WWDC 2018: What's new in LLVM

我就是御姐我摊牌了 + 关注
预计阅读时间18分钟 5 年前

目录

- 目录
  - 一分钟速读
  - 在开始之前，我们再来回顾一下 LLVM 是什么
  - 『What's new in LLVM』 具体讲了什么?
    - Updates on ARC
    - New Diagnostics in Xcode 10
      - Swift and Objective-C Interoperability
      - Packing Struct Members with #pragma pack
    - Clang Static Analyzer
      - Grand Central Dispatch performance anti-pattern
      - Autoreleasing variables outliving autorelease pool
      - Improved Performance and Report Visualizations
    - Increased Security
      - Stack Protector
      - Stack Checking
    - New Instruction Set Extensions
      - New Vector Extension: AVX-512
      - ARM v8.1 Atomics
      - ARM v8.2 Float 16
  - 写在最后

一分钟速读

如果你日常写的比较多的，是业务代码，而非底层的基础库，那么此次 LLVM 的更新你可以基本不看，因为他对你的日常开发不会有太多改变，最多是你会发现 Xcode 变得更加严格了——原来正常的代码出现了新的编译器警告。来简单过一下此次 LLVM 的更新内容：

1. Xcode 10 的 ARC 支持在 C 结构体中直接使用 Objective-C 的对象指针
2. Xcode 10 增加了 100+ 个编译器诊断(Compiler Diagnostic)，你在写代码的时候可能会看到更多的 warning 警告
3. Xcode 10 的静态检查器新增了对 GCD 不当使用以及自动释放池变量不当使用的检查，同时提升了性能，优化了 Xcode 中 Analyze 后的界面展示方式
4. LLVM 编译器通过自动添加一些防御性代码，帮助你防范「Stack Buffer Overflow」和「Stack Clash」两个二进制相关的安全漏洞
5. Xcode 10 中，你可以针对 iMac Pro / iPhone X / iPhone 8 / iPhone 8 Plus 这几款具有新 CPU 指令集扩展的设备进行针对性的优化，这些优化有的是需要自己写针对性的平台代码来实现，有的则是编译器自动帮你实现 。

在开始之前，我们再来回顾一下 LLVM 是什么

LLVM 是包括 Clang、LLDB、LLVM Core 在内的一系列编译器相关的项目的总称。这个项目的主要意义，就是在 GNU 的 GCC 体系之外，创建一套新的编译器以及相关工具的体系，让 Xcode 或者其他工具能够更好的集成进来。我们可以认为，我们的 OC / Swift 代码从编写到 Xcode 为我们生成最终的二进制代码中间的所有的过程，都离不开 LLVM 。即便你对编译器等相关知识不是那么熟悉，也应该记住我们如今习以为常的 ARC ，就是利用 LLVM 实现的。

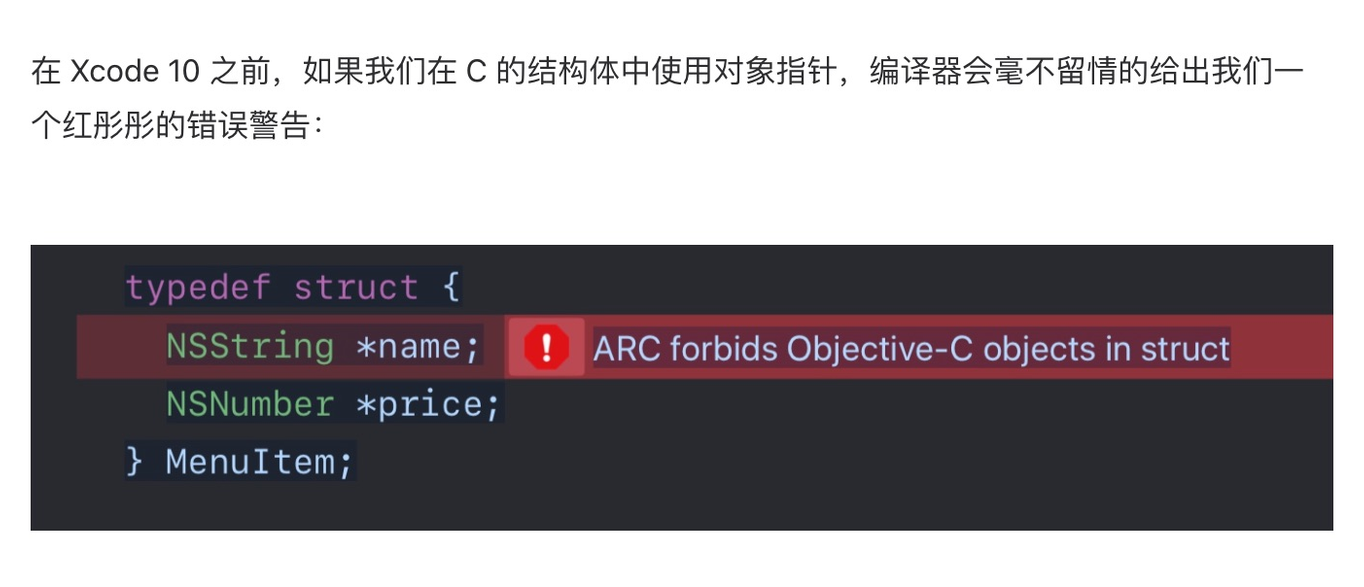
LLVM 的不断迭代，可以为我们带来：

1. 更简洁的编程语言（如 ARC）
2. 更全面的编译期错误检查
3. 更全面的静态分析
4. 更快的编译速度
5. 更高效的二进制代码

『What's new in LLVM』 具体讲了什么?

Updates on ARC

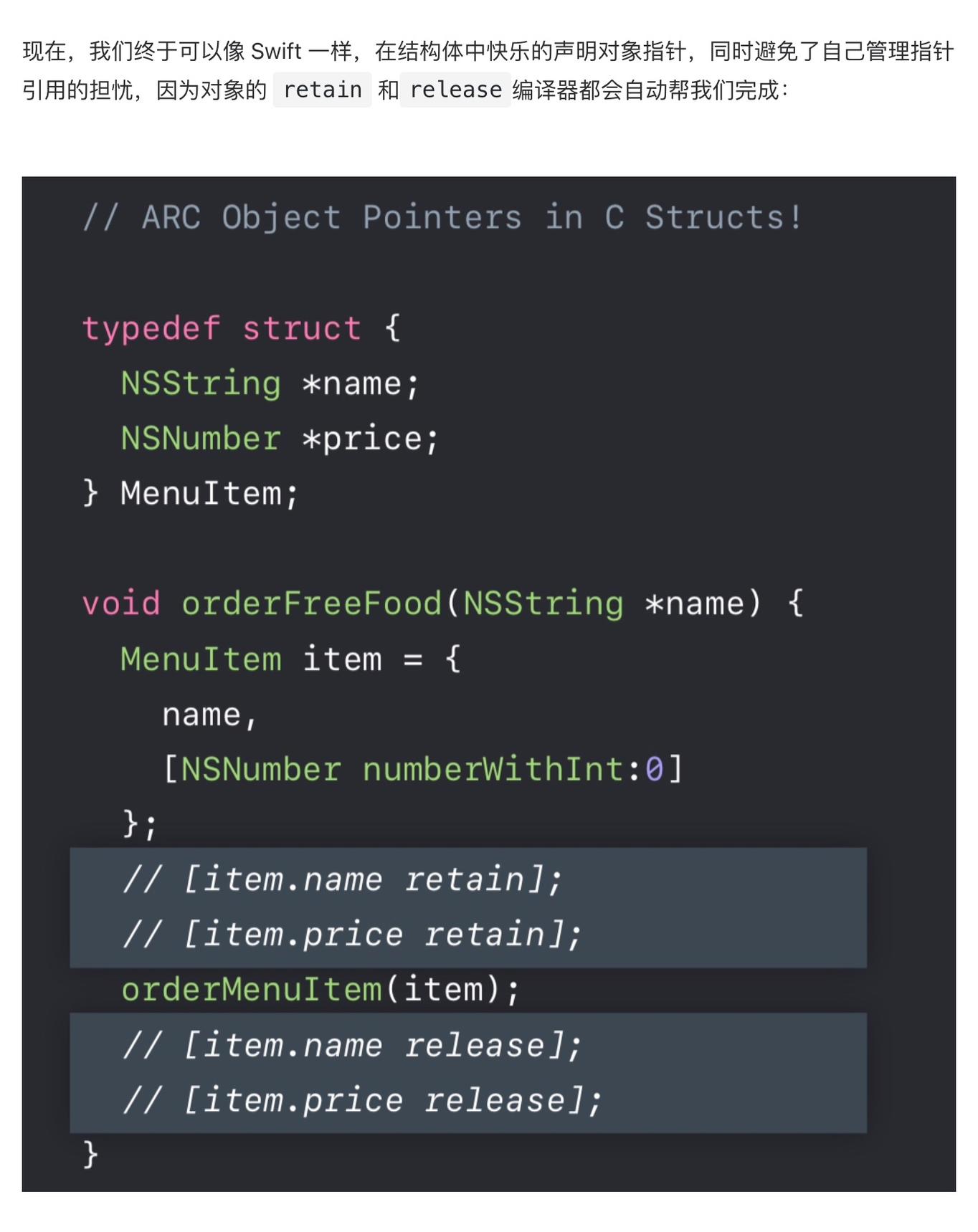
在 Xcode 10 之前，如果我们在 C 的结构体中使用对象指针，编译器会毫不留情的给出我们一个红彤彤的错误警告：



```
typedef struct {
  NSString *name;     ! ARC forbids Objective-C objects in struct
  NSNumber *price;
} MenuItem;
```

