Function-Oriented Programming: A New Class of Code Reuse Attack in C Applications

Yingjie GUO
Institute of Information Engineering,
Chinese Academy of Sciences,
School of Cyber Security,
University of Chinese
Academy of Sciences
Beijing, China
Email:guoyingjie@iie.ac.cn

Liwei Chen
Institute of Information Engineering,
Chinese Academy of Sciences,
School of Cyber Security,
University of Chinese
Academy of Sciences
Beijing, China
Email:chenliwei@iie.ac.cn

Gang Shi
Institute of Information Engineering,
Chinese Academy of Sciences,
School of Cyber Security,
University of Chinese
Academy of Sciences
Beijing, China
Email: shigang@iie.ac.cn

***Abstract*—Control-hijacking attacks include code injection attacks and code reuse attacks. In recent years, with the emergence of the defense mechanism data-execution prevention(DEP), code reuse attacks have become mainstream, such as return-oriented programming(ROP), Jump-Oriented Programming(JOP), and Counterfeit Object-oriented Programming(COOP). And a series of defensive measures have been proposed, such as DEP, address space layout randomization (ASLR), coarse-grained Control-Flow Integrity(CFI) and fine-grained CFI. In this paper, we propose a new attack called function-oriented programming(FOP) to construct malicious program behavior. FOP takes advantage of the existing function of the C program to induce attack. We propose concrete algorithms for FOP gadgets and build a tool to identify FOP gadgets. FOP can successfully bypass coarse-grained CFI, and FOP also can bypass some existing fine-grained CFI technologies, such as shadow stack technology. We show a real-world attack for proftpd1.3.0 server in the Linux x64 environment. We believe that the FOP attack will encourage people to come up with more effective defense measures.**

Keywords*—*Code reuse attacks, return-oriented programming, Counterfeit Object-oriented Programming, function-oriented programming, ASLR, CFI.

I. INTRODUCTION

At present, many large programs are written in C language. C language is an unsafe programming language, and there are lots of vulnerabilities in the programs, such as the buffer overflow vulnerabilities, use-after-free, and direct access to memory [25]. Code injection attacks exploit the memory corruptions to inject malicious code into memory, and execute malicious code by modifying the control-flow. However, with the emergence and deployment of the defense mechanism data execution prevention(DEP)[1], code injection attacks become unfeasible. Subsequently, people proposed a new attack type called code reuse attack.

Code reuse attacks introduce malicious program behaviors by recombining program code fragments(called gadgets). So code reuse attacks can bypass the DEP defense mechanism. So far, many code reuse attack techniques have been proposed, such as Return-into-Libc[2], ROP[3], JOP[4] and COOP[5]. Code reuse attacks own strong destructiveness and extremely high flexibility, which has aroused widespread concern. With the development of code reuse attack techniques, people have put forward a series of defensive measures.

There are three kinds of defense techniques for code reuse attacks. The first is anomaly detection based on the behavior characteristics of code reuse attacks. Pappas et al.[18] and Cheng et al.[17] have proposed methods that monitoring the occurrence frequency of ROP gadgets according to behavior characteristics of ROP attack. The second is to ensure the legitimacy of control flow and prevent the control flow hijacking. Relevant defense mechanisms include CFI[6], Code-Pointer Separation(CPS), and Code-Pointer Integrity(CPI)[7]. Their implementation techniques include shadow stacks, s-tack canaries, vtable pointer verification, Microsoft's Control Flow Guard(CFG) and Return Flow Guard (RFG), Google's Indirect Function-Call Checks(IFCC) and Reuse Attack Protector(RAP). The third is to increase unpredictability of the code layout in the memory such as Address Space Layout Randomization (ALSR)[8], [9], [10]. ASLR can increase the difficulty of the attack, but there is an important flaw: attackers can get the code layout in the program memory through the memory leak vulnerability of a program[22], [23], so this defense mechanism will only slow down attacks, it cannot completely defend against code reuse attacks[11].

