Challenges and techniques for code protection in a distributed environment

Ivica Pesovski*, Katerina Zdravkova**
* Brainster Next College, Skopje, Macedonia
** Ss. Cyril and Methodius University, Skopje, Macedonia
ivica@next.edu.mk, katerina.zdravkova@finki.ukim.mk

***Abstract*—The digital transformation is impacting every aspect of our everyday life. The recent social-distancing and isolation measures accelerated the migration of many very traditional processes to online operation. Different vendors develop their own applications for diverse purposes, which expose end-users to significant security risks. Source code protection is crucial for making the internet world a secure environment. This paper discusses code obfuscation, white-box encryption, tamper-proofing, and diversification techniques. The recommendations and discussions in this article contribute toward ensuring a secure digital world. Following them and comprehending their advantages and disadvantages will enable the delivery of code that end users will trust and use.**

Keywords: browser extensions, source code protection, code obfuscation, white-box encryption, tamper-proofing, code diversification, cybersecurity

I. Introduction

During the COVID-19 era, many new software solutions have arisen aiming to support the increased demand for online activities that partially or entirely replace the traditional face-to-face activities. The transformation of methodologies tech companies implement during software delivery has triggered the necessity to enhance source code protection. Unfortunately, not all parties involved in this migration are familiar with source code protection, which has resulted in an unprecedented number of internet scams and hijackings], caused by the massive high-tech pervasiveness and the diversity of delivery models the internet offers [1]. Software applications are embedded in every single electronic device and gadget and even simple devices, like doorbells, are nowadays smart devices that incorporate software that is shipped together with the hardware. The topic that will be covered in this paper is what are the available techniques for protecting these kinds of software applications.

The motivation for this paper was an extension submitted for the Google Chrome browser that was rejected by Google. The response from Google is shown in Fig. 1. The only reason for rejecting this extension was that the code was obfuscated. Obfuscation is a technique for making the source code obscure or unclear in order to protect it against intellectual property infringement or inclusion in third-party libraries. It is particularly important during the creation of open-source applications where the extensions are automatically open-sourced even without our knowledge or control. This browser extension is the event that inspired this paper.

In order to publish an extension to any of the browser web stores, the developer needs to upload the complete code of the extension to the store's web administration panel for approval. If it is approved, it is added to the store from where the end users can add the extension to their browsers. If they choose to add a certain extension, the code that the developer submits is downloaded to the end-user's computer from where the browser loads it on every start-up. The end-user can then open and examine the code just by manually navigating to the folder location where the extension is stored. This qualifies as open-source, even though it is not explicitly classified as such.

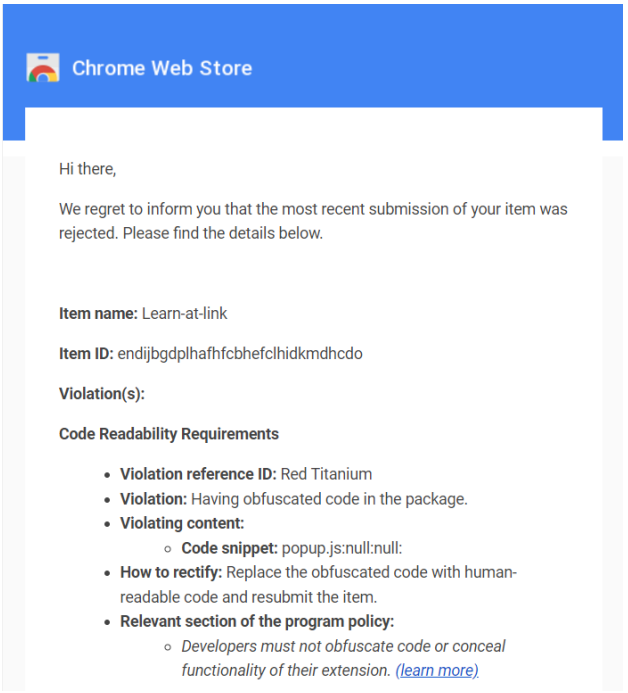

Figure 1. Response from Google after submitting for approval extension with obfuscated code

This paper will be organized into 4 sections. The first section is the introduction which introduces the existing problem that the paper will reason about. The second section contains the methodology and how different vendors have solved the problem. The third section contains the discussion where proposed solutions for the problem are outlined and finally the last section concludes with a brief summary of every exposed approach.