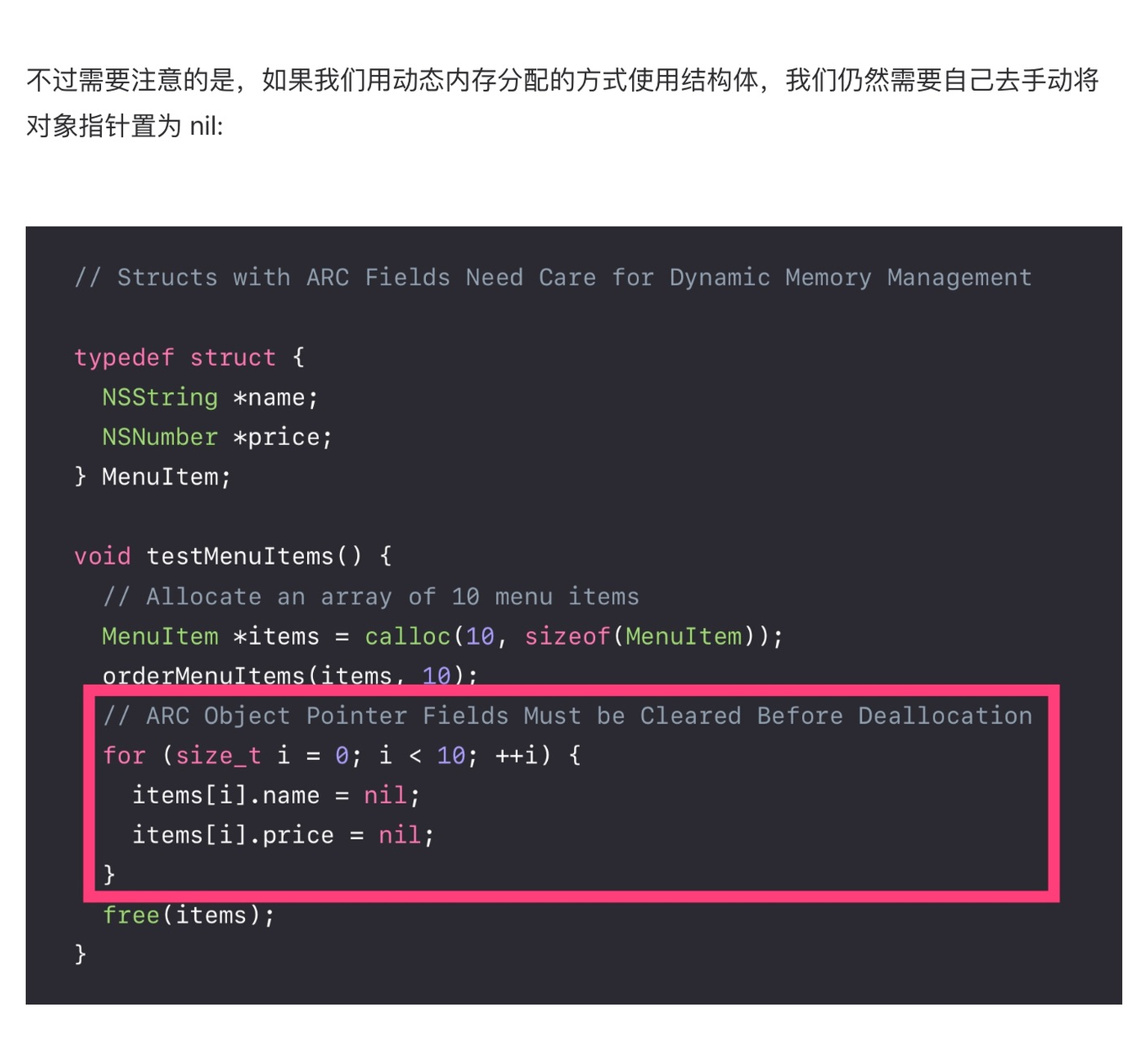
根据 Clang 的 ARC 文档 以及 2011 年的 WWDC Session 323 第 21 页，在之前的 ARC 代码中禁止 C 结构体中使用对象指针，是因为 ARC 要求编译器在编译期就知道结构体中对象指针的引用的变化情况。而在当时，这并不好实现。所以当时苹果的工程师直接建议大家使用类而不是结构体。也有 SO 上的热心网友表示我们可以用 `__unsafe_unretained` 来去掉编译错误。也就是明确告诉编译器，这个对象指针的生命周期你不用管了，我们开发者来管。

现在，我们终于可以像 Swift 一样，在结构体中快乐的声明对象指针，同时对于自己管理指针引用的担忧，因为对象的 `retain` 和 `release` 编译器都会自动帮我们完成：



```
// ARC Object Pointers in C Structs!

typedef struct {
  NSString *name;
  NSNumber *price;
} MenuItem;

void orderFreeFood(NSString *name) {
  MenuItem item = {
    name,
    [NSNumber numberWithInt:0]
  };
  // [item.name retain];
  // [item.price retain];
  orderMenuItem(item);
  // [item.name release];
  // [item.price release];
}
```

不过需要注意的是，如果我们用动态内存分配的方式使用结构体，我们仍然需要自己去手动将对象指针置为 nil:



```
// Structs with ARC Fields Need Care for Dynamic Memory Management

typedef struct {
  NSString *name;
  NSNumber *price;
} MenuItem;

void testMenuItems() {
  // Allocate an array of 10 menu items
  MenuItem *items = calloc(10, sizeof(MenuItem));
  orderMenuItems(items, 10);
  // ARC Object Pointer Fields Must be Cleared Before Deallocation
  for (size_t i = 0; i < 10; ++i) {
    items[i].name = nil;
    items[i].price = nil;
  }
  free(items);
}
```

最后再来说说我个人认为这个变化给我们编程上带来的改变：

我们可以声明一些『轻量』的模型。比如我们有一些非常简单的模型对象，就不用很麻烦的声明一个类，而是用一个结构体迅速解决战斗。虽然这里的结构体并不能像 Swift 中的结构体那样强大，但毕竟是一个进步。

最后需要注意的是，带有对象指针的结构体，是不能引入到 Swift 中的。

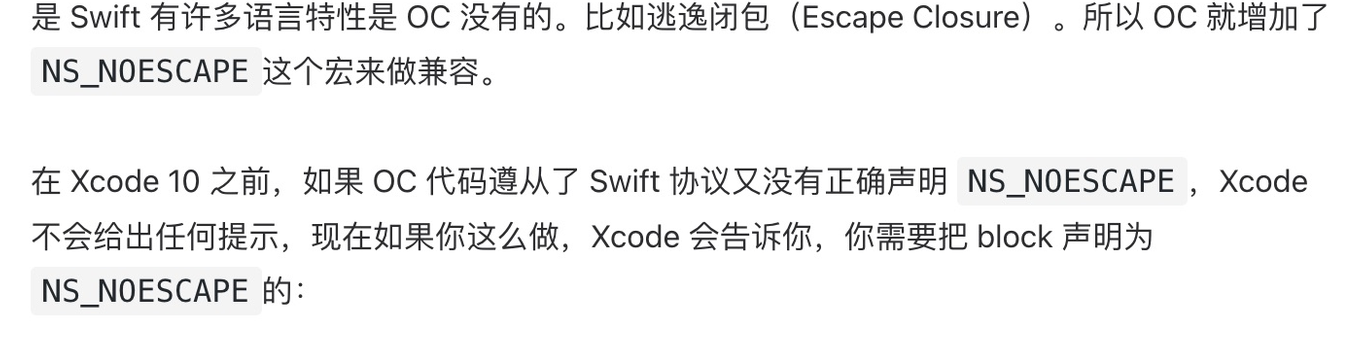
New Diagnostics in Xcode 10

这一部分对我们最大的影响，就是我们在写代码的时候可以看到更多的编译警告了 😅。苹果的工程师宣称这一次他们增加了 100+ 个代码诊断项，以帮助我们写出更有质量的代码。在 Session 中，他挑了两个主要的诊断作为演示。

Swift and Objective-C Interoperability

我们知道，当我们想在 OC 的代码中使用 Swift 代码时，需要为 Swift 代码生成一个头文件。但是 Swift 有许多语言特性是 OC 没有的，比如逃逸闭包（Escape Closure），所以 OC 就增加了 `NS_NOESCAPE` 这个宏来做兼容。

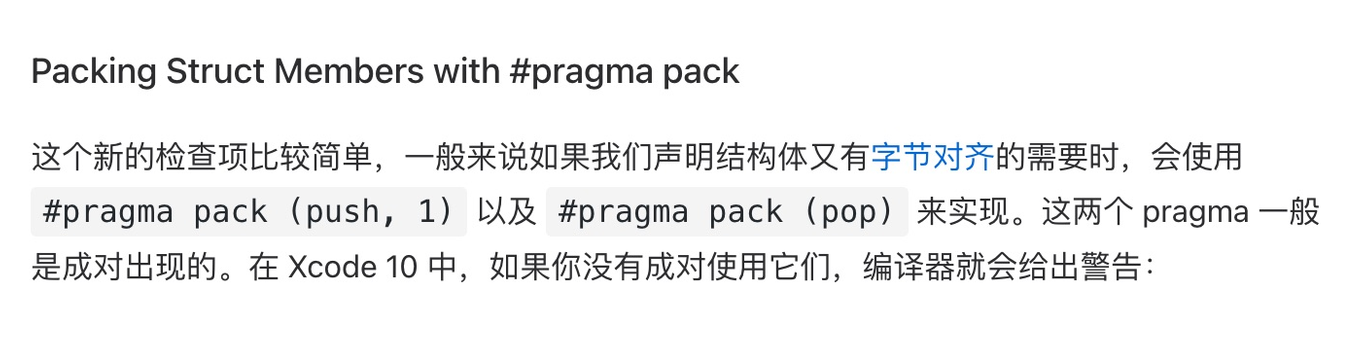
在 Xcode 10 之前，如果 OC 代码遵从了 Swift 协议又没有正确声明 `NS_NOESCAPE` ，Xcode 不会给出任何提示，现在如果你这么做，Xcode 会告诉你，你需要把 block 声明为 `NS_NOESCAPE` 的：



```
// Warning for Missing Noescape Annotations for Method Overrides

#import "Executor-Swift.h"
@interface DispatchExecutor : NSObject<Executor>
- (void) performOperation:(void (^)(void)) handler;   ⚠ Parameter must be annotated with noescape
@end
```

