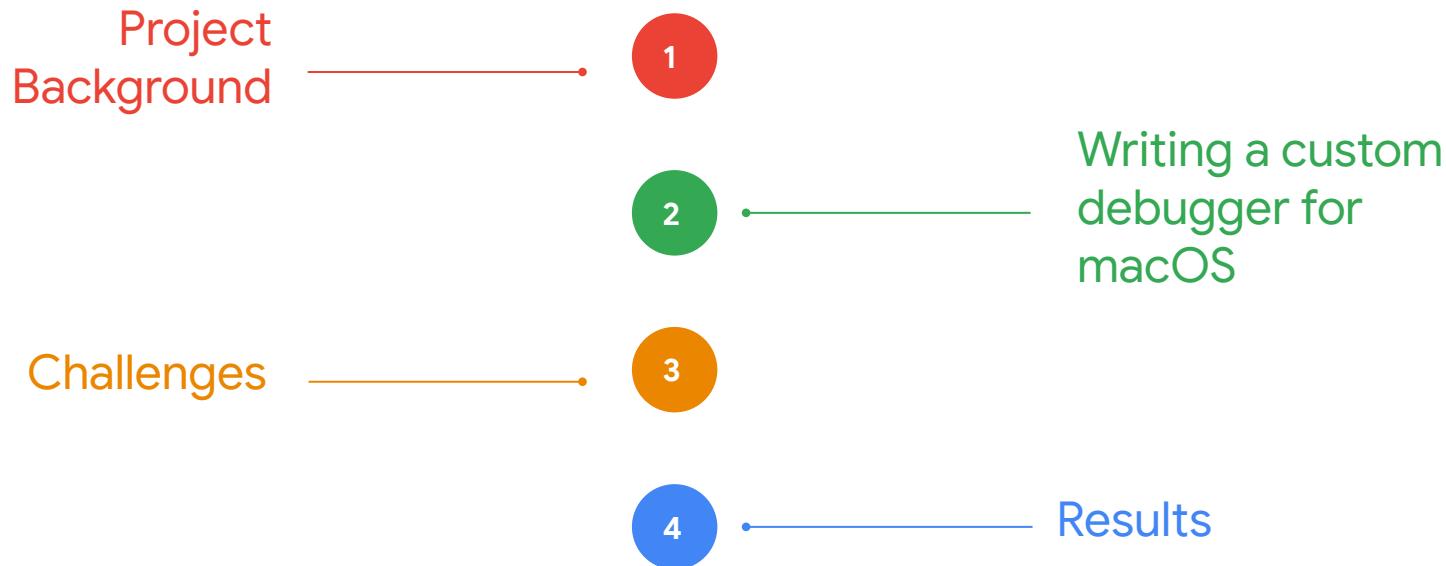
Alexandru-Vlad Niculae



Alexandru-Vlad Niculae

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

Agenda



Project Background

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

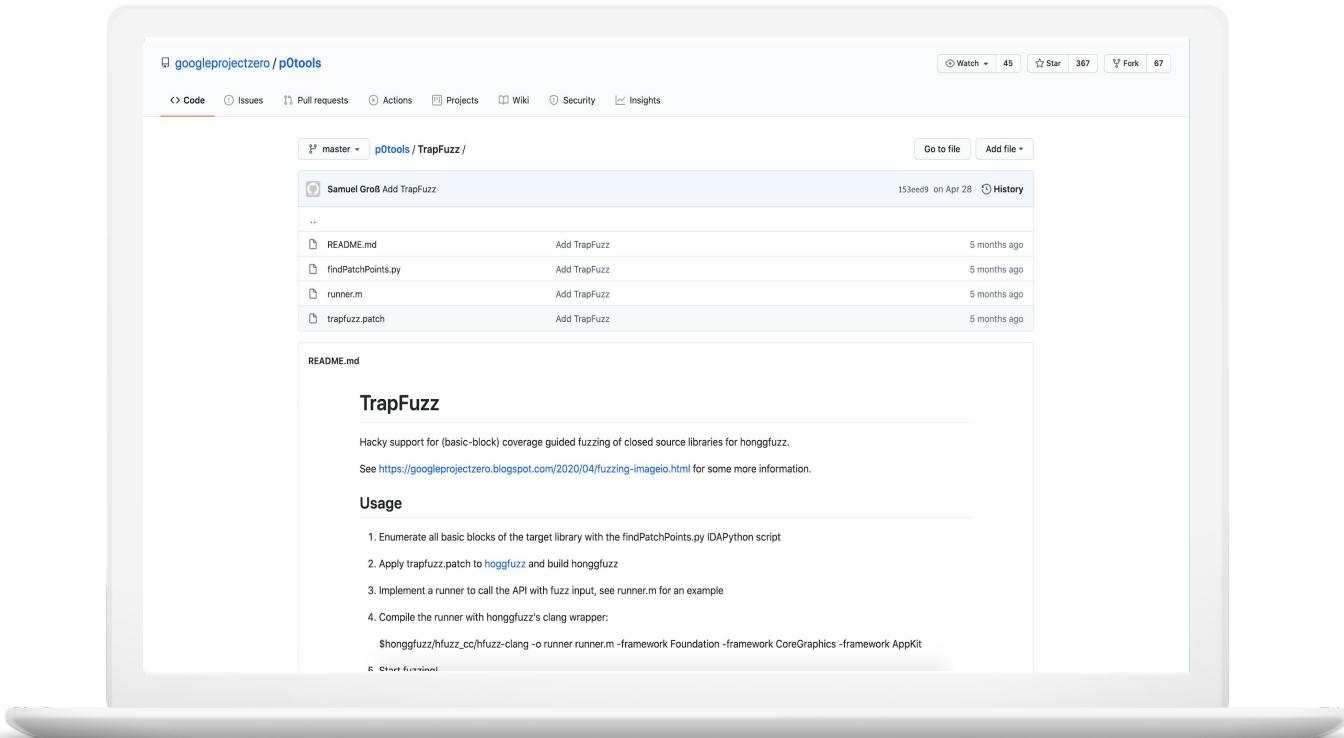
- Effective when dealing with structured inputs (e.g. Javascript code).
- Requires a lot of non-reusable effort.

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

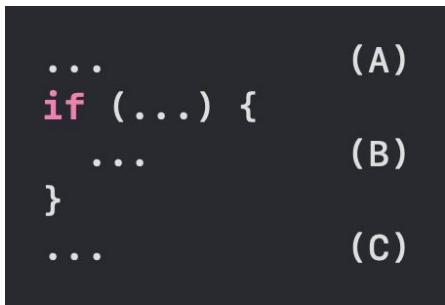
```
...          (A)
if (...) {
    ...
}
...          (C)
```

Counts the number of basic blocks
reached during runtime.

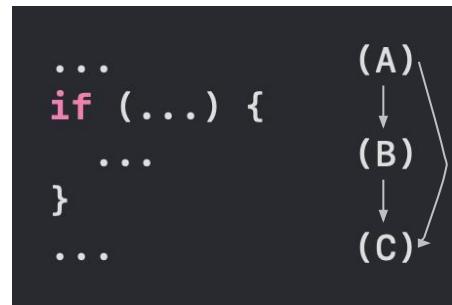
Code Coverage

“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



Edge Coverage



Counts the number of basic blocks reached during runtime.

Counts the number of edges hit at runtime.

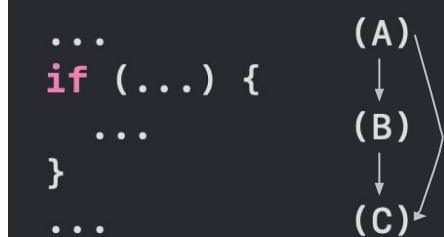
Code Coverage

"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)
```

Edge Coverage



Counts the number of basic blocks reached during runtime.

Compare Coverage

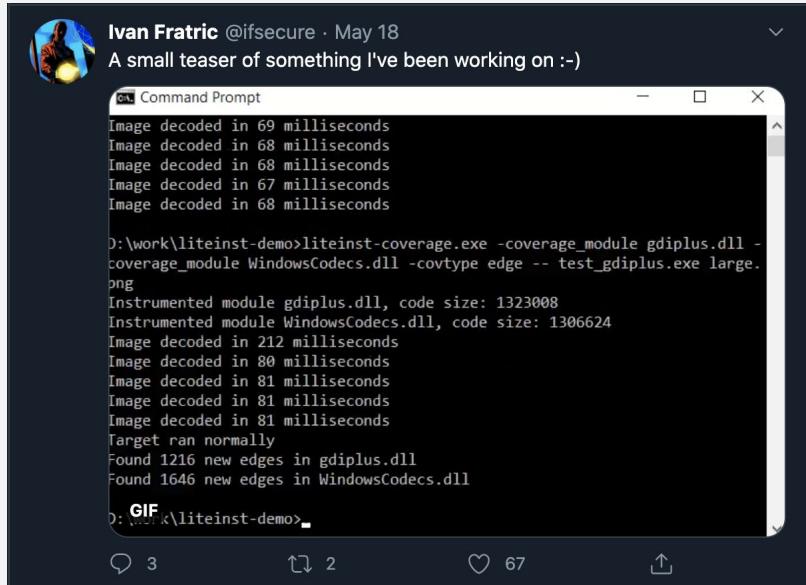
```
...          (A)  
if (var == MAGIC_NUMBER) {  
...          (B)  
}  
...          (C)
```

Counts the number of edges hit at runtime.

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



Ivan Fratric @ifsecure · May 18
A small teaser of something I've been working on :-)

Command Prompt

```
Image decoded in 69 milliseconds
Image decoded in 68 milliseconds
Image decoded in 68 milliseconds
Image decoded in 67 milliseconds
Image decoded in 68 milliseconds

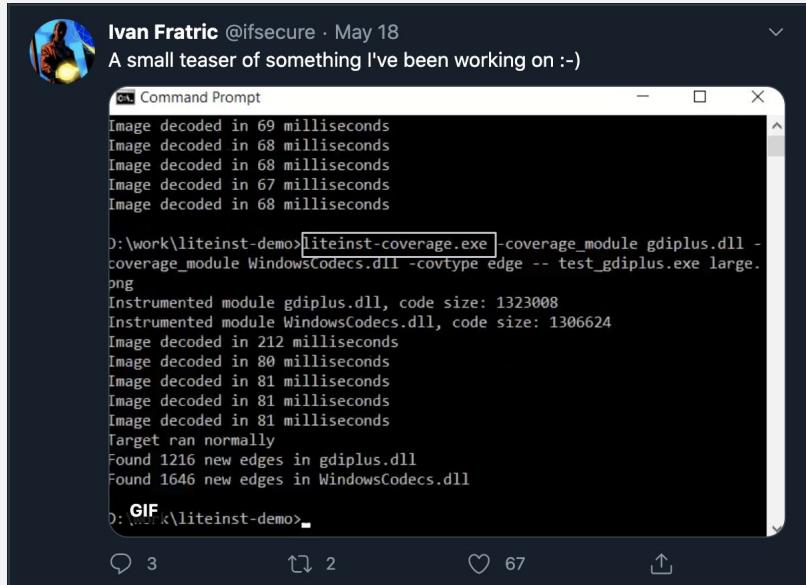
D:\work\liteinst-demo>liteinst-coverage.exe -coverage_module gdiplus.dll -coverage_module WindowsCodecs.dll -covtype edge -- test_gdiplus.exe large.png
Instrumented module gdiplus.dll, code size: 1323008
Instrumented module WindowsCodecs.dll, code size: 1306624
Image decoded in 212 milliseconds
Image decoded in 80 milliseconds
Image decoded in 81 milliseconds
Image decoded in 81 milliseconds
Image decoded in 81 milliseconds
Target ran normally
Found 1216 new edges in gdiplus.dll
Found 1646 new edges in WindowsCodecs.dll
```

D:\work\liteinst-demo>

3 2 67

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



Ivan Fratric @ifsecure · May 18
A small teaser of something I've been working on :-)

Command Prompt

```
Image decoded in 69 milliseconds
Image decoded in 68 milliseconds
Image decoded in 68 milliseconds
Image decoded in 67 milliseconds
Image decoded in 68 milliseconds

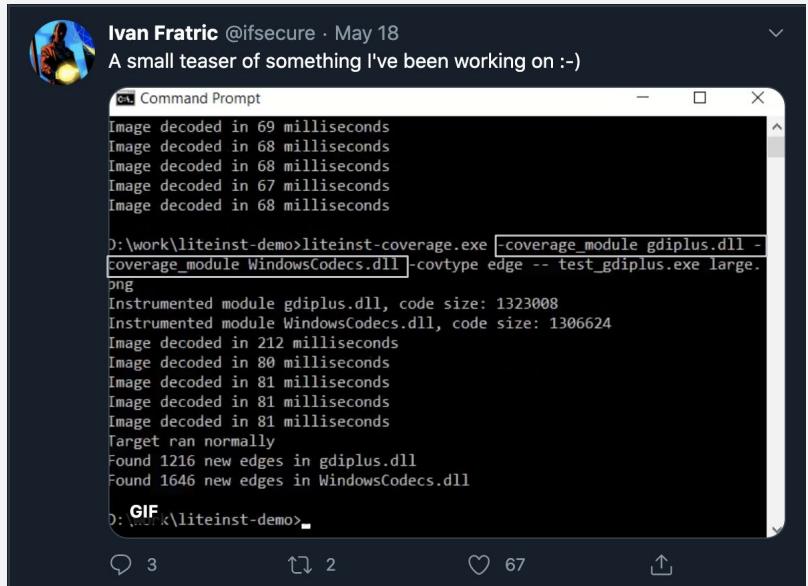
D:\work\liteinst-demo>liteinst-coverage.exe -coverage_module gdiplus.dll -
-coverage_module WindowsCodecs.dll -covtype edge -- test_gdiplus.exe large.
png
Instrumented module gdiplus.dll, code size: 1323008
Instrumented module WindowsCodecs.dll, code size: 1306624
Image decoded in 212 milliseconds
Image decoded in 80 milliseconds
Image decoded in 81 milliseconds
Image decoded in 81 milliseconds
Image decoded in 81 milliseconds
Target ran normally
Found 1216 new edges in gdiplus.dll
Found 1646 new edges in WindowsCodecs.dll

D:\work\liteinst-demo>
```

3 2 67

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



Ivan Fratric @ifsecure · May 18
A small teaser of something I've been working on :-)

Command Prompt

```
Image decoded in 69 milliseconds
Image decoded in 68 milliseconds
Image decoded in 68 milliseconds
Image decoded in 67 milliseconds
Image decoded in 68 milliseconds

