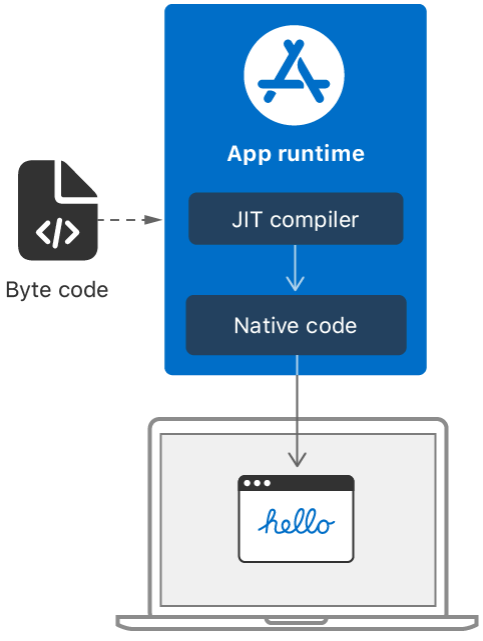
Article

Porting just-in-time compilers to Apple silicon

Update your just-in-time (JIT) compiler to work with the Hardened Runtime capability, and with Apple silicon.

Overview

A just-in-time (JIT) compiler translates byte-code or intermediate script code into machine-language instructions, and makes those instructions available for execution. An app initiates JIT compilation as needed to support relevant tasks, and the compilation process takes place within the app's process space. For example, a web browser uses JIT compilation to transform a web page's script code into runnable code when the user interacts with the appropriate page elements.



Because JIT compilation uses some techniques that the Hardened Runtime capability specifically disallows, you need to update your app if it supports that capability. Even if your app doesn't adopt

the Hardened Runtime, you need to still make changes to support Apple silicon. For more information about configuring the Hardened Runtime capability, see Hardened Runtime.

Enable the JIT entitlements for the Hardened Runtime

The Hardened Runtime capability prohibits the execution of code in a memory page, unless that code is accompanied by a valid code signature. Because a JIT compiler doesn't sign the code it generates, it technically violates the rules of the Hardened Runtime. To fix this issue, you need to adjust your Hardened Runtime settings to allow JIT-related activities.

To allow JIT compilation in your app, navigate to the Hardened Runtime capability in Xcode and enable the Allow Execution of JIT-compiled Code option for your app. When you enable this option, Xcode adds the `com.apple.security.cs.allow-jit` entitlement to your app. When this entitlement is present, the system allows your app to call `mmap` with the `MAP_JIT` flag. If you don't have this entitlement, calls using that flag return an error.

When your app has the Hardened Runtime capability and the `com.apple.security.cs.allow-jit` entitlement, it can only create one memory region with the `MAP_JIT` flag set. The `com.apple.security.cs.allow-jit` entitlement is required only when an app adopts the Hardened Runtime capability. If you don't adopt this capability, you don't need the entitlement to use the `MAP_JIT` flag. For more information about the entitlement, see `Allow execution of JIT-compiled code entitlement`.

Additionally, add the `com.apple.security.cs.jit-write-allowlist` entitlement with the value `true`, to enable JIT callback allow lists. Adding this entitlement allows your to call `pthread_jit_write_with_callback_np()`, which you use to write to your app's JIT region.

> **Note**
>
> On macOS, when you add the `com.apple.security.cs.jit-write-allowlist` entitlement, your app can no longer call `pthread_jit_write_protect_np()`. The `pthread_jit_write_protect_np()` function isn't available on iOS.

When memory protection is enabled, a thread cannot write to a memory region and execute instructions in that region at the same time. Apple silicon enables memory protection for all apps, regardless of whether they adopt the Hardened Runtime. Intel-based Mac computers enable memory protection only for apps that adopt the Hardened Runtime.

