ISSN 2277-3061
Volume 15 Number 8
International Journal of Computer and Technology

SECURITY MECHANISMS AND ANALYSIS FOR INSECURE DATA STORAGE AND UNINTENDED DATA LEAKAGE FOR MOBILE APPLICATIONS

Vanessa M. Santana, Paolina Centonze
Department of Computer Science, Iona College, 715 North Ave, New Rochelle NY 10801
vsantana2@gaels.iona.edu
Department of Computer Science, Iona College, 715 North Ave, New Rochelle NY 10801
pcentonze@iona.edu

ABSTRACT

Using one mobile programming language like Objective-C, Swift or Java is challenging enough because of the many things that need to be considered from a security point of view, like the programming language secure guidelines and vulnerabilities. With the introduction of Swift in 2014 it's now possible to build Swift/Objective-C mobile applications. Building a mobile application using two languages also adds a greater attack surface for hackers because of the need for developers to stay up to date on vulnerabilities on more than one language and operating system.

To our best knowledge, since as of today, there is no academic-research based effort comparing Swift, Objective-C and Android from a programming language and platform security point of view. Our comparative analysis covers a subset of OWASP top ten mobile vulnerabilities and seeing how Swift, Objective-C and Android programming languages safeguard against these risks and how the built-in platform security mechanisms for Android and Apple for the chosen subset of OWASP vulnerabilities compare when placed side-by-side.

Indexing terms/Keywords

OWASP, Mobile, Security, Vulnerabilities, Coding Guidelines, Swift, Objective-C, Java, Android, iOS.

Academic Discipline And Sub-Disciplines

Computer Science, Cyber security, Mobile Application Development, Mobile Application Security, Mobile Application Code Hardening;

SUBJECT CLASSIFICATION

Computer Science Subject Classification;

TYPE (METHOD/APPROACH)

Pragmatic (data triangulation, investigator triangulation); Deductive (Testing, Confirmation/Rejection) [1][2]

INTRODUCTION

Apple's introduction of Swift in 2014 into the mobile application programming language arena brought many programming opportunities for developers as well as for malicious attackers. The introduction of a language that claimed to be safe leaves the question as to how safe did Apple really make Swift in comparison to Objective-C and Java for Android. We focus on OWASP's 2014 Top Ten Mobile Vulnerability [3] Second, Insecure Data Storage, and Fourth, Unintended Data Leakage, and look at what Apple and Android provide on their developer websites as guidelines/aids they claim defend against attacks that fall within the two vulnerabilities. Moreover, we also look at what built-in security mechanisms the programming language or platform provides against attacks within these two vulnerabilities, and test both the built-in mechanisms as well as code in Swift, Objective-C and Java (for Android) that uses said guidelines to provide a comparative analysis of these three languages with respect to the two mentioned vulnerabilities.

MOBILE ANALYSIS TESTING TOOLS

To successfully test the secure guidelines posted by Apple and Google Android on their developer websites and built-in security mechanisms built into the programming language, we use open-source static analysis tools and dynamic analysis to confirm the static analysis results. Static analysis is used in this study to ensure that no device is compromised because the code and behavior can be examined without executing the program being analyzed. For Swift, we use Xcode's built-in and Tailor static analyzers [5][6]. The static analyzer built into Xcode parses project source code and identifies Logic flaws (accessing uninitialized variables and dereferencing null pointers), Memory management flaws (leaking allocated memory), Dead store flaws (unused variable), and API-usage flaws (not following the policies required by the frameworks and libraries the project is using)[5]. The open-source static analyzer Tailor focuses on enforcing style guidelines outlined in the Swift Programming Language, GitHub, Ray Wenderlich, and Coursera [6]. For Objective-C, we use static analyzer Clang [7]. Clang checks for logical errors for function calls, uninitialized arguments, null function pointers, division by zero, null pointers passed as arguments to a function whose arguments are marked with the nonnull attribute, dereferences of null pointers, sending retain, release, or autorelease directly to a class, incompatible type signature when overriding an Objective-C method, methods that lack a necessary call to super, suboptimal uses of NSAutoreleasePool in Objective-C GC mode, usage of NSError** parameters, prohibited nil arguments in specific Objective-C method calls (caseInsensitiveCompare, compare, compare:options, compare:options:range, compare:options:range:locale,

componentsSeparatedByCharactersInSet, initWithFormat), leaks and violations of the Cocoa Memory Management rules, self is properly initialized inside an initializer method, private ivars that are never used, passing non-Objective-C types to variadic collection initialization methods that expect only Objective-C types, and mismatched deallocators. For Android-Java, we use the Lint static analyzer [8]. Lint helps look at Java Android project source file code for potential bugs and optimization improvements for correctness, security, performance, usability, accessibility, and internationalization [8]. Needless to say, based on the issues these tools are able to identify is what will limit the extent to which this study will confirm or deny whether the secure guidelines and built-in security mechanisms defend against Insecure Data Storage and Unintended Data Leakage attacks.