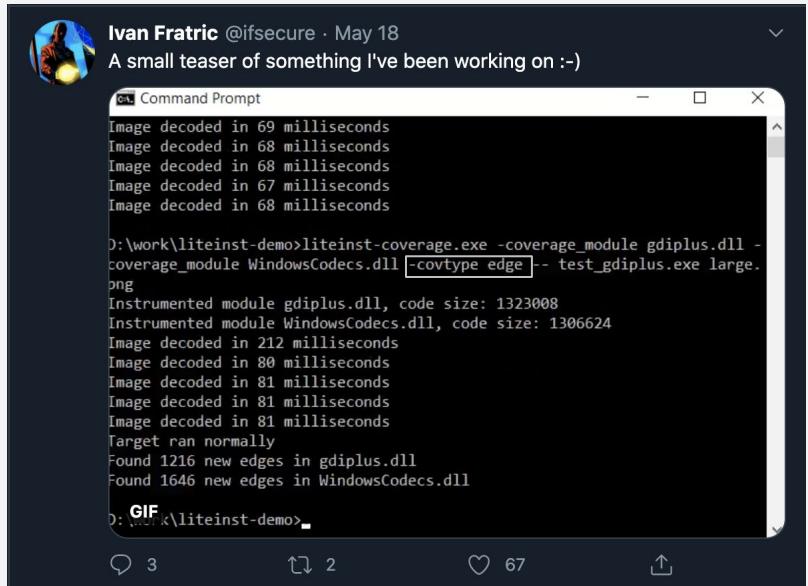
D:\work\liteinst-demo>liteinst-coverage.exe [-coverage_module gdiplus.dll -coverage_module WindowsCodecs.dll] covtype edge -- test_gdiplus.exe large.
png
Instrumented module gdiplus.dll, code size: 1323008
Instrumented module WindowsCodecs.dll, code size: 1306624
Image decoded in 212 milliseconds
Image decoded in 80 milliseconds
Image decoded in 81 milliseconds
Image decoded in 81 milliseconds
Image decoded in 81 milliseconds
Target ran normally
Found 1216 new edges in gdiplus.dll
Found 1646 new edges in WindowsCodecs.dll

D:\work\liteinst-demo>
```

3 2 67

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



Ivan Fratric @ifsecure · May 18
A small teaser of something I've been working on :-)

Command Prompt

```
Image decoded in 69 milliseconds
Image decoded in 68 milliseconds
Image decoded in 68 milliseconds
Image decoded in 67 milliseconds
Image decoded in 68 milliseconds

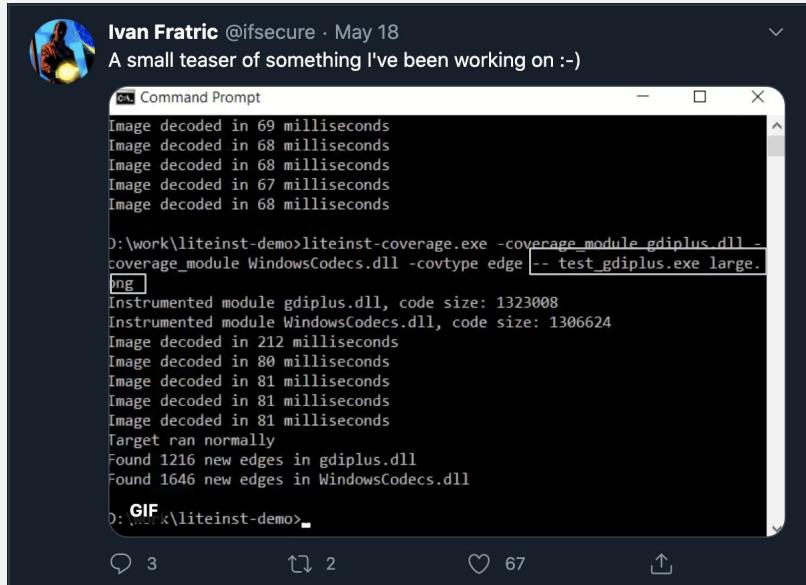
D:\work\liteinst-demo>liteinst-coverage.exe -coverage_module gdiplus.dll -
-coverage_module WindowsCodecs.dll [covtype edge]- test_gdiplus.exe large.
png
Instrumented module gdiplus.dll, code size: 1323008
Instrumented module WindowsCodecs.dll, code size: 1306624
Image decoded in 212 milliseconds
Image decoded in 80 milliseconds
Image decoded in 81 milliseconds
Image decoded in 81 milliseconds
Image decoded in 81 milliseconds
Target ran normally
Found 1216 new edges in gdiplus.dll
Found 1646 new edges in WindowsCodecs.dll

D:\work\liteinst-demo>
```

3 2 67

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



Ivan Fratric @ifsecure · May 18
A small teaser of something I've been working on :-)

Command Prompt

```
Image decoded in 69 milliseconds
Image decoded in 68 milliseconds
Image decoded in 68 milliseconds
Image decoded in 67 milliseconds
Image decoded in 68 milliseconds

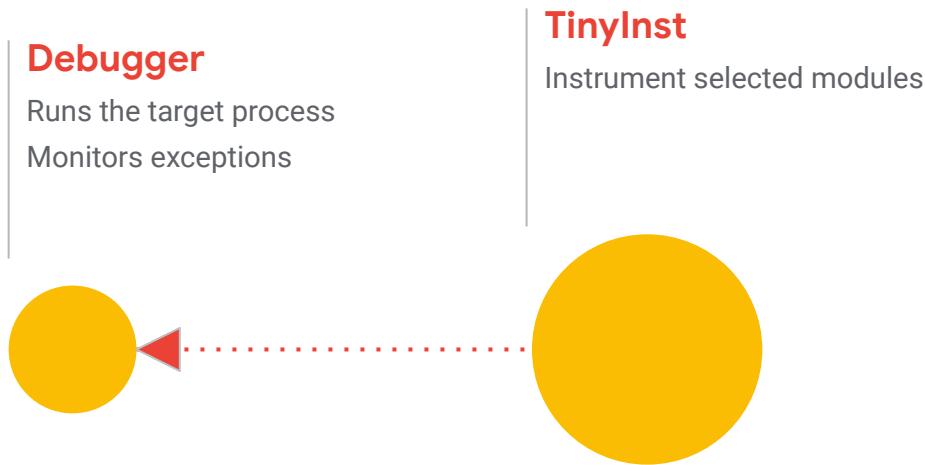
D:\work\liteinst-demo>liteinst-coverage.exe -coverage_module gdiplus.dll -
-coverage_module WindowsCodecs.dll -covtype edge [- test_gdiplus.exe large.
[redacted]
Instrumented module gdiplus.dll, code size: 1323008
Instrumented module WindowsCodecs.dll, code size: 1306624
Image decoded in 212 milliseconds
Image decoded in 80 milliseconds
Image decoded in 81 milliseconds
Image decoded in 81 milliseconds
Image decoded in 81 milliseconds
Target ran normally
Found 1216 new edges in gdiplus.dll
Found 1646 new edges in WindowsCodecs.dll

D:\work\liteinst-demo>
```

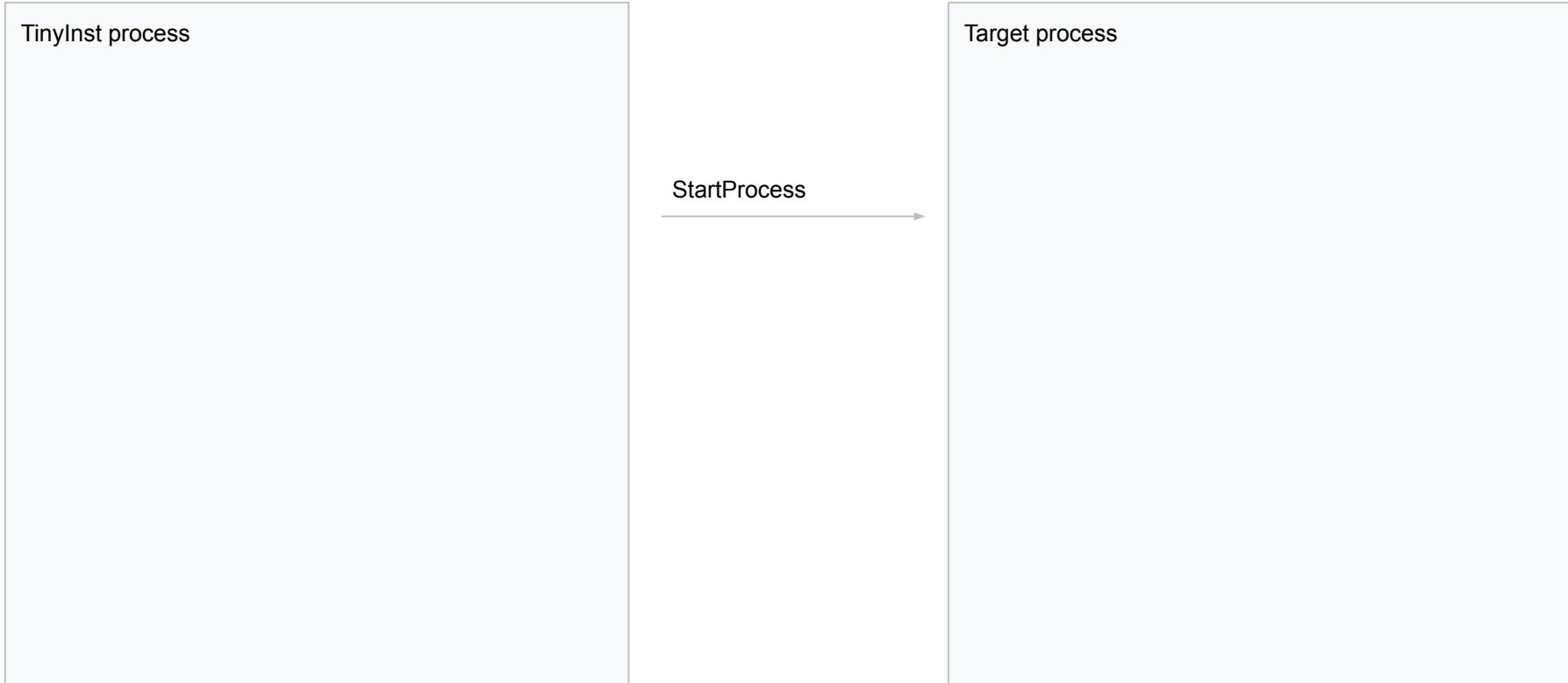
3 2 67

How is TinyInst instrumenting binaries?

How is TinyInst instrumenting binaries?



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
```

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_local

VirtualAllocEx

mach_vm_allocate,
mach_vm_protect

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
```

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_local



instrumented_code_remote



Target wants to execute code in an “instrumented” module

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
```

RIP

→

instrumented_code_remote



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

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
```



RIP

instrumented_code_local



instrumented_code_remote



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

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
```

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
```



RIP

instrumented_code_local

instrumented_code_remote

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.

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
```

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
```



RIP

instrumented_code_local

```
mov    [rsp+8], rbx
```

instrumented_code_remote

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.

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
```

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
```



RIP

instrumented_code_local

```
mov    [rsp+8], rbx  
push  rdi
```

instrumented_code_remote

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.

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
```

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_local

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

instrumented_code_remote

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.

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
```

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_local

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

instrumented_code_remote

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

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  
...  
...
```

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
```



RIP

WriteProcessMemory

mach_vm_write

instrumented_code_remote

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

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
...
...
```

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
```

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_local

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

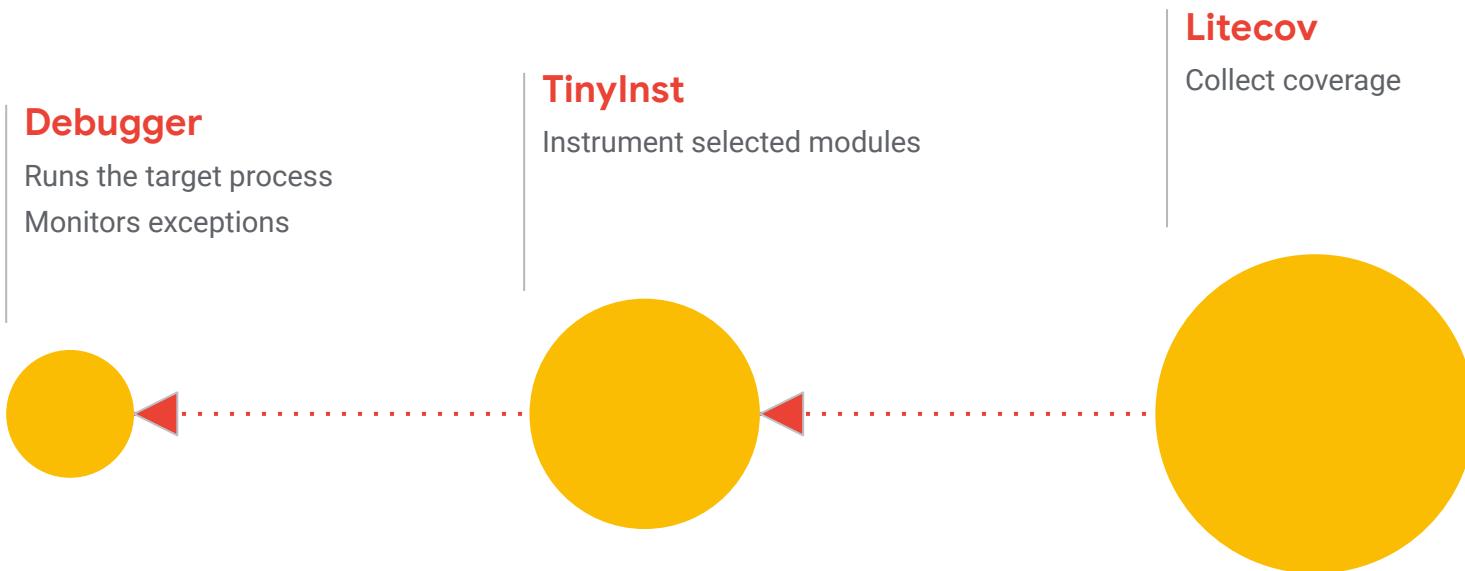
instrumented_code_remote

```
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?

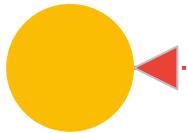


How is TinyInst used to extract code coverage?

Platform dependent

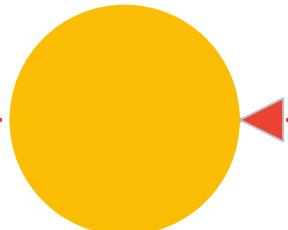
Debugger

Runs the target process
Monitors exceptions



TinyInst

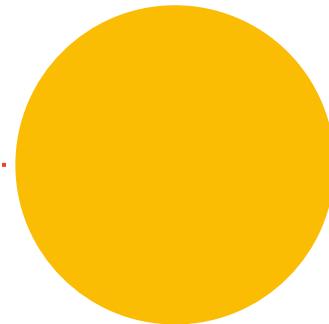
Instrument selected modules



Platform independent

Litecov

Collect 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

The screenshot shows a Microsoft Docs page titled "Creating a Basic Debugger". The page is part of the "Windows Dev Center" under the "Windows" category. The URL is <https://docs.microsoft.com/windows/desktop/Debug/basic-debugger>. The page content includes a sidebar with navigation links like "Error Handling", "Basic Debugging", "About Basic Debugging", "Using Basic Debugging", and "Creating a Basic Debugger" (which is highlighted). The main content area discusses attaching a debugger to a running process and writing the debugger's main loop. At the bottom, there are links for "Download PDF", "English (United States)", "Theme", and various footer links including "Previous Version Docs", "Blog", "Contribute", "Privacy & Cookies", "Terms of Use", "Site Feedback", "Trademarks", and "© Microsoft 2020".