Packing Struct Members with #pragma pack

这个新的检查项比较简单，一般来说如果我们声明结构体又有字节对齐的需要时，会使用 `#pragma pack (push, 1)` 以及 `#pragma pack (pop)` 来实现，这两个 pragma 一般是成对使用的。在 Xcode 10 中，如果你没有成对使用它们，编译器就会给出警告：



```
#pragma pack (push, 1)   ⚠ Unterminated '#pragma pack (push, ...)' at end of file
struct PackedStruct {
  uint8_t a, b;
  uint32_t c;
};
// The programmer forgot #pragma pack (pop)
```

Clang Static Analyzer

静态检查器和编译诊断不同，静态检查需要我们手动的执行 Product -> Analyze 才能看到结果。在这一部分，苹果的工程师表示，我们努力了一年，为大家带来了两个新的静态检查， 同时优化了界面展示效果并提升了静态检查的性能。

Grand Central Dispatch performance anti-pattern

所谓 anti-pattern（反模式），一般指的是哪些乍一眼看上去还 OK ，但是实际运行起来往往得不偿失没有益处的代码编写模式。一个典型的例子莫过于『单例模式』，当大家滥用单例模式以后，单例模式就成了反模式。

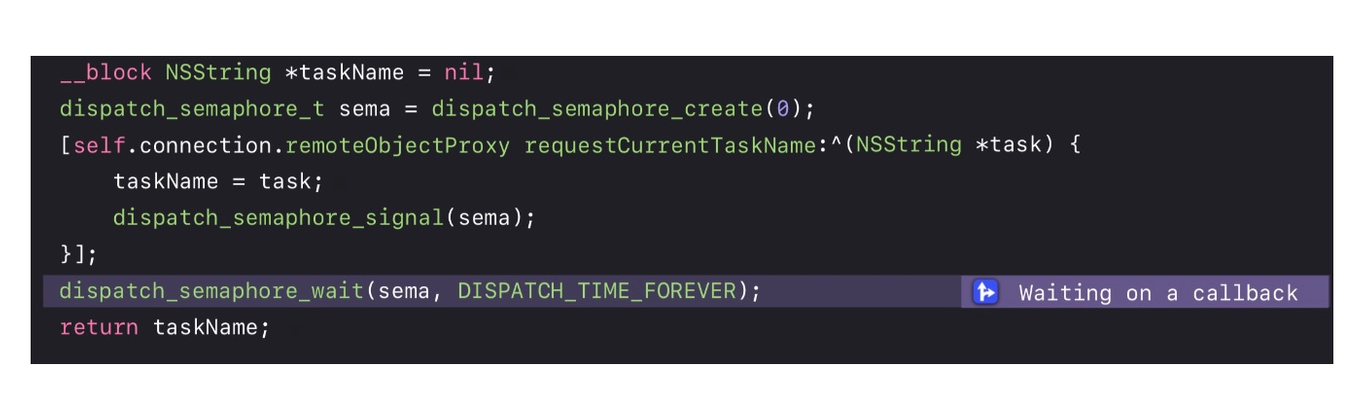
而在使用 GCD 的过程中也有一种不太好的模式，我们先来看下面一段代码：

```
+ (NSString *)requestCurrentTaskName {
    __block NSString *taskName = nil;
    dispatch_semaphore_t sema = dispatch_semaphore_create(0);
    NSXPCConnection *connection =
    [[NSXPCConnection alloc] initWithServiceName:@"MyConnection"];
    id remoteObjectProxy = connection.remoteObjectProxy;
    [remoteObjectProxy requestCurrentTaskName:^(NSString *task) {
        taskName = task;
        dispatch_semaphore_signal(sema);
    }];
    dispatch_semaphore_wait(
                            sema,
                            dispatch_time(DISPATCH_TIME_NOW, 100)
                            );
    return taskName;
}
```

乍一看，这段代码好像没有什么问题，我们使用信号量来保证当前方法在异步获取到 taskName 之后才会返回。然后仔细推敲一下，这段代码是有问题的：

1. 当前线程在等待另外一个异步线程结束，而这个异步线程一般是优先级比较低的线程，因此就出现了优先级高的线程等待优先级低的线程这种优先级逆转的不合理现象。
2. 新建了一个线程，并使用了锁，产生了多余的线程间交互的成本

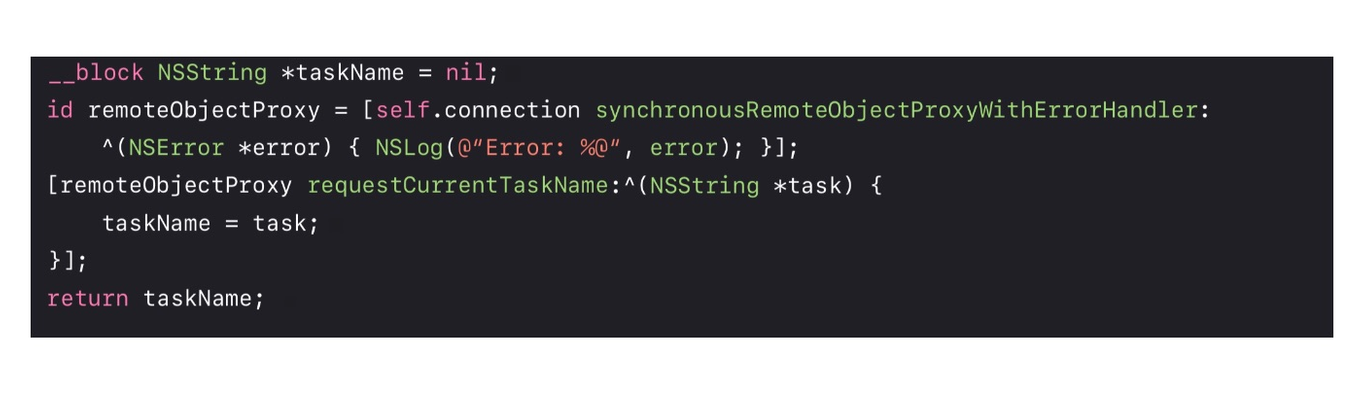
Xcode 10 会帮我们检查出来这样的问题代码，并给与我们善意的提醒：



```
__block NSString *taskName = nil;
dispatch_semaphore_t sema = dispatch_semaphore_create(0);
[self.connection.remoteObjectProxy requestCurrentTaskName:^(NSString *task) {
    taskName = task;
    dispatch_semaphore_signal(sema);
}];
dispatch_semaphore_wait(sema, DISPATCH_TIME_FOREVER);          Waiting on a callback
return taskName;
```

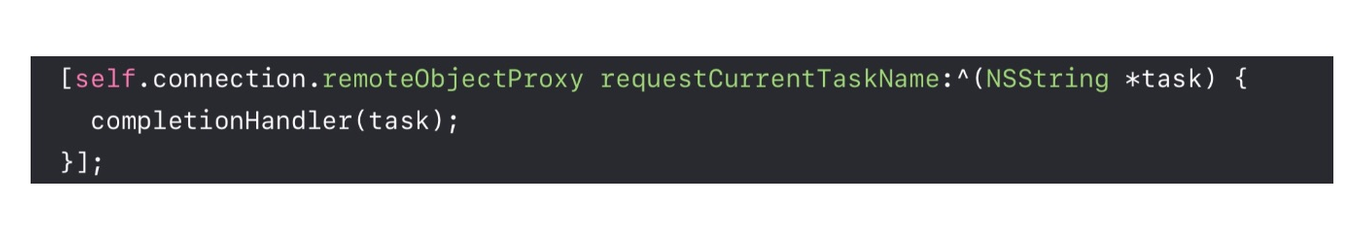
那这段代码应该如何改正呢?

一种方法是如果当前的返回值是同步的，就使用同步版本的底层 API：



```
__block NSString *taskName = nil;
id remoteObjectProxy = [self.connection synchronousRemoteObjectProxyWithErrorHandler:
    ^(NSError *error) { NSLog(@"Error: %@", error); }];
[remoteObjectProxy requestCurrentTaskName:^(NSString *task) {
    taskName = task;
}];
return taskName;
```

另外一种是，将当前的方法风格改为异步回调风格：

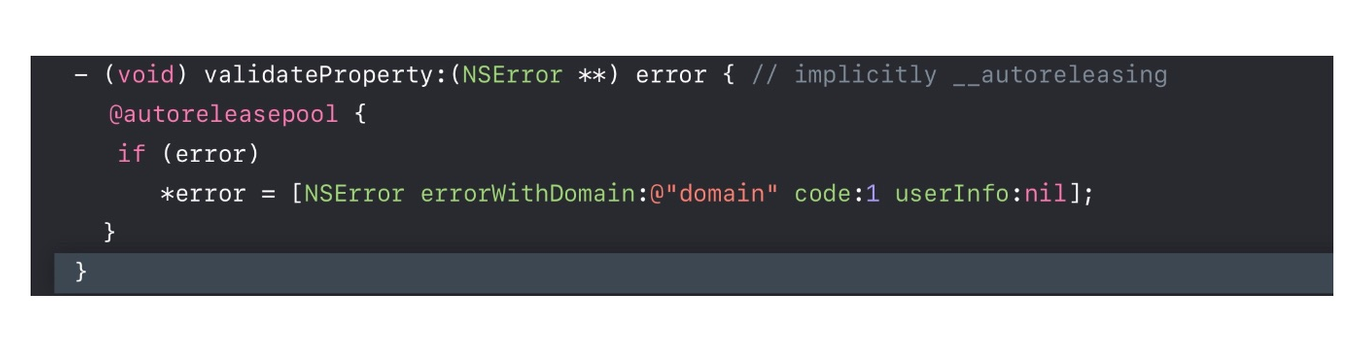


```
[self.connection.remoteObjectProxy requestCurrentTaskName:^(NSString *task) {
  completionHandler(task);
}];
```

