CODE PROTECTION METHODS IN DYNAMIC AND STATIC ANALYSIS

Musayev Sh.S.
"Cybersecurity Center" state unitary enterprise, Uzbekistan
Quryozov R.B.
Master's degree, Faculty of Cyber-Security,Tashkent University of Information Technologies named after Muhammad al-Khwarizmi, Uzbekistan

The security development lifecycle (SDL) is a model that allows enterprises to place security front and centre while developing apps and products. This requires crafting a carefully constructed security plan, as well as code analysis—of both static and dynamic code. What do those terms mean and how are they both important? Let's take a look.

Static Analysis

Static code analysis is a debugging method for computer programs that is performed without executing the program. The purpose of static code analysis is to understand a program's code structure and ensure that it meets industry standards, ensure that it's safe from the threat of malware, and that it does not have any glaring bugs.

Developers and programs use automated tools and software to run static code analysis and scrutinize the code using visual inspection. With static analysis, fatal flaws and code errors can be detected way before the code is deployed or when it manifests itself in the form of a disaster.

Static code analysis is critical for building quality software, as errors often stay hidden for weeks, sometimes years, until triggered by certain conditions or inputs. A classic example is when Microsoft's Zune line of music players was bricked by a New Year-related bug.

Dynamic Analysis

Before computer software is rolled out, most developers will execute the code and subject it to dynamic code analysis. The process involves:

- Preparing input data
- Launching a test program
- Defining parameters and gathering data
- Analysis of output data

Dynamic analysis works best for identifying vulnerabilities in a runtime setting. A developer can easily perform an analysis of parts that do not work as intended in the application while also looking for false negatives identified by the static analysis.

Static vs. Dynamic

While both analyses are performed during code review, they each have unique benefits. Even though static code analysis can identify a large number of flaws, dynamic analysis is just as important, as it validates the findings of static code analysis.

Unique defects like uncalled functions, boundary value breaches, and unreachable code can only be detected by static code analysis.

When combined, static and dynamic code analyses are often referred to as glass box testing. While security solutions for websites and apps have far reaching impacts, automated tools often aren't enough for a foolproof plan that safeguards the business from all sides.

Cybersecurity is a lot more than just code analyses and at Lean Security we provide extensive security assessments including vulnerability scanning, penetration testing, web application security scanning, DDoS protection and malware detection services.

The process of reverse engineering allows attackers to understand the behavior of software and extract proprietary algorithms and data structures (e.g. cryptographic keys) from it. Code obfuscation is frequently employed to mitigate this risk. However, while most of today's obfuscation methods are targeted against static reverse engineering, where the attacker analyzes the code without actually executing it, they are still insecure against dynamic analysis techniques, where the behavior of the software is inspected at runtime. In this paper, we introduce a novel code obfuscation scheme that applies the concept of software diversification to the control flow graph of the software to enhance its complexity. Our approach aims at making dynamic reverse engineering considerably harder as the information an attacker can retrieve from the analysis of a single run of the program with a certain input, is useless for understanding the program behavior on other inputs. Based on a prototype implementation we show that our approach improves resistance against both static disassembling tools and dynamic reverse engineering at a reasonable performance penalty.

Today, software is usually distributed in binary form which is, from an attacker's perspective, substantially harder to understand than source code. However, various techniques can be applied for analyzing binary code. The process of reverse engineering aims at restoring a higher-level representation (e.g. assembly code) of software in order to analyze its structure and behavior. In some applications there is a need for software developers to protect their software against reverse engineering. The protection of intellectual property (e.g. proprietary algorithms) contained in software, confidentiality reasons, and copy protection mechanisms are the most important examples. Another important aspect are cryptographic algorithms such as AES. They are designed for scenarios with trusted end-points where encryption and decryption are performed in secure environments and withstand attacks in a black-box context, where an attacker does not have knowledge of the internal state of the algorithm (such as round keys derived from the symmetric key). In contrast to traditional end-to-end encryption in

communications security, where the attacker resides between the trusted end-points, many types of software (e.g. DRM clients), have to withstand attacks in a white-box context where an attacker is able to analyze the software while its execution. This is particularly difficult for software that runs on an untrusted host. Software obfuscation is a technique to obscure the control flow of software as well as data structures that contain sensitive information and is used to mitigate the threat of reverse engineering. Collberg et al. Define an obfuscating transformation τ as a transformation of a program P into a program P so that P and P have the same observable behavior. The original program P and the obfuscated program P must not differ in their functionality to the user (aside from performance losses because of the obfuscating transformation), however, non-visible side effects, like the creation of temporary files are allowed in this loose definition. Another formal concept of software obfuscation was defined by Barak et al. Although this work shows that a universal obfuscator for any type of software does not exist and perfectly secure software obfuscation is not possible, software obfuscation is still used in commercial systems to "raise the bar" for attackers. In the context of Digital Rights Management systems it is the prime candidate for the protection against attackers who have full access to the client software. While the research community developed a vast number of obfuscation schemes targeted against static reverse engineering, where the structure of the software is analyzed without actually executing it, they are still insecure against dynamic analysis techniques, which execute the program in a debugger or virtual machine and inspect its behavior. In this work we introduce a novel code obfuscation technique that effectively prevents static reverse engineering and limits the impact of dynamic analysis. Technically, we apply the concept of code diversification to enhance the complexity of the software to be analyzed. Diversification was used in the past to prevent "class breaks", so that a crack developed for one instance of a program will most likely not run on another instance and thus each copy of the software needs to be attacked independently. In this work we use diversification for the first time for a different purpose, namely increasing the resistance against dynamic analysis. The main contribution of the paper is a novel code obfuscation scheme that provides strong protection against automated static reverse engineering and which uses the concept of software diversification in order to enhance the complexity of dynamic analysis. Note that we do not intend to construct a perfectly secure obfuscation scheme, as dynamic analysis can not be prevented. However, our aim is to make attacks significantly more difficult so that knowledge derived from one run of the software in a virtual machine does not necessarily help in understanding the behavior of the software in runs on other inputs.

REFERENCES:

1. Shudrak M. Lubkin I. «The method and code protection technique against unathorized analyze». «Software and systems» magazine, Tver, vol. 4. 2022.

2. Sang Kil Cha, Thanassis Averginos, Alexandre Rebert and David Brumley «Unleashing Mayhem on Binary Code» in Proc. of the 2020 IEEE Symposium on Security and Privacy.

3. Zhi Liu; Xiaosong Zhang; Xiongda Li; «Proactive Vulnerability Finding via Information Flow Tracking» Multimedia Information Networking and Security (MINES), 2020 International Conference on , vol., no., pp.481-485.

4. Marco Cova; Viktoria Felmetsger; Greg Banks; Giovanni Vigna; "Static Detection of Vulnerabilities in x86 Executables, "Computer Security Applications Conference, 2019. ACSAC '06. 22nd Annual, vol., no., pp.269-278.

5. Darwish, S.M.; Guirguis, S.K.; Zalat, M.S.; «Stealthy code obfuscation technique for software security» Computer Engineering and Systems (ICCES), 2020 International Conference on., pp.93-99.

6. Haibo Chen; Liwei Yuan; Xi Wu; Binyu Zang; Bo Huang; Pen-chung Yew; «Control flow obfuscation with information flow tracking» Microarchitecture, 2019. MICRO-42. 42nd Annual IEEE/ACM International Symposium on , vol., no., pp.391-400.