Create a callback to write JIT instructions

Your app writes to the `MAP_JIT` memory region in a callback function, that you pass to the `pthread_jit_write_with_callback_np()` function. The callback needs to take a context

pointer as an argument, and return its result as an integer. The callback needs to assume that the memory pointed to by the context pointer can be controlled by an attacker, and validate that the instructions to be written are permitted. For example:

```
// Initialize this pointer using mmap() with the MAP_JIT flag.
static void *jit_region;

struct jit_code {
  void *instructions;
  ptrdiff_t entry_point;
  size_t instructions_length;
}

enum jit_code_safety {
  JIT_CODE_VALID,
  JIT_CODE_INVALID,
};

// Ensure that the JIT code is safe to execute.
enum jit_code_safety validate_jit_code(struct jit_code *code);

int jit_writing_callback(void *context) {
  struct jit_code *code = (struct jit_code *)context;
  if (validate_jit_code(code) == JIT_CODE_VALID) {
    memcpy(jit_region, code->instructions, code->instructions_length);
    return 0;
  } else {
    return -1;
  }
}
```

Add your JIT callback to the allowlist

Each executable, for example, your app binary, can define at most one allowlist of functions that the executable uses as callbacks to `pthread_jit_write_with_callback_np()`.

Important

Your app crashes if you call `pthread_jit_write_with_callback_np()` with a callback function that isn't in the allowlist.

Define your app's allowlist using the `PTHREAD_JIT_WRITE_ALLOW_CALLBACKS_NP` macro:

```
PTHREAD_JIT_WRITE_ALLOW_CALLBACKS_NP(jit_writing_callback)
```

On macOS, if you need to update the allowlist with callbacks you load at runtime, for example from dynamic libraries you load using `dlopen()`, follow these steps:

1. Add the `com.apple.security.cs.jit-write-allowlist-freeze-late` entitlement with the value `true`.
2. Define at most one allowlist in each dynamic library using `PTHREAD_JIT_WRITE_ALLOW_CALLBACKS_NP`.
3. Load the dynamic libraries in your main executable.
4. Call `pthread_jit_write_freeze_callbacks_np()`.

If you add the `com.apple.security.cs.jit-write-allowlist-freeze-late` entitlement, you need to call `pthread_jit_write_freeze_callbacks_np()` before your first call to `pthread_jit_write_with_callback_np()`. If you call `pthread_jit_write_with_callback_np()` before freezing the callbacks allowlist, the system terminates your process with an error.

> **Important**
>
> Freeze the callback allowlist by calling `pthread_jit_write_freeze_callbacks_np()` early in your app's startup process, before the app starts processing untrusted inputs.

Write instructions to your JIT memory region

Pass your callback and context pointer to `pthread_jit_write_with_callback_np()` to write code to the `MAP_JIT` memory region. This function:

1. Checks that your callback function is in the allowlist.
2. Makes the memory region writable, and not executable, for the current thread only.
3. Runs your callback function, passing the context pointer as the argument.
4. Makes the memory region executable, and not writable, for the current thread.
5. Returns the value that your callback function returned.

```
struct jit_code code = {0};
int status = jit_compile("console.log(\"Hello, world!\");", &code);
```

```
if (status == JIT_COMPILATION_SUCCESS) {
  pthread_jit_write_with_callback_np(jit_writing_callback, &code);
} else {
  // Handle a compilation error.
}
```

Invalidate caches and execute the code

Always call `sys_icache_invalidate(_:_:)` before you execute the machine instructions on a recently updated memory page. On Apple silicon, the instruction caches aren't coherent with data caches, and unexpected results might occur if you execute instructions without invalidating the caches. It's also safe to call the `sys_icache_invalidate(_:_:)` function on Intel-based Mac computers, where the function does nothing.

```
sys_icache_invalidate(jit_region, code.instructions_length);
void (*run_jit)(void) = jit_region + code.entry_point;
run_jit();
```

See Also

General porting tips

- Addressing architectural differences in your macOS code

  Fix problems that stem from architectural differences between Apple silicon and Intel-based Mac computers.

- Porting your audio code to Apple silicon

  Eliminate issues in your audio-specific code when running on Apple silicon Mac computers.