SCTE CABLE-TEC EXPO 22
SEPTEMBER 19-22 • PHILADELPHIA, PA

2022 Fall Technical Forum
SCTE • CABLELABS • NCTA

Bitcode Obfuscation

Protecting Software Without Source Code Access

A Technical Paper prepared for SCTE by

Rafie Shamsaasef
Director of Software Engineering
CommScope
6450 Sequence Dr, San Diego, CA 92121
1 (858) 404-2205
rafie.shamsaasef@commscope.com

Lex Aaron Anderson
Senior Security Architect
CommScope
PO Box 37-942 Parnell 1052, Auckland, New Zealand
+64 935 803 75
aaron.anderson@commscope.com

© 2022, SCTE®, CableLabs® and, NCTA. All rights reserved.

Table of Contents

Title **Page Number**

1. Introduction 3
2. Introduction to Software Obfuscation 3
3. Control-flow and Data-flow Obfuscation 5
   3.1. Control Flow Obfuscation 5
   3.2. Data Flow Obfuscation 5
4. Bitcode obfuscation techniques 5
   4.1. What is bitcode? 6
   4.2. Why bitcode protection? 6
   4.3. How does it work? 7
5. Exploring the usage of bitcode obfuscation 7
   5.1. Mobile applications 8
   5.2. Cloud Server Applications and Web Services 10
   5.3. IoT Applications 10
6. Conclusion 11

Abbreviations 12

Bibliography 13

List of Figures

Title **Page Number**

Figure 1 - Encryption vs Obfuscation 4
Figure 2 - LLVM Architecture 6
Figure 3 - Cloud-based bitcode obfuscation use-case. 7
Figure 4 - iOS Mobile App Framework 8
Figure 5 - iOS Mobile App Framework with App Obfuscation 9
Figure 6 - Sample Cloud App with Obfuscation 10

© 2022, SCTE® CableLabs® and NCTA. All rights reserved.

1. Introduction

Software obfuscation techniques have become increasingly popular in recent decades due to their broad applicability toward malware threats and intellectual property protection. Reverse engineering and tampering attacks are prominent means for software piracy and exploitation. Obfuscation is a collection of techniques for securing software and protecting applications from harmful malware. The goal of these techniques is to increase the cost and feasibility for attackers to exploit security vulnerabilities and carry out successful attacks against software implementations.

Bitcode obfuscation is a form of obfuscation that operates on an intermediate code representation rather than the original source code or native binary. This technique is based on sound and proven mathematical principles that retain the security characteristics of native binary while also maintaining the portability and platform independence of source code. Obfuscating at the bitcode level enables developers to utilize obfuscation-as-a-service without exposing their source code or proprietary libraries, while achieving levels of protection not possible with source-code obfuscation. As such, bitcode obfuscation is a promising field for developing the next generation of software protection services in the cloud.

In this paper, we offer technical details of control-flow and data-flow obfuscation techniques based on the idea that no source code or other dependencies are required to apply strong obfuscation directly to the intermediate bitcode representation rather than the original source code or the native binaries. We conclude by providing insights into the usage of obfuscation in relation to the security requirements of connected and cloud software systems.

2. Introduction to Software Obfuscation

In software development, obfuscation is the act of generating source or machine code that is difficult for humans and automated tools to reverse engineer. The goal is to achieve maximally unintelligible code without introducing unacceptable levels of overhead. Many techniques have been described in the literature that have both heuristic and tractable security basis. A combination of techniques can provide synergy in terms of the work factor required to reverse engineer the resulting binary code.

Programs often use high-level programming language constructs, common design patterns, and reusable components. These software engineering practices make code easier to reverse engineer and exploit (Wikipedia, 2022). It is therefore not appropriate (nor sufficient) for programmers to deliberately obfuscate their code to attempt conceal its purpose (security through obscurity), since many advanced tools exist to deobfuscate code that has not been sufficiently protected (OWASP, 2016).

Obfuscation is not the same as encryption. There is a common misconception to think of obfuscated data or code being the same as encrypted ones. While they both manipulate the original data/code to a different form, they are fundamentally different processes. Encryption requires a key and deploys a well-known cipher algorithm such as AES to convert the data or code to an unreadable form; where a decryption process is needed to perform reverse operation and get back the original data/code. Obfuscation on the other hand utilizes algorithms which often do not require a key. The obfuscated data and code can be used as-is without a need to de-obfuscate them. In-fact the goal is to make the code hard to de-obfuscate while retaining the original purpose and minimizing overhead as much as possible.