In this paper, we introduce function-oriented programming(FOP), a new code reuse attack technique. FOP takes advantage of the existing functions of the C program to induce the attack. FOP uses existing functions as gadgets. FOP mainly defines two types of gadgets, dispatcher gadgets and function gadgets. Dispatcher gadgets are responsible for scheduling function gadgets, and function gadgets achieve the expected attack through mutual cooperation. We divide the function gadgets into different categories in detail and propose concrete algorithms for FOP gadgets. We have implemented a tool called fop-gadgets which can automatically find FOP gadgets. To a certain extent, the tool fop-gadgets reduces the complexity of the attack and enhances the attack efficiency.

With the progress of code reuse attack techniques, many defense mechanisms[6], [7], [17], [18], [19], [20]have been proposed, but these defensive mechanisms are more or less defective. In the real world, attackers can complete the FOP attack without triggering existing security mechanisms. FOP can successfully bypass coarse-grained CFI, and FOP also can bypass some existing fine-grained CFI technologies. The shadow stack is used to protect the return address, each function call will save the return address into the original stack and the shadow stack. When the function returns, two return addresses will be compared. If two return addresses are not same, the program is subjected to attack. FOP attack ensures the matching of a call instruction and a return instruction, so FOP can bypass the shadow stack technology. FOP can also bypass other defensive technologies, such as CFG, RFG and RAP. For example CFG technology, it ensures that each call instruction jumps to the function head. FOP makes use of the existing functions as gadgets and each call instruction must jump to the function head, so FOP can bypass CFG defense technology. CPI and CPS defense techniques mainly protect code pointers, they put code pointers into a safe area. CPI can defense control-hijacking attacks. But the premise is that the safe area is safe enough and cannot be found, and we must make use of a specific instruction to access safe area[7]. Evans et al.[24] have proven that CPI can be broken. Safe area does not achieve absolute isolation. So attackers can still find the position of the safe region. So we can still break the CPI, and we can access and control sensitive pointers.

We have proven that FOP attack technology is feasible. In the Linux x64 environment, we achieved a real-world attack. We have chosen a server, ProFTPD1.3.0, that is written in C language[15]. This server provides critical network services and there is a remote buffer overflow vulnerability on this server. In this server, we used the tool fop-gadgets to find enough dispatcher gadgets and function gadgets which can meet attack requirements. The dispatcher gadgets are reachable in accordance with the normal control flow of the program and are able to sew these function gadgets. We make use of the ProFTPD1.3.0 server buffer overflow vulnerability to successfully complete the FOP attack. We made use of this FOP attack to execute arbitrary machine code.

II. Technical Background

Return-into-libc[2] is a code reuse attack. It leads to the return address of the function in the program stack is covered with the address of another function that the attacker can control through the buffer overflow vulnerability, and the function parameters are injected into the stack together. Attackers mainly choose the functions in the library, because the library will be linked to the program, and the library provides many very useful functions for attackers, such as system() and mprotect(). Return-into-libc can bypass DEP defense mechanism. Shadow stack can prevent this attack. So far, Return-into-libc attack has become impractical and gradually eliminated.

ROP attacks use the existing short instruction sequences as gadgets, and ROP gadget ends in a return instruction and completes a certain function, such as arithmetic operation and logical operation[3]. ROP attacks firstly find a series of gadgets that meeting the requirement of the attack from the program, and then according to the order of gadgets execution, stitch addresses of gadgets and parameters into the ROP chain, finally inject ROP chain into the stack by making use of memory vulnerabilities and execute the ROP chain by changing the program control flow. Shadow stack can prevent ROP attacks. ROP attack supports multiple architectures and systems, so ROP attack has been used and there are many variants of ROP[12], [13], [14].

JOP attack is a variant of ROP attack, ROP gadget ends with a return instruction, but JOP gadget ends with an indirect jump instruction or indirect call instruction[4], [14]. In the ROP attack, it needs to inject ROP chain into the stack, and the stack pointer acts as the role of rip. But JOP attack saves the gadget chain into any readable and writable data area. ROP and JOP all use short instruction sequence as a gadget, and they have a certain regularity, so some abnormal detection defense measures can be used to defend against ROP and JOP attacks. And in the ROP and JOP attacks, call and return instructions do not match, therefor CFG, RAP and IFCC can defend against them.

