2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) | 978-1-6654-2603-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISSREW53611.2021.00083

SourceSnippet2Binary: A Method for Searching Vulnerable Source Code Snippets in Binaries

Qinqin Wu *, Hao Huang *, Yi Tang *, Zhenwei Gu *, and Ang Jia †
*Department of Cyber Security, Guangdong Power Dispatching and Controlling Center, Guangdong Power Grid Co., Ltd
†School of Cyber Science and Engineering, Xi'an Jiaotong University, China
149679600@qq.com; 380772757@qq.com; tangyicsg@hotmail.com; 847748468@qq.com; jiaang@stu.xjtu.edu.cn

***Abstract*—Vulnerability detection, known as research to detect whether a target code contains vulnerabilities, is often conducted at source code level or binary code level. However, in some cases, the vulnerable code is usually at source-level as they are found by software developers, and target code is usually in binary-level as they are released to be executed in different operating systems, which pose new requirements for vulnerability detection. In this paper, we illustrate a new method named SourceSnippet2Binary, trying to use vulnerable source code snippets to search whether a binary code contains the vulnerability. By proposing such a method, we hope further studies to spend more effort in source2binary vulnerability detection and make vulnerability detection more comprehensive.**

***Index Terms*—vulnerability detection, source2binary, code snippets**

I. Introduction

In recent years, vulnerabilities are increasing in their amount and bringing severe financial losses to software users. On the one hand, 18325 vulnerabilities have emerged during 2020 [1], almost three times as many as in 2016. On the other hand, software bugs and other errors related to bad software quality have cost businesses and organizations more than 1.7 trillion in financial losses in 2017, which corresponds to an estimated total of 268 years of downtime [2].

To help software get rid of vulnerabilities, many works have been carried out, of which one way is to search whether a piece of code with this vulnerability appears in the target code. According to the format of the target code focused on, these researches can be classified as source-code-level methods and binary-code-level methods. However, in some real cases, the vulnerable code is usually at the source level as they are found by software developers, and the target code is usually in binary-level as they are released to be executed in different operating systems. Existing methods cannot be directly applied to this kind of vulnerability detection scenario.

To make it worse, the source-level vulnerable code is usually code snippets as shown in CVE summaries. It is often some fixes in several functions, which is hard to compile without human effort. And the vulnerabilities are hidden in the logic detail of these code snippets, which is hard to extract without compilation.

To search such vulnerable source code snippets in target binaries, we have to overcome the following challenges: First, the semantic difference between source and binary. As the source code and binary code are two different formats of code representations, we cannot directly compare source code to binaries. And the code snippets are just part of code, which prevents them to be compiled into binaries. Second, the representation of logic in vulnerabilities. To detect whether a binary contains the vulnerable code, we need to characterize the behavior of the vulnerable code in detail. What representation is suitable for vulnerability detection is unknown.

To solve such a source2binary code search task, we propose our method, SourceSnippet2Binary. In our method, we first find a way to translate source snippets and binary into a uniform representation as LLVM IR. And then extract the data flows of the critical path in the vulnerability code and target binary code and obtain the data flow strands. Finally, after the normalization of these strands, we calculate the similarities between source code and binaries and then give our judgment of whether this binary contains vulnerability code.

The contributions of this paper are: (a) We propose a method to communicate non-complete source code and binary code, making them feasible to be compared. (b) We extract a representation by data flow analysis, which helps to extract the semantics of vulnerabilities and conduct detection.

II. SourceSnippet2Binary

Figure 1 shows the overview of SourceSnippet2Binary. It consists of three main components: Convert to LLVM IR, Strands Extraction, and Function similarity calculation.

A. Convert to LLVM IR

As source snippets and binary are two distant code representations, an intermediate representation is needed for comparison between them. In our work, we select LLVM IR [3], which is the intermediate representation of the LLVM compiler. LLVM IR is in SSA (Single-Static Assignment) form, which is convenient for conducting data flow analysis.

As source code snippets are not complete, they cannot be directly compiled into LLVM IR, so another process must be applied. We assume source code snippets are function snippets without global variable definition and other called functions, which is often the case of CVE summary that describe which function is vulnerable. For that single function, we use *Psyche-C* [4] to complement the missing components. *Psyche-C* will add missing functions or struct definitions but won't provide

This work is sponsored under Science and Technology Project of China Southern Power Grid Corporation under Grant No.036000KK52190037.