Available resources

macOS

Windows

The screenshot shows a Microsoft Docs page titled "Writing the Debugger's Main Loop" from May 01, 2018. The page has a "Filter by title" sidebar on the left containing links like Error Handling, Basic Debugging, About Basic Debugging, Using Basic Debugging, Using Basic Debugging, Configuring Automatic Debugging, Creating a Basic Debugger, Creating a Basic Debugger, Debugging a Running Process, Writing the Debugger's Main Loop, Communicating with the Debugger, Debugging Reference, Debug Help Library, Structured Exception Handling, Wait Chain Traversal, and Intel AVX. The main content discusses the debugger's main loop, the `WaitForDebugEvent` function, and the `GetThreadContext`, `GetThreadSelectorEntry`, `ReadProcessMemory`, `SetThreadContext`, and `WriteProcessMemory` functions. It also covers memory management with `VirtualAlloc`, `VirtualAllocEx`, and `VirtualFree`. A C++ code block at the bottom shows event handlers for thread creation, process creation, exit, and debug events.

```
#include <windows.h>
DMODL OnCreateThreadDebugEvent(const LPDEBUG_EVENT);
DMODL OnCreateProcessDebugEvent(const LPDEBUG_EVENT);
DMODL OnExitProcessDebugEvent(const LPDEBUG_EVENT);
DMODL OnThreadDebugEvent(const LPDEBUG_EVENT);
DMODL OnLoadModuleEvent(const LPDEBUG_EVENT);
DMODL OnUnloadModuleEvent(const LPDEBUG_EVENT);
DMODL OnDebugStartStopEvent(const LPDEBUG_EVENT);
DMODL OnDebugEvent(const LPDEBUG_EVENT);
```

Available resources

macOS

Windows

The screenshot shows the Microsoft Windows Dev Center documentation page for the `ReadProcessMemory` function. The URL is <https://docs.microsoft.com/windows/desktop/api/memoryapi/nf-memoryapi-readprocessmemory>. The page includes the function's syntax in C++, parameters (`hProcess`, `lpBaseAddress`, `lpBuffer`, `nSize`, `nNumberOfBytesRead`), and a detailed description of each parameter. It also includes sections for `Return value` and `Requirements`.

```
BOOL ReadProcessMemory(
    _In_      HANDLE hProcess,
    _In_      _In_opt_ LPVOID lpBaseAddress,
    _Out_     _Out_   LPVOID lpBuffer,
    _In_      _In_     SIZE_T nSize,
    _Out_     _Inout_ SIZE_T *nNumberOfBytesRead
);
```

Syntax

Parameters

- `hProcess`**
A handle to the process with memory that is being read. The handle must have `PROCESS_VM_READ` access to the process.
- `lpBaseAddress`**
A pointer to the base address in the specified process from which to read. Before any data transfer occurs, the system verifies that all data in the base address and memory of the specified size is accessible for read access; and if it is not accessible the function fails.
- `lpBuffer`**
A pointer to a buffer that receives the contents from the address space of the specified process.
- `nSize`**
The number of bytes to be read from the specified process.
- `nNumberOfBytesRead`**
A pointer to a variable that receives the number of bytes transferred into the specified buffer. If `nNumberOfBytesRead` is `NULL`, the parameter is ignored.

Return value

Available resources

macOS

A screenshot of a Stack Overflow question titled "How to develop a debugger". The question asks for a way to attach to a process, set breakpoints, view memory, and other things that gdb can do. It notes that this is similar to another question but for macOS instead of Windows. A note specifies: "Note: I want to make a debugger, not use one." Another note says: "Another thing is that I don't want this debugger to be super complicated, all I need is just reading/writing memory, breakpoint handling, and viewing the GPU." The question has two answers. The first answer by "TheDarkKnight" (24.7k reputation) provides a link to "HighPerformanceMark" and suggests using LLDB. The second answer by "Camden Weaver" (24.7k reputation) links to "Create your own what? Your own debugger?" by Mike Harris. The question has 143 views and 7 upvotes.

How to develop a debugger

Asked 2 years ago Active 2 years ago Views 14 times

I am looking for a way to do things such as attach to a process, set breakpoints, view memory, and other things that gdb can do. I cannot, however, find a way to do these things.

This question is similar to [this one](#), but for macOS instead of Windows. Any help is appreciated!

Note: I want to make a debugger, not use one.

Another thing is that I don't want this debugger to be super complicated, all I need is just reading/writing memory, breakpoint handling, and viewing the GPU

share Improve this question follow

edited Aug 22 '18 at 18:06 by TheDarkKnight 24.7k • 4 43 • 74

asked Aug 22 '18 at 14:20 by Camden Weaver 24.7k • 4 43 • 74

HighPerformanceMark I want to create my own using C++, not use xcode - [Camden Weaver](#) Aug 22 '18 at 14:28

@CamdenWeaver Create your own what? Your own debugger? - Mike Harris Aug 22 '18 at 14:29

@HighPerformanceMark thanks, I updated my question - [Camden Weaver](#) Aug 22 '18 at 14:34

Have a look here for a simple debugger idea. [uninformed.org/index.cgi?4+k&38=c14](#) - Matthew Fisher Aug 29 '18 at 1:41

add a comment

6 Answers

LLDB has an API that can be consumed from [C++](#) or [Python](#). Maybe this is what you're looking for.

Unfortunately the documentation is fairly threadbare, and there don't seem to be usage examples. This will therefore entail some reading of the source and a lot of trial and error.

share Improve this answer follow

answered Aug 22 '18 at 14:35 by Konrad Rudolph 29k • 117 • 480 • 1158

Windows

A screenshot of the Microsoft Docs page for the "ReadProcessMemory" function. The page includes the function signature, parameters, and a detailed description. The parameters are: hProcess (A handle to the process with memory that is being read. The handle must have PROCESS_VM_READ access to the process), lpBaseAddress (A pointer to the base address in the specified process from which to read. Before any data transfer occurs, the system verifies that all data in the base address and memory of the specified size is accessible for read access; and if it is not accessible, the function fails), lpBuffer (A pointer to a buffer that receives the contents from the address space of the specified process), nSize (The number of bytes to be read from the specified process), and lpNumberOfBytesRead (A pointer to a variable that receives the number of bytes transferred into the specified buffer. If lpNumberOfBytesRead is `NULL`, the parameter is ignored).

ReadProcessMemory function

03/10/2020 · 2 minutes to read

In this article

Syntax

```
BOOL ReadProcessMemory(
  _In_     HANDLE hProcess,
  _In_     _In_opt_ LPVOID lpBaseAddress,
  _Out_    _Out_   _In_opt_ LPVOID lpBuffer,
  _In_     _In_     SIZE_T  nSize,
  _Out_    _Out_   _In_opt_ LPVOID *lpNumberOfBytesRead
);
```

Parameters

- hProcess**
A handle to the process with memory that is being read. The handle must have `PROCESS_VM_READ` access to the process.
- lpBaseAddress**
A pointer to the base address in the specified process from which to read. Before any data transfer occurs, the system verifies that all data in the base address and memory of the specified size is accessible for read access; and if it is not accessible, the function fails.
- lpBuffer**
A pointer to a buffer that receives the contents from the address space of the specified process.
- nSize**
The number of bytes to be read from the specified process.
- lpNumberOfBytesRead**
A pointer to a variable that receives the number of bytes transferred into the specified buffer. If `lpNumberOfBytesRead` is `NULL`, the parameter is ignored.

Return value

Available resources

macOS

Mach IPC Interface

Mach IPC presents itself in a few forms: message queues, lock-sets, and semaphores (more may be added in the future). All share one common specific representation in these Mach port capability handles allow the underlying IPC object to be used and manipulated in consistent ways.

Message Queue Interface

mach_msg - Send and/or receive a message from the target port.
mach_msg_overwrite - Send and/or receive messages with possible overwrite.
Mach Message Queue Data Structure

Jack Jack - Set 2

- lock acquire** - Acquire ownership a lock.
- lock handoff** - Hand-off ownership of a lock.
- lock handoff accept** - Accept lock ownership from a handoff.
- lock make stable** - Stabilize the state of the specified lock.
- lock release** - Release ownership of a lock.
- lock set create** - Create a new lock set.
- lock set destroy** - Destroy a lock set and its associated locks.
- try** - Attempt to acquire access rights to a lock.

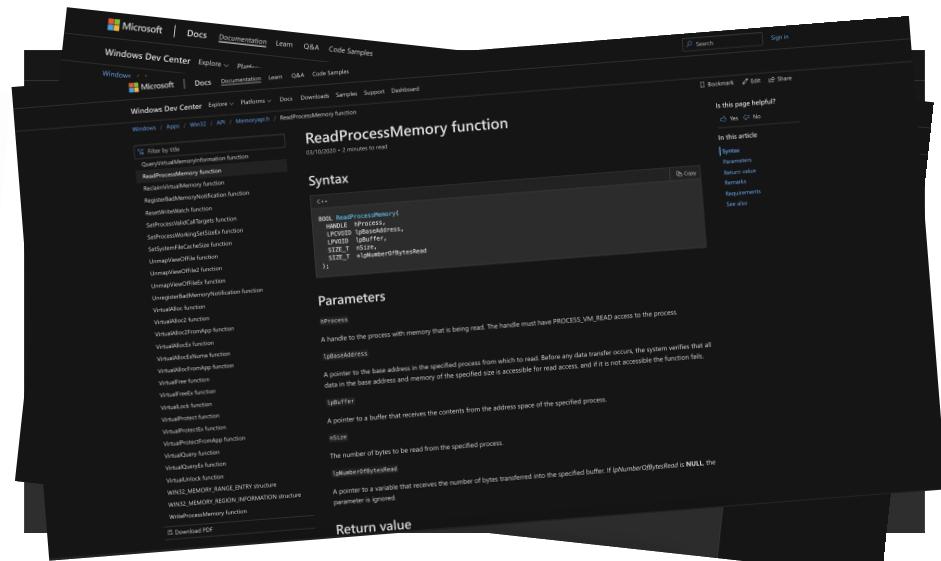
Mach Semaphore Interf.

semaphore - Interface
 create - Create a new semaphore.
 destroy - Destroy a semaphore.
 signal - Increments the semaphore count.
 signal_all - Wake up all threads blocked on a semaphore.
 wait - Wait on the specified semaphore.

Jack Port Management Inc.

much_port_allocate - Create caller-specified type of port right.
much_port_allocate_all - Create a port right with full MacP port semantics.
much_port_allocate_glo - Allocate a port right with the caller-specified name.
much_port_allocate_glo - Allocate a port right with the caller-specified name.
much_port_create_subsystem - Create a port right with the caller-specified subsystem.
much_port_destroy - Delete the target port right's user references.
much_port_destroy - Delete all the port rights associated with specified name.
much_port_get_name - Return the name of the target port and return it to the caller.
much_port_get_right - Remove the specific information about target port right from the caller.
much_port_get_set - Returns the target port right's user references on the target port right.
much_port_inet_set - Return the port right's user references on the target port right set.
much_port_inet_set - Modify the specific port right on the target port right set.
much_port_inet_set - Modify the specific port right on the target port right set.
much_port_inet_set - Move the specific port right's count of user references.
much_port_inet_update - Return information about a specific port right.

Windows



Available resources

macOS

The screenshot shows the Apple Developer Documentation for the `mach_vm_read` function. At the top, there's a navigation bar with links like Documentation, Virtual Memory, and mach_vm_read. The main content area has a title "Function" and "mach_vm_read". Below it, a section says "No overview available." Then there's a "Declaration" block containing C code:

```
 kern_return_t mach_vm_read(vm_msp_read_t target_task, mach_vm_address
    _t address, mach_vn_size_t size, vm_offset_t *data, mach_msg_type_number
    _t *dataCnt);
```

Under "See Also", there's a "Mach VM" section with links to `mach_vm_allocate`, `mach_vm_deallocate`, `mach_vm_map`, `mach_vm_write`, and `mach_vm_protect`.

Windows

The screenshot shows the Microsoft Windows Dev Center documentation for the `ReadProcessMemory` function. At the top, there's a navigation bar with links like Documentation, Learn, Q&A, and Code Samples. The main content area has a title "ReadProcessMemory function". Below it, there's a "Syntax" block with C++ code:

```
 BOOL ReadProcessMemory(
    _In_opt_ HANDLE hProcess,
    _In_opt_ _In_ LPOVOID lpBaseAddress,
    _In_opt_ _In_ LPOVOID lpBuffer,
    _In_ SIZE_T nSize,
    _Out_opt_ _Inout_ _In_ LPOWORD lpNumberOfBytesRead
);
```

Under "Parameters", there are several parameters with descriptions:

- `hProcess`: A handle to the process that contains the memory to be read.
- `lpBaseAddress`: A pointer to the base address in the specified process from which to read. Before any data transfer occurs, the system verifies that all data in the base address and memory of the specified size is accessible for read access, and if it is not accessible the function fails.
- `lpBuffer`: A pointer to a buffer that receives the contents from the address space of the specified process.
- `nSize`: The number of bytes to be read from the specified process.
- `lpNumberOfBytesRead`: A pointer to a variable that receives the number of bytes transferred into the specified buffer. If `nNumberOfBytesRead` is `NULL`, the parameter is ignored.

At the bottom, there's a "Return value" section and a "Download PDF" link.

Available resources

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

23.captures
23.Jun.2008 – 22.Dic.2018

Go MAP JUN MAY
◀ 27 ▶
2008 2009 2010 2011 2012

INTERNET ARCHIVE
PageArchiver

matasano
SECURITY

HOME CAREERS CONTACT BLOG PRODUCT SERVICES TEAM

What I've Been Doing On My Summer Vacation or, "It has to work;
Otherwise gdb wouldn't"

Author | July 17th, 2008 | Filed Under: Apple, Uncategorized

An intern expects to be given simple projects, like coffee retrieval, or "Hello, World." So I've been sorely disappointed by Matasano. I have been offered coffee retrieval services by senior engineers and my latest project has been anything but "Hello, World."

In fact, it's been more like, "Hello, OS X. Tell me your secrets!"

This is the story of one trial-by-fire project handed to an intern that turned out to be more complicated than anyone expected.

It started with Thomas, innocently enough, handing me some debugger code. It was both C and Ruby, and for Solaris and Win32. He said, "I would like you to port this Win32 Ruby code to OS X."

"Um, okay."

At that point I'd just finished learning the basics of Ruby via my previous Matasano project, a database backed HTTP proxy. I knew nothing about debuggers, let alone the low level C library calls I'd need and Ruby bindings to make them work. I know, fun, right?

I started simply and dumped the C off in my head so I could begin to read and understand the code Thomas dumped on me, and perhaps learn how a debugger works and gets used. It took a day or two just to read it. I'd ask the office some fairly basic question about debuggers, and receive a much longer response than I'd anticipated. Like a tutorial on the workings of x86 assembly. Eventually, I got to a point where I was almost comfortable with how the C debugger worked.

When staring at C code stopped doing me any good, and writing Ruby code started seeming feasible, I moved on to porting the Ruby code. "How hard could it be?"

2.

Thomas gave me a starting point. Our Ruby code called directly into libraries using Win32API and Ruby/DL. We have wrapper libraries that make those C calls look like Ruby library functions. So, for instance, in our Wrap32 library, we have:

```
# just grab some local memory
def malloc(sz)
  r = CFunc::newscript(malloc,[&L"],call(sz)
  raise Win32X.new(:malloc) if r == 0
  return r
end
```

We had a small piece of this written for OS X as well. I had to build it out. I started with `getpid()`, a simple system call I could make sure worked before I moved on to something harder. It worked right away. My confidence was high. I was feeling cocky.

