Poster: Static Detection of API Call Vulnerabilities in iOS Executables*

Chun-Han Lin
National Chengchi University, Taiwan
john.lin0420@gmail.com

Fang Yu
National Chengchi University, Taiwan
yuf@nccu.edu.tw

Jie-Hong Roland Jiang
National Taiwan University, Taiwan
jhjiang@ntu.edu.tw

Tevfik Bultan
University of California, Santa Barbara, CA
bultan@cs.ucsb.edu

ABSTRACT

We propose a static analysis technique for iOS executables for checking API call vulnerabilities that can cause 1) app behaviors to be altered by malicious external inputs, and 2) sensitive user data to be illegally accessed by apps with stealthy private API calls that use string obfuscation. We identify sensitive functions that dynamically load classes/frameworks, and, for each parameter that corresponds to a dynamically loaded class/framework, we construct a dependency graph that shows the set of values that flow to that parameter. A sensitive function that has its class name or framework path parameter depending on external inputs is considered to contain a vulnerability. We further conduct string analysis on these dependency graphs to determine all potential string values that these parameters can take, which identifies the set of dynamically loaded classes/frameworks. Taking the intersection of these values with patterns that characterize Apple's API policies (such as restricted use of private/sensitive APIs), we are able to detect potential policy violations and vulnerabilities.

CCS CONCEPTS

• **Security and privacy → Logic and verification**;

KEYWORDS

String analysis, iOS mobile application, API call vulnerability

ACM Reference Format:
Chun-Han Lin, Fang Yu, Jie-Hong Roland Jiang, and Tevfik Bultan. 2018. Poster: Static Detection of API Call Vulnerabilities in iOS Executables. In *Proceedings of 40th International Conference on Software Engineering Companion, Gothenburg, Sweden, May 27-June 3, 2018 (ICSE '18 Companion),* 2 pages.
https://doi.org/10.1145/3183440.3195024

*This project is funded in part by MOST 106-2221-E-004 -002 -, MOST 105-2923-E-002-016-MY3, NSF CCF-1548848 and DARPA FA8750-15-2-0087.

1 API CALL VULNERABILITY

Most malicious behaviors and violations of security policies, such as IDFA abuse and private API usage, are related to the loaded classes and invoked methods of mobile applications. Due to the flexibility of Objective-C (actually most modern programming languages, such as PHP, Java reflections), developers can use string variables to load classes and invoke methods dynamically. This hinders effective program analysis and verification and detection of potential policy violations. Furthermore, mobile applications downloaded as executables are not available with source code. This requires significant work to rebuild program flows at the assembly level. The closed system nature makes the iOS mobile applications even harder to analyze.

Listing 1: Load a class dynamically

```
NSBundle *b = [NSBundle bundleWithPath:@"/System/Library/
          Frameworks/AdSupport.framework"];
[b load];
Class c = NSClassFromString(@"ASIdentifierManager");
id si = [c valueForKey:@"sharedManager"];
```

Listing 2: Load a class with string manipulations

```
NSString *name = [NSString stringWithFormat:@"%c%c%c%c%
          c%c%c%c%c%c%c%c%c%c%c%c%c%c%c",'A','S','I','d','e
          ','n','t','i','f','i','e','r','M','a','n','a','
          g','e','r'];
Class c = NSClassFromString(name);
```

We start from a simple code in listing 1 that would be caught by Apple's app review process for loading the ASIdentifierManager class with the class name (a string value) by calling a C-function called NSClassFromString. After the class was loaded, it then gets the value of a static field named sharedManager to access users' private information. Note that the class is loaded dynamically via the NSClassFromString function. While the loaded class depends on the value of the parameter that can be manipulated through string operations, the dynamism could lead to a loophole in Apple's app review process. In fact, one could load the class ASIdentifierManager without having any class associated with ASIdentifierManager at compile time, and bypass the check on IDFA abuse.

Listing 2 is a modified version of dynamically loading the ASIdentifierManager class. The parameter of NSClassFromString is no longer a literal but a string variable called name. As the listing shows, the value of name is synthesized at runtime via concatenating 19 characters (by calling stringWithFormat function in NSString). In this case, resolving static methods [3, 5] or searching constants appearing in assembly [6] would fail to find the correct class ASIdentifierManager associated with the app. Constant propagation techniques [1, 4] could be used to reveal the loaded classes. However, since the loaded classes depend on the values of string variables, there are various ways to obfuscate the loaded classes and invoked methods by manipulating string values with advanced string operations such as replacement, by composing string operations with branch and loop structures, and by using external calls to get string values from Internet or user inputs.

