FIFA: Finding Indirect Forward edge Attack path

Kyuwon Cho
College of Computing and Informatics
Sungkyunkwan University
Suwon, Korea
kyuwoncho18@gmail.com

Byeongsu Woo
College of Computing
Sungkyunkwan University
Suwon, Korea
wbs79@skku.edu

Hajeong Lim
College of Computing
Sungkyunkwan University
Suwon, Korea
doolim98@gmail.com

I. Background and Related Work

A. Background

1) LLVM Indirect Function Call Checks(IFCC): To become more resistance to control flow hijacking attack, LLVM introduces some CFI(Control Flow Integrity) techniques. One of them is called IFCC [1], Indirect Function Call Check. It protects C-style indirect call, which is function call with function pointer which decides its value in runtime. IFCC collects the function pointer call in compile time, and makes a jump table in binary. Function call checking is done through the type of the pointer. Both static cast and dynamic type can be considered. The jump table contains the function addresses which is same type of the function pointer. In runtime, the indirect call is invoked, it checks whether the target address is in jump table and decides that this call can be invoked or denied.

2) Symbolic Execution: Symbolic execution [2] is a static binary analysis technique. For input generation, it is important to create an input that can execute each part of the program. The simplest way is to input a concrete value and execute the program, but it is impractical to run the entire program for many inputs. The basic idea of the symbolic execution is not to find a concrete value, but to find and solve constraints of the symbols. It runs as binary as *symbolically*, and shows what happened. If the executor walks around the program's state graph and meet the terminate condition, it tries to solve the constraints of the path. It is very useful to find the exact input of hitting the target code point, but the constraint solving is very time consuming job. Also, since exploring all paths is not feasible in large programs, path explosion is the fundamental limitation of symbolic execution, so it needs lot of heuristic optimization.

B. Related Work

1) Attacks on CFI: DOP [3] and BOP [4] show that leverage data-only attacks to control hijacking attack. They do not violate the benign CFG(Control Flow Graph), but corrupt the variable which decides the branch condition. It is perfect counterfeit of the state of the art CFI. However, they have fewer gadgets than control flow hijacking attack because it can use only benign CFG. C++ uses vtable to decide object method function call, and it is also kind of indirect branch so it need to be verified. LLVM and GCC introduce some CFI technique to defeat the vtable corrupting exploit, COOP [5] and Vscape [6] show that vtable CFI solutions are not perfect.

2) Exploit automation: Automatic Exploit Generation(AEG) [7] made an automatic tool that traditional control flow hijack exploits about target binaries automatically using symbolic execution. They solved the scaling issue of symbolic execution by proposing preconditioned symbolic execution which restrict exploration to only likely-exploitable regions. *Automatic Generation of Data-Oriented Exploits* [8] made an automatic tool using Data-only attacks which purpose of sensitive data leak or privilege escalation.

II. Problem Statement and Motivation

In the history of exploit, there is an attack that controls a program flow using stack buffer overflow. In the beginning, attackers inject exploit code, then corrupt the return address to injected code to execute the exploit code. Because of this attacks, W⊕X is introduced, which is an mitigation that removes memory segment having both write and execute privilege. Even though an attacker injects exploit code in somewhere writable memory segment, the code is not executable because the memory segment does not have execution privilege.

An attacker introduces Return Oriented Programming (ROP) to bypass W⊕X. ROP is an attack that chaining some useful gadgets. A gadget is a code block that performs a simple operation such as popping or pushing some registers, loading or storing data to memory, and some other works. A gadget is ended with ret, jmp, or call operation. This allows attackers to chain gadgets. Attackers can execute exploit codes by chaining some gadgets without writable and executable memory segment.

Some mitigations such as shadow stack are introduced to defend above attacks changing program control flows by corrupting return address. Shadow stack copies the return address to another memory space, and when ret instruction is performed, compare the return address with the saved one so check that is corrupted or not. This mitigation makes backward edge attacks almost impossible. However, the possibility of forward edge attack still remains relatively high. Forward edge attack is an

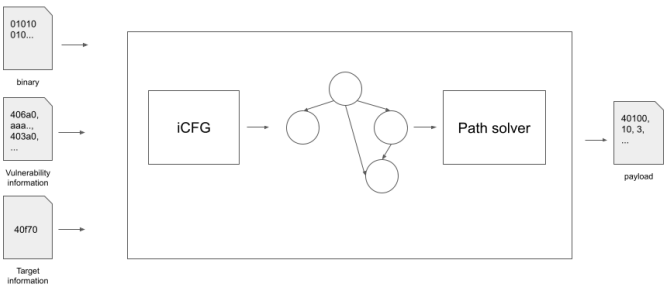

Fig. 1. Overall process of FIFA. It takes the target binary, and the information of vulnerability, and the target code address as input and calculate the reachability and payload.

attack changing control flows by changing a target of some indirect branches. There are some mitigations that defend the forward edge attack. Intel CET [9] checks whether the operation at the destination of an indirect branch is `endbranch` operation. LLVM CFI [1] restrict control flow which violate the compile-time control flow graph. The goal of above mitigations is limiting the range of destinations of indirect branches not preserving the original code flow strictly. They make attackers hard to analyze the binary and to leverage the forward edge attack. When an attacker try to attack a CFI-disabled binary, all the attacker have to do is just jump to target address directly. But if an attacker try to attack CFI-enabled binary, CFI limits the range of destination of branches, thus it effectively reduces the possibility of attack. However, that kind of forward-edge CFI makes the attack *difficult*, not *impossible*. Because of performance overhead and intentional design, LLVM CFI check only the function signature of target, so attacker leverage the function that has same signature with target function to exploit the control flow.

