2018 IEEE International Conference on Software Quality, Reliability and Security Companion

Automatic Exploit Generation for Buffer Overflow Vulnerabilities

Luhang Xu
National University of Defense Technology
NUDT
Changsha, China
me@xuluhang.cn

Weixi Jia
National University of Defense Technology
NUDT
Changsha, China
wxjia92@163.com

Wei Dong
National University of Defense Technology
NUDT
Changsha, China
wdong@nudt.edu.cn

Yongjun Li
Northwestern Polytechnical University
NWPU
Xi'an, China
lyj@nwpu.edu.cn

***Abstract*—Buffer overflow vulnerabilities are widely found in software. Finding these vulnerabilities and identifying whether these vulnerabilities can be exploit is very important. However, it is not easy to find all of the buffer overflow vulnerabilities in software programs, and it is more difficult to find and exploit these vulnerabilities in binary programs. This paper proposes a method and a corresponding tool that automatically finds buffer overflow vulnerabilities in binary programs, and then automatically generate exploit for the vulnerability. The tool uses symbolic execution to search the target software and find potential buffer overflow vulnerabilities, then try to bypass system protection by choosing different exploiting method according to the different level of protections. Finally, the exploit of software vulnerability is generated using constraint solver. The method and tool can automatically find vulnerabilities and generate exploits for three kinds of protection: without system protection, with address space layout randomization protection, and with stack non-executable protection.**

Keywords—binary program; symbolic execution; automatic exploit generation

I. INTRODUCTION

The security and reliability of the software is very important, and the buffer overflow vulnerability is an important kind of vulnerabilities that destroys the security of the software. Most software systems have lurked buffer overflow problems, the hackers and security researchers have great interest in that, especially in the case of no source code situation. They can mine and exploit vulnerabilities directly on the binary program with buffer overflow.

To exploit the buffer overflow vulnerabilities of binary programs, first, we need to reverse the binary codes to get basic program information such as control flow, data dependence and so on. According to the program logic, people find the location of the buffer overflow that may exist in the program. Next, by tracing the program execution, they infer an input to arrive bug location and trigger a potential buffer overflow vulnerability. Finally, they design a special program input, to achieve an unbelievable effect through the buffer overflow vulnerability that found.

The above process is very complicated. In the absence of source code, it is very easy to make mistakes in manual way, and it is also time consuming to discover and verify an available buffer overflow vulnerability. Our work is dedicated to automate the process, which searches and exploits the vulnerabilities automatically.

The buffer overflow vulnerability is one of the most widely existing and serious harm to the modern software, so the security strategy of modern operating system had many prevention measures for buffer overflow, alleviating the harm of buffer overflow vulnerability to software and system. The hackers didn't stop there. They constantly studied the technologies that can bypass these protective means, so that they could continue to exploit the buffer overflow vulnerability under the protection of modern system.

Based on the modern operating system's protective measures, we studied how to automatically exploit the buffer overflow vulnerabilities and achieve the ability of vulnerability mining as deeply as possible, to ensure the security and reliability of the software and system.

The tool in this paper is implemented based on symbolic execution technology and uses the test case generation ability of constraint solving to implement the automatic exploit generation of vulnerability. The main contributions of the paper include:

- We propose an automatic exploit generation method based on symbolic execution for buffer overflow vulnerabilities.
- We propose an exploit method of automatically bypassing a series of security protections of modern operating system.

978-1-5386-7839-8/18/$31.00 ©2018 IEEE
DOI 10.1109/QRS-C.2018.00085

Authorized licensed use limited to: XIDIAN UNIVERSITY. Downloaded on June 24,2021 at 08:43:19 UTC from IEEE Xplore. Restrictions apply.

- We implement the above two methods as an available software tool.

Setion II presents an overview of our approach. Section III describes the method of mining buffer overflow vulnerabilities in detail. Section IV describes the bypassing strategies according to different system protection levels and studies the automatic exploit generation method in detail. Section V describes the implementation of the method and the experiment result. Section VI summaries the related work in automatic exploit generation and compares the differences between our tool and AEG [2]. Section VII concludes the contributions and future work of our method.

