Understanding Undefined Behavior

Session 407

Fred Riss, Clang Team
Ryan Govostes, Security Engineering and Architecture Team
Anna Zaks, Program Analysis Team
What is undefined behavior?
The compiler and undefined behavior
Security implications
Tools can help
Swift is safer by default
What Is Undefined Behavior?
“undefined behavior: behavior for which this International Standard imposes no requirements.”

ISO C++14 Standard
What Can the Compiler Do with Undefined Behavior?
What Can the Compiler Do with Undefined Behavior?

Diagnose using warnings or errors
What Can the Compiler Do with Undefined Behavior?

Diagnose using warnings or errors

Act in a documented manner
What Can the Compiler Do with Undefined Behavior?

Diagnose using warnings or errors

Act in a documented manner

Produce unpredictable results
Signed integer overflow  Use of uninitialized values

Out-of-bounds array subscript  Data races  NULL dereference

Use of uninitialized values  C++ dynamic type violation

Division by 0  Shift amounts bigger than type  Misaligned access

Access to an object past end of lifetime  Missing return statement
Undefined Behavior Is About Tradeoffs

Performance over safety
Some Undefined Behavior Examples

Use of an uninitialized variable

```c
int uninitialized_variable(int arg) {
    int value;

    if (arg <= 0)
        value = 42;

    return arg + value;
}
```
Some Undefined Behavior Examples
Use of an uninitialized variable

```c
int uninitialized_variable(int arg) {
    int value;

    if (arg <= 0)
        value = 42;

    return arg + value;
}
```
Some Undefined Behavior Examples

Use of an uninitialized variable

```c
int uninitialized_variable(int arg) {
    int value;

    if (arg <= 0)
        value = 42;

    return arg + value;
}
```
Some Undefined Behavior Examples

Use of an uninitialized variable

```c
int uninitialized_variable(int arg) {
    int value;
    if (arg <= 0)
        value = 42;
    return arg + value;
}
```
Some Undefined Behavior Examples
Use of an uninitialized variable

```c
int uninitialized_variable(int arg) {
    int value;

    if (arg <= 0) {
        value = 42;

        return arg + value;
    }
}
```

⚠️ Variable value is used uninitialized whenever the ‘if’ condition is false

⚠️ Compiler warnings
Some Undefined Behavior Examples
Use of an uninitialized variable

```c
int uninitialized_variable(int arg) {
    int value;
    if (arg <= 0) {
        value = 42;
    }
    return arg + value;
}
```

Variable value is used uninitialized whenever the 'if' condition is false
Some Undefined Behavior Examples

Misaligned pointers

def serialize_misaligned(char *buffer, int a, int b) {
    *(int *)buffer = a;
    buffer += sizeof(a);
    *(int *)buffer = b;
    buffer += sizeof(b);
    return buffer;
}
Some Undefined Behavior Examples

Misaligned pointers

```c
char *serialize_misaligned(char *buffer, int a, int b) {
    *(int *)buffer = a;
    buffer += sizeof(a);
    *(int *)buffer = b;
    buffer += sizeof(b);
    return buffer;
}
```
Some Undefined Behavior Examples
Misaligned pointers

```c
char *serialize_misaligned(char *buffer, int a, int b) {
    *(int *)buffer = a;
    buffer += sizeof(a);
    *(int *)buffer = b;
    buffer += sizeof(b);
    return buffer;
}
```
Some Undefined Behavior Examples

Misaligned pointers

```c
char *serialize_misaligned(char *buffer, int a, int b) {
    *(int *)buffer = a;
    buffer += sizeof(a);
    *(int *)buffer = b;
    buffer += sizeof(b);
    return buffer;
}
```

```
char *buffer = ...;  
int a = ...;        
int b = ...;
```
Some Undefined Behavior Examples

Misaligned pointers

```c
char *serialize_misaligned(char *buffer, int a, int b) {
    *(int *)buffer = a;
    buffer += sizeof(a);
    *(int *)buffer = b;
    buffer += sizeof(b);
    return buffer;
}
```
Some Undefined Behavior Examples

Misaligned pointers

```c
char *serialize_misaligned(char *buffer, int a, int b) {
    *(int *)buffer = a;
    buffer += sizeof(a);
    *(int *)buffer = b;
    buffer += sizeof(b);
    return buffer;
}
```
Some Undefined Behavior Examples