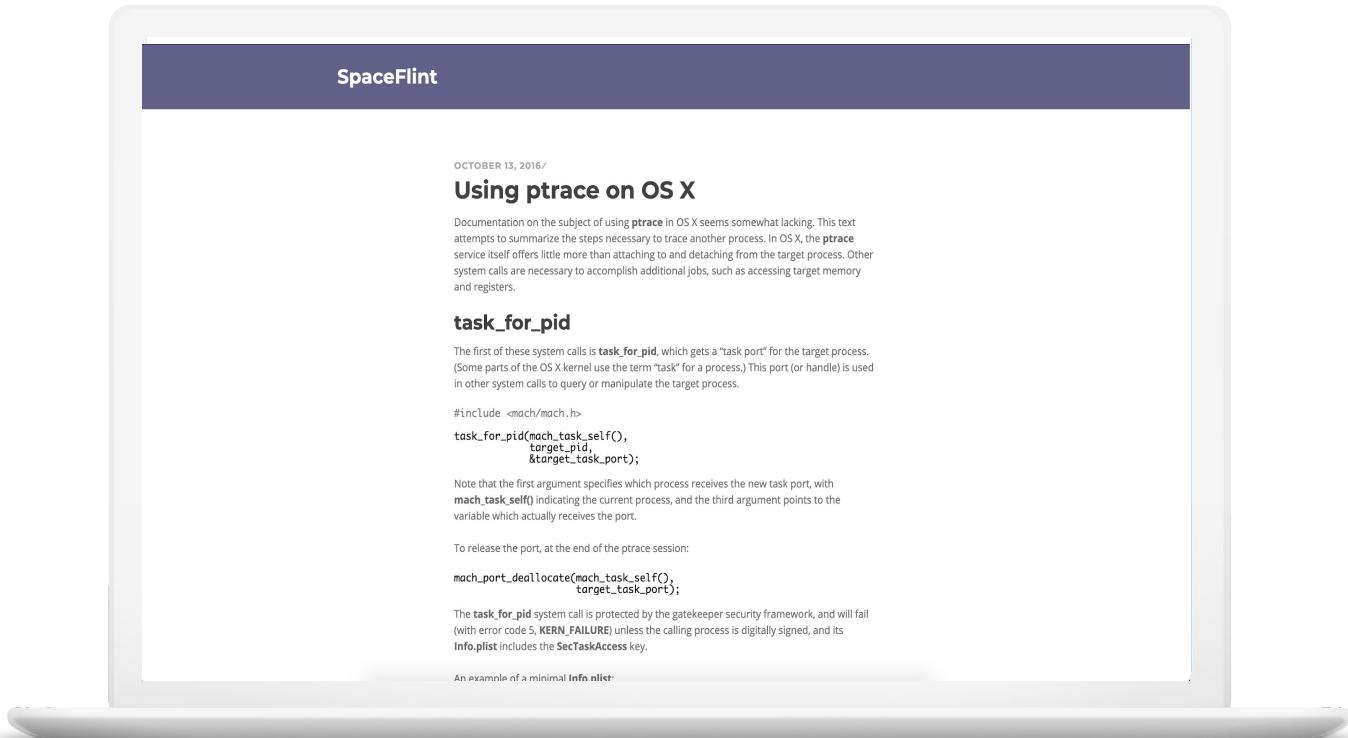
People We Read

- Adam Shostack & Friends
- Aki Kamozawa & H. Alexander Talbot
- Amrit Williams
- Andrew Donofrio
- Bunnie Huang
- Cambridge Security Lab Blog
- David Litchfield
- Dowd, McDonald, and Schuh
- Eric Rescorla
- Halvar Flake
- Iftek Gulifanov
- Jeremiah Grossman
- Ken "Skywing" Johnson
- Metasploit Team
- Mike Rothman
- Nate Lawson
- nCircle Team
- Peter Lindstrom
- Richard Bejtlich
- The Veracode Blog

Things We Write

- Apple (51)

Available resources



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 execution of another process
- Examine and change the memory and registers of another process
- Used to implement breakpoints, debugging and system call tracing



ptrace on macOS

- Observe and control the execution of another process
- Examine and change the memory and registers of another process
- Used to implement breakpoints, debugging and system call tracing
- Very limited on macOS

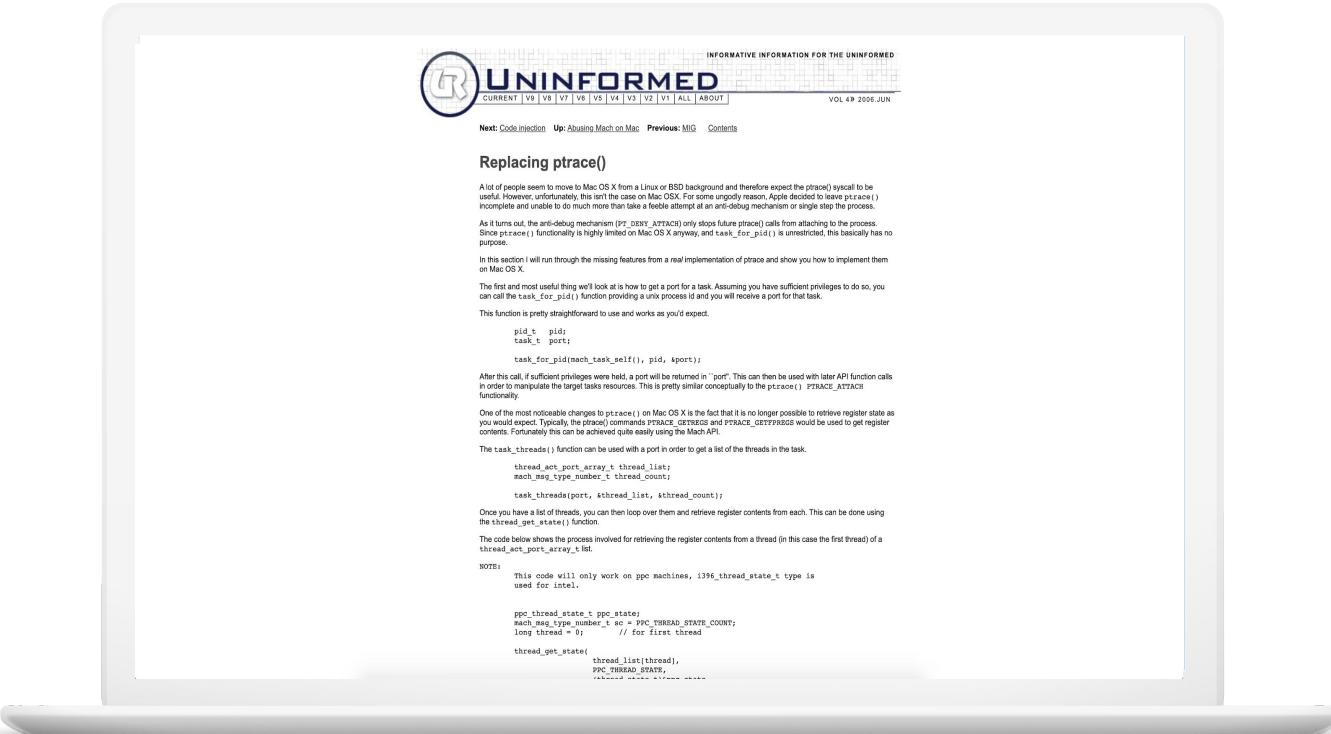


ptrace on macOS

- Observe and control the execution of another process
 - Examine and change the memory and registers of another process
 - Used to implement breakpoints, debugging 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?



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



mach_msg

MACH_RCV_MSG

MACH_RCV_TIMEOUT

MACH_RCV_INTERRUPTED

Parse the debugging event



```
/**  
 * 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



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

Debugger's Main Loop

Wait for debugging

event



mach_msg

MACH_RCV_MSG

MACH_RCV_TIMEOUT

MACH_RCV_INTERRUPTED

Parse the debugging

event



Handle the debugging

event



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

```
/**  
 * 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) {  
|
```

EXC_BAD_ACCESS

Debugger's Main Loop

Wait for debugging event



mach_msg
MACH_RCV_MSG
MACH_RCV_TIMEOUT
MACH_RCV_INTERRUPTED

Parse the debugging event



```
/**  
 * 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



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

EXC_BAD_INSTRUCTION

Debugger's Main Loop

Wait for debugging event



mach_msg
MACH_RCV_MSG
MACH_RCV_TIMEOUT
MACH_RCV_INTERRUPTED

Parse the debugging event



event



Handle the debugging event



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

```
/**  
 * 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) {  
|
```

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

Mach-O File

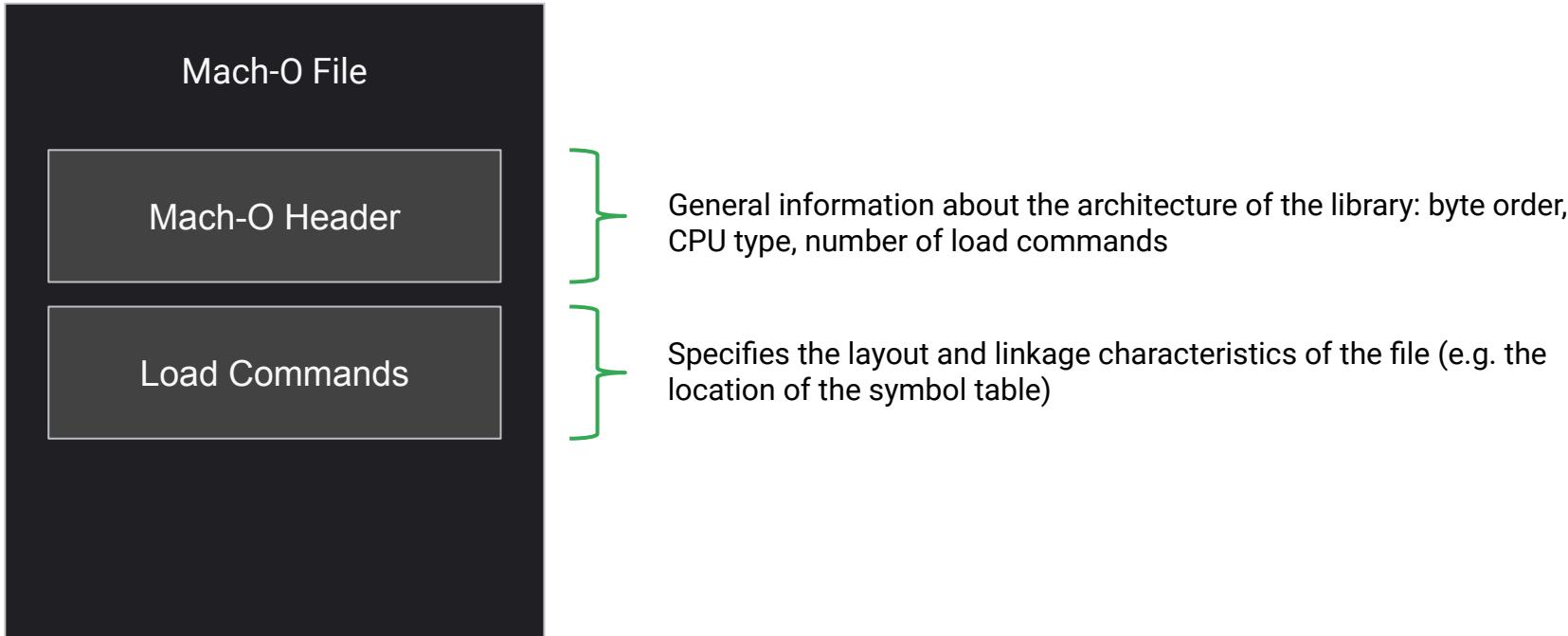


Examine loaded libraries in the target

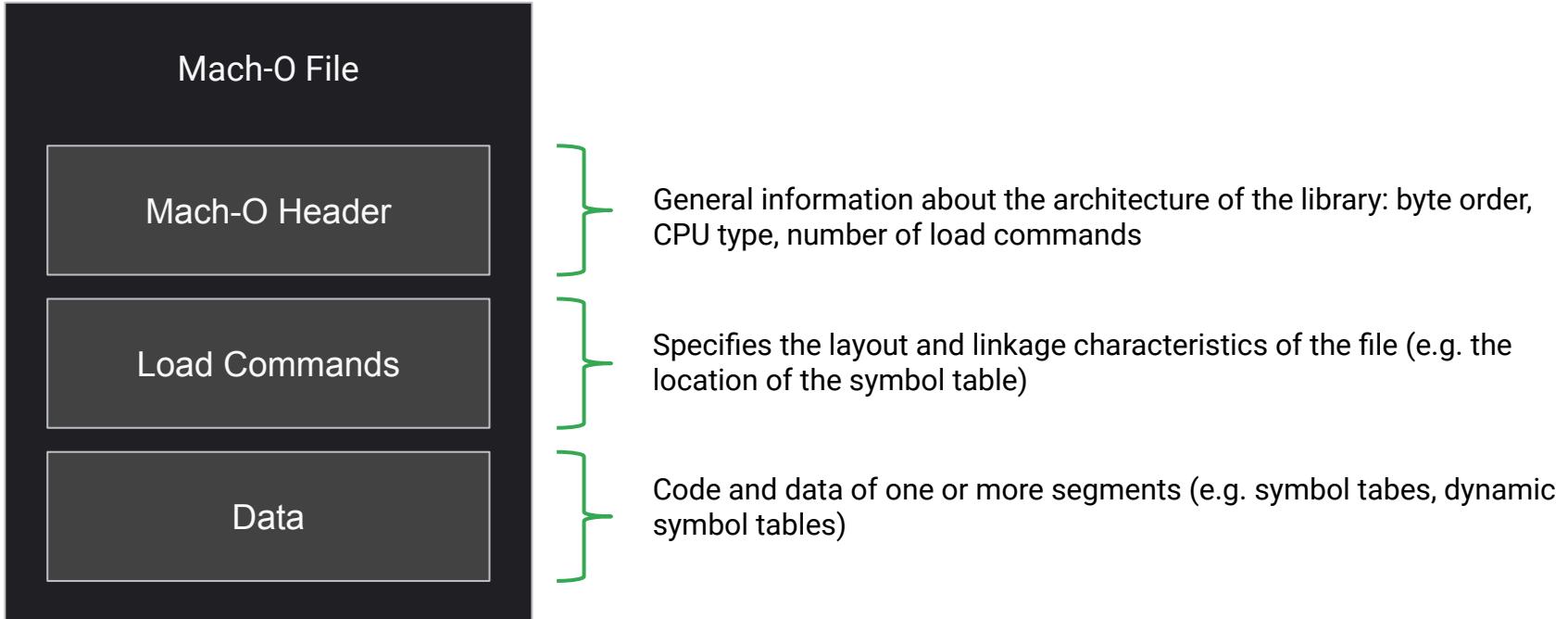


General information about the architecture of the library: byte order, CPU type, number of load commands

Examine loaded libraries in the target



Examine loaded libraries in the target



Detect target's entrypoint

Detect target's entrypoint

Mach-O File for
the Main binary

Mach-O Header

Load Commands

Data

```
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

Mach-O File for
the Main binary

Mach-O Header

Load Commands

Data

```
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

Mach-O File for
the Main binary

Mach-O Header

Load Commands

Data

```
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

Mach-O File for
the Main binary

Mach-O Header

Load Commands

Data