II. Approach Framework

The approach proposed in this paper is mainly divided into two parts: automatic vulnerability mining and automatic exploit generation. In addition, the approach also includes pre-processing before main process and exploit verification after main process. Fig.1 describes the overall framework of the approach.

Firstly, the pre-processing will obtain the basic properties of the operating system and analyzed program. It then provides the information to the following steps for using. Pre-processing operations mainly include: disassembling binary code to get the IR (Intermediate representation) and CFG (Control Flow Graph) of the program, checking whether there is stack non-executable protection or stack protection for the program, and checking whether there is ASLR (Address Space Layout Randomization) protection in the operating system. In addition to these necessary attribute checking, pre-processing is also used to check program information that will be used in the exploit process, such as whether there is assembly code 'JMP ESP' and string 'SH' in the binary code.

Secondly, the automatic vulnerability mining module uses symbolic execution to search the paths of the program and excavate vulnerabilities based on the control flow graph of binary code. In our work, we use Breadth-First Search (BFS) strategy to traverse the control flow graph of programs. It will add a program state constraint and check whether the program's current state is defective if passing any basic block of program. When we find potential vulnerabilities in program, we will record the path from the program entry point to the current defect location and submit to next step. Due to the problem of path space explosion in symbolic execution, we will limit the search depth.

Next, the exploit generation of vulnerabilities is divided into two parts: bypassing system protection and automatic exploit code generation. The step of bypassing system protection determines the protection level of the system by checking the system attribute and program attribute information acquired in the pre-processing stage. After selecting the system protection bypass strategy according to the type of system protection, the step of automatic exploit generation will aggregate all the constraints and solve them by SMT solver to get the final exploit input of vulnerability. At present, there are two kinds of system protection strategies that can be bypassed: stack non-executable protection and address space layout randomization protection. We use the way of returning to system library to bypass the stack non-executable protection. We use jump to special assembly instruction to bypass the address space layout randomization protection. Finally, we use constraint solver to calculate the input that is consistent with the above bypass method. The input is what we want, an exploit of vulnerability.

Finally, the validation step is carried out to verify the generated exploit input by executing binary code with this exploit. Through the verification part, we can reduce the false positive of the tool to zero.