Misaligned pointers

```c
char *serialize_misaligned(char *buffer, int a, int b) {
    *(int *)buffer = a;
    buffer += sizeof(a);
    *(int *)buffer = b;
    buffer += sizeof(b);
    return buffer;
}
```
Some Undefined Behavior Examples

Misaligned pointers

```c
char *serialize_misaligned(char *buffer, int a, int b) {
    *(int *)buffer = a;
    buffer += sizeof(a);
    *(int *)buffer = b;
    buffer += sizeof(b);
    return buffer;
}
```

Store of misaligned address 0x7fff5fbff642 for type 'int', which requires 4 byte alignment

Store of misaligned address 0x7fff5fbff646 for type 'int', which requires 4 byte alignment
Some Undefined Behavior Examples
Access to an object past end of lifetime

```c
int lifetime_issue(int *value) {
    if (value == NULL) {
        int default_value = 42;
        value = &default_value;
    }
    return *value;
}
```
Some Undefined Behavior Examples
Access to an object past end of lifetime

```c
int lifetime_issue(int *value) {
    if (value == NULL) {
        int default_value = 42;
        value = &default_value;
    }
    return *value;
}
```
Some Undefined Behavior Examples
Access to an object past end of lifetime

```c
int lifetime_issue(int *value) {
    if (value == NULL) {
        int default_value = 42;
        value = &default_value;
    }
    return *value;
}
```
Some Undefined Behavior Examples

Access to an object past end of lifetime

```c
int lifetime_issue(int *value) {
    if (value == NULL) {
        int default_value = 42;
        value = &default_value;
    }
    return *value;
}
```

Thread 1: Use of out of scope stack memory
The Compiler and Undefined Behavior
Undefined Behavior Provides Information
## Undefined Behavior Provides Information

<table>
<thead>
<tr>
<th>Undefined Behavior</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed integers cannot overflow</td>
<td>$x &lt; x+1$</td>
</tr>
</tbody>
</table>
### Undefined Behavior Provides Information

<table>
<thead>
<tr>
<th>Undefined Behavior</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed integers cannot overflow</td>
<td>$x &lt; x+1$</td>
</tr>
<tr>
<td>Pointers are naturally aligned</td>
<td>Can use vector instructions</td>
</tr>
<tr>
<td>Undefined Behavior</td>
<td>Information</td>
</tr>
<tr>
<td>---------------------------------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>Signed integers cannot overflow</td>
<td>$x &lt; x+1$</td>
</tr>
<tr>
<td>Pointers are naturally aligned</td>
<td>Can use vector instructions</td>
</tr>
<tr>
<td>NULL cannot be dereferenced</td>
<td>A dereferenced pointer cannot be NULL</td>
</tr>
</tbody>
</table>
### Undefined Behavior Provides Information

<table>
<thead>
<tr>
<th>Undefined Behavior</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed integers cannot overflow</td>
<td>$x &lt; x + 1$</td>
</tr>
<tr>
<td>Pointers are naturally aligned</td>
<td>Can use vector instructions</td>
</tr>
<tr>
<td>NULL cannot be dereferenced</td>
<td>A dereferenced pointer cannot be NULL</td>
</tr>
</tbody>
</table>
The Compiler Executes an Optimization Pipeline

- Source code (.c, .m, .cpp, .mm)
- Analyze
- Intermediate representation
- Optimize
- Object file (.o)
Dereferencing NULL Might Not Always Crash

Source code
.c, .m, .cpp, .mm → Dead Code Elimination → Object file
.o
Dereferencing NULL Might Not Always Crash

```c
int foo(int *P) {
    int var = *P;
    return 42;
}
```
Dereferencing NULL Might Not Always Crash

Source code .c, .m, .cpp, .mm

Dead Code Elimination

Object file .o

int foo(int *P) {
    int var = *P;
    return 42;
}

int foo(int *P) {
    
    return 42;
}
Dereferencing NULL Might Not Always Crash

```c
int foo(int *P) {
    int var = *P;
    return 42;
}
```

After dead code elimination:

```c
int foo(int *P) {
    return 42;
}
```
Let's Experiment: A Very Simple Optimization Pipeline

Compiler 1

Source code
.c, .m, .cpp, .mm

Redundant Null Check Elimination

Dead Code Elimination

Object file
.o

```c
void contains_null_check(int *P) {
    int unused = *P;
    ...
    if (P == NULL)
        return;
    *P = 4;
}
```
Let’s Experiment: A Very Simple Optimization Pipeline