MOBILE VULNERABILITIES

Unintended Data Leakage – OWASP Mobile Top Two

Unintended Data Leakage is a vulnerability exploitable by anyone who has acquired a lost or stolen mobile device, malware, or any repackaged application executing on the device on an attacker's behalf. If an attacker has gained physical access to a device, using open source tools, an attacker may be able to access third party application directories that may contain stored sensitive data, or an attacker may construct malware or modify a legitimate application to steal sensitive data. Development teams or organizations that assume users or malware will not have access to the mobile device's file system and data storage on the device are at the highest risks and may also forget that devices lose any encryption protections once rooted or jail broken.

Threat-modeling an application is the best way to understand how it processes data and how underlying API's handle said data. Places where OWASP has often seen insecure data stored includes SQLite databases, Log Files, Plist Files, XML Data Stores or Manifest Files, Binary data stores, Cookie stores and SD Cards [21].

Insecure Data Storage - OWASP Mobile Top Four

Insecure Data Storage is a vulnerability exploitable by mobile malware, modified versions of legitimate apps, or an attacker that has physical access to the victim's mobile device. With physical access to the mobile device, an attacker can use open source tools to use permissible and documented API calls to conduct this attack via malicious code. This vulnerability when a developer unknowingly places sensitive information in a location on the mobile device that is easily accessible by other apps on the device. Keeping up to date on the technologies and programming languages used is very important as a developer since a developer that does not have intimate knowledge of how information is stored or processed by the underlying OS can lead to this vulnerability. Inspecting all mobile device locations accessible to other applications for stored sensitive information can help detect data leakage.

Unintended data leakage may result from vulnerabilities in the underlying OS, frameworks, compiler environment, or new hardware. Undocumented or under-documented processes OWASP has most often seen this vulnerability includes in the way OS, platforms, and frameworks handle URL caching request and response, keyboard press caching, copy/paste buffer caching, application backgrounding, logging, HTML5 data storage, browser cookie objects, and analytics data sent to 3rd parties [22].

SECURITY STUDY – iOS

iOS executables targeted towards versions 4.3 and later have the necessary flags for Address Space Layout Randomization and Position Independent Executable Flags enabled by default, and earlier versions of iOS can add this security measure by adding the Position Independent Flag. Address Space Layout Randomization (ASLR) along with the compiler's Position Independent Executable adds a layer of protection towards buffer overflow attacks. Address Space Layout Randomization randomizes where data and code is mapped in a processes address space to defend against attacks that target specific addresses in a process's layout. In applications compiled with Position Independent Execution (PIE), the application's memory regions are randomized and the binary is loaded at a random address for each execution [18].

iOS adds another layer of protection towards buffer overflow attacks by employing Non-Executable Stack and Heaps by making use of the processors NX bit feature, and marking the stack and heap as non-executables. Memory pages cannot be marked as both executables and writeable at any time and, once a memory page is marked from an executable to writeable, it later cannot be marked back as executable. Any attack that places executable code on the stack or heap and tries to run will fail as a result. Return Orientated Programming (ROP) based payloads can bypass the non-executable memory security feature, which is why non-executable memory is used in combination with ASLR to increase exploit complexity [18].

Unintended Data Leakage

Heap Overflow (String Handling)

Because many string handling functions have no built-in checks for string length, strings are prone to buffer overflow attacks [9]. Apple warns about using strcpy in Objective-C, since it overwrites anything afterwards:

ISSN 2277-3061
Volume 15 Number 8
International Journal of Computer and Technology

```
Char destination[5]; char *source = "LARGER";

strcpy(destination, source);
```

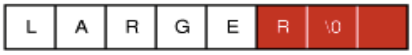


Fig. 1 Objective-C Unsafe String Handling

The built-in analyzer for Xcode successfully catches that buffer overflow and terminates the application:

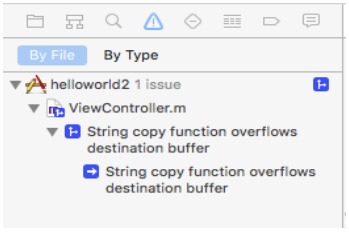


Fig. 2 Xcode Analyzer Buffer Overflow (By File)

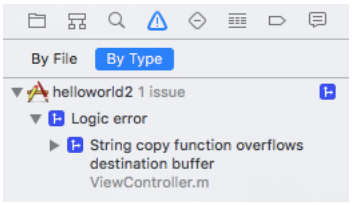


Fig. 3 Xcode Analyzer Buffer Overflow (By Error Type)

```
23 - (void)viewDidLoad {
24     [super viewDidLoad];
25     // Do any additional setup after loading the view, typically from a nib.
26     char destination[5];
27     char *source = "LARGER";
28
29     strcpy(destination, source);
30     //strncpy(destination, source, sizeof(destination));
31     //strlcpy(destination, source, sizeof(destination));
32 }
```

Fig. 4 Xcode Analyzer Error Line

The string-handling function Apple recommends is safe is strlcpy, because it truncates the string to one byte smaller than the buffer size while also adding the terminating null character:

```
strlcpy(destination, source, sizeof(destination));
```

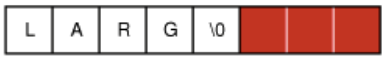


Fig. 5 Objective-C Correct String Handling

The built-in Xcode analyzer does not detect any buffer overflow issues when the app uses this string-handling function and builds successfully.