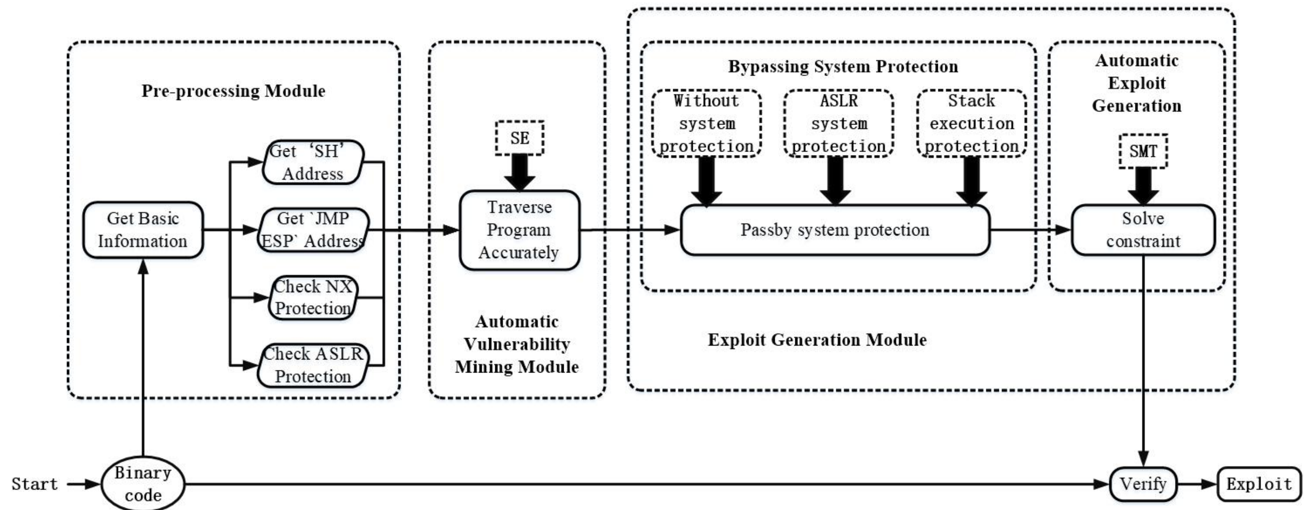

Fig. 1. Framework of proposed apporach

III. Mining Buffer Overflow Vulnerability

The process of automatic vulnerability mining is based on symbolic execution. Symbolic execution is one of the most important technologies for software analysis. It can be used to

Authorized licensed use limited to: XIDIAN UNIVERSITY. Downloaded on June 24,2021 at 08:43:19 UTC from IEEE Xplore. Restrictions apply.

traverse program paths accurately, generate corresponding test input for every path passed, and provide support for stain analysis and fuzzy testing. Because symbolic execution records a lot of program state information and stores branch nodes for every different path, there will be the problem of path space explosion. Here we will use the Breadth-First Search and depth restriction strategy to prevent the mining and search process of software inexhaustibly.

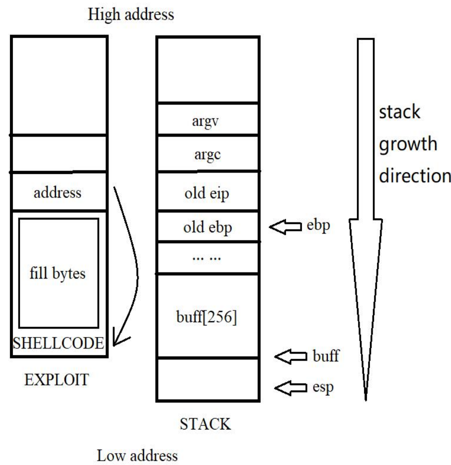

Fig. 2. Exploit without Protection

As shown in Fig.2, buffer overflow vulnerability is caused by data overlaying on the stack or heap, which is larger than the buffer size. Because the buffer overflow on the heap is very complex, we only consider the buffer overflow on the stack in this paper. The most common way of exploiting stack buffer overflow vulnerabilities is executing arbitrary instructions by control-flow hijacking. We achieve the purpose of controlling the flow hijacking, and finally achieve the effect of arbitrary instruction execution by hijacking the EBP pointer and the key stack area near it. So, we firstly search software vulnerabilities that can be covered to the EBP pointer and the stack area near it. The program is loaded into virtual memory before it is executed, and not all memory areas can be used by programs. Once a buffer overflow occurs, we can use the input hijacked instruction pointer (IP) to jump to any location, which conflict with the nature of the program. If the instruction pointer is in an unconstrained state which can point to any program position, we believe it is a typical buffer overflow vulnerability, and it can be exploited to achieve the effect of control-flow hijacking.

As shown in Fig.3, we obtain the intermediate representation of programs by reverse engineering technology. Then we convert the intermediate representation into control flow graph. In the program pre-processing, we obtain the reachable memory space interval when the program executes normally. We use the symbolic execution to traverse the program, and check whether the location of IP pointer is in normal interval. If the instruction pointer is over the maximum value of the reachable memory address or below the minimum reachable memory address, we regard the program state point as a typical buffer overflow vulnerability point that can be exploited. After the discovery of a buffer overflow vulnerability, the current program execution path and current program state constraints will be recorded. This information will be submitted to the next step which will generate exploit of vulnerability.

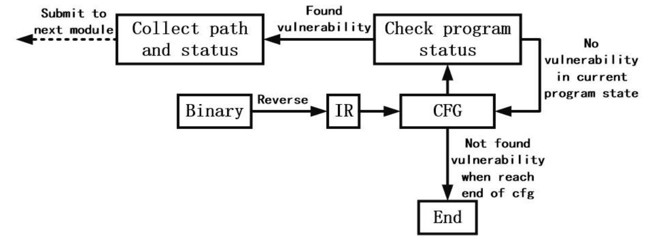

Fig. 3. Framework of mining buffer overflow vulnerability

Since the tool is based on binary program level, there are a large amount of binary codes between the entry point of the program and the entry point of the main function. These codes are automatically generated by the compiler, so they have no possibilities containing buffer overflow vulnerability, and we do not need to check the state of these code. It is necessary to skip these codes.