© 2022, SCTE® CableLabs® and NCTA. All rights reserved.

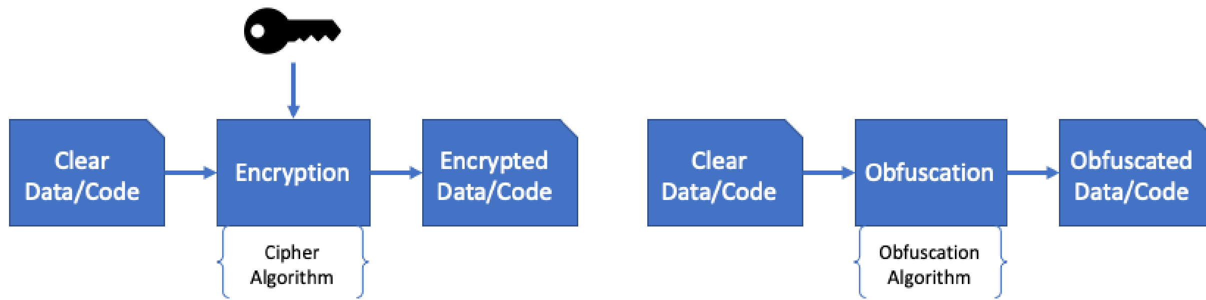

Figure 1 - Encryption vs Obfuscation

When it comes to source code or binary object encryption versus obfuscation, the encrypted object must be decrypted prior to execution rather the obfuscated object can be executed while remaining in the obscured form. It is possible for obfuscated elements to be captured and de-obfuscated, but the obfuscation presents a barrier to understanding and analysis (Arini Balakrishnan, 2005).

Several notions of obfuscation were shown in (Barak, et al., 2001) and (Balakrishnan & Schulze, 2005). The main result is that a strong notion of obfuscation cannot always be achieved. Over a decade later, Garg, Gentry, and Halevi (Garg, Gentry, & Halevi, 2013) gave the first candidate construction of an efficient general-purpose indistinguishability obfuscator, called multilinear jigsaw puzzles. While indistinguishability obfuscation promises secure general-purpose obfuscation, it remains an open question as to whether any practical real-world indistinguishability obfuscators can be implemented under this new model.

There are many mature obfuscation techniques offering a wide range of protections based on heuristic approaches to manipulate source code and even binary without altering the logic or the purpose of the code. The strength and weakness of obfuscation approaches can only be determined by effectiveness of the attacking and exploitation mechanism. Tools such as disassemblers and decompilers are often deployed to perform reverse engineering attacks by extracting sensitive information, adding malicious code and application cloning (Barak, et al., 2001). Although code obfuscation can thwart many attacks, given enough time and effort any of these techniques can be eventually overcome by reverse engineering process. Regardless, programs are obfuscated every day in the real world without any provable security guarantee. Nevertheless, obfuscation and diversification remain viable protection tools from the practical perspective (Garg, Gentry, & Halevi, 2013) (Goldwasser & Rothblum, 2007).

While obfuscation can be applied to programs written in any language, they are more effective for programs that are compiled to binary without the need for a virtual machine (VM). There are limited techniques that can be considered to obscure high level languages (i.e. Java and C#) instructions and control flow such that are heavily relying on their underlying VMs. Unlike C/C++, decompilation of Java programs is a much simpler task and therefore may be fully automated by attackers. Class hierarchy, high-level statements, names of classes, methods and fields can be retrieved from class files emitted by the standard javac compiler. Every current obfuscation product is easily circumvented by off-the-shelf de-obfuscation tools for java. The challenge gets even worse when trying to obfuscate a Python program. For Python, it basically comes down to renaming and hiding some of the instructions that can even be easily de-obfuscated (OWASP, 2016).

© 2022, SCTE® CableLabs® and NCTA. All rights reserved.

3. Control-flow and Data-flow Obfuscation

Obfuscation protects an application from reverse engineering, protecting proprietary source code, intellectual property and to increase the difficulty of exploitation. Obfuscation can be split into two main types: Control-flow and data-flow obfuscation. Together, these can provide a synergy that is similar in nature to how individually weak confusion and diffusion components when used together in cryptography lead to strong ciphers like AES (Anderson L. , 2015) (Worldwide Patent No. WO2018106439A1, 2018).