In Swift, to copy a string, an assignment would suffice:

var src = "Hello World"

var dst = src

Because the same string function (strcpy) in Swift would not even allow Memory to be accessed it does not allow the application to build as in figure 6 below:

```
17
18      var str = "Hello, World"
19      let dst = UnsafeMutablePointer.alloc(11)
20
21      strcpy(dst, str)
22      let ss = String.fromCString(dst)
23      print("\(ss)")                    // "Hello, World"
24
```

Fig. 6 Swift Compiler Error Line

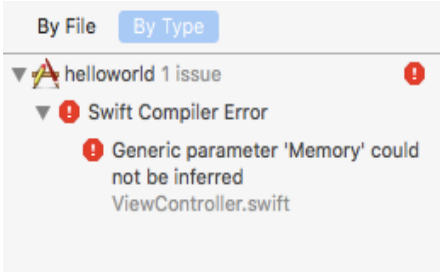


Fig. 7 Swift Compiler Error

Heap Overflow (Buffer Overflow)

We looked into what others might have found that we might've missed while testing buffer overflows in Swift, and found on an online journal: Dr. Dobb's, an article where the author claims the code from Objective-C (top) can produce a buffer overflow in Swift (bottom):

```
1 size_tinput_size = getchar();
2 char* buf = malloc(input_size);
3 gets(buf); // if more than input_size bytes
```

Fig. 8 Objective-C Buffer Overflow Code

```
1 let inputSize : Int32 = getchar()
2 let buf = UnsafePointer<Int8>.alloc(Int(inputSize))
3 gets(buf) // if more than input_size bytes is available
```

Fig. 9 Swift Buffer Overflow Code

When we run the Swift code to analyze it, however, the application does not even build because it does not allow the `alloc` reference to memory and throws an error:

```
16 let inputSize : Int32 = getchar()
17 let buf = UnsafePointer<Int8>.alloc(Int(inputSize))
18 gets(buf)
```

Fig. 10 Swift Buffer Overflow Code

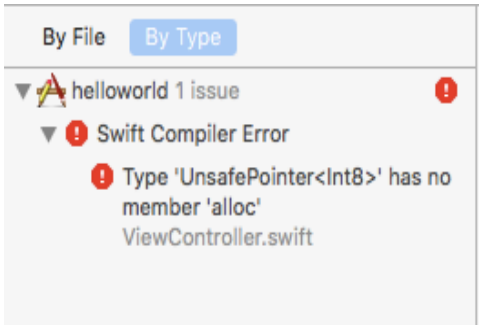


Fig. 11 Swift Compiler Error

Insecure Data Storage

iOS has local on-device storage solutions like NSUserDefaults, Plist Files, CoreData (Sqlite files), and the Keychain.

NSUserDefaults

NSUserDefaults saves user preferences and properties in an application, and the data persists even after the application closes and starts again. Common things saved onto NSUserDefaults are the logged in state of a user (YES or NO), or a user's access token. Saving any of these things into NSUserDefaults makes it possible to not have to ask the user to sign

in all over again if they did not sign out, or customizing what's displayed to a user depending if they are logged in into the application or not.

The data saved by NSUserDefaults is not secure because it can easily be viewed from the application bundle in a plist file that has the name of the bundle id of the application. Using the open-source application from InfoSec Institute localDataStorageDemo, in figure 12 is an example [14] of how to access content stored in NSUserDefaults:

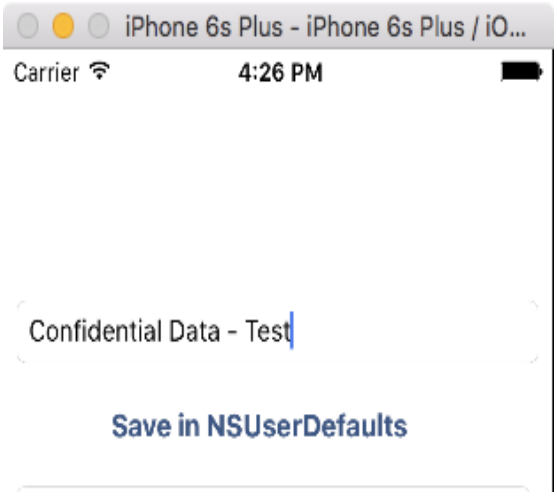


Fig. 12 NSUserDefaults Application Example

If we enter the phrase Confidential Data – Test into the textbox for NSUserDefaults and press the button to Save in NSUserDefaults, if we are running this application through Xcode's iPhone simulator, that data can be found if you search for the plist file for the app like this:

/Users/USERNAME/Library/Developer/CoreSimulator/Devices/3BBDC61C-8D67-454A-B50C-18F677F4223F/data/Containers/Data/Application/2E7059A2-2F53-4E9A-8756-E38961EBF73E/Library/Preferences/com.highaltitudehacks.LocalDataStorageDemo.plist

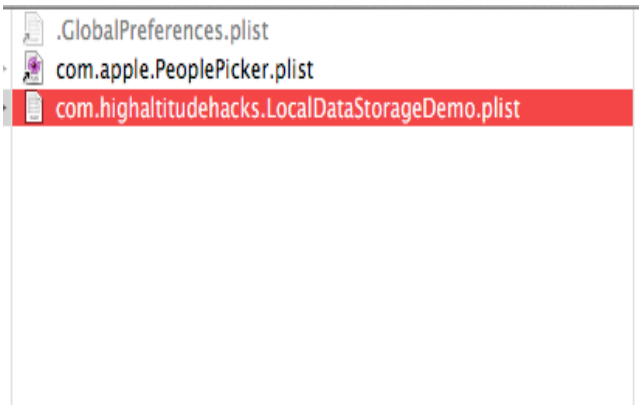


Fig. 13 NSUserDefaults plist File Location

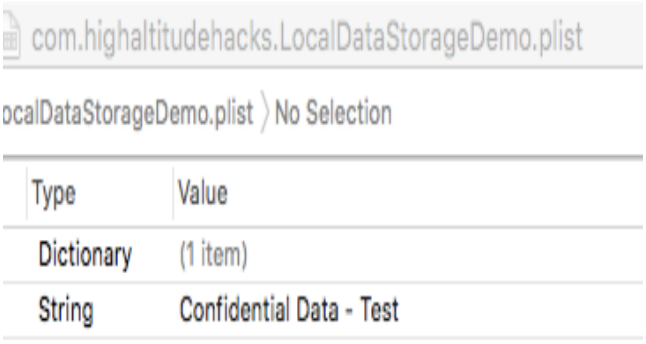


Fig. 14 NSUserDefaults Application Plain Text Data

Plist Files

Plist Files is another method of storing information onto the device unencrypted. That data is easily accessible, so it should not be used to store sensitive data like Access tokens, Usernames or passwords.

Below we show how to access the data stored into a Plist file called userInfo.plist:

```
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,NSUserDomainMask,YES);
NSString *documentsDirectory = [paths objectAtIndex:0];
NSString *filePath = [documentsDirectory stringByAppendingString:@"/userInfo.plist"];
 NSMutableDictionary* plist = [[NSMutableDictionary alloc] init];
[plist setValue:self.usernameTextField.text forKey:@"username"];
[plist setValue:self.passwordTextField.text forKey:@"passwprd"];
[plist writeToFile:filePath atomically:YES];
```

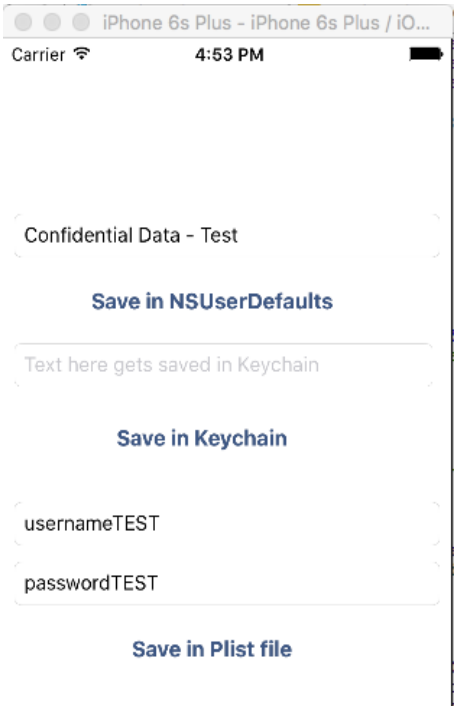


Fig. 15 Plist Storage Application Example

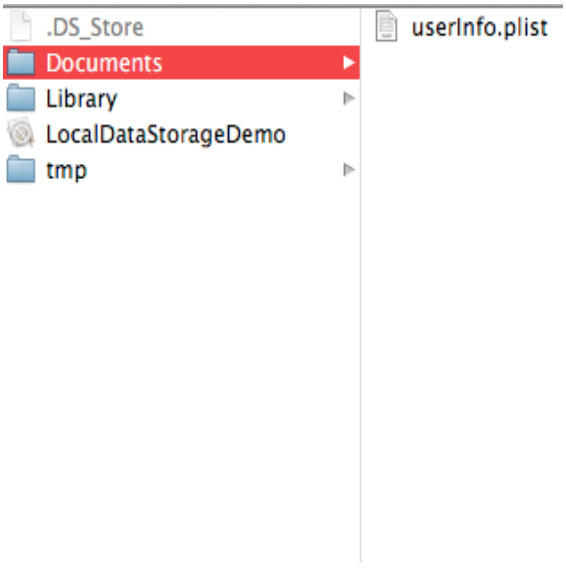


Fig. 16 Plist File Location (Custom File Name)