Autoreleasing variables outliving autorelease pool

我们都知道，在 `@autoreleasepool` 中的对象会在离开 `@autoreleasepool` 作用域后释放。Xcode 10 新增了静态检查器帮助查出来哪些在 `@autoreleasepool` 以外访问变量地址的问题代码。

比如我们有这样的错误代码：



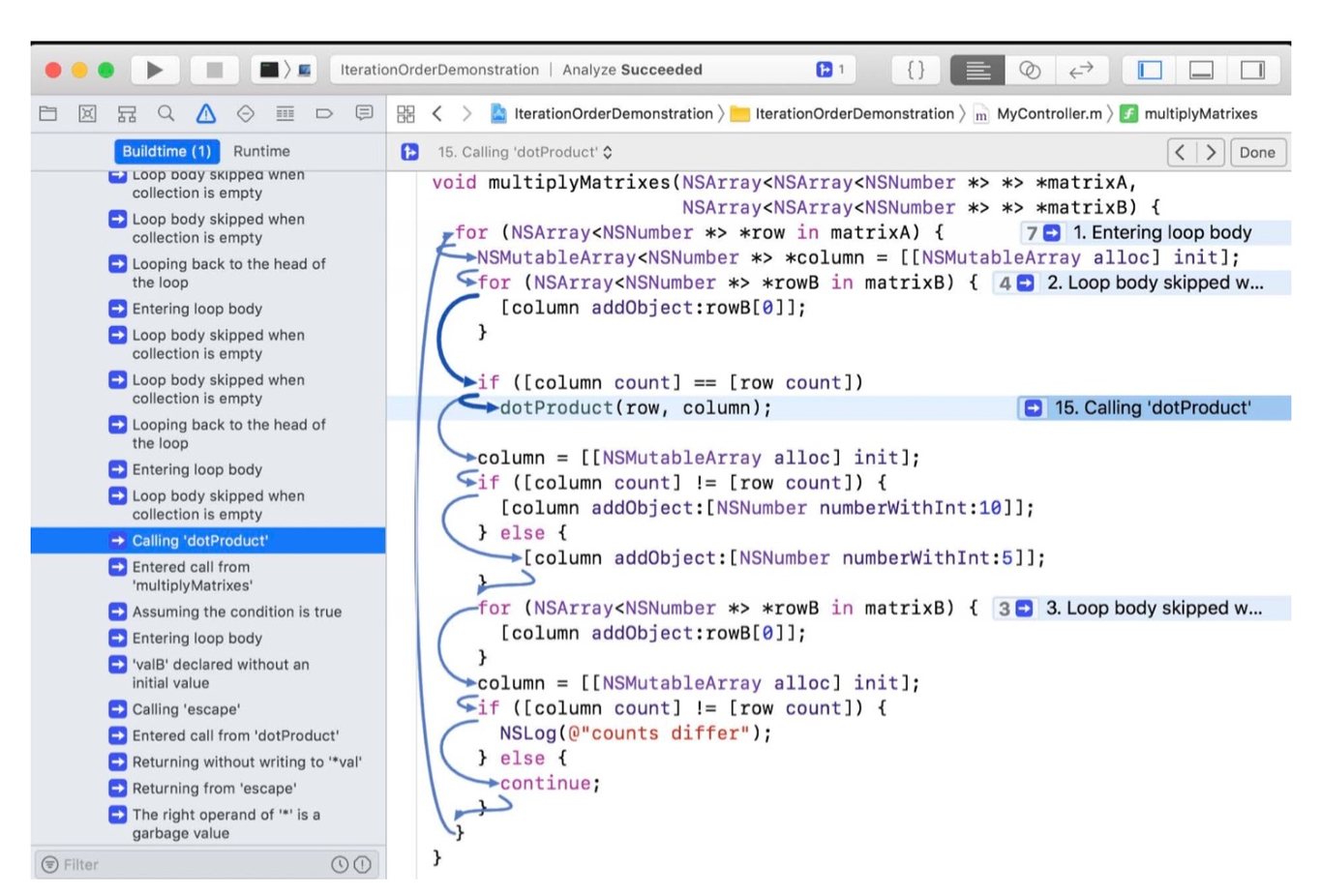
```
- (void) validateProperty:(NSError **) error { // implicitly __autoreleasing
    @autoreleasepool {
     if (error)
        *error = [NSError errorWithDomain:@"domain" code:1 userInfo:nil];
    }
}
```

error 在执行完当前方法后就会被释放，而此时如果外界去访问 error 对象，就会喜闻乐见了 Boom —— 发生 crash 。Xcode 10 的静态检查器可以帮我们在编译器就检查出来这样的问题。

Improved Performance and Report Visualizations

在这一部分，苹果的工程师表示，Xcode 10 上的静态检查器在同样的时间内，平均比 Xcode 9 多检查出来 15% 的问题。

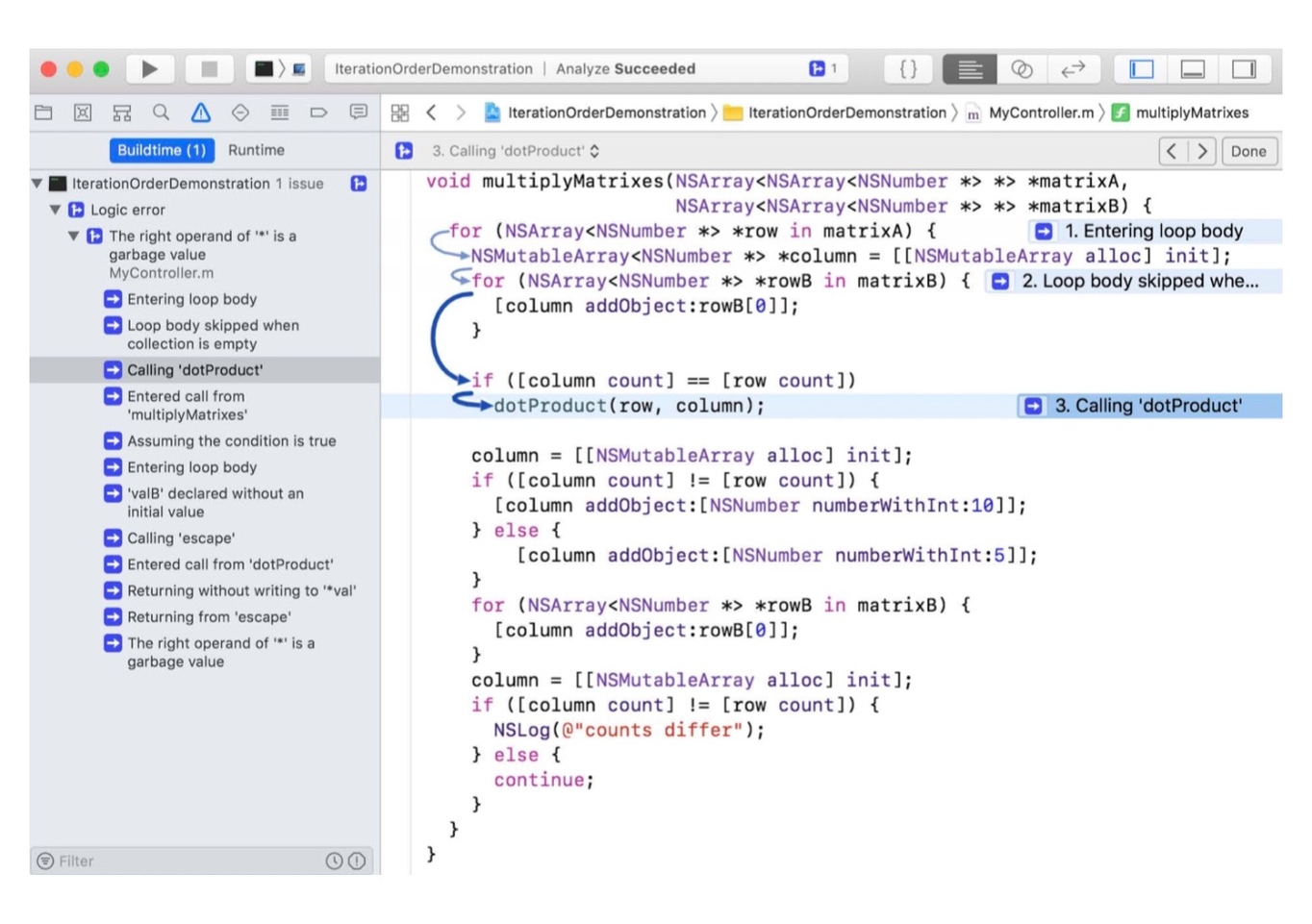
同时他们也优化了问题复现路径的展示，去掉了一些不必要的路径。比如在 Xcode 9 中一个检查结果会这么展示：