We introduce FIFA: Finding Indirect Forward edge Attack path, that finds an attack path in the CFI-enabled program. If the target program has a forward edge attack entry point, the user gives the binary, the forward edge attack entry point, and a target point to FIFA. Then FIFA analyzes the binary, finds a path between the given entry point and the target code and return the payload that exploit the finding path.

III. Design

A. Assumption

We assume that the target binary is compiled with IFCC. The target binary includes a memory bug which allows an attacker controls the one indirect branch. The target binary also has a memory disclosure bug defeating ASLR.

B. iCFG

Almost Control Flow Graph(CFG) generating tools cannot resolve an indirect call because its jump target is decided in runtime, whereas CFG commonly analyze the target statically. But a binary protected by IFCC allows indirect call to jump only limited target, and the target of indirect branch is determined in compile time. This allows CFG to resolve indirect calls. We propose the indirect Control Flow Graph(iCFG), which analyzes indirect calls of IFCC protected binary, resolves its indirect call target and draws CFG.

C. Path Solver

Symbolic execution that finding a path between entry point and target code is difficult because of path explosion and undeterministic indirect target. However, we can get a path easily from entry to target in IFCC-protected binary by using iCFG. However, we do not know how to walk the path because it needs the value of variables that effects to branch conditions on path. We use concolic execution to get the actual value which decide each branch condition. So as to avoid path explosion, we run the target binary as concretely until the vulnerable entry point, then run it symbolically following the iCFG path.

IV. Implementation and Evaluation Plan

A. Implementation

1) iCFG: In iCFG, we focus on resolving jump targets of the IFCC-protected indirect call. First of all, the iCFG scans the whole binary, finds the indirect calls like `call *rcx`, and decides whether it is an IFCC-protected call by searching the `ud2` instruction. After that, If the indirect call is a protected call, iCFG collects the jump table address and its length by backward searching the jump-table load instruction like `movabs rax, 0x400ab0` and length compare instruction. With the jump table address and its length, iCFG can create the angr CFG indirect jump resolver. With this resolver, angr automatically draws the CFG with the information about indirect call targets. We finish implementing the iCFG, and have plan to evaluate the iCFG.

2) Path Solver: If iCFG succeeds finding a path, path solver finds a payload that executes the path. Path solver executes the binary symbolically using path produced by iCFG. After it reaches to target, it generates a payload by resolving constraints of symbolic values, and returns the result to user. Path solver concretely executes the target binary until it reaches to the attack entry point received as input, and then symbolically executes the target binary along the path. We use angr [10], the state-of-art symbolic execution tool to solve this path constraints. And symbion [11] is used to do concrete execution. Currently, implementation of Symbolic execution along the path is done. Implementation of concrete execution until the attack entry point is in progress.

B. Evaluation Plan

We will evaluate the iCFG with standalone, and evaluate the path solver with iCFG. We first plan to evaluate the iCFG to show this tool can resolve indirect branch

target of complex indirect branch-included binary. After that, we evaluate the path solver with CTF challenge that open the source, we compile that challenge with IFCC and test the path solver can hit the target code.

References

[1] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 941–955. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice

[2] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, p. 385–394, Jul. 1976. [Online]. Available: https://doi.org/10.1145/360248.360252

[3] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 969–986.

[4] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1868–1882. [Online]. Available: https://doi.org/10.1145/3243734.3243739

[5] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 745–762.

[6] K. Chen, C. Zhang, T. Yin, X. Chen, and L. Zhao, "Vscape: Assessing and escaping virtual call protections," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1719–1736. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/chen-kaixiang

[7] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Commun. ACM*, vol. 57, no. 2, p. 74–84, Feb. 2014. [Online]. Available: https://doi.org/10.1145/2560217.2560219

[8] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 177–192. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/hu

[9] V. Shanbhogue, D. Gupta, and R. Sahita, "Security analysis of processor instruction set architecture for enforcing control-flow integrity," in *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3337167.3337175

[10] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[11] F. Gritti, L. Fontana, E. Gustafson, F. Pagani, A. Continella, C. Kruegel, and G. Vigna, "Symbion: Interleaving symbolic with concrete execution," in *2020 IEEE Conference on Communications and Network Security (CNS)*, 2020, pp. 1–10.