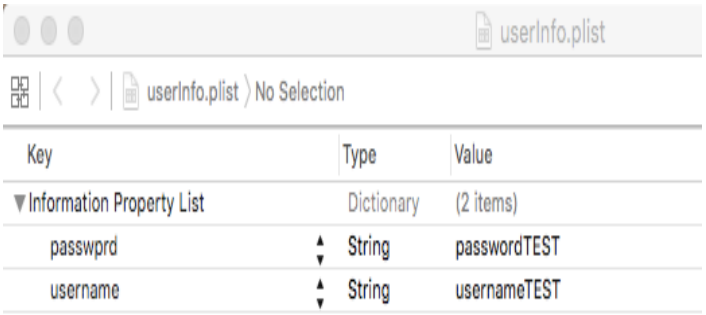


Fig. 17 Plist File Insecure Storage in Plaintext

CoreData and Sqlite Files

Core data uses Sqlite internally to save data, so, it's used to create models of different objects with relationships. That data is saved locally as .db files as in figure 18 below and is fetched from the local cache with queries.

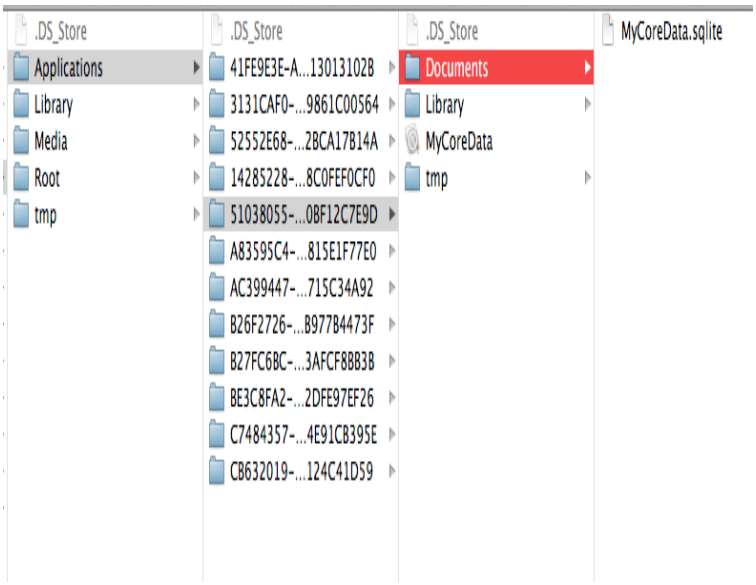


Fig. 18 CoreData database file Location

The values in the tables in these files can easily be dumped into a file. By default, data stored in CoreData is unencrypted and can be easily found within a device.

iOS ENCRYPTION TECHNOLOGIES

Common iOS encryption technologies include support functions like the Keychain services API (stores passwords, keys, etc.), the cryptographic message syntax (encrypts or signs messages commonly used with email), and the certificate key and trust services (provides trust validation and cryptography support functions), and Common Crypto (low-level C support for encryption and decryption). Basic encryption and decryption, signing and verifying of blocks of data is also available through the SecKey functions (SecKeyEncrypt, SecKeyDecrypt, SecKeyRawSign, SecKeyRawVerify) [20].

Secure Data Storage Encryption Example [16] (AES)

Symmetric encryption algorithm (ex. Advanced Encryption Standard or AES), or secret key, uses the same key for both encryption and decryption. Asymmetric encryption algorithms (ex. Rivest, Shamir, Adleman or RSA), or public key, use different keys for encryption and decryption.

In order to store data securely using the above storage options in swift, below is an example of how encrypt a string type using the encryption component (CkoCrypt2) from Chilkat software company, which allows users to set the algorithm (symmetric AES in this case), cipher, key length, padding scheme, encoding mode, initialization vector (if applicable), and secret key.

```
let crypt = CkoCrypt2()
crypt.CryptAlgorithm = "aes"
//  CipherMode may be "ecb", "cbc", "ofb", "cfb", "gcm", etc.
crypt.CipherMode = "cbc"
//  KeyLength may be 128, 192, 256
crypt.KeyLength = 256
//  The padding scheme determines the contents of the bytes
//  that are added to pad the result to a multiple of the
//  encryption algorithm's block size.  AES has a block
//  size of 16 bytes, so encrypted output is always
//  a multiple of 16.
crypt.PaddingScheme = 0
//  EncodingMode specifies the encoding of the output for
//  encryption, and the input for decryption.
//  It may be "hex", "url", "base64", or "quoted-printable".
crypt.EncodingMode = "hex"
//  An initialization vector is required if using CBC mode.
//  ECB mode does not use an IV.
//  The length of the IV is equal to the algorithm's block size.
//  It is NOT equal to the length of the key.
var ivHex: String? = "000102030405060708090A0B0C0D0E0F"
crypt.SetEncodedIV(ivHex, encoding: "hex")
//  The secret key must equal the size of the key.  For
//  256-bit encryption, the binary secret key is 32 bytes.
//  For 128-bit encryption, the binary secret key is 16 bytes.
var keyHex: String? = "000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F"
crypt.SetEncodedKey(keyHex, encoding: "hex")
//  Encrypt a string...
//  The input string is 44 ANSI characters (i.e. 44 bytes), so
//  the output should be 48 bytes (a multiple of 16).
//  Because the output is a hex string, it should
//  be 96 characters long (2 chars per byte).
var encStr: String? = crypt.EncryptStringENC("The quick brown fox jumps over the lazy dog.")
println("\(encStr!)")
```