```
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

Mach-O File for
the Main binary

Mach-O Header

Load Commands

Data

```
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 above and that are contributed under the terms of the 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 refer to 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>

#ifndef __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, llbd 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 {
    uint64_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 above and that are merged hereunder, and any additional source code
 * in such form and/or in such manner as to allow the contents to be incorporated
 * directly into a product by you without prior permission or notice from
 * Apple, and subject to the appropriate license terms governing the
 * main product.
 *
 * 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>

#ifndef __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, llbd 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 {
    uint64_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 above and that are contributed under the terms of the 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 refer to 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 };

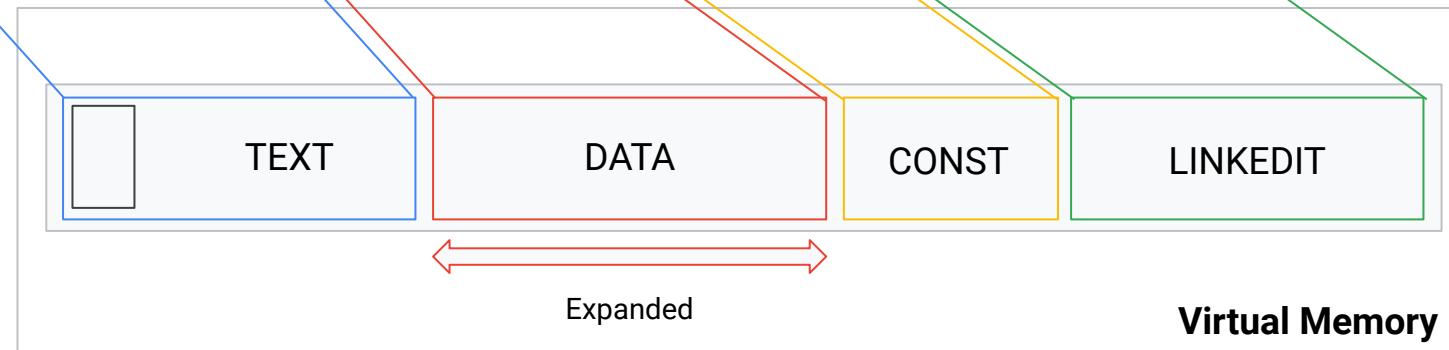
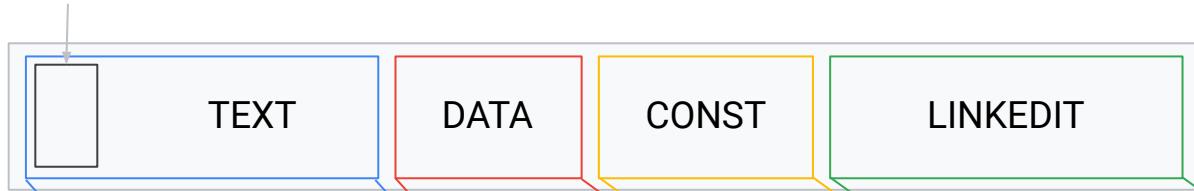
void _dyld_debugger_notification(enum dyld_notify_mode, unsigned long count, uint64_t machHeaders[]);

struct dyld_process_cache_info {
    uint64_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 Commands

File



Mach-O Header
Load Commands

File



dyld_debugger_notification



Expanded

Virtual Memory

Mach-O Header
Load Commands

File



↔ symoff

TEXT

DATA

CONST

LINKEDIT

Virtual Memory

↔

Expanded

Mach-O Header
Load Commands

File



↔ symoff

Module_address - text_cmd->vmaddr

Virtual Memory



Expanded

```
/*
 * 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

Mach-O Header
Load Commands

File



↔ symoff

TEXT

DATA

CONST

LINKEDIT

↔

Expanded

Virtual Memory

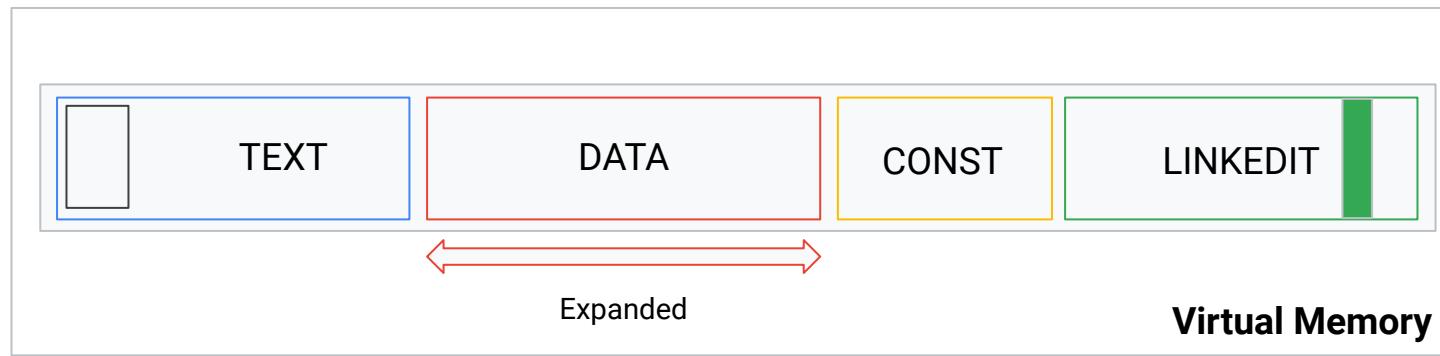
Module_address - text_cmd->vmaddr

Mach-O Header
Load Commands

File



↔ symoff



Expanded

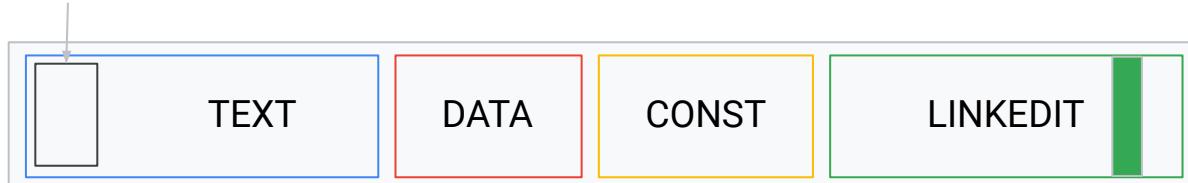
Virtual Memory

Module_address - text_cmd->vmaddr

linkedit_cmd -> vmaddr

Mach-O Header
Load Commands

File



↔ symoff

TEXT

DATA

CONST

LINKEDIT

↔

Expanded

Virtual Memory

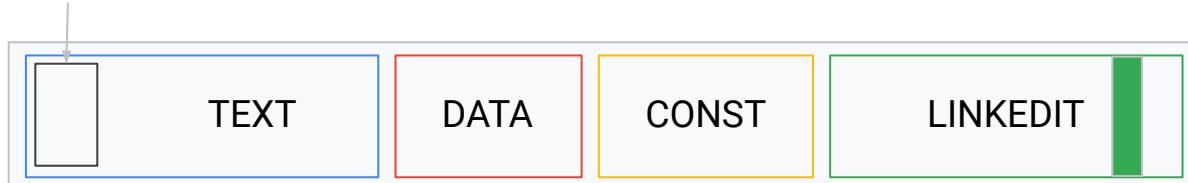
Module_address - text_cmd->vmaddr

linkedit_cmd -> vmaddr

symoff - fileoff

Mach-O Header
Load Commands

File



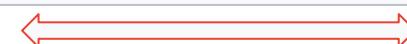
↔ symoff

TEXT

DATA

CONST

LINKEDIT



Expanded

Virtual Memory

Module_address - text_cmd->vmaddr

linkedit_cmd -> vmaddr

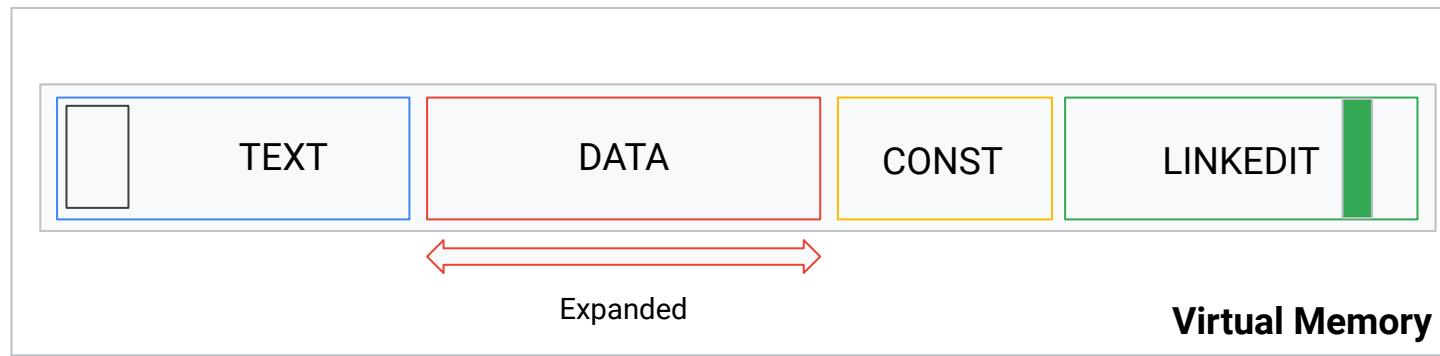
symoff - fileoff

Mach-O Header
Load Commands

File



↔ symoff
↔ fileoff



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:0000000000001201F ; ===== S U B R O U T I N E =====
text:0000000000001201F ; Attributes: bp-based frame
text:0000000000001201F
text:0000000000001201F ; Attributes: bp-based frame
text:0000000000001201F
text:0000000000001201F
text:0000000000001201F
text:0000000000001201F
text:0000000000001201F
text:0000000000001201F
text:0000000000001201F
text:0000000000001201F
text:00000000000012020
text:00000000000012023
text:00000000000012024
text:00000000000012024
text:00000000000012024
public __dyld_debugger_notification
__dyld_debugger_notification proc near ; CODE XREF: gdb_image_notifier(dy
push    rbp
mov     rbp, rsp
pop     rbp
ret
__dyld_debugger_notification endp
```

Get notified when libraries are loaded or unloaded

```
text:000000000001201F ; ===== S U B R O U T I N E =====
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
```

Get notified when libraries are loaded or unloaded

Get notified when libraries are loaded or unloaded

```
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
```

Get notified when libraries are loaded or unloaded

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?

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
```

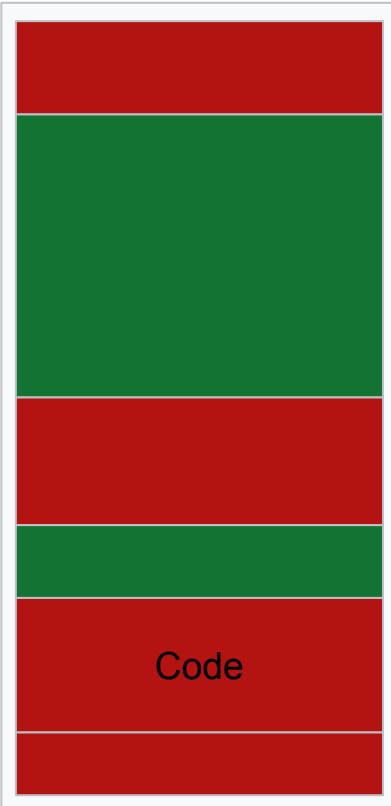
Read Memory →

Target process

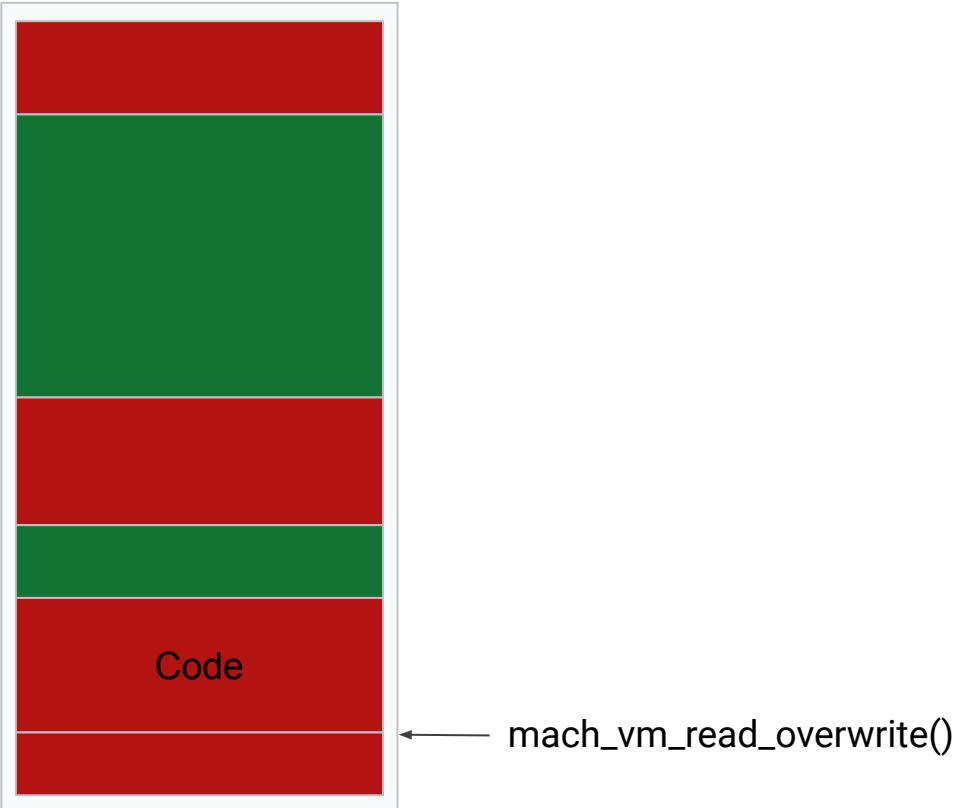
module.dylib:

```
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
```

Read target's memory



Read target's memory



Demo

How does TinyInst work?