**Compiler 1**

- **Source code** (.c, .m, .cpp, .mm)
- **Redundant Null Check Elimination**
- **Dead Code Elimination**
- **Object file** (.o)

```c
void contains_null_check(int *P) {
    int unused = *P;
    ...
    if (P == NULL)
        return;
    *P = 4;
}
```
Let’s Experiment: A Very Simple Optimization Pipeline

Compiler 1

Source code .c, .m, .cpp, .mm

Redundant Null Check Elimination

Dead Code Elimination

Object file .o

```c
void contains_null_check(int *P) {
    int unused = *P;
    ...
    if (P == NULL) return;
    *P = 4;
}
```

```c
void contains_null_check(int *P) {
    int unused = *P;
    ...
    if (P == NULL) return;
    *P = 4;
}
```
Let’s Experiment: A Very Simple Optimization Pipeline

Compiler 1

void contains_null_check(int *P) {
    int unused = *P;
    ...
    if (P == NULL)
        return;
    *P = 4;
}

Redundant Null Check Elimination

Dead Code Elimination

Object file .o
Let's Experiment: A Very Simple Optimization Pipeline

Compiler 1

Source code .c, .m, .cpp, .mm

Redundant Null Check Elimination

Dead Code Elimination

Object file .o

```c
void contains_null_check(int *P) {
    int unused = *P;
    ...

    *P = 4;
}
```
Let's Experiment: A Very Simple Optimization Pipeline

Compiler 1

Source code
.c, .m, .cpp, .mm

Redundant Null Check Elimination

Dead Code Elimination

Object file
.o

void contains_null_check(int *P) {
    int unused = *P;
    ...
    *P = 4;
}

void contains_null_check(int *P) {
    int unused = *P;
    ...
    *P = 4;
}
Let's Experiment: A Very Simple Optimization Pipeline

Compiler 1

Source code .c, .m, .cpp, .mm

Redundant Null Check Elimination

Dead Code Elimination

Object file .o

```c
void contains_null_check(int *P) {
    int unused = *P;
    ...
    *P = 4;
}
```

```c
void contains_null_check(int *P) {
    ...
    *P = 4;
}
```
Let's Experiment: A Very Simple Optimization Pipeline

Compiler 1

Source code
.c, .m, .cpp, .mm

Redundant Null Check Elimination

Dead Code Elimination

Object file
.o

```c
void contains_null_check(int *P) {
    int unused = *P;
    ...
    if (P == NULL)
        return;
    *P = 4;
}
```

```c
void contains_null_check(int *P) {
    ...
    *P = 4;
}
```
Let's Experiment: A Very Simple Optimization Pipeline

Compiler 1

void contains_null_check(int *P) {
    int unused = *P;
    ...
    if (P == NULL)
        return;
    *P = 4;
}

Compiler 1

void contains_null_check(int *P) {
    ...
    *P = 4;
}

Source code
.c, .m, .cpp, .mm

Redundant Null Check Elimination

Dead Code Elimination

Object file
.o
Let’s Experiment: A Very Simple Optimization Pipeline

Compiler 2

Source code
.c, .m, .cpp, .mm

Dead Code Elimination

Redundant Null Check Elimination

Object file
.o

```c
void contains_null_check(int *P) {
    int unused = *P;
    ...
    if (P == NULL)
        return;
    *P = 4;
}
```
Let’s Experiment: A Very Simple Optimization Pipeline

Compiler 2

Source code
.c, .m, .cpp, .mm

Dead Code Elimination

Redundant Null Check Elimination

Object file
.o

```c
void contains_null_check(int *P) {
    int unused = *P;
    ...
    if (P == NULL)
        return;
    *P = 4;
}
```
Let's Experiment: A Very Simple Optimization Pipeline

Compiler 2

Objective code

Redundant Null Check Elimination

Dead Code Elimination

void contains_null_check(int *P) {
    int unused = *P;
    ...
    if (P == NULL)
        return;
    *P = 4;
}

void contains_null_check(int *P) {
    int unused = *P;
    ...
    if (P == NULL)
        return;
    *P = 4;
}
Let’s Experiment: A Very Simple Optimization Pipeline