COOP attack is different from ROP attack and JOP attack, and it is mainly aimed at C++ programs[5]. When we define a class instance(non-global), a class object will be allocated in the stack followed by the virtual function table address and data. A COOP attack introduces malicious program behavior by creating fake class objects and calling C++ virtual functions. Virtual table verification (VTV) technology strictly limits the target set of each virtual function call point, so it can defend against COOP attacks.

Data-oriented programming(DOP)[21] attack is a code reuse attack. DOP is different from control-hijacking attacks. DOP constructs non-control data exploits for arbitrary x86 programs. DOP makes use of short instruction sequences as gadgets, and it is similar to ROP attack. But DOP uses data plane values for malicious purposes, and it maintains the integrity of the control plane.

In this paper, we have proposed FOP attack technology. FOP attack technology uses already existing functions as gadgets in a C program. Compared with ROP and JOP attacks, FOP attack calls a complete function at each time, so FOP attack can bypass the coarse-grained CFI, shadow stack, CFG, RFG, RAP and IFCC.

III. Function-Oriented Programming

FOP is a code reuse attack technology aiming at applications developed in C language. In the following, we first introduce the FOP design goals and the attack model, and then introduce the FOP gadget types. FOP attack technology supports any system. In order to introduce clearly FOP attack technology, the following description is based on the linux x64 operating system.

A. Goals

We define the following goals for FOP:

G-1 does not modify the return address of the function on the stack.

G-2 must use the function that already existing in the program as a gadget.

G-3 be not allowed to jump into the body of a function, must be executed from the function head.

G-4 be widely applicable to for C applications

G-5 achieves FOP attacks in the real world

B. Attack Model

Applications developed in C language may generally exist buffer overflow vulnerability, use-after-free, double-free and a series of other vulnerabilities. Attackers can exploit these vulnerabilities to introduce malicious program behavior. With the emergence of code reuse attack techniques, such as ROP, JOP, and COOP, in order to achieve these attacks, attackers need to know the code layout in the program memory, especially the instruction address. Some defensive techniques increase the attacker's ambiguity of memory layout, such as address space randomization(ASLR) technique[8]. It is difficult to know the memory layout of the program and the specific location of the code, but this defense technique can still be broken. Such as Blind-ROP technology[16], it can successfully bypass the randomization technology to complete the ROP attack. Attackers can also get the program's memory layout through memory leaks[22], [23]. In short, the address space randomization technology can still be bypassed.

FOP attack needs to use a two-level function pointer to circularly call the function gadgets, so the existence of two-level function pointers in the program is very critical. We can also simulate a two-level function pointer. For example, there are a function pointer and a next pointer in the structure, and the next pointer points to the same structure. So we can use this structure type to simulate a two-level function pointer.

In the code reuse attack, the word gadgets have been very common, almost every code reuse attack will define their own gadgets. ROP gadget is a short instruction sequence ending in the return instruction. The JOP gadget is a short instruction sequence ending with the jump instruction or call instruction. The COOP gadget is a virtual function in the C++ program. In FOP attacks, we make use of the C function as a gadget. Attackers can find useful gadgets by analyzing source code or binary file of the program. The tool fop-gadgets is used to find FOP gadgets by analyzing source code of the program.

The FOP attack model is shown in Fig.1. The main function of the dispatcher gadget is to dispatch function gadgets and stitch the function gadgets together to complete the attack. So, firstly we have to find some dispatcher gadgets in the program, and from a large number of dispatcher gadgets, we select a dispatcher gadget that the control flow of the program can reach. The dispatcher gadget what we select must be convenient to control the scheduling of function gadgets. After finding the dispatcher gadget, we can identify function gadgets in the program, and then from a large number of function gadgets, we select some function gadgets that can achieve attack intention. Function gadgets what we select are combined together in a certain execution order. Then we can get the address space of the target program through the memory error. We can infer the address of each gadget and the address of key data that we need to control. After obtaining the above information, we can construct the gadget chain according to the order in which the gadgets are executed. Finally, we can write gadget chain into the memory through the memory vulnerabilities and achieve the FOP attack.