Symbolic execution is limited while dealing with system library functions. The symbol execution tool used in this paper, ANGR [1], rewrites many system library functions, making the symbol execution process more capable of handling the system library functions.

IV. Exploit Generation

Through the above searching process, the vulnerability location will be found in a program containing a specific stack buffer overflow vulnerability. Then we will generate the exploit of the vulnerabilities and implement corresponding bypassing methods for special system protection strategies. This process contains two components: system protection bypassing and automatic exploit generation.

A. Bypass System Protection

Because modern operating systems contain many kinds of system level protection strategies, this section discusses the bypassing strategy for these system protections. Our tool can bypass two system protections: stack non-executable protection, address space layout randomization protection. The corresponding bypassing ways are: returning to system library and jumping to the specific assembly instruction.

a) **Without System Protection**. As shown in Fig.2, when system doesn't exist any protections, the exploit of buffer overflow vulnerabilities is to inject shellcode at the start of the buffer zone and overlay the location which saved EIP on the stack with the buffer header address. Because there are no system protection measure, the buffer header address and EIP location offset from the starting position of the buffer can be obtained in advance. In this case, it is easy to construct exploit of buffer overflow vulnerabilities.

b) **Stack Non-Executable Protection**. Stack non-executable protection is a system level protection, which is set by the compiler parameter options. The main idea is to set the permissions of the program stack area only for reading and

Authorized licensed use limited to: XIDIAN UNIVERSITY. Downloaded on June 24,2021 at 08:43:19 UTC from IEEE Xplore. Restrictions apply.

writing without executing permissions. Under this setting, even if the control flow is hijacked to the stack position, it will be failure to exploit without the executing permissions, thus it ensures the security of the software.

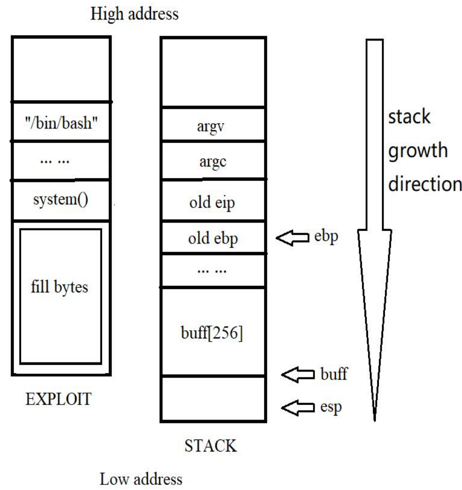

Fig. 4. Exploit under Stack Non-Executable Protection

Bypassing Stack Non-Executable Protection. For stack non-executable protection, we use the way of returning to system library to bypass. As shown in Fig.4, the main idea of returning to system library is that when the stack structure is overlaid, the contents of instructions stored on the stack are not directly executed, but the execution position of the program is set to the function of the system library, and the parameters of the function are constructed at another proper position on the stack, thus the exploit of the software vulnerability is achieved.

c) **Address Space Layout Randomization Protection**. Besides the stack non-executable protection, our tool can also bypass the address space layout randomization protection. In the previous case, the various address areas of the program are fixed. However, the idea of the address space layout randomization protection is to dynamically modify the various addresses of each execution process, so we can't get a fixed function address and stack address ahead of time.

Bypassing Address Space Layout Randomization Protection. We use the strategy that returns to specific assembly instruction to bypass the address space layout randomization protection. As shown in Fig.5, it will no longer statically jump to a certain fixed address but jump to the ESP stack register to achieve a vulnerability exploit process. However, the premise of using this way is that there is "JMP ESP" assembler instruction in the program. Many experiments show that there is a great possibility of existing "JMP ESP" assembly instruction in the program.