IterationOrderDemonstration | Analyze Succeeded
IterationOrderDemonstration 〉 IterationOrderDemonstration 〉 MyController.m 〉 multiplyMatrixes
Buildtime (1) Runtime
15. Calling 'dotProduct'
Done
Loop body skipped when collection is empty
Looping back to the head of the loop
Entering loop body
Calling 'dotProduct'
Entered call from 'multiplyMatrixes'
Assuming the condition is true
'valB' declared without an initial value
Calling 'escape'
Entered call from 'dotProduct'
Returning without writing to '*val'
Returning from 'escape'
The right operand of '*' is a garbage value
Filter

```
void multiplyMatrixes(NSArray<NSArray<NSNumber *> *> *matrixA,
                      NSArray<NSArray<NSNumber *> *> *matrixB) {
  for (NSArray<NSNumber *> *row in matrixA) {                   7  1. Entering loop body
    NSMutableArray<NSNumber *> *column = [[NSMutableArray alloc] init];
    for (NSArray<NSNumber *> *rowB in matrixB) {                4  2. Loop body skipped w...
      [column addObject:rowB[0]];
    }

    if ([column count] == [row count])
      dotProduct(row, column);                                     15. Calling 'dotProduct'

    column = [[NSMutableArray alloc] init];
    if ([column count] != [row count]) {
      [column addObject:[NSNumber numberWithInt:10]];
    } else {
      [column addObject:[NSNumber numberWithInt:5]];
    }
    for (NSArray<NSNumber *> *rowB in matrixB) {                3  3. Loop body skipped w...
      [column addObject:rowB[0]];
    }
    column = [[NSMutableArray alloc] init];
    if ([column count] != [row count]) {
      NSLog(@"counts differ");
    } else {
      continue;
    }
  }
}
```

而在 Xcode 10 中就会仅展示关键的部分：



IterationOrderDemonstration | Analyze Succeeded
IterationOrderDemonstration 〉 IterationOrderDemonstration 〉 MyController.m 〉 multiplyMatrixes
Buildtime (1) Runtime
3. Calling 'dotProduct'
Done
IterationOrderDemonstration 1 issue
Logic error
The right operand of '*' is a garbage value
MyController.m
Entering loop body
Loop body skipped when collection is empty
Calling 'dotProduct'
Entered call from 'multiplyMatrixes'
Assuming the condition is true
Entering loop body
'valB' declared without an initial value
Calling 'escape'
Entered call from 'dotProduct'
Returning without writing to '*val'
Returning from 'escape'
The right operand of '*' is a garbage value
Filter

```
void multiplyMatrixes(NSArray<NSArray<NSNumber *> *> *matrixA,
                      NSArray<NSArray<NSNumber *> *> *matrixB) {
  for (NSArray<NSNumber *> *row in matrixA) {                   1. Entering loop body
    NSMutableArray<NSNumber *> *column = [[NSMutableArray alloc] init];
    for (NSArray<NSNumber *> *rowB in matrixB) {                2. Loop body skipped whe...
      [column addObject:rowB[0]];
    }

    if ([column count] == [row count])
      dotProduct(row, column);                                  3. Calling 'dotProduct'

    column = [[NSMutableArray alloc] init];
    if ([column count] != [row count]) {
      [column addObject:[NSNumber numberWithInt:10]];
    } else {
      [column addObject:[NSNumber numberWithInt:5]];
    }
    for (NSArray<NSNumber *> *rowB in matrixB) {
      [column addObject:rowB[0]];
    }
    column = [[NSMutableArray alloc] init];
    if ([column count] != [row count]) {
      NSLog(@"counts differ");
    } else {
      continue;
    }
  }
}
```

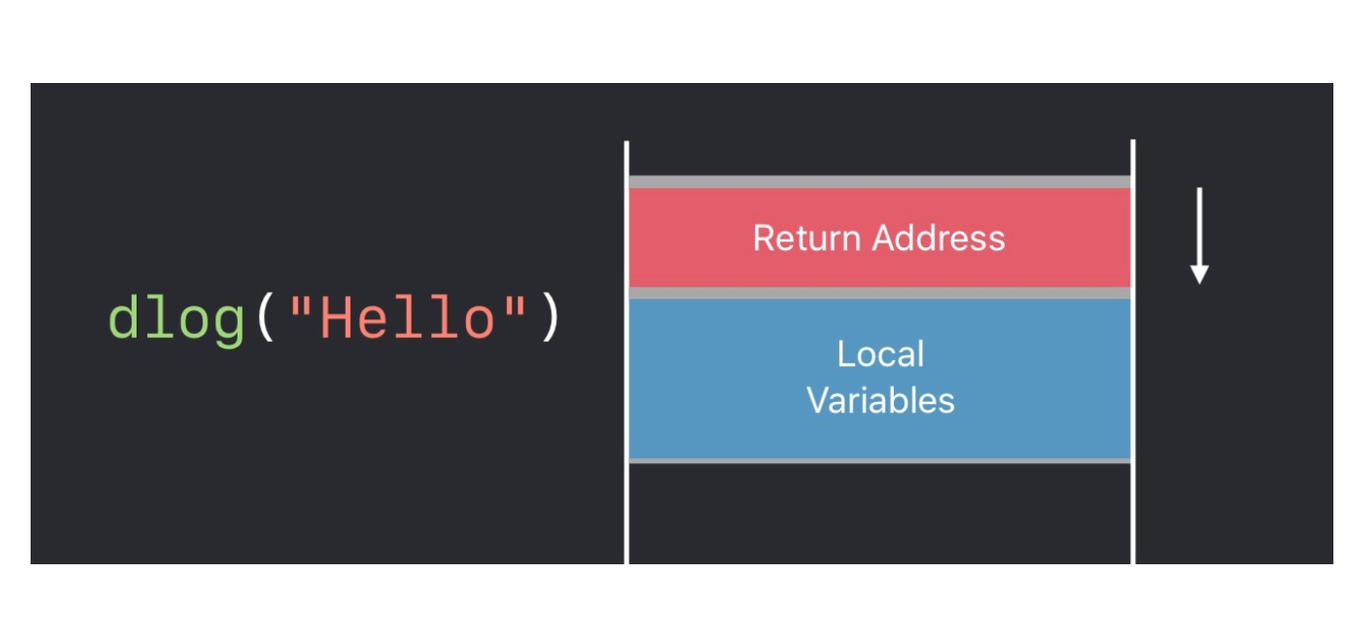
Increased Security

这一部分的内容可能绝大多数 iOS 开发者都不太会关心，除非是做底层安全防护的的开发者。

Stack Protector

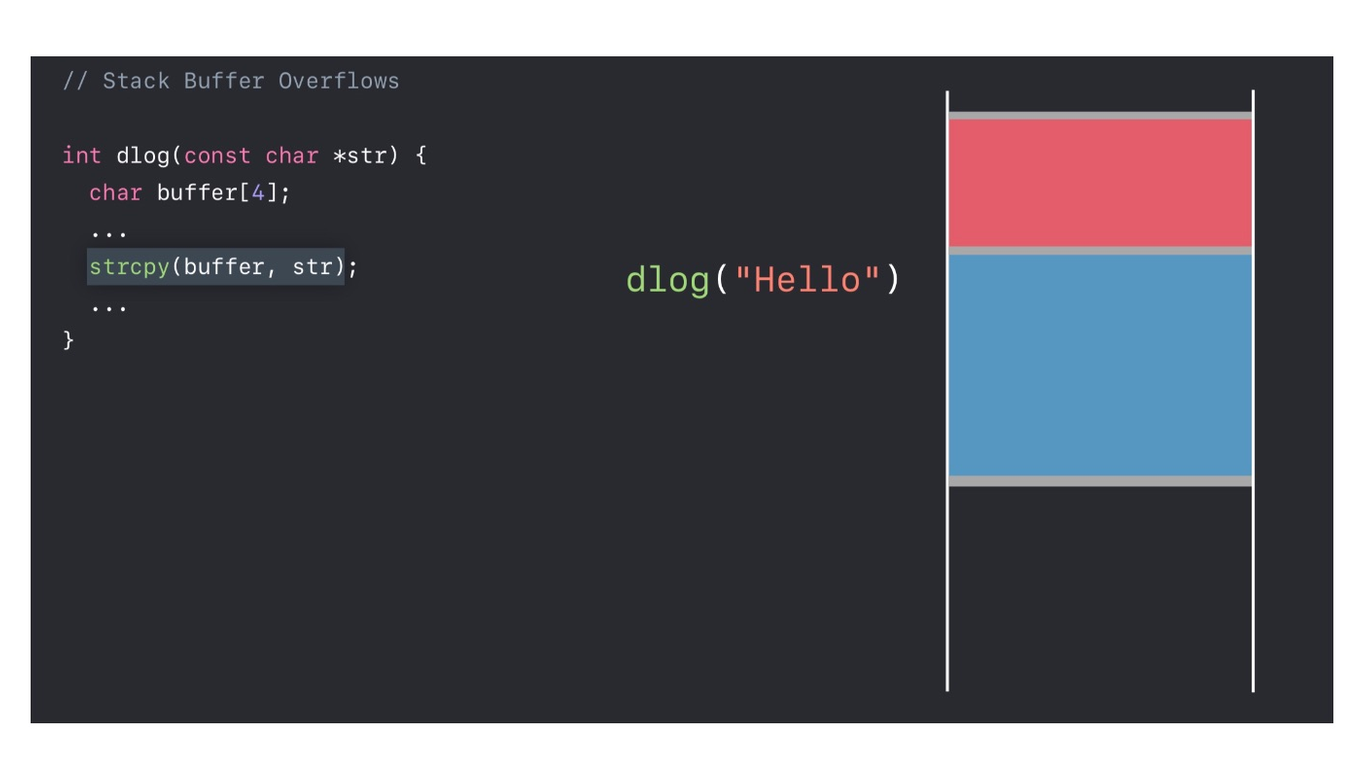
Stack Protector 这个特性，主要是为了防护 Stack Buffer Overflow Attack。什么意思呢？我尽量用简单的语言解释一下。