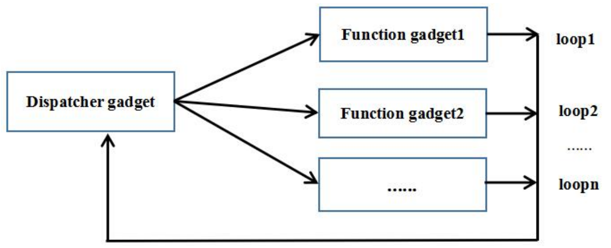


Fig. 1: FOP attack model

C. Gadget Types

In order to achieve FOP attacks, we define a series of gadget types. These gadget types are shown in Table I. In the following, we will give a detailed description of the definition of each gadget. In order to make it easier to understand each gadget type, we provide a simple example for each gadget type. Each example is a function that written in C language, and some examples also include the definitions and uses of global variables. We firstly introduce the dispatcher gadget(D-G), and then introduce the function gadgets. Function gadgets include reading memory gadget(R-M-G), writing memory gadget(W-M-G), arithmetic operations gadget(ARITHMETIC-G), logic operations gadget(LOGIC-G), writing parameter register gadget(W-P-R-G), and conditional branch gadget(C-B-G). For each gadget type, we will provide you detailed descriptions of the features, requirements and limitations, so that you can look for gadgets to achieve the FOP attack. We define a gadget as a C function, so the following will show the gadget type by using the form of high-level language rather than the form of assembly code. Of course, in order to facilitate the understanding, we will appropriately give the corresponding assembly code representation.

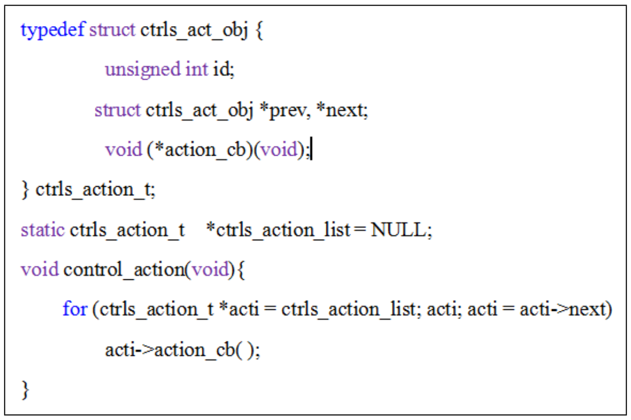
TABLE I: FOP Gadget Types

gadget type	description	example
D-G	Dispatcher gadget is a function in which there is a loop callof a function pointercontainer and there is a function call at each time	See Fig.2
R-M-G	Read data from a particular memory location	No example
W-M-G	Write data to a particular memory location	See Fig.3
ARITHMETIC-G	Perform arithmetic operation	See Fig.4,5,6,7
LOGIC-G	Perform logic operation	No example
W-P-R-G	Write data to parameter registers (rdi, rsi, rdx, rcx, r8, r9,xmm0-xmm7)	Fig.8,9
C-B-G	Achieve conditional branch	Fig.10

1)The Dispatcher Gadget: FOP attack requires the dispatcher gadget to call the function gadgets in a certain execution order. The dispatcher gadget is defined as follows:

Dispatcher gadget is a function in which there is a loop call of a function pointer container (e. g., an array or linked list or structure) and there is a function call at each time.

Fig.2 depicts a dispatcher gadget. Firstly, we define a structure called *ctrls_act_obj*, and there are four elements in the structure *ctrls_act_obj*, an unsigned integer *id*, a forward pointer *prev*, a backward pointer *next*, a function pointer *action_cb*. Then, we define a static global variable *ctrls_action_list* which is a pointer to the *ctrls_action_t* structure type. The function *control_action( )* is a dispatcher gadget. There is a *for* loop, and each time loop variable *acti* points to a structure *ctrls_action_t* and calls the function pointer *action_cb* of the structure *ctrls_action_t*.

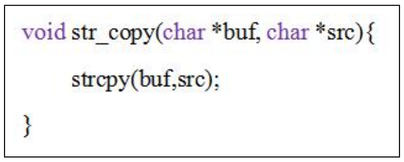


```
typedef struct ctrls_act_obj {
        unsigned int id;
        struct ctrls_act_obj *prev, *next;
        void (*action_cb)(void);
} ctrls_action_t;
static ctrls_action_t   *ctrls_action_list = NULL;
void control_action(void){
    for (ctrls_action_t *acti = ctrls_action_list; acti; acti = acti->next)
        acti->action_cb( );
}
```

Fig. 2: Depict a dispatcher gadget(D-G)

2)The Writing Memory Gadget: Writing memory gadget(W-M-G) is a kind of function gadget. Writing memory gadget is mainly responsible for writing the data into a particular memory location, and the attacker can control the data and memory location.