II. Literature review and Methodology

The goal of this paper is to point out the available techniques for protecting source code in a distributed environment. Browser extensions work in such an environment and are a perfect example of the proposed methods. We will not analyze the security concerns which exist in the distributed environments. Instead, we will focus on analyzing the available solutions that prevent their occurrence. Most of the available approaches related to security concerns of browser extensions are general and cover common problems for each browser [2], [3], [4]. In 2016, Arunagiri, Rakhi, and Jevitha made a systematic literature review of security measures for browser extensions [5]. There is also a tiny amount of dedicated research for different browsers. In their paper from 2012, Carlini, Porter Felt and Wagner proposed solutions for the Google Chrome extensions platform which should increase the security of the end-users [6]. Two papers from nearly the same period can be found as clearly aimed toward improving this segment [7], [8]. Moreover, there are several reports of possible security threats in other browsers too, like Safari, Firefox and Edge [9], [10], [11], but knowing that Google Chrome dominates the browser market with 66.31% share before Safari which has around 15% and all others less than 3% [12], it is understandable that the attackers and the solutions will be aimed towards Google Chrome.

Many of the concerns and solutions outlined in the reviewed articles have been overcome since web browsers are constantly being upgraded with security add-ons. This frequent update cycle means that known bugs and concerns will be fixed often, but also that new opportunities for hackers can be revealed often as well. Because of the dynamic nature of the browser world, this paper will focus on the techniques that were and have remained possible throughout the years. The steps outlined here can and should be used unrelated to the fact of how fast browsers are evolving and also can be used in other situations where source code is shipped to the end-users. Some such scenarios include software shipped in IoT devices, publicly accessible devices, etc.

We have inspected the techniques used for protecting source code since the early 1980s. At that time, some of the recommended solutions included using physical devices known as tamper-resistant modules. These modules, later known as hardware security modules, are still used today, particularly in IoT devices. The first publications offering concrete implementation techniques for code security appeared in the 1990s. Techniques like code obfuscation, encryption, and diversification are among them. These strategies are still the main force behind code protection measures in the early 2000s. Fast forward two decades, and the source code protection strategies in use today are still the same, even though the methods for code distribution have transformed. New and enhanced obfuscation and tamper-proofing methods are still state-of-the-art defense mechanisms against source code corruption [13], [14], [15].

We identified four techniques for source code protection that can be used on any occasion, without regarding the nature of the application. These are: code obfuscation, white-box encryption, tamper-proofing and code diversification. The next section will offer a more in-depth analysis of these four techniques.Results and discussion

The four mentioned techniques will be covered in the dedicated subsections that follow.

A. Code obfuscation

The first observed technique covers code obfuscation as a means of attaining code protection. This is a built-in feature in various application and software development frameworks, which is why many developers are unfamiliar with it. Google, for example, ships all of its products with obfuscated client-side code. They have even patented their own algorithm for obfuscation [14]. Conversely, Google decided to block add-on apps and extensions for its products if they are obfuscated, as a result of numerous malicious programs breaching its firewalls due to a lack of code clarity [15].

Obfuscation by definition is modifying the source code in order to make it not readable for humans, yet still executable for the machine. The purpose of obfuscation is to transition from human-readable to gibberish-looking code without affecting the app's performance. The important thing to understand is that obfuscation differs from minifying and uglifying code because they are both reversible processes, whereas obfuscation is not. Still, we must mention that with enough time and effort, almost any code can be reverse-engineered.