3.1. Control Flow Obfuscation

The aim of control-flow obfuscation is to make a program's execution difficult for an attacker to understand and hence reverse-engineer. Typical control-flow obfuscation methods include:

- **Instruction substitution** replaces assembly-level operations with randomly chosen code blocks that perform the same operation in different ways. The aim is to add resilience against both static analysis and dynamic analysis of the program code as well as automated attacks that look for code fingerprints.
- **Bogus-control-flow** obfuscation modifies a program's control-flow by adding entry points that evaluate complex expressions to determine the outcome of conditional jumps: either to jump to valid program code or to randomly altered "junk" code blocks.
- **Control-flow-flattening** rearranges a program's basic blocks in a randomized manner to give a program a uniformly random structure, where the original program's control-flow is only able to be re-established by the runtime computation of a control variable representing the state of the program.
- **Virtual-machine interpreter obfuscation** incorporates known hard mathematical problems in the computation of the control-flow to further increase the cost of reverse-engineering attacks.

3.2. Data Flow Obfuscation

Data-flow obfuscation is about randomizing the instructions that compute logical and arithmetic operations in a program (Worldwide Patent No. WO2018106439A1, 2018). Data-flow obfuscation methods include:

- **Randomized branch encoding** involves representing data-related logical and mathematical operations as a branching program composed of a sequence of permutations. Sequences of these branching programs are then concatenated together. When these programs are converted back to machine code, the result is uniformly randomized and unintelligible code that bears no resemblance to the original algorithm.
- **Randomized input, output encodings** can be used to make the obfuscated code even harder to reverse-engineer, as well as protecting constants and allowing seamless secure chaining to and from the obfuscate application.

4. Bitcode obfuscation techniques

Obfuscation (and other protection methods) can be applied directly to bitcode files. This enables application protection to be applied via a third-party service, and as such reduces the need for in-house tools and expertise in application security.

© 2022, SCTE® CableLabs® and NCTA. All rights reserved.

4.1. What is bitcode?

Bitcode is a platform-independent, universal low-level intermediate representation (IR) used by LLVM compilers and tools, such as Clang, XCode, Microsoft Clang-CL, Objective C/C++ Swift, GO, EM-Scripten, Rust, and many others. **Bitcode is the file format for LLVM IR** (LLVM Documents, 2003).