Compiler 2

Source code
.c, .m, .cpp, .mm

Dead Code Elimination

Redundant Null Check Elimination

Object file
.o

```c
void contains_null_check(int *P) {
    int unused = *P;
    ...
    if (P == NULL)
        return;
    *P = 4;
}
```

```c
void contains_null_check(int *P) {
    ... if (P == NULL)
        return;
    *P = 4;
}
```
Let's Experiment: A Very Simple Optimization Pipeline

Compiler 2

Source code
.c, .m, .cpp, .mm

Dead Code Elimination

Redundant Null Check Elimination

Object file
.o

void contains_null_check(int *P) {
    ...
    if (P == NULL)
        return;
    *P = 4;
}
Let's Experiment: A Very Simple Optimization Pipeline

Compiler 2

void contains_null_check(int *P) {
    if (P == NULL)
        return;
    *P = 4;
}

Redundant Null Check Elimination
Let's Experiment: A Very Simple Optimization Pipeline

Compiler 2

Source code
.c, .m, .cpp, .mm

Dead Code Elimination

Redundant Null Check Elimination

Object file
.o

void contains_null_check(int *P) {
    int unused = *P;
    ...
    if (P == NULL)
        return;
    *P = 4;
}

void contains_null_check(int *P) {
    int unused = *P;
    ...
    if (P == NULL)
        return;
    *P = 4;
}
Let's Experiment: A Very Simple Optimization Pipeline

Compiler 2

Source code
.c, .m, .cpp, .mm

Dead Code Elimination

Redundant Null Check Elimination

Object file
.o

```c
void contains_null_check(int *P) {
    int unused = *P;
    ...
    if (P == NULL)
        return;
    *P = 4;
}
```

```c
void contains_null_check(int *P) {
    ...
    if (P == NULL)
        return;
    *P = 4;
}
```
Let’s Experiment: A Very Simple Optimization Pipeline

A surprising result

```c
void contains_null_check(int *P) {
    int unused = *P;
    ...
    if (P == NULL)
        return;
    *P = 4;
}
```

Compiler 2

```c
void contains_null_check(int *P) {
    ...
    if (P == NULL)
        return;
    *P = 4;
}
```
Let’s Experiment: A Very Simple Optimization Pipeline
A surprising result

```c
void contains_null_check(int *P) {
    int unused = *P;
    ...
    if (P == NULL)
        return;
    *P = 4;
}
```

**Compiler 1**

```c
void contains_null_check(int *P) {
    ...
    *P = 4;
}
```

**Compiler 2**

```c
void contains_null_check(int *P) {
    int unused = *P;
    ...
    if (P == NULL)
        return;
    *P = 4;
}
```
Compiler Behavior Changes More Often Than You Think
### Compiler Behavior Changes More Often Than You Think

- **Debug**
  - **Release**
- **Fastest [-O3]**
  - Fastest, Smallest [-Os]
  - Fastest, Aggressive Optimizations [-Ofast]
- **None [-O0]**
- **Fast [-O, O1]**
- **Faster [-O2]**
Compiler Behavior Changes More Often Than You Think
Compiler Behavior Changes More Often Than You Think

- Debug
- Release
- None [-O0]
- Fast [-O, -O1]
- Faster [-O2]
- Fastest [-O3]
- Fastest, Smallest [-Os]
- Fastest, Aggressive Optimizations [-Ofast]

Device
- Frédéric's iPhone

Build Only Device
- Generic iOS Device

iOS Simulators
- iPhone 7
- iPhone 7 Plus
- iPhone SE

Add Additional Simulators...
Issues with Undefined Behavior
Issues with Undefined Behavior

Undefined behavior is unpredictable
Issues with Undefined Behavior

Undefined behavior is unpredictable

Consequences can affect the whole program
Issues with Undefined Behavior

Undefined behavior is unpredictable
Consequences can affect the whole program
Bugs may be dormant
Security Implications of Undefined Behavior
Private Keys
Passwords
Application State

E-mails
Business Documents
Photos
Undefined behavior is at the heart of many security vulnerabilities.
Examples of Security Vulnerabilities
Examples of Security Vulnerabilities

Buffer overflow
Examples of Security Vulnerabilities

Buffer overflow

Use of uninitialized variable
Examples of Security Vulnerabilities

- Buffer overflow
- Use of uninitialized variable
- Use-after-free
Examples of Security Vulnerabilities