假设我们有一个 `dlog` 函数，接收一个字符串作为参数。当调用 `dlog` 函数时，CPU 会先在内存上记录一下当前的函数地址，也就是红色的 Return Address，然后根据 `dlog` 的局部变量的大小分配对应的栈。



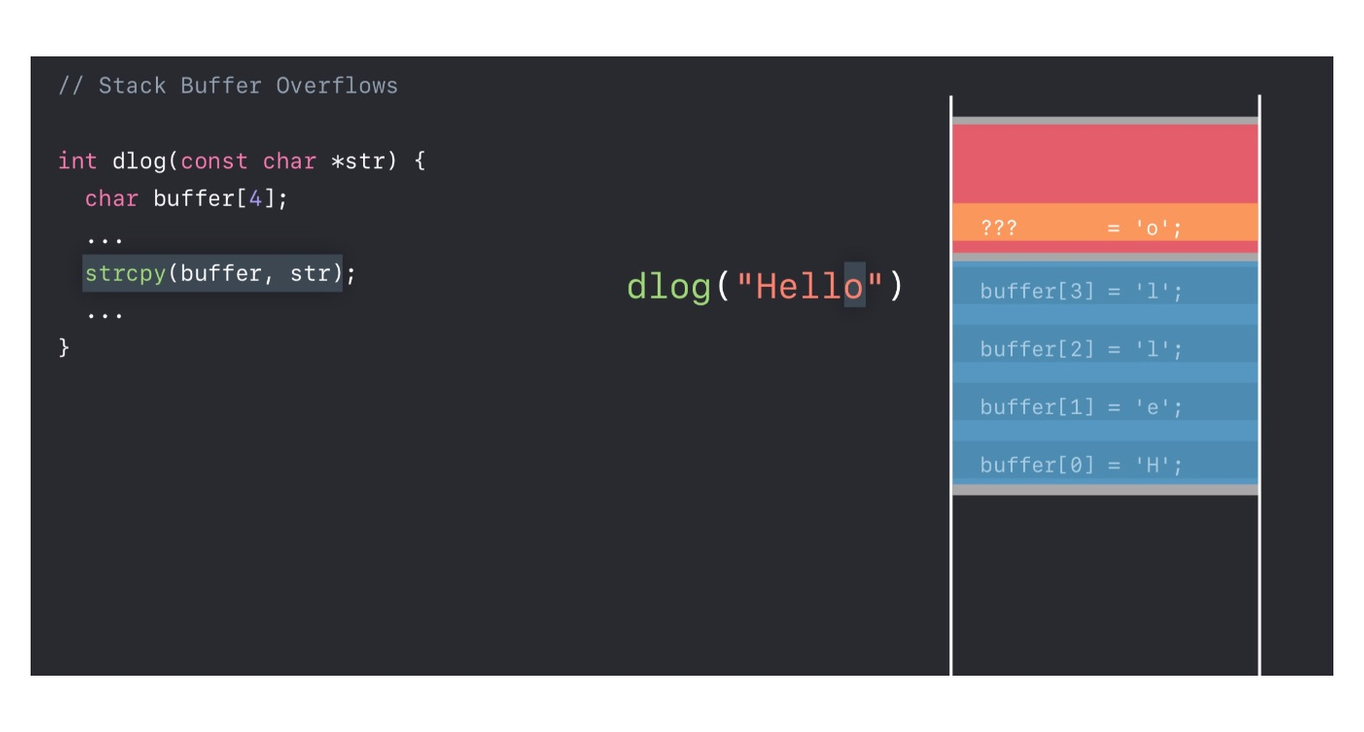
dlog("Hello")
Return Address
Local Variables

假如，我们的 `dlog` 函数中有一个类似 `strcpy` 这样能够写内存的方法，并把函数的参数作为参数，那么，这里就可能会出现一个安全漏洞：Return Address 可能会被改写。



```
// Stack Buffer Overflows

int dlog(const char *str) {
  char buffer[4];
  ...
  strcpy(buffer, str);
  ...
}
```

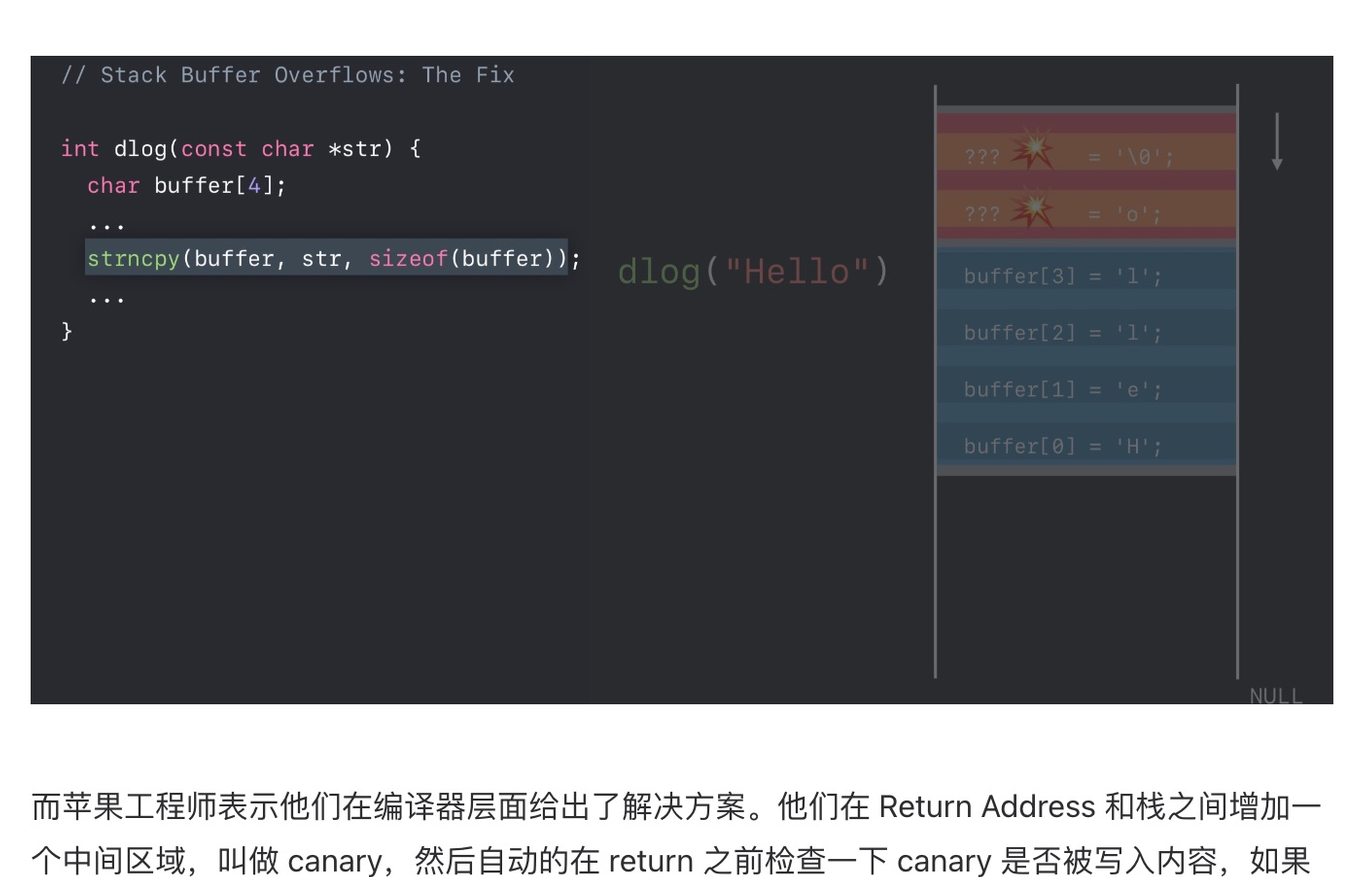
别有用心的攻击者只要将传入的参数的 length 设置为大于 buffer 长度的字符，就可以改写 Return Address 的值，进而控制当前函数执行结束后，下一个调用的函数地址。



```
// Stack Buffer Overflows

int dlog(const char *str) {
  char buffer[4];
  ...
  strcpy(buffer, str);
  ...
}
```

dlog("Hello")
??? = 'o'
buffer[3] = 'l'
buffer[2] = 'l'
buffer[1] = 'e'
buffer[0] = 'H'