Code obfuscation is accomplished by implementing various combinations of code transformations to accomplish creative outcomes. The most used and efficient ones are described below.

- Rename obfuscation

The purpose of renaming is to make source code unclear for the potential reader by renaming variables and methods. New names are usually simple letters or numbers, unprintable or invisible characters or characters from different character sets. This makes the original intent of using variables and methods in source code intelligently masked. This technique not only helps toward protection but also contributes to faster code execution as names can remain small and thus use less memory.

- Control flow obfuscation

This method advises changing conditional statements and iterative constructs so that they still produce valid executable logic, but show non-deterministic results when being reverse-engineered. A visual implementation of this method is shown in Fig. 2 on the next page. The left side presents the normal code execution, while the right side reveals the obfuscated one. If both sequences are run with the same input parameters they will always give the same results, yet the obfuscated variant is extremely hard to decompile by a human. Contrary to the previous technique, control flow obfuscation can impact runtime performance and should not be overused.

- Dummy code insertion

This is a rather simple method of inserting code that doesn’t affect app execution but makes reverse engineering of such code more difficult. The inserted dummy code is usually starving, meaning that machines will never execute it as it is unreachable following the control flow, but it can also be code that is executed and which still doesn’t affect the final outcome in any way.

- Removal of metadata

This is also a simple method to implement and understands removing non-essential metadata from the source code in order to reduce the information available for the potential attacker or debugger.

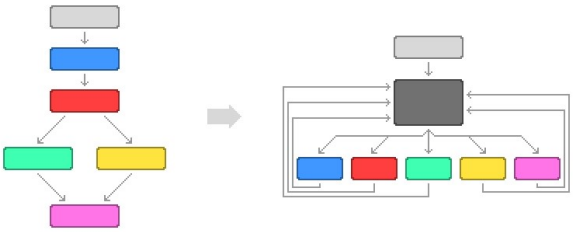

Figure 2. Control flow obfuscation

B. White-box encryption

This protection mechanism enables optimum security on shared platforms by allowing secure applications to run even in hostile environments. Trusted encryption methods typically use encryption keys, and modern software applications usually run on client devices. As a result, the key challenge here is figuring out how to ship encryption keys alongside encrypted software so that it is securely encrypted, can be decrypted for their intended purpose, but cannot be decrypted by malicious users or programs. White-box cryptography (WBC) is the term for this technique [16].

Cryptographic algorithms are by definition public and they provide security guarantees as long as we keep some information secret. This is actually an ancestor to Kerckhoffs's Principle which states that a cryptosystem should be secure even if everything about the system, except the key, is public knowledge [17]. The issue that we are faced with source code protection is how to use cryptography when the complete code is shipped to the client and there is no option for certain code to remain secret. WBC was introduced to provide security in such situations. It is implemented using various mathematical operations and transformations with an aim to tie together the app code and the cryptographic keys so that secure cryptographic operations can be executed. This enables the keys to be shipped together with the application while it also prevents the keys from being observed or extracted from the app. There is no single WBC technique. Usually, such implementations are strictly confidential and known only to the creator.

C. Tamper proofing

Tamper proofing is one of the oldest mechanisms for software protection. Its goal is not to recognize malicious or unwanted software, but to detect changes in the protected software. To detect potential tampering, early computing equipment incorporated hardware devices. Nowadays, we use software to create tamper-resistant applications. The main benefit of this technique is keeping the end-user secure by early and timely detection of possible security threats. Because significant companies' world-driving apps do so by default, we assume the same behavior from every other app we encounter. The technique of using checksums to detect fraud is usually implemented [19]. This technique is especially important for browser extensions since the code is easily available and modifiable by everyone. It is usually implemented in a way in which a hash is stored on a remote server. The hash is generated using a one-way hashing algorithm over all the app's code, Then, whenever the user proceeds to download and install the software or run the application, a match of the hashes is initiated. If there is a match, then the original code from the original author is in use and it hasn't been tampered with since even the slightest change invalidates the hash. Together with obfuscation and white-box cryptography the hash of the original code can be shipped together with the code, so there is no need to validate the hash against a remote computer and everything can be done on the client-side.