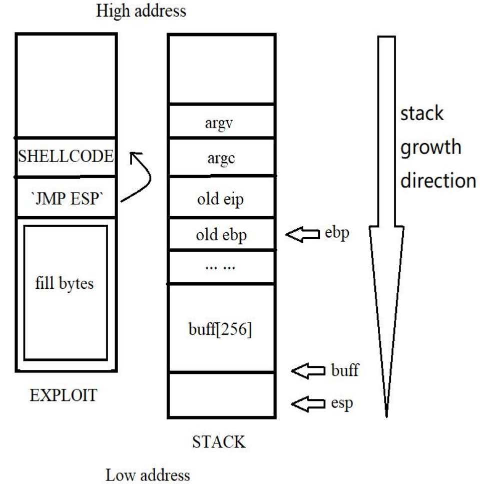

Fig. 5. Exploit under ASLR Protection

To sum up, we implement a vulnerability exploit bypass method of three kinds of system protection strategy: system without protection, the stack non-executable protection, and the address space layout randomization protection. To express the different system protection situations more accurately, we define the following BNF paradigm expression in TABLE I. All the paradigm are expressions like: "<VAL>==<CONTENT>". Among them, the form of "<VAL>" is expressions like: "value(EBP+<NUKBER>)", which represents the value of base stack pointer EBP offset NUMBER constant position. The "<CONTENT>" part represents five other situations, namely: "<NUMBER>" represents integer that can be divided by 4, "<FUNCTION>" represents a function name, "<STRING>" represents a string, "<CONSTANT>" represents a constant without byte code '\x00', "<COMMAND>" represents an assembly instruction, "<SHELLCODE>" represents a shellcode byte string.

TABLE I. BNF PARADIGM EXPRESSION

<Specification>	::=	***<VAL>"=="<CONTENT>***
<VAL>	::=	value(EBP+***<NUKBER>***)
<CONTENT>	::=	address(***<FUNCTION>***)
		| address(`***<COMMAND>***`)
		| address("***<STRING>***")
		| ***<CONSTANT>***
		| "***<SHELLCODE>"***
<NUMBER>	::=	</* Integer that can be divided by 4 */>
<FUNCTION>	::=	</* Function Name */>
<STRING>	::=	</* String */>
<CONSTANT>	::=	</* Constant without '\x00' */>
<COMMAND>	::=	</* Assembly Instruction */>
<SHELLCODE>	::=	</* Shellcode Byte String */>

TABLE II shows the bypassing protocol according to different system protection levels and program attributes. In TABLE II, "-" means there is no need to be discussed, because in the case of stack non-executable protection, the existence of "JMP ESP" assembly instruction in program does not affect the

Authorized licensed use limited to: XIDIAN UNIVERSITY. Downloaded on June 24,2021 at 08:43:19 UTC from IEEE Xplore. Restrictions apply.

exploit generation process of vulnerability; "X" indicates that in this case, the software vulnerability can't be exploited. This is the situation without "JMP ESP" assembler instructions in address space layout randomized protection.

TABLE II. STATUTE IN DIFFERENT CIRCUMSTANCES

Protection level	Attribute of binary code	paradigm
Without protection	with 'JMP ESP' in binary	value(ebp+0)==CONSTANT value(ebp+4)==address(`COMMAND`) value(ebp+8)=="SHELLCODE"
	without 'JMP ESP' in binary	value(ebp+4)==address(FUNCTION) value(ebp+12)==address("STRING") value(ebp+0)== CONSTANT value(ebp+8)== CONSTANT
Address space layout randomization protection	with 'JMP ESP' in binary	value(ebp+0)==CONSTANT value(ebp+4)==address(`COMMAND`) value(ebp+8)== "SHELLCODE"
	without 'JMP ESP' in binary	X
Stack non-executable protection	-	value(ebp+4)==address(FUNCTION) value(ebp+12)==address("STRING") value(ebp+0)== CONSTANT value(ebp+8)== CONSTANT

B. Automatic Exploit Generation

The automatic exploit generation is a process that combines the path and the state of the vulnerability mining module with the bypass method determined by the system protection bypassing module and uses the constraint solver to automatically generate exploit of software vulnerability.