SECURITY STUDY – JAVA ANDROID

Android has built-in security mechanisms like Address Space Layout Randomization (ASLR; Introduced in Ice Cream Sandwich release 4.0), Position Independent Executable (PIE; Support added in 4.1 release) and Data Execution Prevention (DEP; Android 2.3 and later releases have non-executable pages, stack and heap, by default in architectures that support it) that reduce the exploitability of buffer overflow errors.

The Android developer website suggests to minimize using API's that access sensitive or personal user data. If transmitting or storing is absolutely necessary, Android suggests that apps use hash or a non-reversible form of the data in its logic to reduce the chance of unintentional data leakage and insecure data storage. The Android developer website warns against over-permissive Interprocess communications, world writeable files, network sockets, and when writing to on-device logs. Since logs are a shared resource in Android, they are available to applications through the READ_LOGS permission [11].

Unintended Data Leakage

Heap Overflow (String Handling)

In Java, Strings are immutable, meaning they cannot be modified after they have been constructed. Copying one string to another simply gives the new string variable the reference of the first string variable:

String a = "hello";

String b = a;

a = "world";

As in the example above, as long as there is at least one reference to the string, the garbage collector will not get rid of the string, so even though a new string was created and variable a was reassigned to reference “world” instead, variable b still references “hello”.

Insecure Data Storage

Android has different storage options depending on whether an application needs data to be accessed by one or many applications, or depending on how much space the data requires. Android has Shared Preferences, Internal Storage and SQLite Databases. Shared Preferences is a class that provides a framework with persistent save and retrieval key-value pairs capabilities of primitive types.

Internal Storage allows files to be stored on the device internal storage to be only accessed by the application that stored them and not accessible to other applications or the user. SQLite databases on Android are lightweight file-based and only accessible to classes within the application. Similar to iOS’s storage, these storage options store values in plain text, and so unless the values stored are encrypted, then these storage options if used with no encryption are not secure.

Secure Data Storage

Android has a cryptography API which has the main packages javax.crypto (package provides the classes and interfaces for cryptographic applications implementing algorithms for encryption, decryption, or key agreement), javax.crypto.interfaces (package provides the interfaces needed to implement the key agreement algorithm), javax.crypto.spec (package provides the classes and interfaces needed to specify keys and parameter for encryption) [15].

Android’s Crypto API symmetric encryption algorithm example below is used to encrypt and decrypt a String using AES algorithm SHA1PRNG SecureRandom algorithm and a128-bit secret key [15]:

```
// Original text
    String theTestText = "This is just a simple test";
    TextView tvorig = (TextView)findViewById(R.id.tvorig);
    tvorig.setText("\n[ORIGINAL]:\n" + theTestText + "\n");
    // Set up secret key spec for 128-bit AES encryption and decryption
    SecretKeySpec sks = null;
    try {
       SecureRandom sr = SecureRandom.getInstance("SHA1PRNG");
       sr.setSeed("any data used as random seed".getBytes());
       KeyGenerator kg = KeyGenerator.getInstance("AES");
       kg.init(128, sr);
       sks = new SecretKeySpec((kg.generateKey()).getEncoded(), "AES");
    } catch (Exception e) {
       Log.e(TAG, "AES secret key spec error");
    }
// Encode the original data with AES
    byte[] encodedBytes = null;
    try {
       Cipher c = Cipher.getInstance("AES");
       c.init(Cipher.ENCRYPT_MODE, sks);
```

```
            encodedBytes = c.doFinal(theTestText.getBytes());
        } catch (Exception e) {
            Log.e(TAG, "AES encryption error");
        }
        TextView tvencoded = (TextView)findViewById(R.id.tvencoded);
        tvencoded.setText("[ENCODED]:\n" +
            Base64.encodeToString(encodedBytes, Base64.DEFAULT) + "\n");
```