978-1-6654-2603-9/21/$31.00 ©2021 IEEE
DOI 10.1109/ISSREW53611.2021.00083

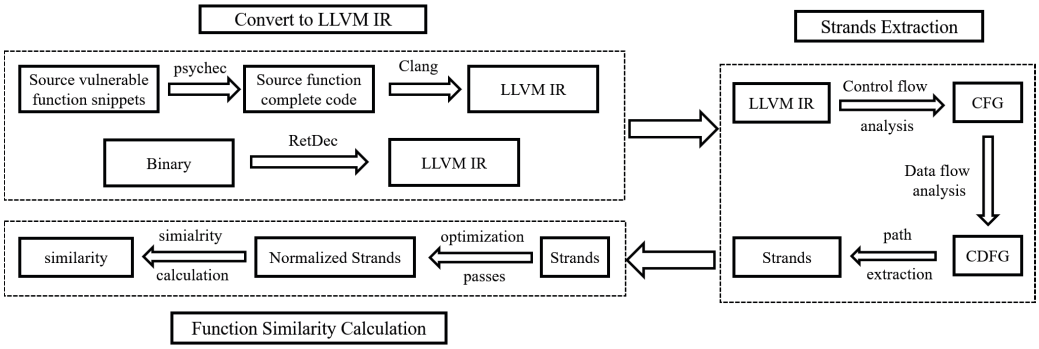

Fig. 1. Overview of SourceSnippet2Binary

a detailed function body, which only facilitates compiling the source code to LLVM IR and the resulted source code cannot be linked during full compilation. Note that *Psyche-C* only adds necessary information, thus it won't introduce noises. Then we can use clang to convert this complemented code into LLVM IR.

For binaries, as they may be compiled for different architectures, using different compilers and options, the content of binaries will diff greatly. Disassembling binaries into assembly is not enough, as binaries in different architectures are using different assembly instruction sets, which makes it complicated to process. In our method, we leverage existing decompilers such as *RetDec* [5], which aimed to translate binaries into source code, and we leverage its intermediate production, decompiled code in LLVM IR. Note that directly decompiling binaries into source code for comparison is not feasible, as the produced source code is much different from the origin code.

Through the above steps, we are able to communicate source and binary in the format of LLVM IR.

B. Strands Extraction

As the semantics of vulnerable code are hidden in its logic for processing variables, we extract its CDFG (control data flow graph) to represent its behavior. To do that, we first extract CFG (control flow graph) of LLVM IR. Then for each basic block, we further extract its DFG (data flow graph) by conducting backward data flow analysis. As LLVM IR is an SSA format representation, there is a special way to process values coming from different branches named *phi* instruction. We conduct a different analysis for instruction *phi* as it will only accept the value from the corresponding control flow, thus we associate *phi* with its formal control flows.

To represent the logic of the vulnerable code, we extract the paths that vulnerable code covered in the CDFG which we named strands. If no specific path is highlighted, we tend to extract data flows associated with function calls. As many vulnerabilities are associated with function invocations and lack proper value check before that, we regard data flow paths to function calls as critical paths of the vulnerable function. And In our early results, matching source code with binaries using function-call-related strands can provide more than 80% accuracy.

C. Function Similarity Calculation

After we process the LLVM IR through control data flow analysis, we get two sets of strands belonging to source snippets and binary code. Before conducting a comparison between them, we first apply normalization to make them more similar.

We leverage the existing optimization passes in LLVM compiler such as " common subexpressions elimination" and "combine redundant instructions" (activated by -early-cse and -instcombine respectively) to optimize strands.

After normalization, we use the following equation to calculate the similarities between optimized strands.

$$similarity = \frac{|Source \bigcap Binary|}{|Souce|} \tag{1}$$

Once getting the similarities, we will compare it with the predetermined threshold, and give the judgment of whether this binary contains the vulnerability.

III. Conclusion

In this paper, we illustrate a new vulnerability detection task, searching source vulnerable code snippets in binaries. To do that, we propose a new method, SourceSnippet2Binary, which is based on LLVM IR and use strands to calculate similarities. In the future, we will complement our work with thorough results and evaluations, and we hope this work could help promote the development of vulnerability detection.

References

[1] Browse cve vulnerabilities by date. https://www.cvedetails.com/browse-by-date.php. 2021
[2] 1.7 trillion: Financial losses caused by software failures in 2017. https://www.alithya.com/en/insights/1.7-trillion-financial-losses-caused-by-software-failures-in-2017. 2020
[3] The LLVM Compiler Infrastructure. https://llvm.org/. 2021
[4] ltcmelo/psychec:A compiler frontend for the C programming language. https://github.com/ltcmelo/psychec. 2021
[5] RetDec. https://retdec.com/. 2021
[6] David, Yaniv, Nimrod Partush, and Eran Yahav. "Similarity of binaries through re-optimization." Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2017.