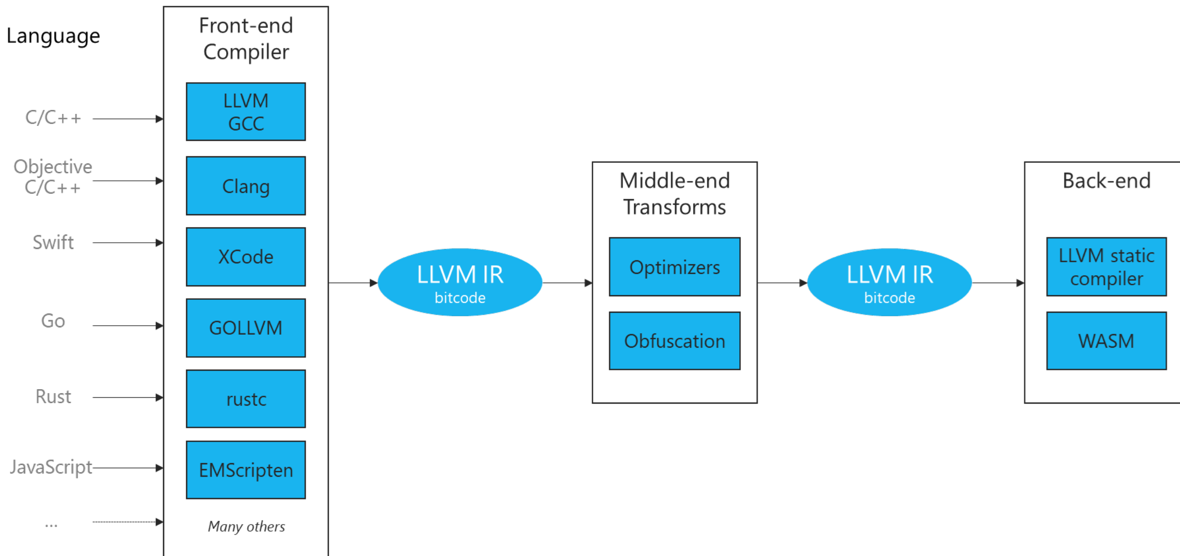

Figure 2 - LLVM Architecture

- Front-end compilers compile source languages to LLVM intermediate representation.
- The LLVM IR can then be optimized and transformed by middle-end tools that are front-end and source language agnostic. In addition to optimization, these transformations can include obfuscation and other protection techniques.
- Finally, back-end tools convert the LLVM IR to native machine language for any of the many supported platforms and architectures. These back-end tools are agnostic to both the front-end and the middle-end layer, thus facilitating broad platform and language independence of the LLVM architecture.

4.2. Why bitcode protection?

Developers are naturally focused on developing and delivering their features and services to their customers. As a result, security is often neglected to varying degrees at both management and operational levels. Security policies may be insufficient to identify unprotected, weak, or otherwise exploitable code, and under-tuned security parameters. Even if good policies are in place, expertise and resources required to manage and implement security may be insufficient to ensure adequate security protections are in place.

A well-designed cloud-based bitcode obfuscation solution can incorporate security policies and parameters to bridge the gap between in-house expertise and practices and robust security implementations.

- The cloud-based service would allow the definition of security policies and would then verify that these policies are correctly implemented during the application of bitcode obfuscation.

© 2022, SCTE® CableLabs® and NCTA. All rights reserved.

- Audit reporting of security coverage and any identified weaknesses can be provided to appropriate parties in the organization.
- Since the compilation of source code is done prior to the application of bitcode protection; and linking is done afterwards, confidential source code and proprietary libraries do not need to be exposed to the middle-end when applying bitcode protection to an application.
- Bitcode is platform and target agnostic, therefore protection can be applied across all LLVM supported source languages and target architectures. This will allow developers to set global security goals or customize security per target platform.
- Alternatives, such as source-code obfuscation methods can be quickly broken and circumvented with readily available de-obfuscation tools (GitHub, 2022). Bitcode obfuscation is not vulnerable to these same attacks.

4.3. How does it work?

Obfuscation is applied directly to bitcode files via a LLVM middle-end according to a set of protection parameters. No source code, header files, libraries or other dependencies are required in order to obfuscate the bitcode. In addition to the protected bitcode, the middle-end can generate audit reports and logs to assist with the monitoring, implementation, and management of the protected code.

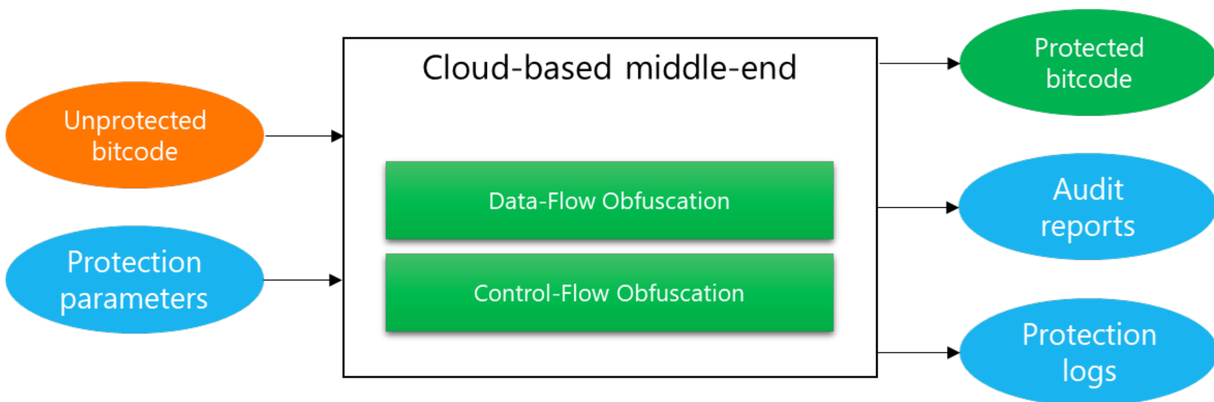

Figure 3 - Cloud-based bitcode obfuscation use-case.