Listing 3 shows a code snippet that has such a vulnerability, where the developer embeds a backdoor to invoke APIs, including private APIs, from specific inputs. The app first loads 3 payload strings from a remote server. The backdoor is triggered while the first payload string $p1$ is equal to the keyword "fire". The value of $p2$ specifies the bundle path. Then an obfuscation with string replacement operation is applied to the third payload $p3$ to generate the invoked class name. With the string manipulation, an input string that contains no harmful strings may result in an attack. For instance, by replacing all characters "x" with "p", an input value "FTDeviceSuxxort" of string $p3$ results in the string "FTDeviceSupport", and is then used to invoke the class. Note that FTDeviceSupport that enables app to access device information is one of the private classes that shall not be invoked in any apps published in Apple's app store.

Such a vulnerability may bypass Apple's app review process. Note that, detecting such a vulnerability with runtime analysis [2] may not be possible since it requires specific inputs to trigger the backdoor. Second, with string manipulation, an input that contains no harmful strings may result in an attack. Pre-screening of user inputs (using a blacklist) cannot detect such a violation. In fact, we have successfully embedded this type of back-door code into an app that has been published in Apple's app store.

Listing 3: Load a class with external inputs

```
NSString *p1, *p2, *p3 = some payload strings from a
        remote server;
if([p1 isEqualToString:@"fire"]){
NSBundle *b = [NSBundle bundleWithPath:p2];
[b load];
NSString* name = [p3 stringByReplacingOccurrencesOfString
        :@"x" withString:@"p"];
Class c = NSClassFromString(name); ...}
```

2 STATIC VULNERABILITY DETECTION

We propose a sound static analysis for systematic API vulnerability checking in iOS executables. It is necessary to determine the possible string values for the name parameters of the functions that dynamically load classes/frameworks.

App fetching and decryption. We first download and install online apps from Apple's app store into a jail-broken iOS device, where we can access the file system directly to fetch the target binary. The binary is encrypted and it is decrypted by the device with authentication upon execution.The decrypted binary can then be analyzed with disassembler toolsto generate the plain text format assembly code.

Segment information extraction and control flow graph (CFG) construction. An iOS app's binary is a Mach Object (Mach-O), and its assembly is split into multiple segments containing various meta information such as subroutine entries, external calls, constant strings, mapping tables, etc., in addition to the assembly body of its subroutines. We extract needed information from assembly segments to construct the control flow graph (CFG) for each subroutine, and resolve register values of indirect jumps to link these routines. During the CFG construction, we also mark dependency relations of registers for each assembly statement. To identify sensitive functions, we find call-external-C-function-node or call-external-method-node and resolve their register values to identify which ones are relevant to the target (sensitive) function. When a sensitive function is identified, we then build the dependency graphs for its parameters. This can be done by traversing the dependency relations from the corresponding register (sink) backwards up to constants or external inputs.

Dependency graph construction and string analysis. For each sink, we build its string dependency graph that specifies how input values flow to the sink. The sink values define the values of the parameters of target functions. For each dependency graph, if it contains an external input, we report a vulnerability. For policy checking, we conduct forward string analysis on the graph to characterize all potential string values at the sink node. We adopt automata-based string analysis where the automata associated with the sink node accepts all possible string values that can reach the sink node. We start from constants and arbitrary values of external inputs and propagate string values through string operations using automata constructions until a fixpoint is reached at the sink node of the dependency graph. The automata are then used to determine all the dynamic loaded classes and invoked methods.

Property checking. We check property violations using automata operations. In our current implementation, we check whether the automata that characterize the set of dynamically loaded classes has a non-empty intersection with automata that characterize policy violations (specified as regular expressions).

3 EXPERIMENTS

We have built an end-to-end tool called BinFlow and used it to analyze more than one thousand popular apps from Apple's AppStore. We identified 435 apps having around 38000 calls in total using dynamically loaded classes/frameworks. We identified 243 apps that contain 385 potential vulnerabilities due to call values constructed from external inputs at run time. We found 18 apps that exploit string obfuscation and illegally use private/sensitive APIs for stealthy user data access.

REFERENCES

[1] Paulo Barros, Rene Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo dAmorim, and Michael D. Ernst. 2015. Static Analysis of Implicit Control Flow: Resolving Java Reflection and Android Intents (T). In *ASE*. 669–679.

[2] Zhui Deng, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2015. iris: Vetting private api abuse in ios applications. In *CCS*. ACM, 44–56.

[3] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2011. PiOS: Detecting Privacy Leaks in iOS Applications. In *NDSS*.

[4] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. 2016. DroidRA: taming reflection to support whole-program analysis of Android apps. In *ISSTA*. 318–329.

[5] Tim Werthmann, Ralf Hund, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2013. PSiOS: bring your own privacy & security to iOS devices. In *ASIACCS*. ACM, 13–24.

[6] Fang Yu, Yuan-Chieh Lee, Steven Tai, and Wei-Shao Tang. 2013. AppBeach: Characterizing App Behaviors via Static Binary Analysis. In *Mobile Services*. IEEE Computer Society, 86.