Fig. 19 Android AES Encryption Application

Android's Crypto API asymmetric encryption algorithm example below is used to encrypt and decrypt a String using a 1024-bit RSA encryption algorithm [15]:

```
// Original text
    String theTestText = "This is just a simple test!";
    TextView tvorig = (TextView)findViewById(R.id.tvorig);
    tvorig.setText("\n[ORIGINAL]:\n" + theTestText + "\n");
    // Generate key pair for 1024-bit RSA encryption and decryption
    Key publicKey = null;
    Key privateKey = null;
    try {
        KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
        kpg.initialize(1024);
        KeyPair kp = kpg.genKeyPair();
        publicKey = kp.getPublic();
        privateKey = kp.getPrivate();
    } catch (Exception e) {
        Log.e(TAG, "RSA key pair error");
    }
    // Encode the original data with RSA private key
    byte[] encodedBytes = null;
    try {
        Cipher c = Cipher.getInstance("RSA");
```

```
    c.init(Cipher.ENCRYPT_MODE, privateKey);
    encodedBytes = c.doFinal(theTestText.getBytes());
} catch (Exception e) {
    Log.e(TAG, "RSA encryption error");
}
TextView tvencoded = (TextView)findViewById(R.id.tvencoded);
tvencoded.setText("[ENCODED]:\n" +
    Base64.encodeToString(encodedBytes, Base64.DEFAULT) + "\n");
```

Fig. 20 Android RSA Encryption Application

RELATED WORKS

Buffer Overflow Detection using Static Analysis (Critical Embedded Programs in C Language)

In Allamigeon's paper Static Analysis of String Manipulations in Critical Embedded C Programs [13], we saw that buffer overflows were being looked at for a subset of the C language found in critical embedded software using Static Analysis based on the theory of abstract interpretation. In comparison to this approach, our study takes a different approach by using relatively recently released open source Static Analysis tools on code Apple and Google have posted in their secure guidelines websites to test Insecure Data Storage and Unintended Data Leakage attacks.

Unintended Data Leakage (Android - ASLR By-Pass Exploit)

Android's ASLR security feature was thought to be a difficult feature to exploit until a 2016 research paper by Hanan Be'er of software research company NorthBit was released [12]. This exploit of Android's ASLR feature affects Android devices with versions 2.2 – 4.0 and 5.0 – 5.1 (since Android 2.2 – 4.0 don't implement ASLR).

The research focuses on Metaphor, which is the name given to the paper's implementation of stagefright; An Android multimedia library that gained attention in July 2015 when several of its heap overflow vulnerabilities were discovered. The paper refers to the library as "libstagefright" and to the bug as "stagefright." Metaphor bypasses Android's ASLR by using Android's heap allocator, jemalloc, and the way it allocates objects of similar sizes in the same run. The web browser is used as the attack vector and techniques like heap grooming (allocates many objects vs spraying the heap) are employed to control the order of allocations and deallocations, and thus, design the order of heap objects in a predictable manner.

Successful exploitation of this library requires JavaScript execution to be enabled, among other architecture specific specifications to also match (like the ASLR algorithm to run on a 32 bit ARM to be able to translate gadget offsets to absolute addresses). But in summary, to allow for Data Leakage, this exploit can be used to overwrite the contents of the mStorage array (mStorage is an array of keys and MetaData::typed_data elements. Mediaserver parses and sends metadata from within the media file back to the web browser. The metadata is stored inside MetaData objects, that store all data in their mItems fields, which are a dictionary of FourCC (4 characters code) keys to MetaData::typed_data values) to point to arbitrary locations in memory and thus leaking information back to the web browser.

RESULTS

The introduction of Swift did not bring much underlying change into iOS mobile application security. The language does introduce an advance Error Handling Model and Syntax improvements, among other things [4]. But, the fact that it can be used along with Objective-C indicates that any vulnerabilities found in the underlying libraries and frameworks in iOS with the use of Objective-C can still be exploited through Swift.

iOS has added safer memory management and memory allocation with Swift which make certain buffer and stack overflows no longer possible in Swift because the application throws a compiler error. Apple's secure coding guidelines provide heap and stack overflows prevention examples and precautions but no encryption examples to use for when storing data in any of their local stage options. Both the iOS and Android local storage options store data in plain text. Encryption is the best solution for securely storing data into any of these storage options. Since there is an OWASP Mobile Vulnerability M6 [3] (Broken Cryptography) dedicated to the misuse of cryptography, the topic of testing the cryptographic algorithms available goes beyond the scope of this study.

In comparison to Objective-C, Android did prove to not be as easily to exploit in terms of heap and stack overflows. Also, the application sandboxing in Android also adds a layer of security into the storage options in comparison to the storage options in iOS.

FUTURE WORK

The OWASP Mobile Top Ten Risks had Broken Cryptography as their sixth vulnerability, which we did not include in this study. After researching the available security mechanism built in the storage options for both iOS and Android it was clear the importance encryption has in securing the information stored in these local storage options.

As a future addition to this study, the fact that, not the lack of encryption used, but rather that the encryption added to applications is broken and as a result is vulnerable leads to question as to how safe really is an application if even after using encryption, there is a chance that it might still be vulnerable due to the cryptography being "broken."

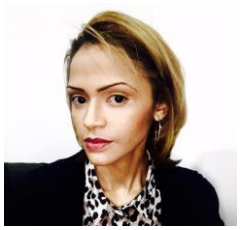
In June 2015, a year after its introduction, the programming language Swift also became open source. It would be interesting to add to this study how this affects the safety of the applications developed in Swift as opposed to using a non open source application.