The stack structure, the memory structure in the program is particularly complex, and there are many registers. The program state information and path information obtained by the vulnerability mining process are also particularly complex. But the constraint information required for different system protection bypassing modes is very simple. To simplify the process of combining program path and state information with system protection bypassing method, we add the constraints of the system protection bypassing method to the program state obtained by the symbolic execution process, and then use the constraint solver integrated in symbolic execution tool to generate the final exploit.

As shown in Table II, when there is an address space layout randomization protection and there is a "JMP ESP" assembly instruction in the program, we can inject any "SHELLCODE" into the special position to generate an exploit of vulnerability. In this situation, our tool will accept a user input, and user enters a system command, our tool will generate an exploit which is able to achieve the result the same as the command executes.

V. IMPLEMENT AND EXPERIMENT

The implementation of this approach is based on the existing symbolic execution tool ANGR and integrates RP++ tool with Python to achieve the final automatic generation tool for vulnerability exploit.

To prevent the problem of path space explosion and make full use of the ability of symbolic execution tool ANGR, the exploration depth of symbolic execution is set to 400.

The test cases used in experiments include the artificial program containing buffer overflow vulnerability, the test programs of the CGC competition, and the program generated automatically by the CSMITH tool.

The experimental system environment is ubuntu14.04 with python2.7. The system environment has optional address space layout randomization protection and optional stack non-executable protection.

To demonstrate the process and effect of our tool better, we record and upload the complete exploit process of buffer overflow vulnerability with address space layout randomization system protection to the online web [3]. The experimental results of other test cases are shown in TABLE III. Our tool can successfully exploit three different test cases in three different level system protection situations.

TABLE III. EXPERIMENT RESULT

	Artificial test case	CGC test case	Test case generated by CSMITH tool
Without system protection	Exploit success	Exploit success	Exploit success
address space layout randomization system protection	Exploit success	Exploit success	Exploit success
stack non-executable system protection	Exploit success	Exploit success	Exploit success

VI. RELATED WORK

In 2011, CMU published the AEG: Automatic Exploit Generation in the Internet Society [2]. The tool implemented by them can generate the exploit of software vulnerabilities automatically, but the method and the system is based on source code level. Source code and binary code compiled by the same source code are needed. Because of that paper, automatic generation exploit of software vulnerabilities is becoming popular. Later, DARPA held a CGC (Computer Great Challenge) game, to mine software vulnerabilities in binary code automatically. Automatic vulnerability mining and exploit technology were focused on at the same time.

The goal of CGC game is to achieve the following two purposes:

- Task 1: Crashing at the specified invalid address AND control a register as the specified value.
- Task 2: Divulging any four bytes in the flag memory page.

The above two studies are the most influential in the field of automatic exploit generation. However, neither of the two studies has considered the safety protection strategy of modern operating system.

Authorized licensed use limited to: XIDIAN UNIVERSITY. Downloaded on June 24,2021 at 08:43:19 UTC from IEEE Xplore. Restrictions apply.

TABLE IV. COMPARED WITH THE AEG SYSTEM PROPOSED BY CMU, OUR TOOL SYSTEM HAS STRONGER ABILITY TO EXPLOIT VULNERABILITIES, WHICH IS SHOWN IN DETAIL IN TABLE IV.COMPARE WITH AEG

Compare content	Our Tool	AEG
Binary code without source code	Yes	No
Rotate ASLR	Yes	No
Rotate NX	Yes	No
Generate exploit by user input	Yes	No
Automatic completely	Yes	Yes
Deal with large program	No	No
Deal with various vulnerability	No	No

VII. CONCLUSION

In this paper, we take the buffer overflow vulnerability as the research object and propose an automatic vulnerability mining and exploiting method based on symbolic execution, which can generate exploit of the buffer overflow vulnerability automatically. The method proposed in this paper can bypass the address space layout randomization protection and stack non-executable protection, and our tool implemented is able to operate directly on the binary program without source code.