Buffer overflow
Use of uninitialized variable
Use-after-free
Double free
Examples of Security Vulnerabilities

- Buffer overflow
- Use of uninitialized variable
- Use-after-free
- Double free
- Race condition
Defend Your Users

Build secure apps

Protect your reputation

Framework bugs are inherited
Tools Can Help
How Address Sanitizer “Saved” macOS Yosemite
CFString
CFString

/Users/tim /Library/Caches /com.apple.hypercard /startup.db
<table>
<thead>
<tr>
<th>CFString</th>
<th>Character Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>/Users/tim</td>
<td></td>
</tr>
<tr>
<td>/Library/Caches</td>
<td></td>
</tr>
<tr>
<td>/com.apple.hypercard</td>
<td></td>
</tr>
<tr>
<td>/startup.db</td>
<td></td>
</tr>
</tbody>
</table>
CFString

/Users/tim
/Library/Caches
/com.apple.hypercard
/startup.db

Character Buffer

/Users/tim/Library/Caches/com.apple.hypercard/startup.db
CFString

/Users/tim  /Library/Caches  /com.apple.hypercard  /startup.db

Character Buffer

/Users/tim/Library/Caches/com.apple.hypercard/startup.db/NUL
CFString

/Users/jappleseed4
/Library/Caches
/com.apple.hypercard
/startup.db

Character Buffer

/Users/jappleseed4
/Library/Caches
/com.apple.hypercard
/startup.db
/NUL
CFString

/Users/jappleseed4 /Library/Caches /com.apple.hypercard /startup.db

Character Buffer

/ U s e r s / j a p p l e s e e d e e e e d 4 / L i b r a r y / C a c h e s / c o m . a p p l e . h y p e r c a r d / s t a r t u p . d b N U L
Tools for Addressing Undefined Behavior
Tools for Addressing Undefined Behavior

Compiler
Tools for Addressing Undefined Behavior

Compiler
Static Analyzer
Tools for Addressing Undefined Behavior

Compiler

Static Analyzer

Address Sanitizer
Tools for Addressing Undefined Behavior

- Compiler
- Static Analyzer
- Address Sanitizer
- Thread Sanitizer
Tools for Addressing Undefined Behavior

Compiler

Static Analyzer

Address Sanitizer

Thread Sanitizer

Undefined Behavior Sanitizer
Trust the Compiler

Pay attention to compiler warnings

Every release of Xcode has better warnings

Modernize your project (Editor → Validate Settings)
Trust the Compiler

Pay attention to compiler warnings

Every release of Xcode has better warnings

Modernize your project (Editor → Validate Settings)

What’s New in LLVM

Hall 2

Thursday 4:10PM
Run the Static Analyzer

Explores your code

Analyze during every build

Analyze in Continuous Integration
Run the Static Analyzer

Explores your code

Analyze during every build

Analyze in Continuous Integration
Use the Runtime Sanitizers
Use the Runtime Sanitizers

<table>
<thead>
<tr>
<th>Tool</th>
<th>Undefined Behavior</th>
</tr>
</thead>
</table>

# Use the Runtime Sanitizers

<table>
<thead>
<tr>
<th>Tool</th>
<th>Undefined Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Sanitizer</td>
<td>buffer overflow, use-after-free, double free, use after end of scope</td>
</tr>
</tbody>
</table>
Use the Runtime Sanitizers

<table>
<thead>
<tr>
<th>Tool</th>
<th>Undefined Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Sanitizer</td>
<td>buffer overflow, use-after-free, double free, use after end of scope</td>
</tr>
<tr>
<td>Thread Sanitizer</td>
<td>data race</td>
</tr>
</tbody>
</table>
## Use the Runtime Sanitizers

<table>
<thead>
<tr>
<th>Tool</th>
<th>Undefined Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Sanitizer</td>
<td>buffer overflow, use-after-free, double free, use after end of scope</td>
</tr>
<tr>
<td>Thread Sanitizer</td>
<td>data race</td>
</tr>
</tbody>
</table>
| Undefined Behavior Sanitizer        | misaligned pointer, null pointer dereference, integer overflow, type mismatch, and more | **NEW**
- (void)touchDownAtPoint:(CGPoint)pos {
    dispatch_queue_t q = dispatch_queue_create("com.example.queue",
                                             DISPATCH_QUEUE_CONCURRENT);