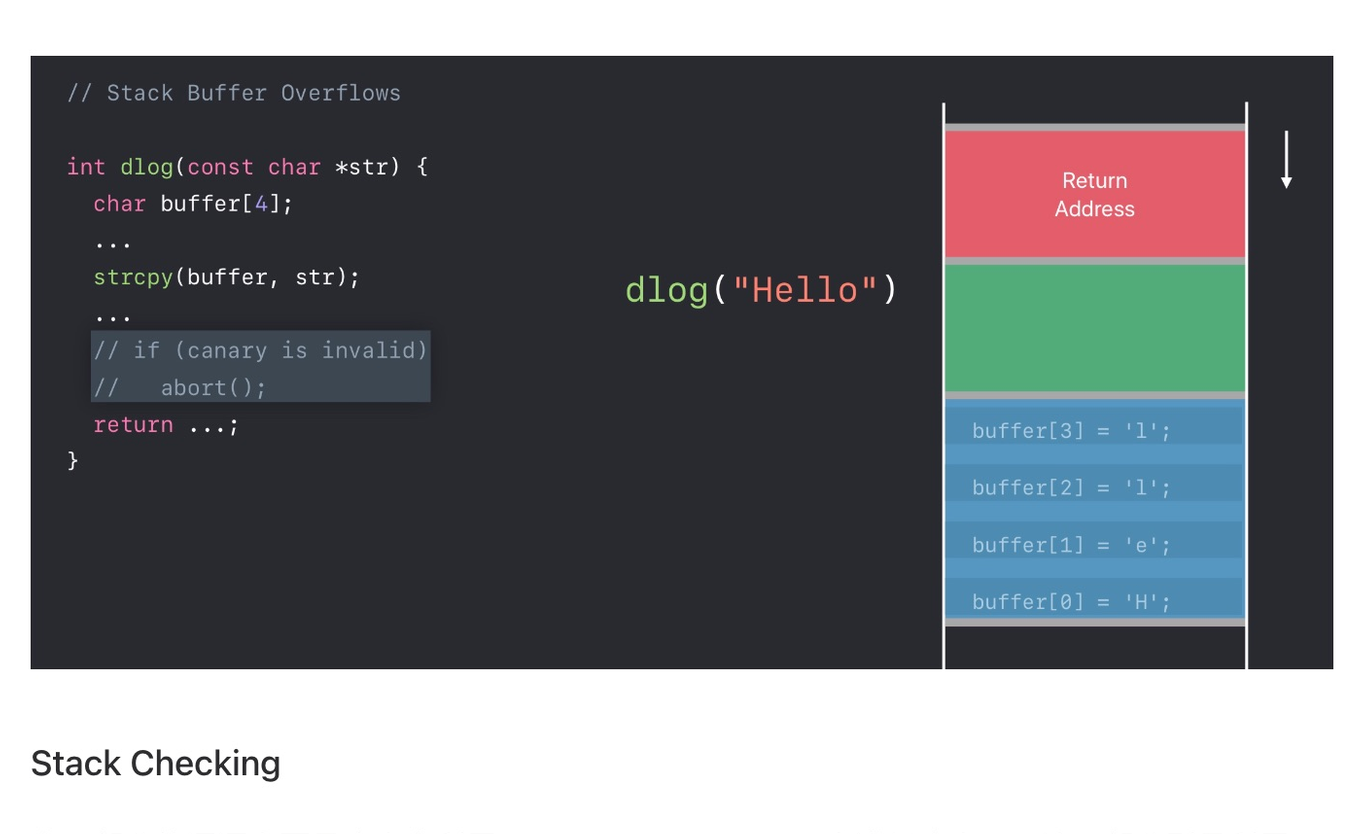
一般代码编写者在编写的过程中，需要自己去注意这个问题，写对应的判断逻辑，避免预期之外的内存写入：



```
// Stack Buffer Overflows: The Fix

int dlog(const char *str) {
  char buffer[4];
  ...
  strcpy(buffer, str, sizeof(buffer));
  ...
}
```

dlog("Hello")
buffer[3] = 'l'
buffer[2] = 'l'
buffer[1] = 'e'
buffer[0] = 'H'

而苹果工程师表示他们在编译器层面给出了解决方案，他们在 Return Address 和栈之间增加一个中间区域，叫做 canary，然后自动的在 return 之前检查一下 canary 是否被写入内容，如果写入，就利用 `abort()` 当前程序的执行，阻止攻击者的攻击。



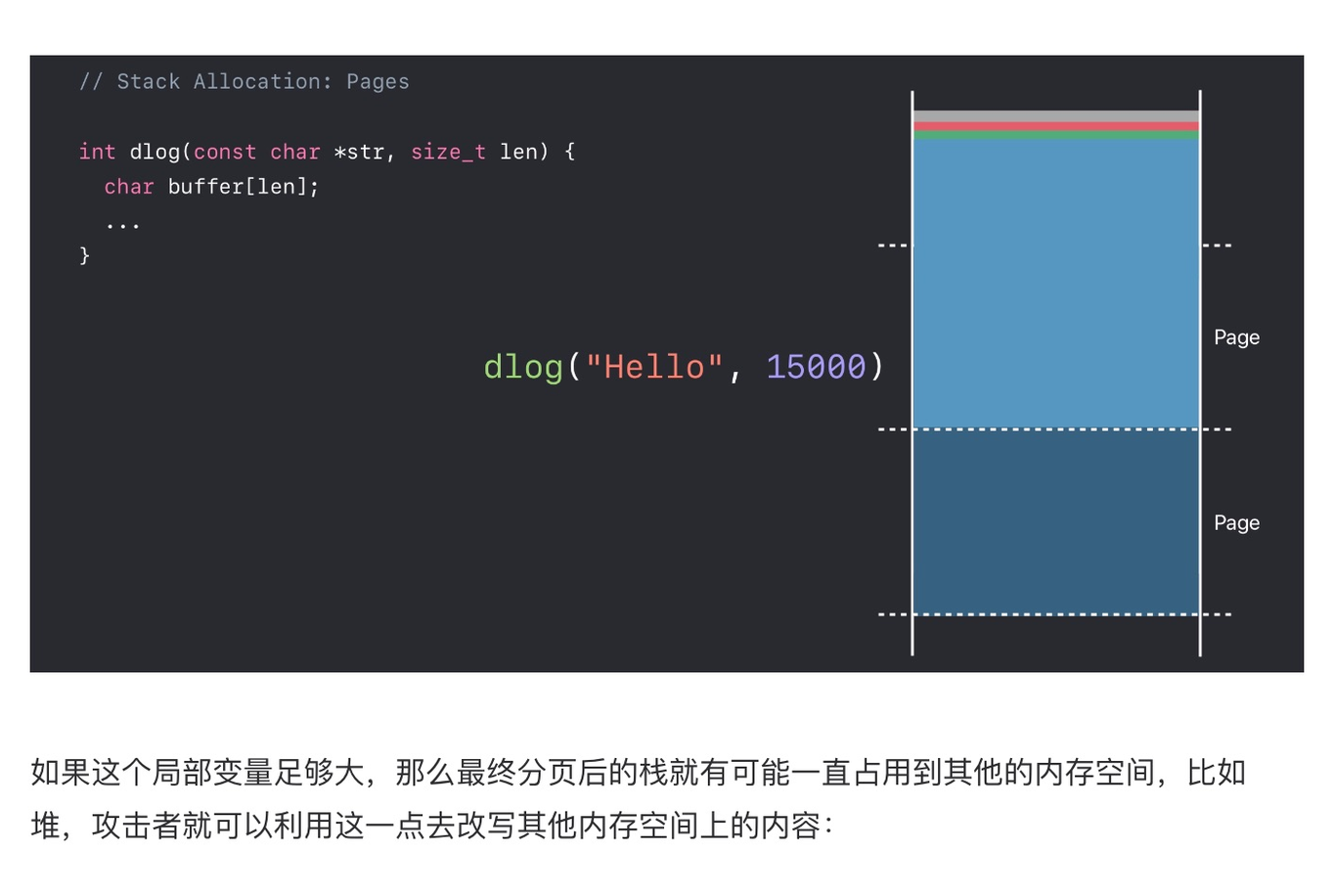
```
// Stack Buffer Overflows

int dlog(const char *str) {
  char buffer[4];
  ...
  strcpy(buffer, str);
  ...
  // if (canary is invalid)
  //   abort();
  return ...;
}
```

dlog("Hello")
Return Address
buffer[3] = 'l'
buffer[2] = 'l'
buffer[1] = 'e'
buffer[0] = 'H'

Stack Checking

上一部分的漏洞主要是攻击者利用 Stack Buffer Overflow 来进行攻击，而这一部分则是利用 Stack Clash 来进行攻击。该漏洞的影响可以参考 2017-06-20 的这篇信息安全新闻。同时 StackExchange 论坛也有关于此的讨论。

