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

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

Control flow exploits allow an attacker to execute code on a target. This is one of the dangerous attacks, mainly caused by memory bugs. Because the attack space of memory is too broad, it was difficult to prevent control-flow hijacking due to memory corruption. Many early exploits used the method of inserting and executing arbitrary code through stack buffer overflow. Because of this attacks, W⊕X is introduced, which is an mitigation that removes memory segment having both write and execute privilege. However attackers introduced Return Oriented Programming (ROP) to bypass W⊕X. To defend ROP, Intel came up with a fundamental solution in Intel CET, hardware supported shadow stack. So the last field in a control flow exploit is the forward-edge control flow. Forward-edge code flow is determined at runtime, so it is difficult to defend. Many methods were introduced to protect CFI, and their basic method is to limit the forward-edge. We observed that the more the forward edge is restricted, the more the chance of binary analysis increases. Paradoxically, the methods for CFI increased the chance of symbolic execution, which was previously difficult because of path explosion problem. We introduce FIFA, Finding Indirect Forward edge Attack path, that finds an attack path in the CFI-enabled program. Then FIFA analyzes the binary, finds a path between the given entry point and the target code and returns the payload that exploits the finding path.

II. Background and Related work

A. LLVM Indirect Function Call Checks(IFCC)

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 jump tables 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.

B. 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 executing the program with concrete value, 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 meets 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 lots of heuristic optimization.

C. Attacks on CFI

DOP [3] and BOP [4] 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 needs to be verified. LLVM and GCC introduce some CFI techniques to defeat the vtable corruption attack, COOP [5] and Vscape [6] show that vtable CFI solutions are not perfect. However, this study conducted an attack on Virtual Call Protections. We target IFCC, which is a more finegrained CFI.

D. 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 restricts exploration to only likely-exploitable regions. *Automatic Generation of Data-Oriented Exploits* [8] made an automatic tool using Data-only attacks whose purpose is leaking some sensitive data or getting privilege escalation.

III. Design

A. Scope

In this paper, we targeted the IFCC compiled binary. The target binary has a memory bug which allows an attacker to control the one indirect branch, also the target binary has a memory disclosure bug that defeating ASLR.

B. iCFG

The most challenge of Control Flow Graph(CFG) generating tools is resolving indirect calls. Since they analyze the target binary statically, determining the jump target is difficult. However, we observed that IFCC allows to analyze indirect calls statically. IFCC limits indirect calls to preserve control flow at runtime and the jump targets are determined in compile time, but at the same time it gives a clue to generating CFG. We propose the indirect Control Flow Graph(iCFG), which analyzes indirect calls of IFCC protected binary, resolves its indirect call target and generates CFG.

To implement iCFG, we use the Capstone, the disassembler package. The iCFG takes two input, the target binary which is protected by IFCC, and the entry point which CFG will start. In the frontend step of iCFG, it disassemble the binary and recognize the protected indirect call by pattern matching. After listing the protected call, it collects the jump table address and length belong to each call in call listing. Now iCFG get the whole information about the jump target of indirect call, so it pass the information to angr CFG tool to draw the 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. The another challenge is finding a path between entry and target point. For Symbolic execution, it is difficult to solve path explosion. 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 values which decide each branch conditions. 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. Evaluation

In this section, we will show an example how FIFA finds a path and generates input.

A. Toy program

We made a toy program to make an attack scenario for CFI enabled program. In this program, the `target` function includes the `setuid`, which leads the privilege escalation. Attacker's goal is calling the target function, and `vuln` function has stack overflow bug leading calling the attacker-controlled function. However, the binary is protected by CFI, attacker can't directly put the target function address to vulnerable function pointer. Instead, the attacker must find the path that reaches the target function. In this case, `first_gate` has same signature with `origin_flow`, so attacker can leverage this function as stepping stone to target function.

```
void (*un_init_func_table[16])(int) = {0};

void target(int x){
    setuid(0);
}

int origin_flow(void (*arg_func)(int, int), int num)
{
    return 0;
}

int first_gate(void (*arg_func)(int, int), int num)
{
    if(num == -1) return -1;
    if(num == 0) arg_func(num, 0);
    return 0;
}

int vuln(){
    int (*vuln_ptr)(void (*)(int, int), int) =
    origin_flow;
    int array[10] = {};
    // Stack overflow occured!
    read(STDIN, array, 1024);

    // vuln_ptr is attacker controlled value, but
    because of CFI, it can only call the origin_flow
     and first_gate
    vuln_ptr(arg_func, arg_num);
    return 0;
}

int init_table_on_runtime(){
    un_init_func_table[0] = target;
    return 0;
}

int main(){
    init_table_on_runtime();
    return vuln();
}
```

Listing 1. Toy Program

The angr provides CFG tool for binary, but unlike source-based CFG tool, it cannot resolve the indirect call because it does not contain any information which function they arrived. In fig. 1, the left side one is result of the unmodified angr CFG tool. It cannot resolve the `call rcx`, so any node is not connected to the call instruction. By comparison, iCFG can resolve the indirect call and find the connected node on the instruction.

Path Solver gets a path between attack entry point and target. Then it executes the basic blocks in the path

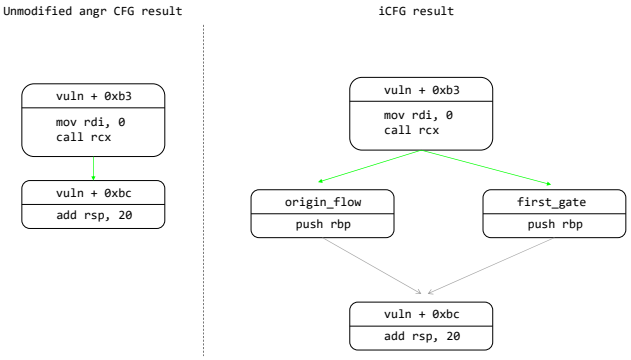

Fig. 1. With our toy program binary, angr CFG generator cannot resolve the indirect function call on function `vuln`. However, our iCFG can resolve the indirect call and recover the appropriate program flow.

sequentially. Path explosion can be prevented by using concrete execution from program starting point to attack entry point. To check the necessity of getting path from iCFG, we compare FIFA with angr code that just tries to find target directly after concrete execution.

The angr code that just tries to find target directly cannot reach to target because of `un_init_func_table`. `un_init_func_table` is a global array initialized in `init_table_on_runtime` function. This means that the global array is initialized during the concrete execution. However, in the angr, the result of concrete execution is not applied automatically to symbolic state. Thus, there is no address of `target` in `un_init_func_table` in symbolic execution. FIFA can solve this problem. FIFA uses time out mechanism. If path solver cannot move to next basic block during the threshold time, time out occurs. Path solver gets MOV instructions in the basic block and inspects the memory spaces that MOV instruction accesses. Then path solver reads the memory space from concrete state, then writes the value to symbolic state. And it retries resolving the basic block again. Thus, getting path from iCFG is needed for not only preventing path explosion but also finding missed memory region.

The elapsed time of FIFA finding the proper input for the toy program is 39.312 seconds which is an average of 10 executions.

V. Conclusion

In this project, we automated the process finding an input that attacker can arrive target point from forward edge attack entry point in LLVM IFCC enabled C program by using symbolic execution framework angr. We made iCFG for finding path between forward edge attack entry point and target point. We made path solver which finds an input that attacker desired and handles path explosion.

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