    // Thread 2
    dispatch_async(q, ^{
        [d setObject:@42 forKey:@"answer" ];
    });

    SKShapeNode *n = [_spinnnyNode copy];
    n.position = pos;
    n.strokeColor = [SKColor greenColor];
    int fastWidth = [self getFastWidth];
    int fastHeight = [self getFastHeight];
    if (_expandWidth == YES)
        n.lineWidth = fastWidth * fastHeight / 10000;
    [self addChild:n];
}
ExampleGame - 36620 issues

Threading Issues
- Data race in -[GameScene sceneDidLoad] at d
  - 'd' is a global variable (0x100008a70)
- Write of size 8 by thread 1
  - 0 -[GameScene sceneDidLoad]
  - 1 -[SKScene initWithCoder:]
  - 3 start
- Read of size 8 by thread 5
  - 0 __-25-[GameScene sceneDidLoad]
  - _tsan::invoke_and_release...
  - _dispatch_client_callout

Undefined Behavior
- Signed integer overflow: 10240000 * 768000 cannot be represented in type 'int'

Signed integer overflow
- 0 -[GameScene touchDown...]
- 1 -[GameScene mouseDown...]
- 2 -[SKView mouseDown:]
- 8 UIApplicationMain
- 9 main

```
touchDownAtPoint:(CGPoint)pos {
    dispatch_queue_t q = dispatch_queue_create("com.example.queue",
        DISPATCH_QUEUE_CONCURRENT);
    read 2
    dispatch_async(q, ^{
        setObject:@42 forKey:@"answer";
    });

    SpinnyNode *n = _spinneyNode copy;
    position = pos;
    strokeColor = [SKColor greenColor];
    fastWidth = [self getFastWidth];
    fastHeight = [self getFastHeight];
    expandWidth == YES
    lineWidth = fastWidth * fastHeight / 10000;
    addChild:n;
```
Turn on Sanitizers
Edit scheme – diagnostics tab

- Runtime Sanitization
- Address Sanitizer
- Detect use of stack after return
- Undefined Behavior Sanitizer
- Pause on issues
- Thread Sanitizer
- Pause on issues
Turn on Sanitizers
Edit scheme – diagnostics tab

Finding Bugs Using Xcode Runtime Tools
Tools for Addressing Undefined Behavior

Compiler
Static Analyzer
Address Sanitizer
Thread Sanitizer
Undefined Behavior Sanitizer
Tools for Addressing Undefined Behavior

Compiler
Static Analyzer
Address Sanitizer
Thread Sanitizer
Undefined Behavior Sanitizer
Language
Use Safe Language Features

Prefer safe constructs

• Automatic Reference Counting
• C++ smart pointers (`std::shared_ptr`, `std::unique_ptr`)
• Bounds-checked containers (NSArray)
Use Safe Language Features

Prefer safe constructs

• Automatic Reference Counting
• C++ smart pointers (\texttt{std::shared\_ptr}, \texttt{std::unique\_ptr})
• Bounds-checked containers (NSArray)

Consider using Swift
Swift Is Safer by Default

Anna Zaks, Program Analysis Team
“Undefined behavior is the enemy of safety”
Safety Enforced on Many Levels
<table>
<thead>
<tr>
<th>C Language Family</th>
<th>Swift</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null pointer dereferences</td>
<td>Stricter type system - Optionals</td>
</tr>
<tr>
<td>Use of uninitialized variables</td>
<td>Definite initialization</td>
</tr>
<tr>
<td>Buffer and integer overflows</td>
<td>Runtime checks</td>
</tr>
<tr>
<td>Use-after-free</td>
<td>ARC (Automatic Reference Counting)</td>
</tr>
</tbody>
</table>
Optionals

Answer to NULL pointer dereferences
Optionals
Answer to NULL pointer dereferences

Non-optional and optional are different kinds of types
Optionals

Answer to NULL pointer dereferences

Need to check before using

```swift
func receivePackage() -> Cake?
...

guard let cake = receivePackage() else {
  // The cake is a lie.
  return
}
println("Jump with joy! Eat \\
(cake.kind)!")
```
Optionals
Do not abuse forced unwrap
**Optionals**

Do not abuse forced unwrap

Optional return type, means the API can return `nil`

```swift
cake = receivePackage()!
```
Optionals
Do not abuse forced unwrap

Optional return type, means the API can return `nil`

```
cake = receivePackage()!
```

Use forced unwrap only if:
• You can guarantee the value is never `nil`
• Cannot encode this in the type system
• For example: loading an image asset from the app bundle
Optionals

Implicitly-unwrapped optional (Cake!)
Optionals

Implicitly-unwrapped optional (Cake!)