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
```

Read Memory →

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?

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.dylib:

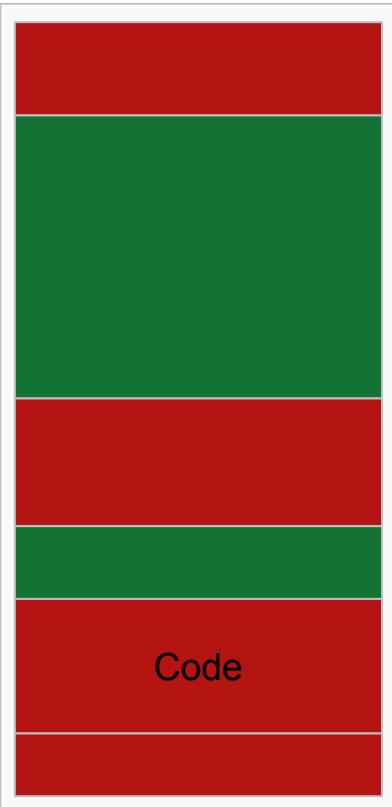
```
00000001800889F4 mov    [rsp+8], rbx  
00000001800889F9 push   rdi  
00000001800889FA sub    rsp, 30h  
00000001800889FE mov    rdi, rcx  
0000000180088A01 mov    rdx, rcx  
0000000180088A04 xor    ecx, ecx  
0000000180088A06 mov    r8d, 104h
```

instrumented_code_remote

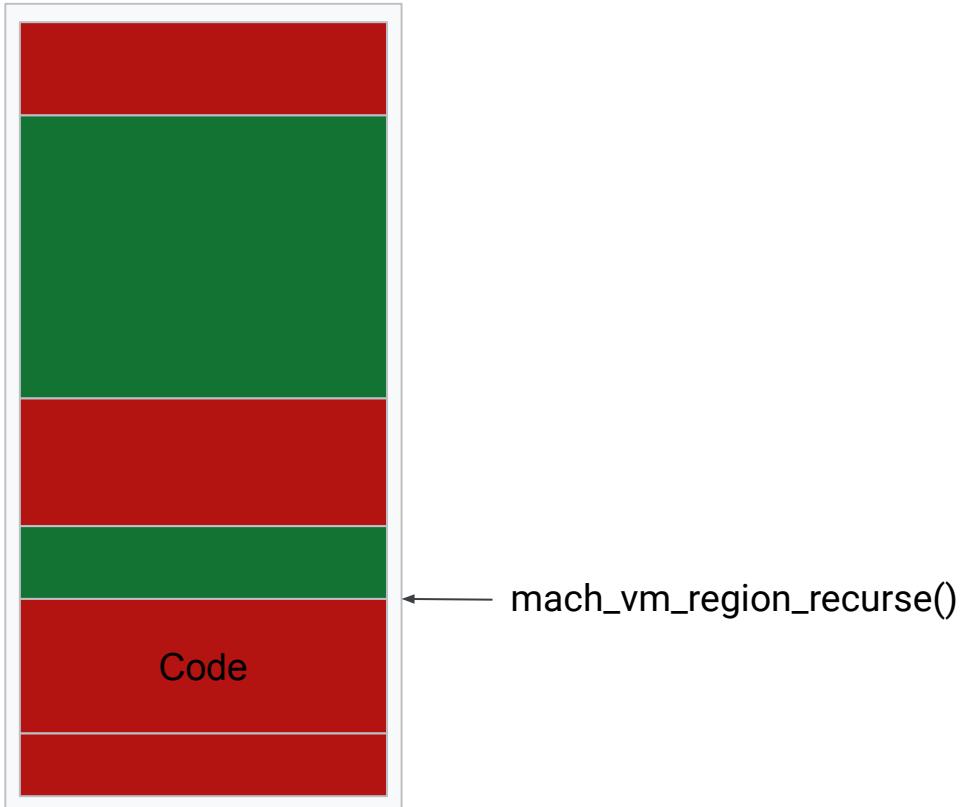


Within 2 GB

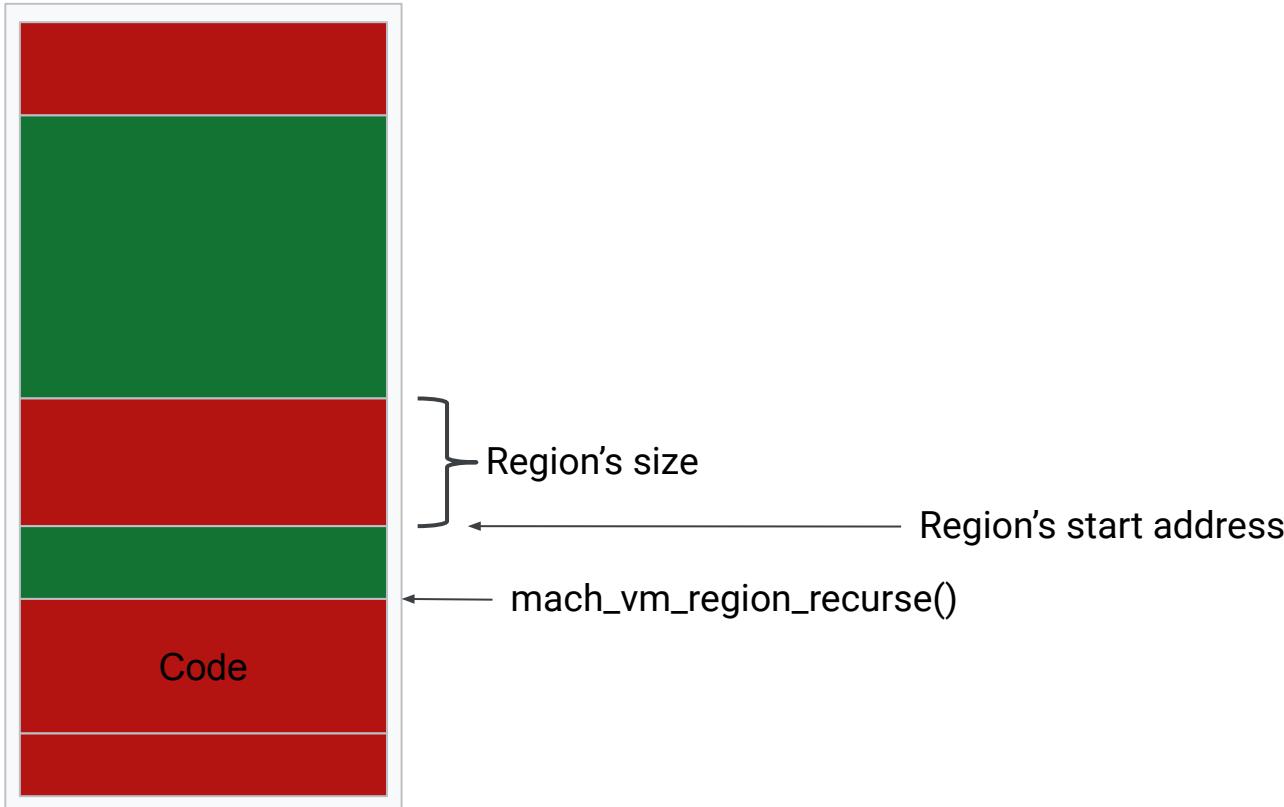
Allocate memory region in the target process



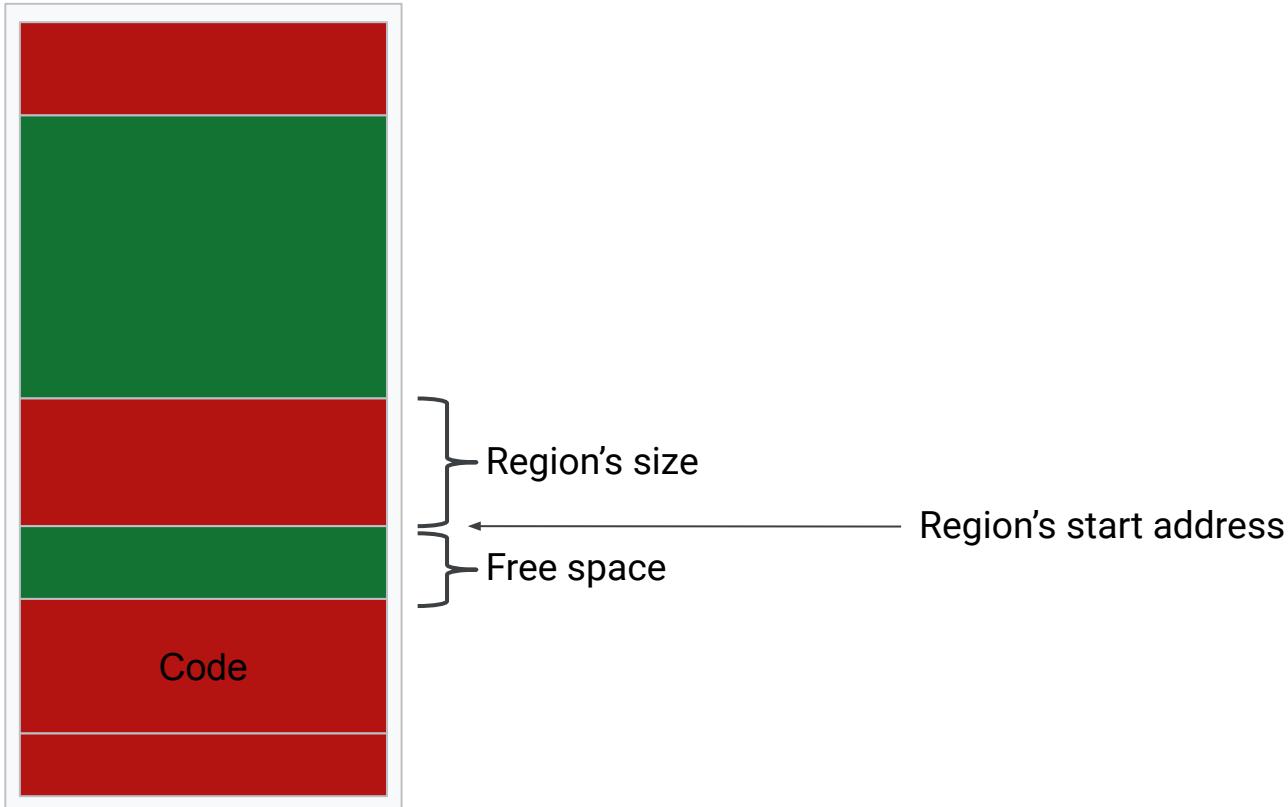
Allocate memory region in the target process



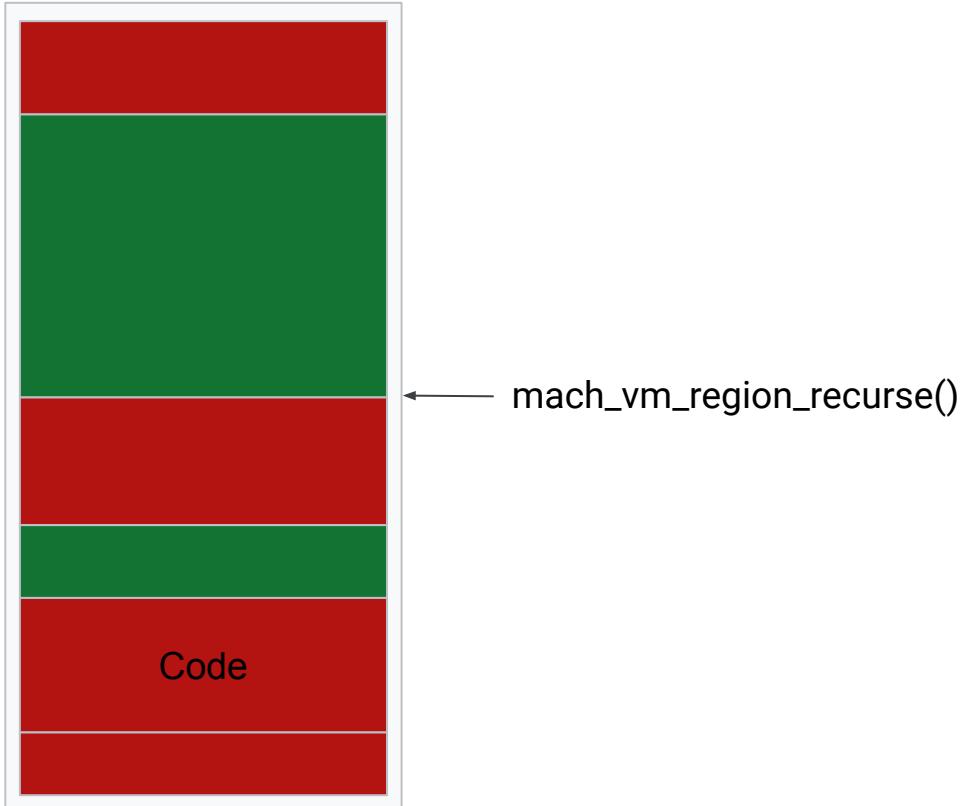
Allocate memory region in the target process



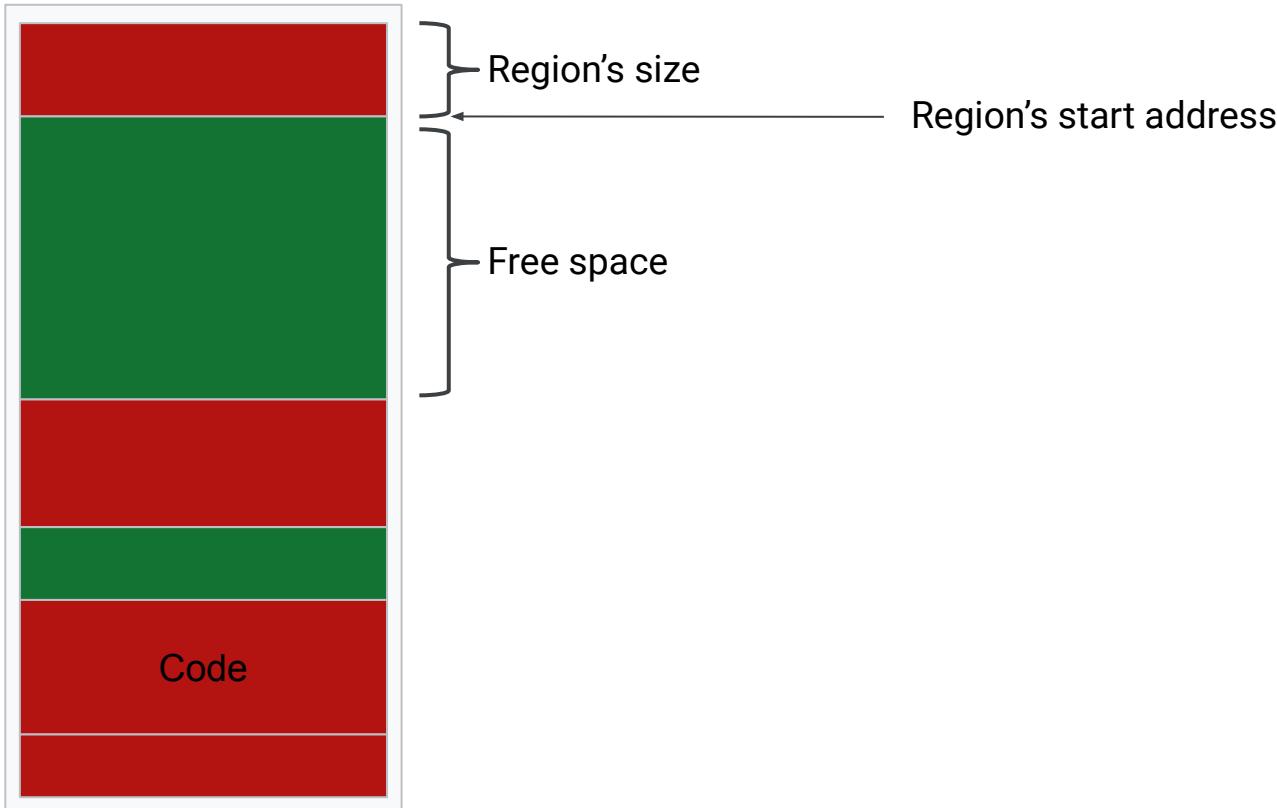
Allocate memory region in the target process



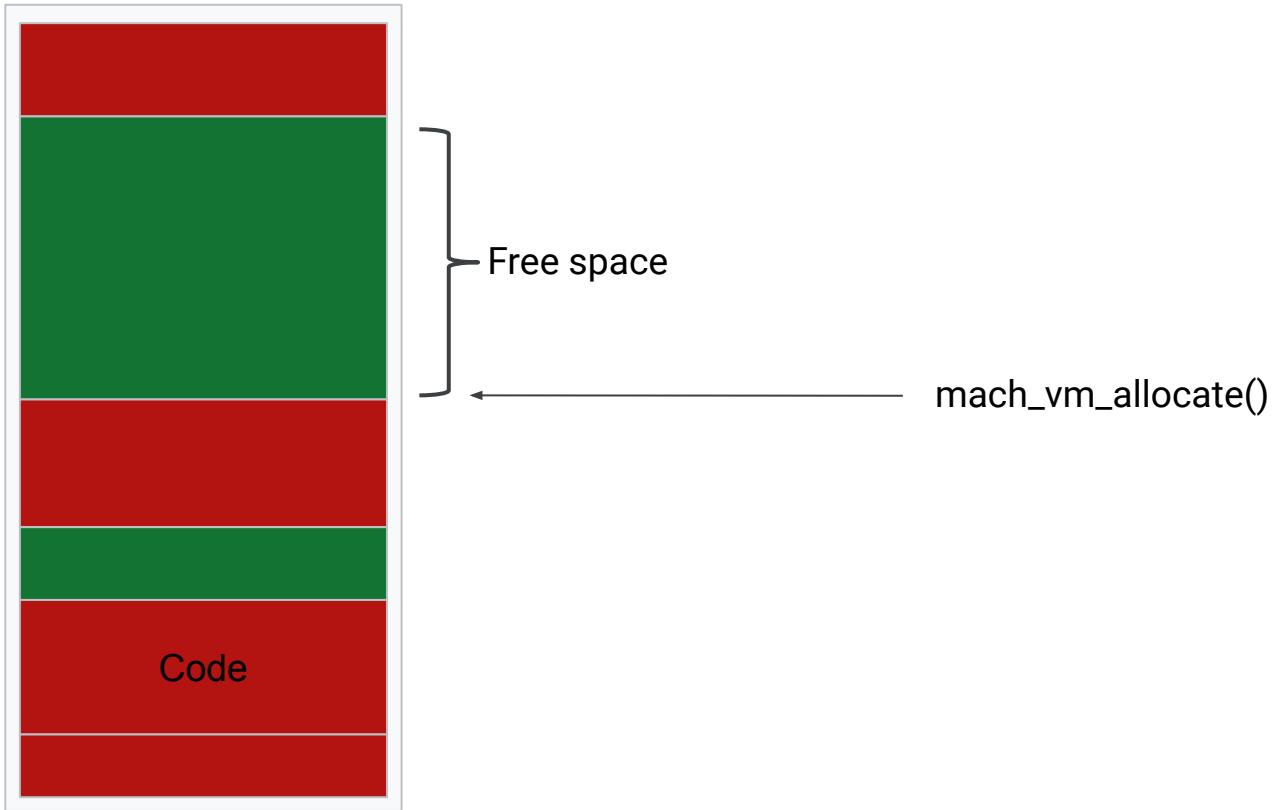