Fig.3 depicts a writing memory gadget(W-M-G). The function *str_copy( )* is a W-M-G. There are two arguments *buf* and *src*, they are both the string pointer type. The function *str_copy()* calls a string copy function *strcpy()* and it copies the string that is pointed by the variable *src* into the particular memory location that is pointed by the variable *buf*. If we control the source and target parameters before we use W-M-G, we can write a specific data to a particular memory location.



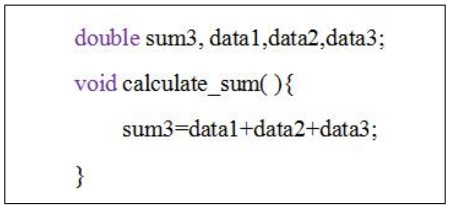
```
void str_copy(char *buf, char *src){
    strcpy(buf,src);
}
```

Fig. 3: Depict a writing memory gadget(W-M-G)

3)The Reading Memory Gadget: Reading memory gadget(R-M-G) is mainly responsible for reading data from a particular memory location, and its function is similar to writing memory gadget.

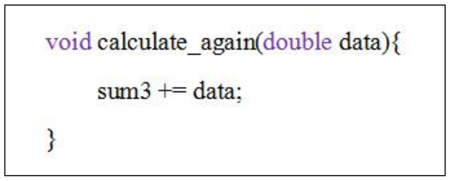
4)The Arithmetic Operations Gadget: The Arithmetic operations gadget (ARITHMETIC-G) is mainly responsible for arithmetic operations of several numbers and writing the result to a specific location. We can make use of the ARITHMETIC-G to calculate the address and data. Meanwhile, we can also set the register value so that the next function gadget use this value.

Fig.4 and Fig.5 depict two ARITHMETIC-G. The functions *calculate_sum()* and *calculate_again()* are both ARITHMETIC-G. The function *calculate_sum()* sets the variable *sum3* to the sum of other three variables. We can use this gadget to calculate the memory address and data, and write the result to the position of variable *sum3*. At the same time, we can use this gadget to set the value of the floating-point register. Fig.6 and Fig.7 depict assembly codes of the function *calculate_sum()* and *calculate_again()*. Fig.6 depicts the assembly code of the function *calculate_sum()*, we can see that the sum of three numbers is stored into the floating-point register *xmm0* and then the floating-point register *xmm0* is stored into the memory. So we can not only set the value of the memory but also set the value of the register by ARITHMETIC-G. Fig.7 depicts the assembly code of the function *calculate_again()*, we can see that the function calculates the result by directly using the value of the floating-point register *xmm0*. Therefore, if we firstly call the function *calculate_sum()* to save the result of the sum of three numbers into the floating-point register *xmm0*, and then call the function *calculate_again()* to calculate the result by directly using the previous calculation results *xmm0*. Here, there is just a brief introduction to the collaboration among function gadgets, and attackers can make full use of gadgets according to features of function gadgets.



```
double sum3, data1,data2,data3;
void calculate_sum( ){
    sum3=data1+data2+data3;
}
```

Fig. 4: ARITHMETIC-G1

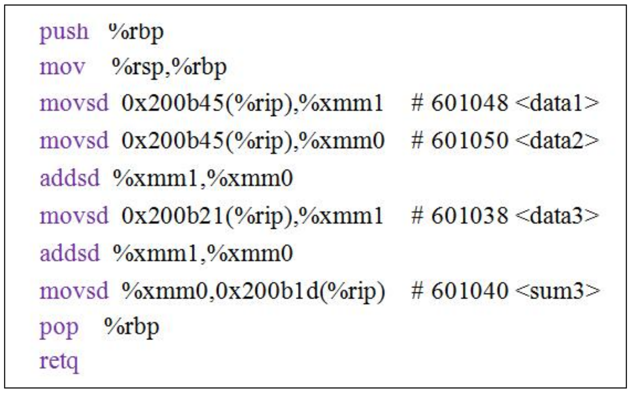


```
void calculate_again(double data){
    sum3 += data;
}
```