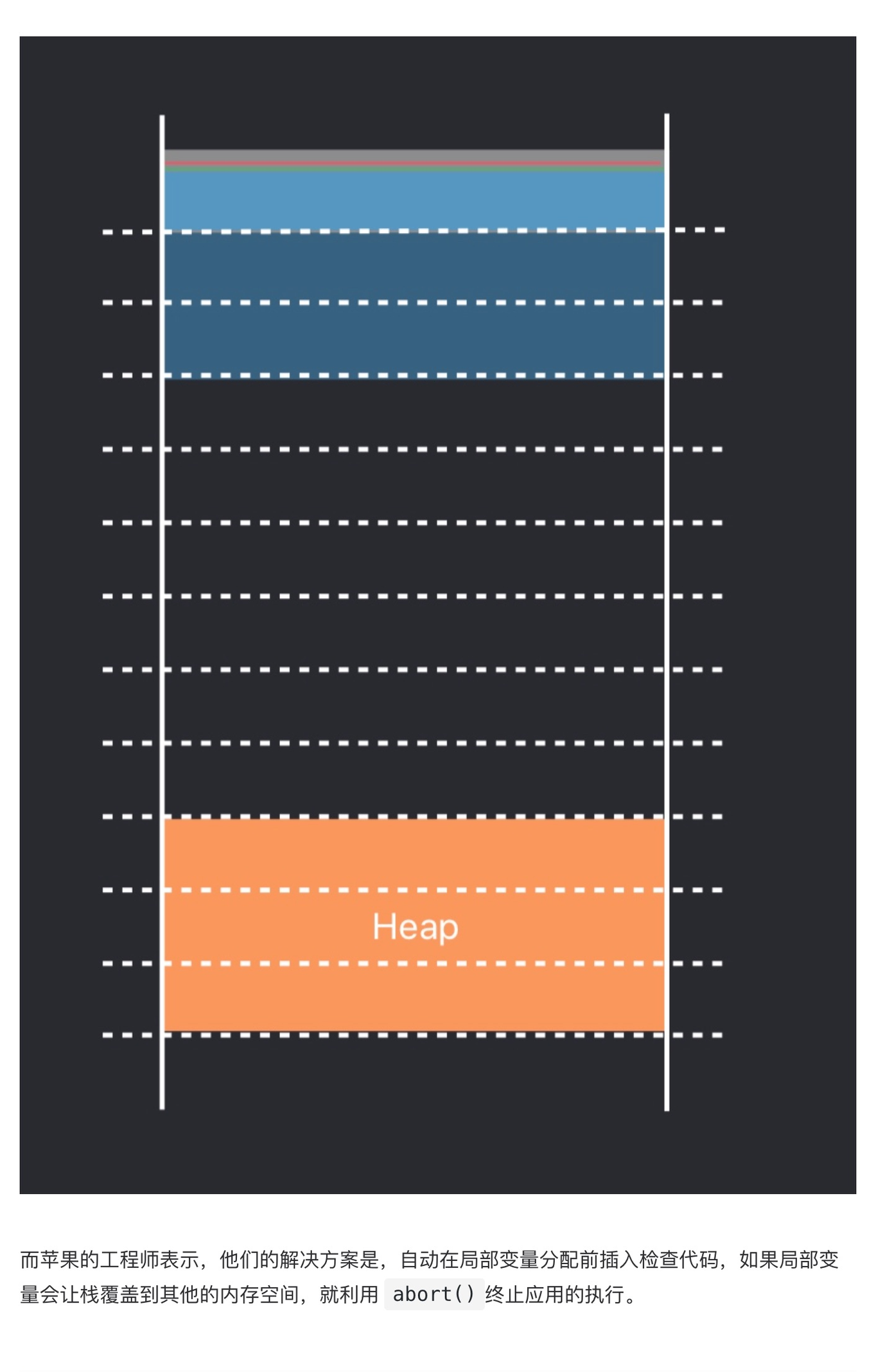
简单来说，假如我们在栈上为局部变量分配了过多的内存，操作系统就会将栈进行分页：



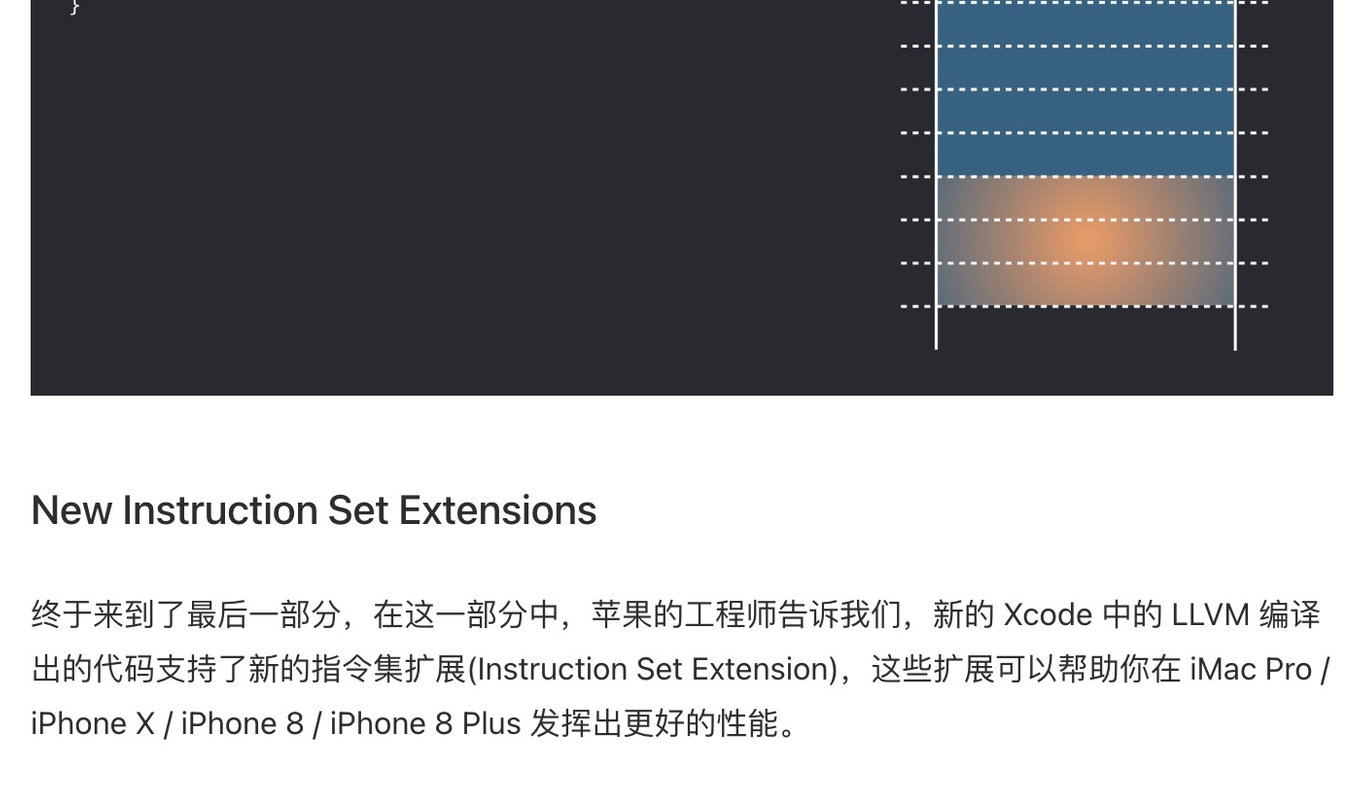
```
// Stack Allocation: Pages

int dlog(const char *str, size_t len) {
  char buffer[len];
  ...
}
```

dlog("Hello", 15000)
Page
Page

如果这个局部变量足够大，那么最终分页后的栈就有可能一直占用到其他的内存空间，比如堆，攻击者就可以利用这一点去改写其他内存空间上的内容：



而苹果的工程师表示，他们的解决方案是，自动在局部变量分配前插入检查代码，如果局部变量会让栈覆盖到其他的内存空间，就利用 `abort()` 终止应用的执行。



```
// Stack Checking

int dlog(const char *str, size_t len) {
  // if (buffer[len] not in stack)
  //   abort();
  char buffer[len];
  ...
}
```

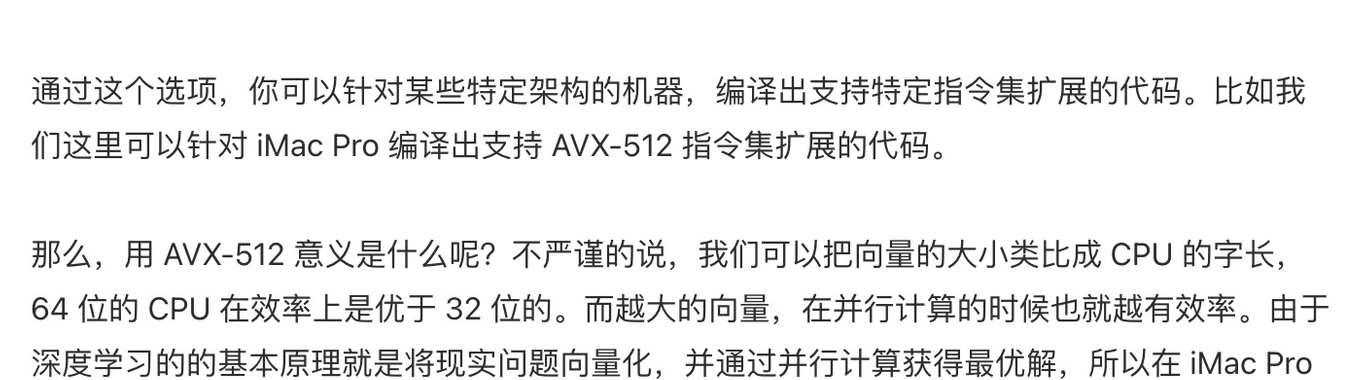
New Instruction Set Extensions

终于来到了最后一部分，在这一部分中，苹果的工程师告诉我们，新的 Xcode 中的 LLVM 编译出的代码支持了新的指令集扩展(Instruction Set Extension)，这些扩展可以帮助你在 iMac Pro / iPhone X / iPhone 8 / iPhone 8 Plus 发挥出更好的性能。

New Vector Extension: AVX-512

第一部分的扩展内容，是对 Intel 的 AVX-512 扩展的支持，这个扩展简单来说，就是在 Intel 的至强(Xeon)处理器上，我们可以使用最大 512 bit 的向量来提升计算密集型工作的效率。

在 Xcode 的 build settings 中， 有一个叫做 `Enable Additional Vector Extensions` 的选项，可能我们大部分人都不会注意到：



TestSwiftWithOC
General
Capabilities
Resource Tags
Info
Build Settings
Build Phases
Build Rules
PROJECT
TestSwiftWithOC
TARGETS
TestSwiftWithOC
Basic
Customized
All
Combined
Levels
vector
Apple Clang - Code Generation
Setting
TestSwiftWithOC
Quick Help
Summary
Declaration
Value Type

Enable Additional Vector Extensions: ✓ Platform default / SSE 3 / SSE 3 (with supplemental extensions) / SSE 4.1 / SSE 4.2 / AVX / AVX 2 / AVX 512

Summary: Enables the use of extended vector instructions. Only used when targeting Intel architectures.

Declaration: CLANG_X86_VECTOR_INSTRUCTIONS

Value Type: Enumeration (String)

通过这个选项，你可以针对某些特定架构的机器，编译出支持特定指令集扩展的代码。比如我们这里可以针对 iMac Pro 编译出支持 AVX-512 指令集扩展的代码。

那么，用 AVX-512 意义是什么呢？不严谨的说，我们可以把向量的大小类比成 CPU 的字长，64 位的 CPU 在效率上是优于 32 位的。而越大的向量，在并行计算的时候也就越有效率。由于深度学习的的基本原理就是将现实问题向量化，并通过并行计算获得最优解，所以在 iMac Pro 上，你可以利用这一点，让你的深度学习模型学习的更有效率。不过需要注意的是，这是对 CPU 的提升，现在的深度学习应该还是主要用 GPU 来做学习运算的，其具体意义，可以参考知