- Unprotected bitcode is sent to a cloud-based middle-end that can apply dataflow and control-flow obfuscation to the bitcode according to a set of supplied protection parameters.
- The middle-end pass manager executes a series of passes to apply the control and dataflow obfuscating transforms. These passes are interleaved with each other and with optimization passes to ensure that the obfuscation complexity matches the tuning requirements specified in the protection parameters.
- The higher the complexity, the higher the runtime overhead, thus the importance of enabling developers to tune the amount of obfuscation applied by the middle-end. Further tuning is possible via LLVM function and inline attributes (LLVM Project, 2022), which can be embedded in the source code and remain readable from the bitcode by the middle-end transform passes.

5. Exploring the usage of bitcode obfuscation

Bitcode obfuscation offers strong protection against tampering and reverse-engineering attacks for software programs running in non-secure environments. These techniques allow tunability to achieve a balance between security and performance; and can be applied to a wide range of applications targeted to

© 2022, SCTE® CableLabs® and NCTA. All rights reserved.

various platforms and devices. In this chapter we explore how obfuscation can be utilized to protect specific types of applications in different industries.

5.1. Mobile applications

Mobile applications (running on iOS or Android devices) often process customer credentials and other sensitive data, while containing differentiated business logic. With stiff competition in the mobile application space as well as an increasing background security of exploits and attacks, it is crucial to protect applications against reverse-engineering by adversaries seeking to gain proprietary knowledge of the app. It is additionally important to prevent adversaries seeking to circumvent authorization and authentication, and to protect against the extraction of sensitive and confidential information.

Secret business logic embedded in the app's source code is considered intellectual property of the app owner and should not be exposed. Generally, all mobile code is susceptible to reverse engineering according to OWASP (OWASP, 2016). Even though security of iOS and Android mobile platforms has always been improving, access to low-level security components is not always available to apps. An attacker will typically download the targeted app from an app store and analyze it within their own local environment using a suite of different tools to statically analyze the code/binary and perform reverse engineering attack.

Let's take an iOS app for example and examine its structure to understand how obfuscation can protect it. The language of choice to develop an app for iOS devices is either Objective C or Swift. They are both compiled into LLVM bitcode binary suitable for Apple App Store submission. The following picture shows the application framework and underlying components of iOS platform (Lucideus, 2019).

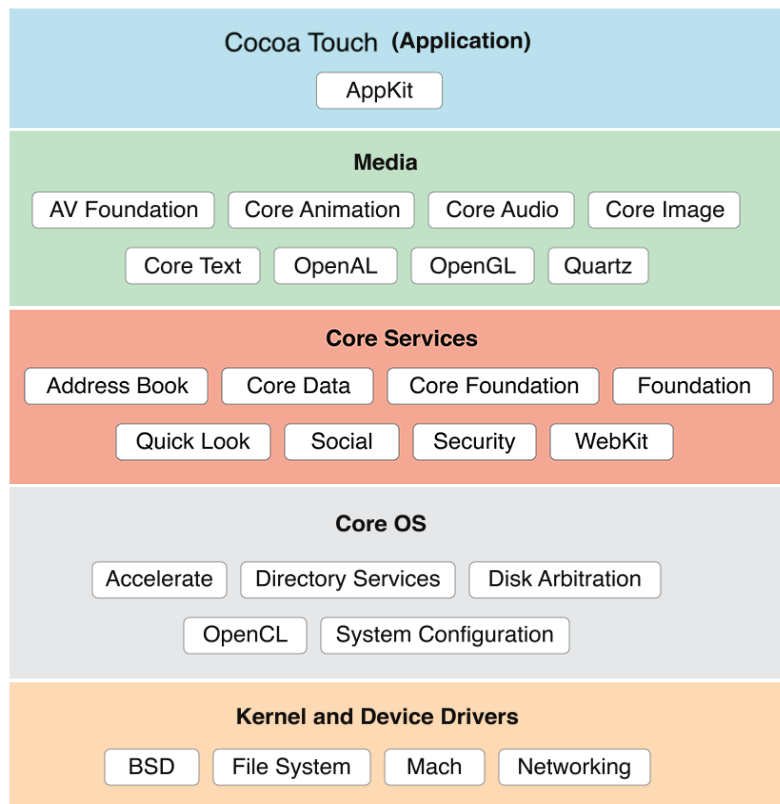

Figure 4 - iOS Mobile App Framework

© 2022, SCTE® CableLabs® and NCTA. All rights reserved.