Although compared with to the AEG system, our method has great improvements, our tool still has some shortcomings and limitations. It mainly includes two aspects: the tool cannot deal with large-scale software program, because this method is based on symbolic execution technology that may cause path space explosion problem; this method can deal with a single type vulnerability, just buffer overflow. Our method and tool need to be further enhanced and promoted in future work.

ACKNOWLEDGMENT

We would like to thank the authors of ANGR for opening their source code. We also would like to thank the authors of tutorials: Linux (x86) Exploit Development Series [4]. This work is funded by National Natural Science Foundation of China (No.61690203, No.61532007), and 973 National Program on Key Basic Research Project of China (No.2014CB340703).

REFERENCE

[1] Yan S, Wang R, Salls C, et al. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis[C] Security & Privacy. IEEE Computer Society, 2016:138-157.

[2] Avgerinos T, Sang K C, Hao B L T, et al. AEG: Automatic Exploit Generation[J]. Internet Society, 2011, 57(2).

[3] Weixi J. The exploit process demonstration of our tool system with ASLR protection. [EB/OL]. https://asciinema.org/a/jkrle7lmKDgGmkluiV6xBwQ7D

[4] Sploitfun. Linux(x86) Exploit Development Series [EB/OL]. https://sploitfun.wordpress.com/2015/06/26/linux-x86-exploit-development-tutorial-series/, 2015.06.26

[5] Heelan S. Automatic generation of control flow hijacking exploits for software vulnerabilities[M]. 2009.

[6] Ramos D A, Engler D. Under-constrained symbolic execution: correctness checking for real code[C] Usenix Conference on Security Symposium. USENIX Association, 2015:49-64.

[7] Stephens N, Grosen J, Salls C, et al. Driller: Augmenting fuzzing through selective symbolic execution[C] Proceedings of the Network and Distributed System Security Symposium. 2016.

[8] Hme M, Pham V T, Roychoudhury A. Coverage-based Greybox Fuzzing as Markov Chain[C] ACM Sigsac Conference on Computer and Communications Security. ACM, 2016:1032-1043.

[9] Kulenovic M, Donko D. A survey of static code analysis methods for security vulnerabilities detection[C] International Convention on Information and Communication Technology, Electronics and Microelectronics. IEEE, 2014:1381-1386.

[10] Pistoia M, Chandra S, Fink S J, et al. A survey of static analysis methods for identifying security vulnerabilities in software systems[J]. Ibm Systems Journal, 2007, 46(2):265-288.

[11] Bao T, Burket J, Woo M, et al. BYTEWEIGHT: learning to recognize functions in binary code[C] Usenix Security Symposium. USENIX Association, 2014.

[12] Hanov S. Static Analysis of Binary Executables[J]. Stevehanov Ca.

[13] Eschweiler S, Yakdan K, Gerhards-Padilla E. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code[C] The Network and Distributed System Security Symposium. 2016.

[14] Wang S, Wang P, Wu D. Reassembleable disassembling[C] Usenix Conference on Security Symposium. USENIX Association, 2015:627-642.

[15] Meng X, Miller B P. Binary code is not easy[C] International Symposium on Software Testing and Analysis. 2016:24-35.

[16] Heelan S. Automatic generation of control flow hijacking exploits for software vulnerabilities[M]. 2009.

[17] Avgerinos T, Sang K C, Rebert A, et al. Automatic exploit generation[J]. Communications of the Acm, 2014, 57(2):74-84.

[18] Kapus T, Cadar C. Automatic testing of symbolic execution engines via program generation and differential testing[C] Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. IEEE Press, 2017: 590-600.

[19] Wang S, Wang P, Wu D. Reassembleable disassembling[C] Usenix Conference on Security Symposium. USENIX Association, 2015:627-642.

[20] Pewny J, Garmany B, Gawlik R, et al. Cross-Architecture Bug Search in Binary Executables[C] Security and Privacy. IEEE, 2015:709-724.

Authorized licensed use limited to: XIDIAN UNIVERSITY. Downloaded on June 24,2021 at 08:43:19 UTC from IEEE Xplore. Restrictions apply.