Allocate memory region in the target process



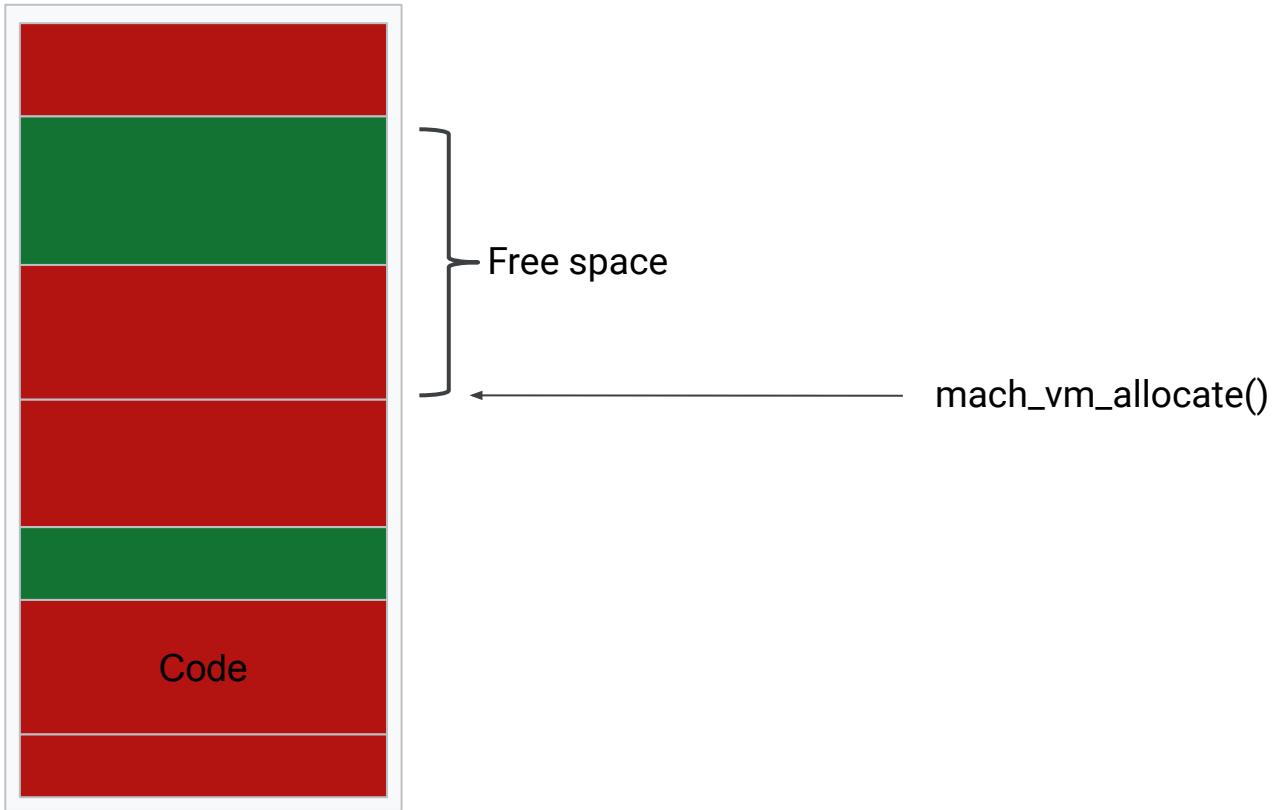
Allocate memory region in the target process



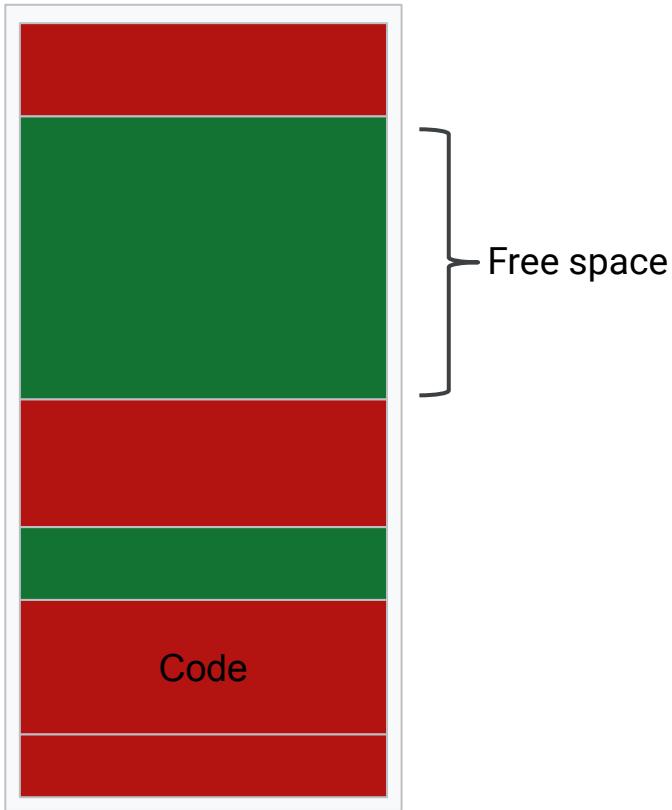
Allocate memory region in the target process



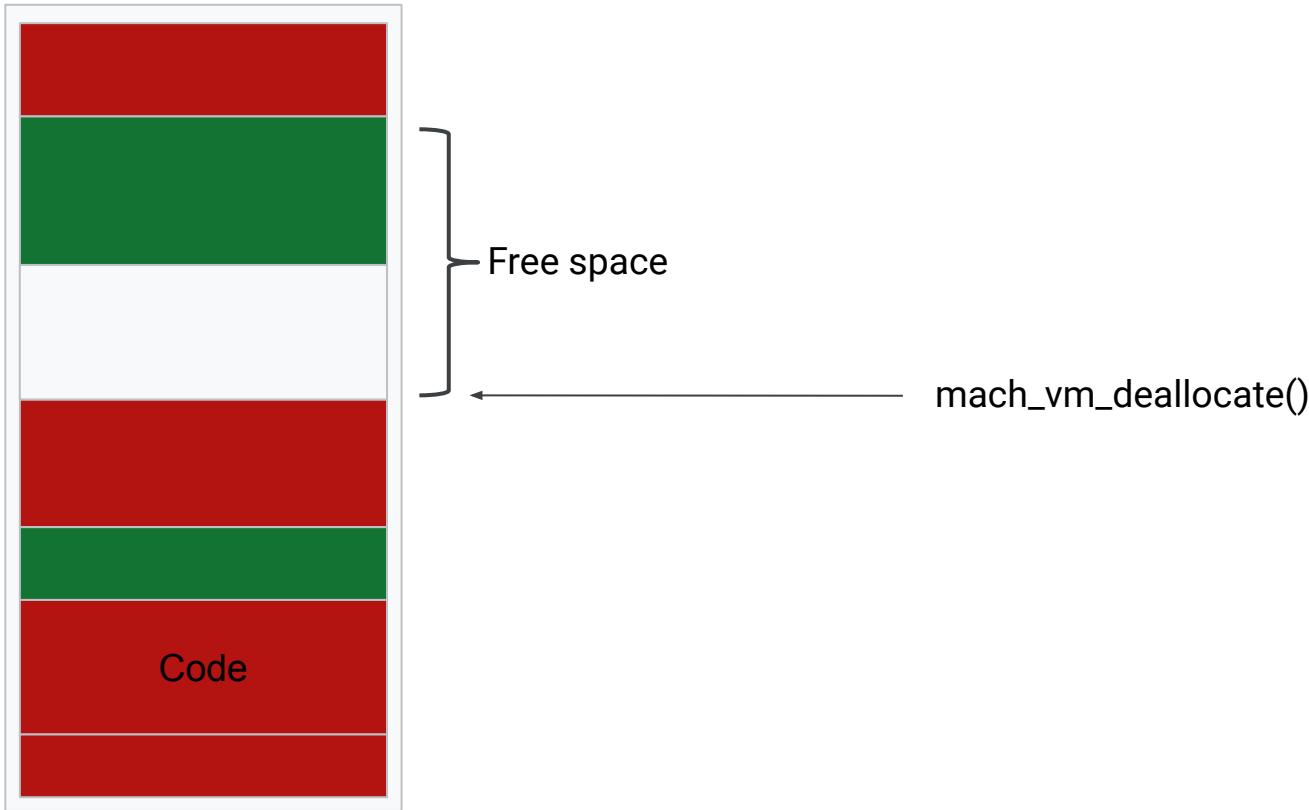
Allocate memory region in the target process



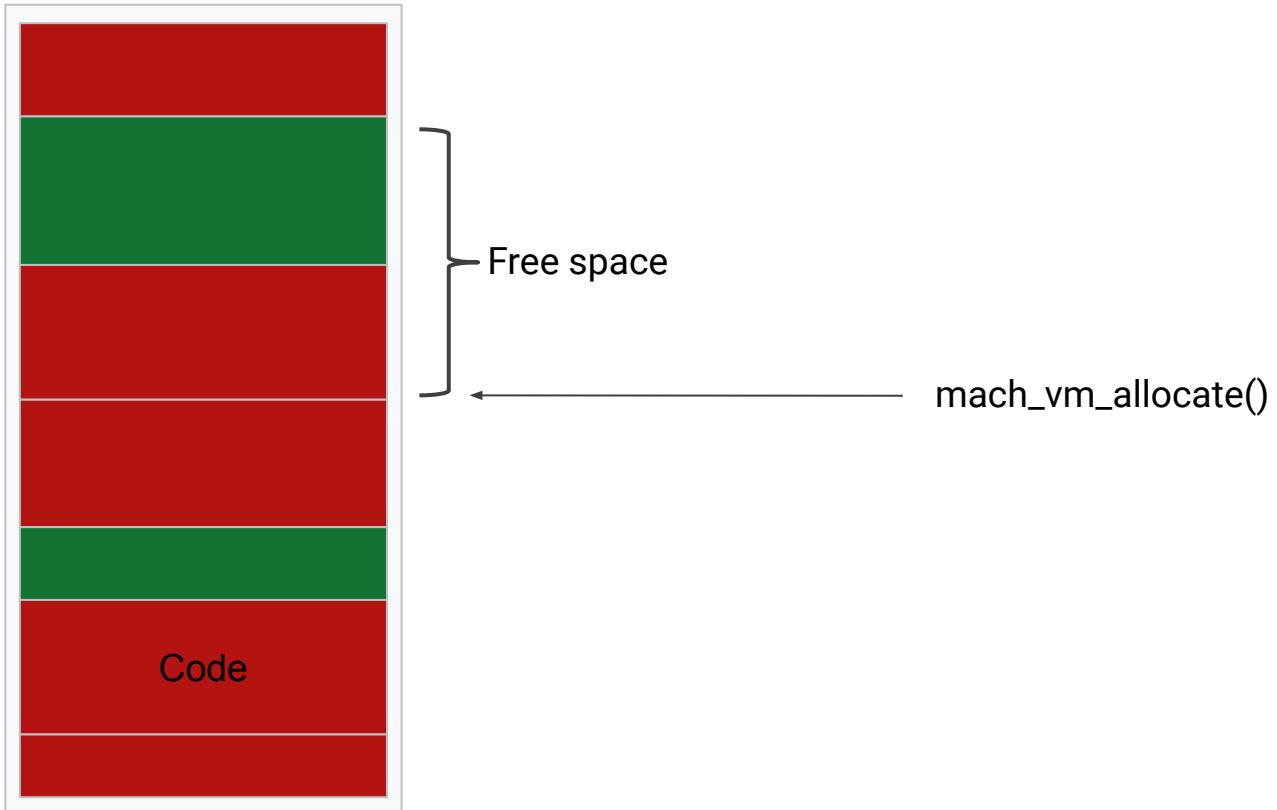
Allocate memory region in the target process



Allocate memory region in the target process

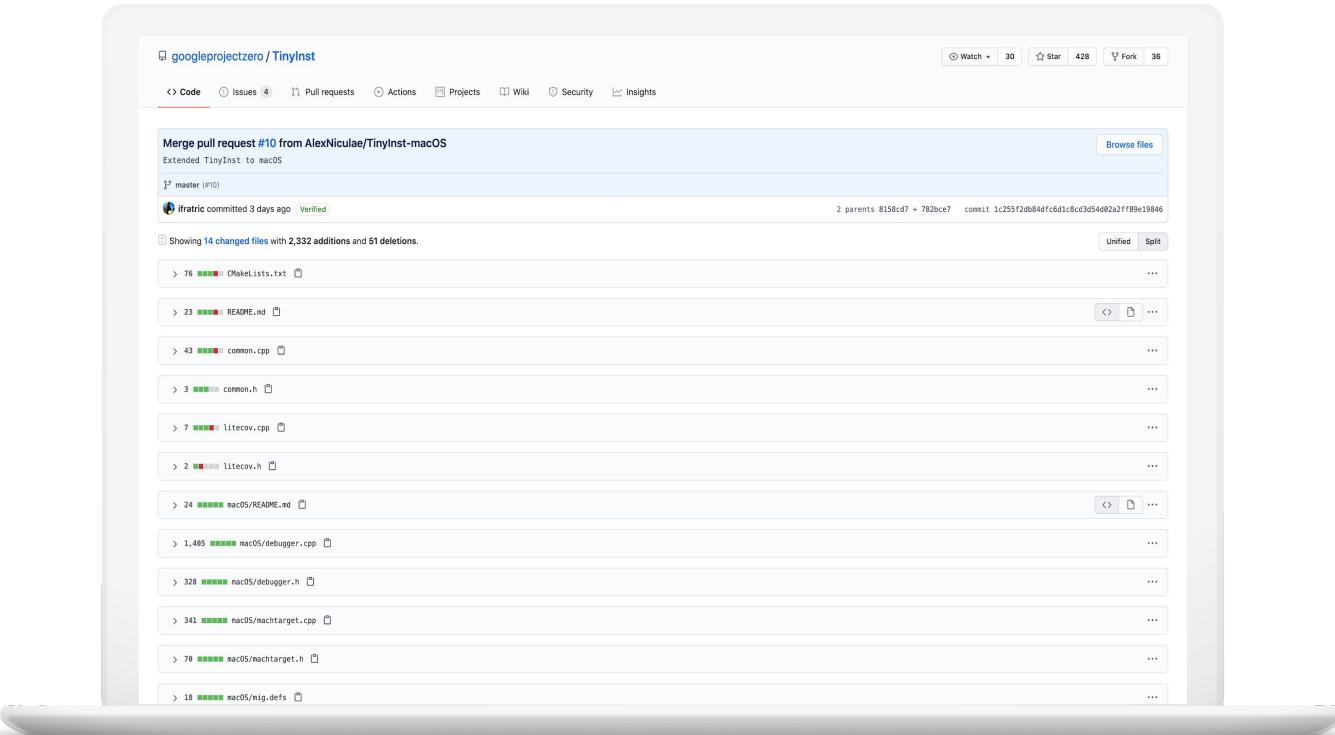


Allocate memory region in the target process



Results

TinyInst on macOS is public



TinyInst on macOS is public

Alexandru-Vlad Niculae @_aniculae · Sep 11

Today, TinyInst became available on macOS, as part of my Google Project Zero Internship project!

As it enables coverage-guided binary fuzzing, I am very excited about how this project will improve macOS security in the future.

Special thanks to [@ifsecure](#) and [@_bazad!](#)