The app utilizes iOS SDK APIs to access certain functionalities performed by low level libraries while containing a series of business logic and confidential codes within the app layer. App developers are essentially creating complicated puzzles of these APIs glued together with their app's specific codes to build a unique experience to serve the purpose of the app. While there is no secret about the usage of the iOS SDK APIs, the very experience created by the app becomes the intellectual property and requires proper protection. Furthermore, Apple App Store recommends iOS app developers to submit a bitcode version of their app for Apple to perform post-processing on the submitted app for thinning or optimizing purposes. Including the bitcode version of the app will allow Apple to re-optimize the app binary in the future without the need to submit a new version of the app to the Apple App Store (Apple Help, 2020).

An effective obfuscation should be at the bitcode layer to preserve protection against reverse engineering of the binary while still being compliant to Apple App Store. The final iOS app will be placed in a sandbox to create isolation from other apps running on the same device providing some level of security and protection. However, the app is still susceptible to reverse engineering attacks using static analysis offline. Applying obfuscation in the application layer and entry calls to iOS SDK APIs makes it hard for the advisories to identify and follow the app logic. Of course, any software obfuscation comes with an overhead associated with making control flow unpredictable. A bitcode obfuscation level can then be tuned to an acceptable level to balance desire app performance and protection.

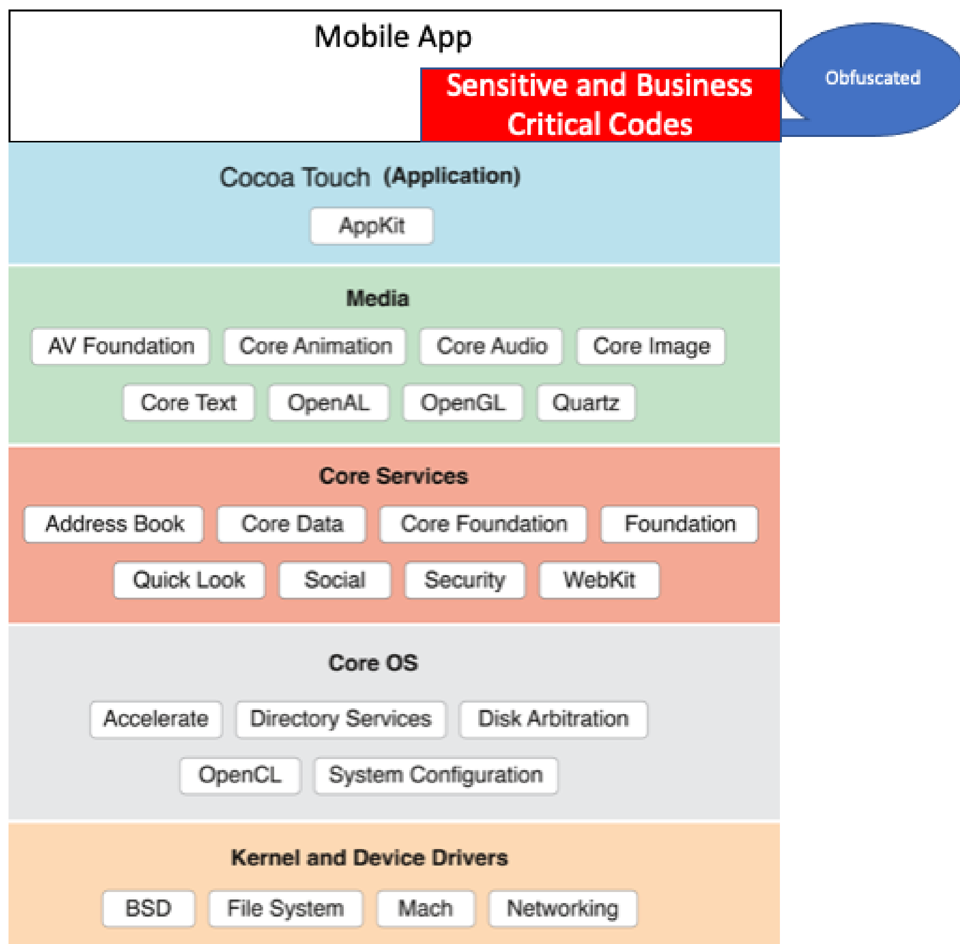

Figure 5 - iOS Mobile App Framework with App Obfuscation

© 2022, SCTE® CableLabs® and NCTA. All rights reserved.