REFERENCES

1. http://www.alzheimer-europe.org/Research/Understanding-dementia-research/Types-of-research/The-four-main-approaches , ALZHEIMER EUROPE OFFICE, Alzheimer Europe Office, The four main approaches - Types of research, Friday 21 August 2009
2. http://research-methodology.net/research-methodology/research-approach/ , Research Methodology - Research Approach, John Dudovskiy
3. https://www.owasp.org/index.php/OWASP_Mobile_Security_Project#tab=Top_10_Mobile_Risks , OWASP Mobile Security Project, 14 March 2016, at 02:32
4. https://developer.apple.com/swift/ , Apple Developer – Swift, 2016
5. https://developer.apple.com/library/ios/recipes/xcode_help-source_editor/chapters/Analyze.html , iOS Developer Library, Performing Static Code Analysis, 2016-03-21
6. https://tailor.sh , Tailor - Tailor. Cross-platform static analyzer and linter for Swift., 2015
7. http://clang-analyzer.llvm.org , Clang Static Analyzer
8. http://developer.android.com/tools/help/lint.html , Android Developers - Lint
9. https://developer.apple.com/library/mac/documentation/Security/Conceptual/SecureCodingGuide/Articles/TypesSecVuln.html#//apple_ref/doc/uid/TP40002529-SW5 , Apple Secure Coding Guidelines (Buffer Overflows, Secure Storage)
10. http://www.drdobbs.com/security/security-issues-in-swift-what-the-new-la/240168882 , Dr. Dobb's, Security Issues in Swift: What the New Language Did Not Fix, Denis Krivitski, August 19, 2014
11. http://developer.android.com/training/articles/security-tips.html#InputValidation , Android Developers – Security Tips
12. Hanan Be'er, NorthBit, Metaphor – A (real) real-life Stagefright exploit. Revision 1.1, 2016
13. Allamigeon, Xavier; Godard, Wenceslas ; Hymans, Charles. Static Analysis of String Manipulations in Critical Embedded C Programs
14. http://resources.infosecinstitute.com/ios-application-security-part-20-local-data-storage-nsuserdefaults-coredata-sqlite-plist-files/ , IOS Application Security Part 20 – Local Data Storage (NSUserDefaults, CoreData, Sqlite, Plist files), Prateek Gianchandani, INFOSEC Institute

15. http://www.developer.com/ws/android/encrypting-with-android-cryptography-api.html , Android Encryption with the Android Cryptography API, Chunyen Liu, 5/20/13
16. http://www.example-code.com/swift/crypt2_aes.asp , (Swift) AES Encryption
17. http://www.example-code.com/swift/rsa_encryptStrings.asp , (Swift) RSA Encrypt and Decrypt Strings
18. http://blog.mdsec.co.uk/2012/05/introduction-to-ios-platform-security.html, Introduction to iOS Platform Security, MDSec - Consultancy, Training and Research from a global authority on Information Security, 5/10/2012
19. https://www.chilkatsoft.com/corporate.asp , About Chilkat Software Inc., 2016
20. https://developer.apple.com/library/mac/documentation/Security/Conceptual/cryptoservices/GeneralPurposeCrypto/GeneralPurposeCrypto.html , Encrypting and Hashing Data, Cryptographic Services Guide, 7/15/2014
21. https://www.owasp.org/index.php/Mobile_Top_10_2014-M2 , Insecure Data Storage, Mobile Top 10 2014-M2
22. https://www.owasp.org/index.php/Mobile_Top_10_2014-M4 , Unintended Data Leakage, Mobile Top 10 2014-M4

Author' biography with Photo



Vanessa Santana graduated Iona College with a Masters in Computer Science in 2016. There she worked on Mobile Security research with the help of her advisor Dr. Paolina Centonze. She currently works in Electric, Gas and Steam utility company Consolidated Edison in New York where she develops and updates applications used by internal and external customers.

Paolina Centonze, Ph.D. has been a professor in the Computer Science Department of Iona College since August 2011. Her areas of research include language-based security and mobile computing. At Iona College, she has been responsible for extending the Computer Science curricula into the field of Cyber Security.

Before joining Iona College, Dr. Centonze was a researcher at IBM's Thomas J. Watson Research Center in Yorktown Heights, N.Y. She has published extensively at numerous conferences worldwide, such as ISSTA, ECOOP, ACSAC, MDM, MOBILESoft, MobileDeLi. Dr. Centonze is also the author of two book chapters in the area of cloud and mobile security, which will appear in 2016 in books published by IGI Global and John Wiley & Sons. She is also the inventor of 10 patents issued by the United States Patent and Trademark Office.

Dr. Centonze received her Ph.D. in Mathematics and MS degree in Computer Science from New York University (NYU) Tandon School of Engineering in Brooklyn, N.Y., and her BS degree in Computer Science from St. John's University in Queens, N.Y.