Compiler does not enforce that value is checked before use
Optionals

Implicitly-unwrapped optional (Cake!)

Compiler does not enforce that value is checked before use

Safer than a pointer type in C

• Defined behavior

• Guaranteed to stop on nil
Optionals

Implicitly-unwrapped optional (Cake!)

Compiler does not enforce that value is checked before use

Safer than a pointer type in C
  • Defined behavior
  • Guaranteed to stop on nil

Useful for delayed initialization
Optionals

Implicitly-unwrapped optional (Cake!)

Compiler does not enforce that value is checked before use

Safer than a pointer type in C
  • Defined behavior
  • Guaranteed to stop on nil

Useful for delayed initialization

May come from Objective-C APIs
Optionals
Nullability annotations for safer ecosystem
Optionals
Nullability annotations for safer ecosystem

Nullability in C languages affects Swift interfaces

- (nullable NSView *)ancestorSharedWithView:(nonnull NSView *)aView; // Objective-C

func ancestorShared(with view: NSView) -> NSView? // Swift
Optionals
Nullability annotations for safer ecosystem

Nullability in C languages affects Swift interfaces

- (nullable NSView *)ancestorSharedWithView:(nonnull NSView *)aView; // Objective-C

func ancestorShared(with view: NSView) -> NSView? // Swift
Optionals
Nullability annotations for safer ecosystem

Nullability in C languages affects Swift interfaces

- *(nullable NSView *)ancestorSharedWithView:(nonnull NSView *)aView; // Objective-C

func ancestorShared(with view: NSView) -> NSView? // Swift
Optionals
Nullability annotations for safer ecosystem

Nullability in C languages affects Swift interfaces

- `(nullable NSView *)ancestorSharedWithView:(nonnull NSView *)aView;` // Objective-C

```swift
func ancestorShared(with view: NSView) -> NSView? // Swift
```
Optionals
Nullability annotations for safer ecosystem

Nullability in C languages affects Swift interfaces

- (nullable NSView *)ancestorSharedWithView:(nonnull NSView *)aView; // Objective-C

func ancestorShared(with view: NSView) -> NSView? // Swift

Apple APIs are annotated for nullability
Optionals
Nullability annotations for safer ecosystem

Nullability in C languages affects Swift interfaces

- (nullable NSView *)ancestorSharedWithView:(nonnull NSView *)aView; // Objective-C

func ancestorShared(with view: NSView) -> NSView? // Swift

Apple APIs are annotated for nullability

Use nullability on your Objective-C code!
Optionals
Nullability annotations for safer ecosystem

Nullability in C languages affects Swift interfaces

- (nullable NSView *)ancestorSharedWithView:(nonnull NSView *)aView; // Objective-C

func ancestorShared(with view: NSView) -> NSView? // Swift

Apple APIs are annotated for nullability

Use nullability on your Objective-C code!

Find nullability inconsistencies with tools

• Static Analyzer, -Wnullability, Undefined Behavior Sanitizer
Definite Initialization

Answer to use of uninitialized variables
Definite Initialization
Answer to use of uninitialized variables

Checks that all values are initialized before use
Definite Initialization
Answer to use of uninitialized variables

Checks that all values are initialized before use

```swift
var myInstance: MyClass

if x > 42 {
    myInstance = MyClass(intValue: 13)
} else {
    myInstance = MyClass(floatValue: 92.3)
}

// myInstance has been initialized on all branches leading here!
myInstance.printIt()
```
Definite Initialization
Answer to use of uninitialized variables

Checks that all values are initialized before use

var myInstance: MyClass

if x > 42 {
    myInstance = MyClass(intValue: 13)
} else {
    myInstance = MyClass(floatValue: 92.3)
}

// myInstance has been initialized on all branches leading here!

myInstance.printIt()
Checks that all values are initialized before use

```swift
var myInstance: MyClass

if x > 42 {
    myInstance = MyClass(intValue: 13)
} else {
    myInstance = MyClass(floatValue: 92.3)
}

// myInstance has been initialized on all branches leading here!
myInstance.printIt()
```
Definite Initialization
Answer to use of uninitialized variables