The bitcode obfuscation provides a protection mechanism at the app layer that does not depend on underlying mobile platform security while conforming with Apple App Store requirements. It makes static analysis of the mobile app more complicated even with sophisticated tools.

5.2. Cloud Server Applications and Web Services

With the advent of cloud virtualization, more and more companies are pushing their server applications and web services to public cloud environments where they can clearly benefit from the scale, reliability, and availability of such services. This process requires a comprehensive analysis of cloud readiness of such applications. Companies typically rush to virtualize their server app to cloud without performing much needed due diligence evaluations. As a result, these server apps are posed to attacks when deployed in the cloud. Relying on cloud provider security alone is not an option when it comes to securing an app running in the cloud. Even though cloud providers typically come with a certain level of industry accepted secure environment, the security of data and application remains the ultimate responsibility of the cloud customers in every cloud model.

With exploitation of server apps using reverse engineering methods, an attacker can reveal information about back-end processes, steal intellectual property and gain intelligence needed to perform subsequent code modification (Dolan, Ray, & Majumdar, 2020). Web services are not exempt from these types of attacks either. A viable protection solution would require independence of the app protection from the underlying cloud platform. The notion of In-App protection means that the app is self-contained in terms of protecting its code and data within its binary regardless of its deployed environment.

Modern server applications and web services are typically written in high languages such as Java, JavaScript and GO. As discussed in the introduction, protecting such apps with obfuscation is not effective. Therefore, it’s recommended to move all business critical and secret sauces in the app source code to native languages such as C/C++ for the obfuscation to be more appropriate. For example, if the cloud application is written in Java, there should be a Java Native Interface (JNI) layer to access C/C++ native code that are obfuscated as shown in the following diagram.

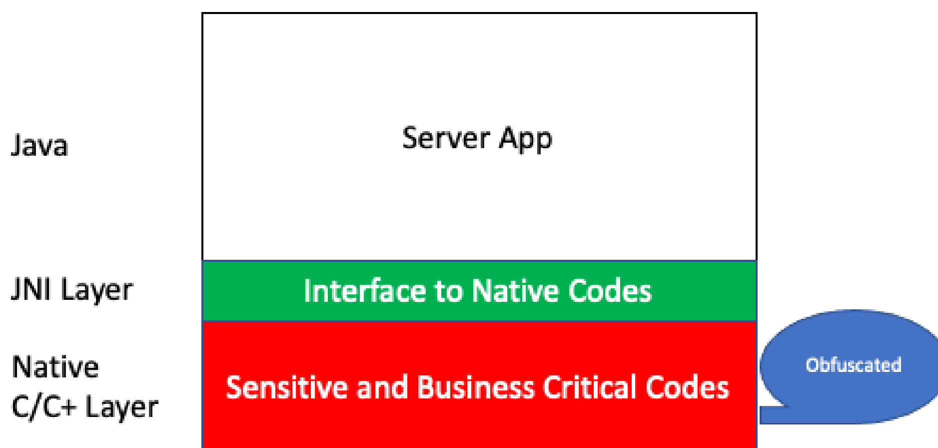

Figure 6 - Sample Cloud App with Obfuscation

5.3. IoT Applications

The rapid expansion of IoT devices creates opportunities for companies to be more innovative with the purpose and scope of their IoT applications. This large ecosystem comprises a variety of devices that are being used in diverse environments including healthcare, industrial control, and homes (Dolan, Ray, &

© 2022, SCTE® CableLabs® and NCTA. All rights reserved.

Majumdar, 2020). From SmartCity to SmartHome to Wearables, IoT devices are entering our world and the list of IoT apps will grow as technology evolves in the years ahead (Placeholder9) These IoT apps are edge devices collecting and processing sensitive data on end users' behavior, the user's devices, habits and their actions in addition to managing Personally identified information (PII).

Naturally the security of IoT apps remains the focus of various standard and industry bodies to regulate and advise. Limited security capabilities along with time-to-market pressure leaves IoT application developers with little or no provisions on securing the app. IoT devices typically come with no hardware security module, making them vulnerable to various attacks. These sophisticated apps analyze the collected data with rich business algorithms all within the IoT device. Even though most attack surfaces are runtime in nature and occur when the IoT app is in action, there is potentially a lot that can be discovered with statically analyzing and reverse engineering these apps.