Fig. 5: ARITHMETIC-G2

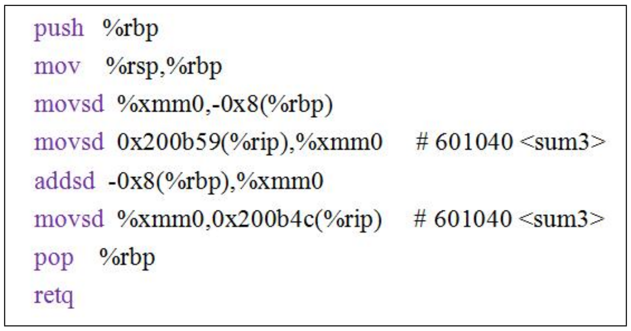
5)The Logic Operations Gadget: The logic operations gadget (Logic-G) is mainly responsible for logic operations of several numbers, such as and-operation, or-operations, non-operation, and then writing the result to a specific location. Its function is similar to ARITHMETIC-G.

6)The Writing Parameter Register Gadget: In the Linux x32 environment, when we call a function, the parameters what the function requires are stored into the program stack. But in the Linux x64 environment, the first six integer parameters are stored in the registers RDI, RSI, RDX, RCX, R8 and R9,



```
push   %rbp
mov    %rsp,%rbp
movsd  0x200b45(%rip),%xmm1    # 601048 <data1>
movsd  0x200b45(%rip),%xmm0    # 601050 <data2>
addsd  %xmm1,%xmm0
movsd  0x200b21(%rip),%xmm1    # 601038 <data3>
addsd  %xmm1,%xmm0
movsd  %xmm0,0x200b1d(%rip)    # 601040 <sum3>
pop    %rbp
retq
```

Fig. 6: Depict assembly code of function calculate_sum



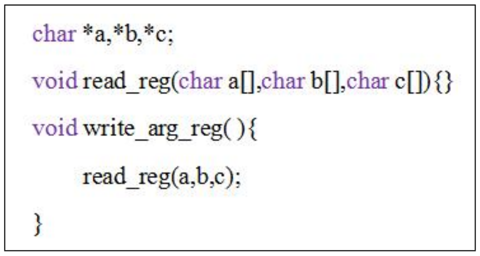
```
push   %rbp
mov    %rsp,%rbp
movsd  %xmm0,-0x8(%rbp)
movsd  0x200b59(%rip),%xmm0     # 601040 <sum3>
addsd  -0x8(%rbp),%xmm0
movsd  %xmm0,0x200b4c(%rip)     # 601040 <sum3>
pop    %rbp
retq
```

Fig. 7: Depict assembly code of function calculate_again

and floating point registers xmm0 to xmm7 are used to place floating point arguments. And if there are extra parameters, they will be saved in the stack. We call these registers as parameter registers. Therefore, in the Linux x64 environment, we need to set the parameter registers when we call a function. The writing parameter register gadget (W-P-R-G) is mainly responsible for setting the parameter registers.

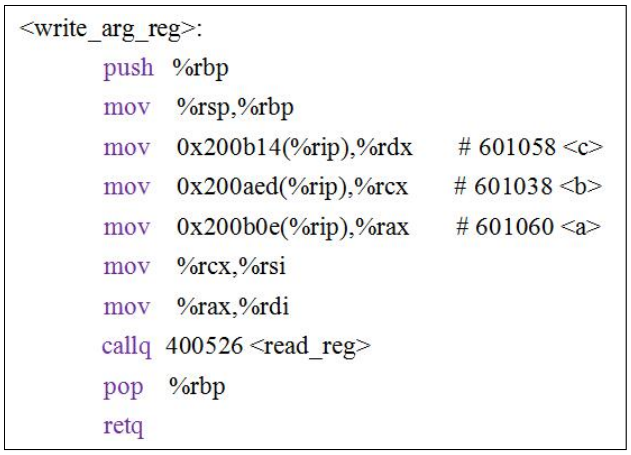
Fig.8 depicts a writing parameter register gadget (W-P-R-G). Function *write_arg_reg( )* is a writing parameter register gadget(W-P-R-G). We define the three character pointer global variables *a*, *b* and *c*. The function *write_arg_reg()* calls another function *read_reg( )*. The function *read_reg( )* can perform some operations that do not affect the parameter registers. Fig.9 depicts the assembly code of the function *write_arg_reg( )*. From the assembly code of the function *write_arg_reg( )*, we can see that function *write_arg_reg( )* sets the argument registers rdi, rsi, and rdx, and respectively assigns the values of the variables *a*, *b*, and *c* to the argument registers rdi, rsi and rdx. FOP attack technology makes use of a function as a gadget. So, before we call a function in the Linux x64 environment, if the function requires parameters, we need to set the parameter registers. The writing parameter register gadget(W-P-R-G) mainly implements the configurations of the parameter registers.