Checks that all values are initialized before use

```swift
var myInstance: MyClass

if x > 42 {
    myInstance = MyClass(intValue: 13)
} else {
    myInstance = MyClass(floatValue: 92.3)
}

// myInstance has been initialized on all branches leading here!
myInstance.printIt()
```
**Definite Initialization**

Answer to use of uninitialized variables

Checks that all values are initialized before use

```swift
var myInstance: MyClass

if x > 42 {
    myInstance = MyClass(intValue: 13)
} else {
    myInstance = MyClass(floatValue: 92.3)
}

// myInstance has been initialized on all branches leading here!
myInstance.printIt()
```
Runtime Checks
Answer to buffer and integer overflows
Runtime Checks
Answer to buffer and integer overflows

Execution ends on **Array** and **Int** overflows
Runtime Checks
Answer to buffer and integer overflows

Execution ends on *Array* and *Int* overflows

Runtime checking is better than undefined behavior
  • Predictable
  • Provides security guarantees
Runtime Checks
Answer to buffer and integer overflows

Execution ends on Array and Int overflows

Runtime checking is better than undefined behavior
• Predictable
• Provides security guarantees

Integer wrapping behavior with &+, &−, &∗
Does undefined behavior exist in Swift?
Unsafe Types
Unsafe Types

Need interoperability with C APIs
Unsafe Types

Need interoperability with C APIs

UnsafePointer<T>, UnsafeMutableRawBufferPointer
Unsafe Types

Need interoperability with C APIs

UnsafePointer<T>, UnsafeMutableRawBufferPointer
Unsafe Types

Need interoperability with C APIs

UnsafePointer<T>, UnsafeMutableRawBufferPointer

Use Address Sanitizer
Exclusive Memory Accesses
Enforcement in Swift 4
func have(_ x: inout Cake, andEat y: inout Cake)

Exclusive Memory Accesses
Enforcement in Swift 4
Exclusive Memory Accesses
Enforcement in Swift 4

```swift
func have(_ x: inout Cake, andEat y: inout Cake)

have(&cake, andEat: &cake)
```

Triple Chocolate Delight

x

y
Exclusive Memory Accesses
Enforcement in Swift 4

```swift
func have(_ x: inout Cake, andEat y: inout Cake)

have(&cake, andEat: &cake)
```

Similar to `restrict` in C but with different default behavior.
exclusive memory accesses

func have(_ x: inout Cake, andEat y: inout Cake)

have(&cake, andEat: &cake)

Triple Chocolate Delight

Similar to restrict in C but with different default behavior
Enforcement of Exclusive Memory Accesses

Options considered

Declare to be undefined behavior (like C)

Provide language guarantees
Enforcement of Exclusive Memory Accesses
Intricate balancing act
Enforcement of Exclusive Memory Accesses

Proposed solution

Enforce a slightly stricter language rule

Enforcement at compile time

Enforcement at run time

Guarantee exclusive access within a thread
Enforcement of Exclusive Access Across Threads

Too expensive to check by default

Access races can lead to memory corruption in Swift

Thread Sanitizer catches most violations
Enforcement of Exclusive Access Across Threads

Too expensive to check by default

Access races can lead to memory corruption in Swift

Thread Sanitizer catches most violations
Summary

C languages rely on undefined behavior
Leads to unpredictability and security issues
Swift is safer by default
Use tools to make your code safe and reliable
More Information

<table>
<thead>
<tr>
<th>Session Title</th>
<th>Hall</th>
<th>Date/Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finding Bugs Using Xcode Runtime Tools</td>
<td></td>
<td>WWDC17</td>
</tr>
<tr>
<td>What's New in Swift</td>
<td></td>
<td>WWDC17</td>
</tr>
<tr>
<td><strong>What's New in LLVM</strong></td>
<td>Hall 2</td>
<td>Thursday 4:10PM</td>
</tr>
<tr>
<td>Labs</td>
<td>Technology Lab</td>
<td>Time</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>----------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Performance Profiling and Runtime Analysis Tools Lab</td>
<td>Lab K</td>
<td>Thur 1:00PM – 4:10PM</td>
</tr>
<tr>
<td>LLVM Compiler, Objective-C, and C++ Lab</td>
<td>Lab E</td>
<td>Fri 9:00AM – 11:00AM</td>
</tr>
</tbody>
</table>