D. Diversification

A single point of failure in an application is a big concern and can quickly drive a company out of business. Diversity has its root in nature, where different species of the same kind have different responses to the same stressors differently. This trait has actually allowed humans to persevere in the face of adversity. It is an exciting challenge to recreate similar features in the digital realm. The goal of code diversification is to offer multiple versions of the same software so that malware finding a bug in one instance doesn't affect the whole ecosystem. Its goal is to prevent the `break once, break everywhere` attacks (software monoculture). Its most common usage is in IoT devices since they usually come distributed with the same source code and are therefore susceptible to such attacks. There are different methods to provide code diversification. Some of the most widely used are:

- N-version programming: different teams developing the same application using the same specifications [18]. The developed applications are functionally equivalent, but their source code and possible bugs are completely different since the multiple applications were developed independently. The goal is to reduce the chance of identical software faults occurring in all application instances.

- In-Place Code Randomization: changing the binary executable files to accomplish code randomization. The main goal is to prevent return-oriented programming (ROP) attacks. Because ROP attacks rely on the predictable order of execution of different code blocks, attackers try to exploit it by inserting data on carefully selected memory blocks in order to interfere with code execution [19]. The in-place code randomization makes it almost impossible for the attackers to predict the memory blocks suitable for inserting the malicious data because the code binaries are randomized in-place, just before app execution starts.

- Versioning: keeping different versions of the software. Although the goal of versioning is to have a way to chronologically follow code changes and recover from possible bugs introduced with new software versions, it can also be used as a measure to provide diversified software for the software consumers. Many applications are being offered in different versions and it is often up to the user to decide which version to use. If one version malfunctions, users should be able to easily switch to another version.

III. Conclusion

The repercussions of source code leakage should be sufficient to persuade and open the eyes of those developing software applications whose source code should be shipped to the end-user. The principle "better prevent than cure" should be deeply ingrained in the activities of all individuals and institutions that deliver software applications to customers. In general, source code should not be shipped if it is not necessary, but as we saw, there are certain situations when the developers do not have any control over the manner in which the code they write is distributed.

The different obfuscation techniques discussed in the previous sections are usually combined together to produce a variant of the source code that is hard for the attacker to reverse-engineer but is still straightforward for the machine executing the code. Some of the advised methods contribute toward code optimization and faster execution as they decrease the amount of memory needed (this mechanism is called rename-obfuscation) while others can increase execution times as unnecessary cycles that have no impact on the final outcome are performed with the intent to aggravate code decompiling (like control-flow obfuscation). Developers should consider these things when combining the adviced or any other mechanism.

White-box cryptography is used when sensitive data needs to be operated from within the app. It offers a way to encrypt this data using keys that are embedded in the source code for it to use but are still inaccessible to potentially malicious attackers. There is no unique way to implement this technique for source code protection because algorithms that provide such functionalities are most often strictly confidential and known only to their creator.

Tamper proofing as a protection mechanism makes the end-user sure that the genuine version of the desired software is used. It is most often implemented using checksums calculated with a one-way hashing algorithm on the entire code base so that even the slightest change in the source code will invalidate the checksum and the user will be warned that the application is potentially harmful.

Finally, code diversification guarantees both the end-users and the app developers that a malfunction in one version of the code will not bring the whole system down. This technique is especially useful as it provides a fallback option to both concerned groups in case of an attack or an exploit.

Combining all of these techniques is not mandatory and is redundant in certain scenarios. If the application doesn't utilize encryption techniques and doesn't require any secret keys to be shipped with the source code then white-box cryptography is a surplus. We recommend always combining obfuscation and diversification techniques in order to protect code from being reverse-engineered and limit the impact of potentially harmful operations on end-users if that happens. Tamper-proofing techniques are always a good feature and are becoming a standard in shareable code, especially code distributed through content delivery networks (CDNs), since they guarantee the authenticity and genuineness of the source code.