```
aniculae-macbookpro:Release aniculae$ ./litecov -instrument_module ImageIO -target_module ImgTarget -target_method _fuzz -n
rgs % instrument -start-loop -- ./ImgTarget
Instrumented module ImageIO, code size: 1429564
Target function returned normally
Found 72 new offsets in ImageIO
Target function returned normally
aniculae-macbookpro:Release aniculae$
```

GIF

3 70 245

Show this thread

TinyInst on macOS is public

A screenshot of a Twitter post from user [@_aniculau](#) dated Sep 11. The post reads:

Very cool!

Alexandru-Vlad Niculae @_aniculau · Sep 11
Today, TinyInst became available on macOS, as part of my Google Project Zero Internship project!
As it enables coverage-guided binary fuzzing, I am very excited about how this project will improve macOS security in the future.
Special thanks to @ifsecure and @_bazad!

[Show this thread](#)

aniculau-macbookpro:Release aniculau\$./litecov -instrument_module ImageIO -target_module ImgTarget -target_method _fuzz -n/a

```
aniculau-macbookpro:Release aniculau$ ./litecov -instrument_module ImageIO -target_module ImgTarget -target_method _fuzz -n/a
Instrumented module ImageIO module size: 1429584
Target function returned normally
Found 72 new offsets in ImageIO
Target function returned normally
aniculau-macbookpro:Release aniculau$
```

GIF

Below the post are standard Twitter interaction icons: a speech bubble, a retweet icon, a heart with the number 1, and an upvote icon.

TinyInst on macOS is public



TinyInst on macOS is public

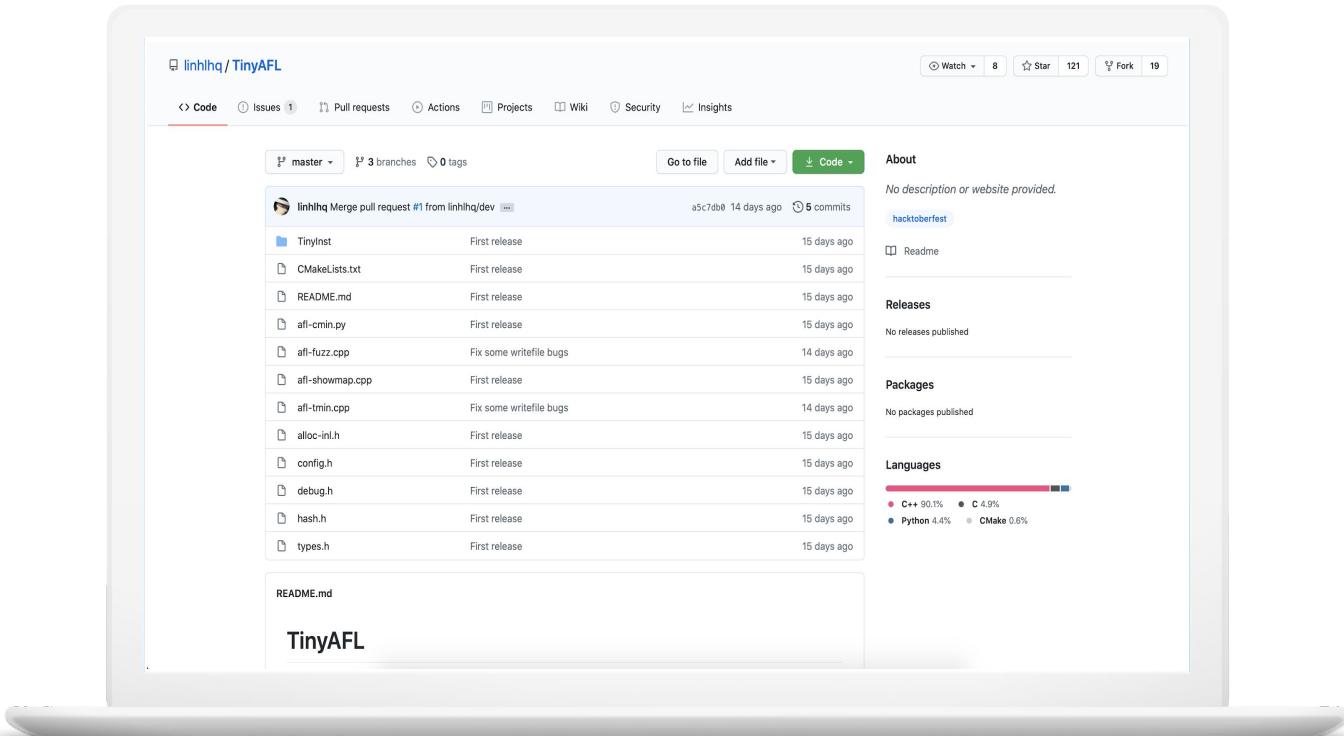


TinyInst on macOS is public

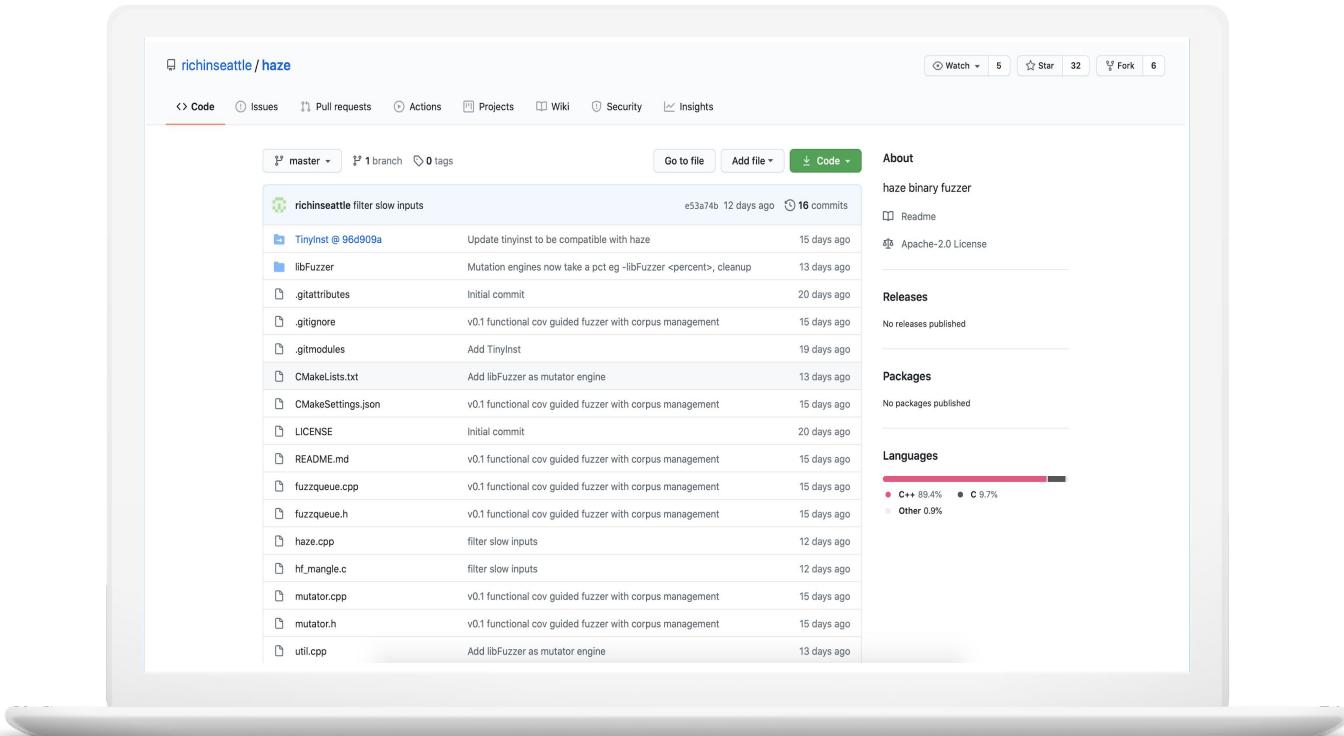


TinyInst Fuzzers

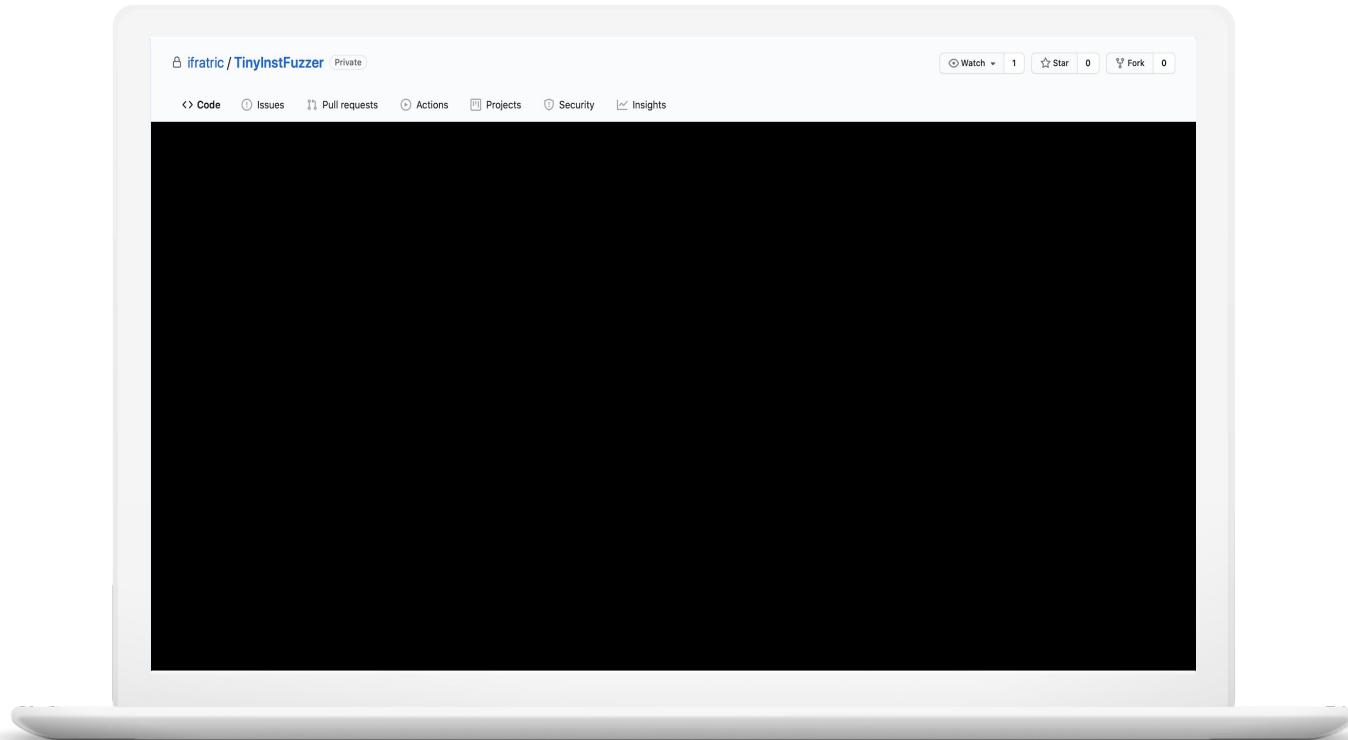
TinyInst Fuzzers



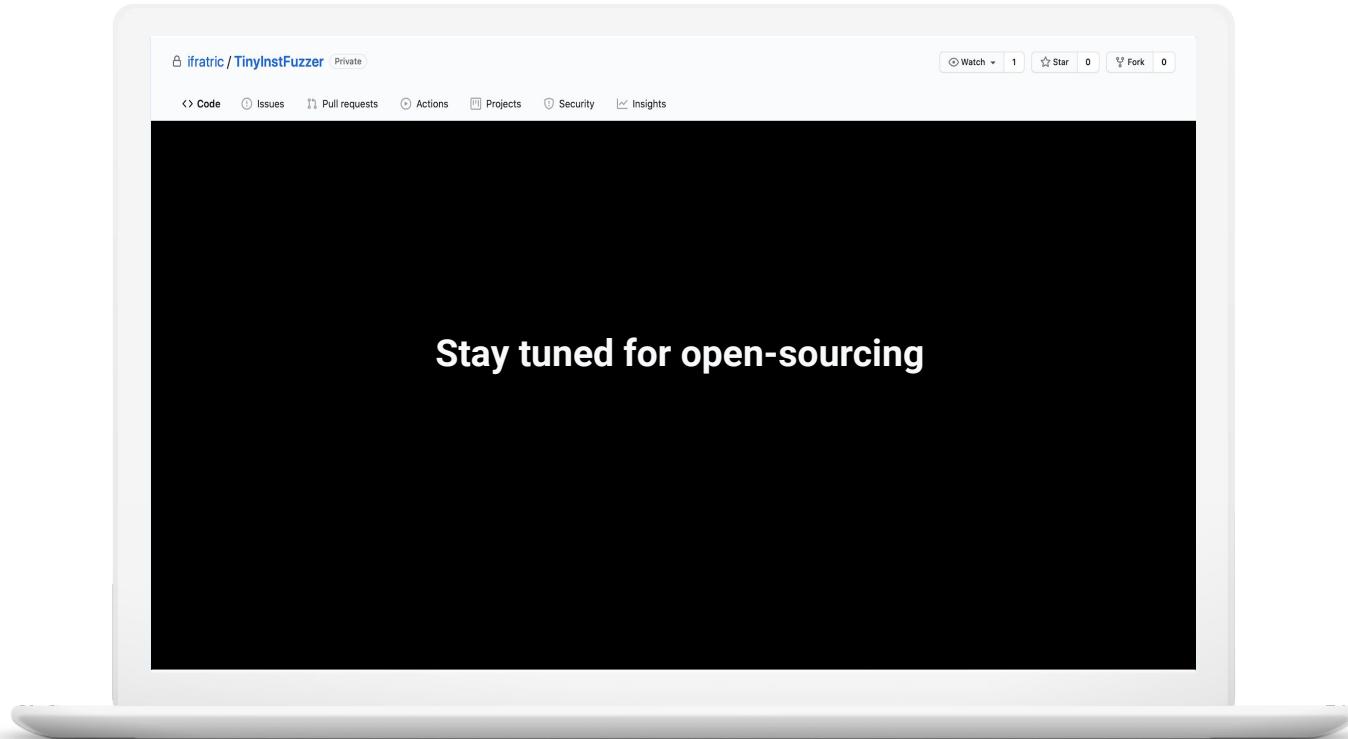
TinyInst Fuzzers



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?