As a result, the root of trust, device identity, keys and crypto operations conducted by IoT applications are exposed. Bitcode obfuscation can potentially reduce and even eliminate such concerns with IoT applications independent of the IoT device security.

6. Conclusion

In this paper, we introduced bitcode obfuscation as a very effective and powerful tool in protecting software application against reverse engineering attack. We explored different obfuscation techniques such as control flow and data flow obfuscation without accessing the original source code. The idea of operating on the intermediate binary without the need for the source code makes the bitcode obfuscation more practical to be offered as a cloud service (obfuscation-as-service). We also offered a few examples of how this protection can be applied to different software application domains and industries.

© 2022, SCTE® CableLabs® and NCTA. All rights reserved.

Abbreviations

AES	Advanced Encryption Standard
API	Application Programming Interface
GO	Google Programming Language
IoT	Internet of Things
JNI	Java Native Interface
LLVM	Refers to the LLVM compiler infrastructure project.
LLVM-IR	A platform-independent, universal low-level intermediate representation (IR) used by LLVM compilers and tools.
SCTE	Society of Cable Telecommunications Engineers
SDK	Software Development Kit
VM	Virtual Machine

© 2022, SCTE® CableLabs® and NCTA. All rights reserved.

Bibliography

Anderson, L. (2015). A survey of control-flow obfuscation methods used in N-Mesh 2. (October).

Anderson, L. A. (2018, 06 14). *Worldwide Patent No. WO2018106439A1.*

Apple Help, A. (2020). *What is app thinning? (iOS, tvOS, watchOS)*. Retrieved from https://help.apple.com/xcode/mac/current/#/devbbdc5ce4f

Arini Balakrishnan, C. S. (2005). Code Obfuscation Literature Survey. *University of Wisconsin, Madison*, https://pages.cs.wisc.edu/~arinib/writeup.pdf.

Balakrishnan, A., & Schulze, C. (2005). Code Obfuscation Literature Survey. *https://pages.cs.wisc.edu/~arinib/writeup.pdf*, pp. 1-10.

Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., & Yang, K. (2001, apr). On the (im)possibility of obfuscating programs (http://dl.acm.org/citation.cfm?doid=2160158.2160159). *Journal of the ACM, 59*(2), pp. 1-48.

Brekne, T. (2001). *Encrypted Computation.* Department of Telematics.

Collberg, C. S., Thomborson, C., & Low, D. (1997). A taxonomy of obfuscating transformations.

Dolan, A., Ray, I., & Majumdar, S. (2020). Proactively Extracting IoT Device Capabilities: An Application to Smart Homes. (A. Singhal, & J. Vaidya, Eds.) 42-63.

Garg, S., Gentry, C., & Halevi, S. (2013). Candidate indistinguishability obfuscation and functional encryption for all circuits. *Proc. of FOCS ...*.

GitHub. (2022). *GitHub deobfuscation topic*. Retrieved from https://github.com/topics/deobfuscation

Goldwasser, S., & Rothblum, G. N. (2007). On best-possible obfuscation. Springer.

Hosseinzadeh, S., Rauti, S., Laurén, S., Mäkelä, J. M., Holvitie, J., Hyrynsalmi, S., & Leppänen, V. (2018). Information and Software Technology: A systematic literature review. 72-93.

LLVM Doc, D. (2003). *LLVM Bitcode File Format*. Retrieved from https://llvm.org/docs/BitCodeFormat.html

LLVM Project. (2022, 06 20). *LLVM Language Reference Manual*. Retrieved from https://llvm.org/docs/LangRef.html#function-attributes

Lucideus. (2019, Jan 6). *Understanding the Structure of an iOS Application*. Retrieved from https://medium.com/@lucideus/understanding-the-structure-of-an-ios-application-a3144f1140d4

OWASP. (2016). *OWASP M9: Reverse Engineering*. Retrieved from https://owasp.org/www-project-mobile-top-10/2016-risks/m9-reverse-engineering

Wikipedia. (2022, 5 24). *Obfuscation (software)*. Retrieved from https://en.wikipedia.org/wiki/Obfuscation_(software)

© 2022, SCTE® CableLabs® and NCTA. All rights reserved.