The actions recommended and discussed in this study provide a strong starting ground for being on the safe side of things. Following these techniques and understanding their benefits and drawbacks will allow delivering software applications that end-users will trust and use. This will lead to attaining business objectives and driving digital transformation with no roadblocks along the way.

References

[1] Weil, T., & Murugesan, S. (2020). IT risk and resilience — Cybersecurity response to COVID-19. IT professional, 22(3), 4-10.

[2] Barth, A., Felt, A. P., Saxena, P., & Boodman, A. (2010). Protecting browsers from extension vulnerabilities.

[3] Sanchez-Rola, I., Santos, I., & Balzarotti, D. (2017). Extension breakdown: Security analysis of browsers extension resources control policies. In 26th USENIX Security Symposium (USENIX Security 17) (pp. 679-694).

[4] Guha, A., Fredrikson, M., Livshits, B., & Swamy, N. (2011, May). Verified security for browser extensions. In 2011 IEEE symposium on security and privacy (pp. 115-130). IEEE.

[5] Arunagiri, J., Rakhi, S., & Jevitha, K. P. (2016). A systematic review of security measures for web browser extension vulnerabilities. In Proceedings of the International Conference on Soft Computing Systems (pp. 99-112). Springer, New Delhi.

[6] Carlini, N., Felt, A. P., & Wagner, D. (2012). An evaluation of the google chrome extension security architecture. In 21st USENIX Security Symposium (USENIX Security 12) (pp. 97-111).

[7] Liu, L., Zhang, X., Yan, G., & Chen, S. (2012, February). Chrome Extensions: Threat Analysis and Countermeasures. In NDSS.

[8] Aravind, V., & Sethumadhavan, M. (2014). A framework for analysing the security of chrome extensions. In Advanced Computing, Networking and Informatics-Volume 2 (pp. 267-272). Springer, Cham.

[9] Verdurmen, J. (2008). Firefox extension security. Raboud University, Netherlands.

[10] Saini, A., Gaur, M. S., & Laxmi, V. (2013, November). The darker side of firefox extension. In Proceedings of the 6th International Conference on Security of Information and Networks (pp. 316-320).

[11] Ursell, S., & Hayajneh, T. (2019, March). Desktop Browser Extension Security and Privacy Issues. In Future of Information and Communication Conference (pp. 868-880). Springer, Cham.

[12] Browser market share. (n.d.). Retrieved May 15, 2022, from https://netmarketshare.com/browser-market-share.aspx

[13] Collberg, C. S., Thomborson, C. D., & Low, D. W. K. (1999). Obfuscation techniques for enhancing software security (Patent No. WO1999001815A1). https://patents.google.com/patent/WO1999001815A1/en

[14] Trustworthy Chrome Extensions, by default. (n.d.). Chromium Blog. Retrieved May 15, 2022, from https://blog.chromium.org/2018/10/trustworthy-chrome-extensions-by-default.html

[15] Beunardeau, M., Connolly, A., Geraud, R., & Naccache, D. (2016). White-box cryptography: Security in an insecure environment. IEEE Security & Privacy, 14(5), 88-92.

[16] Petitcolas, F. (1883). La cryptographie militaire. J. des Sci. Militaires, 9, 161-191.

[17] N-version programming. (2020). In Wikipedia. https://en.wikipedia.org/w/index.php?title=N-version_programming&oldid=957678903

[18] Pappas, V., Polychronakis, M., & Keromytis, A. D. (2012, May). Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In 2012 IEEE Symposium on Security and Privacy (pp. 601-615). IEEE.

[19] Chang, H., & Atallah, M. J. (2001, November). Protecting software code by guards. In ACM Workshop on Digital Rights Management (pp. 160-175). Springer, Berlin, Heidelberg.