7)The Conditional Branch Gadget: The conditional branch gadget(C-B-G) mainly implements the conditional branch operation in the program. According to different branch operations, conditional branch gadget(C-B-G) can be broken down into conditional writing, conditional reading and conditional



```
char *a,*b,*c;
void read_reg(char a[],char b[],char c[]){}
void write_arg_reg( ){
      read_reg(a,b,c);
}
```

Fig. 8: Depict a writing parameter register gadget(W-P-R-G)

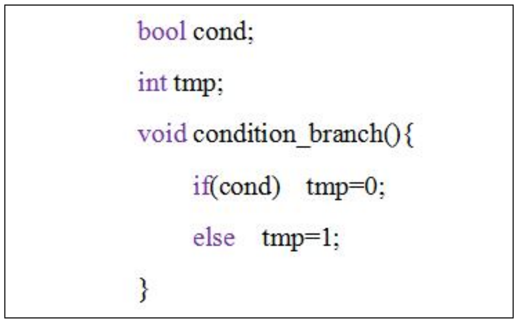


```
<write_arg_reg>:
        push  %rbp
        mov   %rsp,%rbp
        mov   0x200b14(%rip),%rdx      # 601058 <c>
        mov   0x200aed(%rip),%rcx      # 601038 <b>
        mov   0x200b0e(%rip),%rax      # 601060 <a>
        mov   %rcx,%rsi
        mov   %rax,%rdi
        callq 400526 <read_reg>
        pop   %rbp
        retq
```

Fig. 9: Depict assembly code of W-P-R-G

calculation.

Fig.10 depicts a conditional branch gadget(C-B-G). The function *condition_branch( )* is a simple C-B-G. If the Boolean variable *cond* is true, the variable *tmp* is assigned zero, otherwise, the variable *tmp* is assigned one. This is a simple conditional writing operation. For the conditional branch gadget(C-B-G), the number of basic block should be as small as possible, so that we can control the direction of branching and branch operations.



```
bool cond;
int tmp;
void condition_branch(){
      if(cond)   tmp=0;
      else   tmp=1;
}
```

Fig. 10: Conditional branch gadget(W-B-G)

IV. A Tool for Function-Oriented Programming

The entire FOP attack process requires four steps. In the first step, we need to identify gadgets through analyzing the source code or binary code of the program; in the second step, we need to pick out gadgets what we need; in the third step, according to the purpose of the attack, we need to construct gadget chain; in the fourth step, we need to insert gadget chain into the program memory by making use of the program vulnerabilities. Every step of FOP attack process is complex and difficult. So, in order to reduce the complexity of the FOP attack, we designed a tool called fop-gadgets to identify gadgets. We automatically achieve the first step in the FOP attack process. The attacker needs to manually construct remaining three steps according to the attack intention.

Tool fop-gadgets mainly analyze source code of the C program. We make use of the clang and llvm compiler to develop the tool fop-gadgets. fop-gadgets provides some options, we can set the options to identify all the gadgets or only identify a certain type of gadgets. For each gadget, we can view the brief or detail information of the gadget. We can select gadget type and gadget information by setting fop-gadgets options. We require the number of instructions to be as little as possible so that it is easier for attackers to use gadgets. In the design of the tool fop-gadgets, we control the number of a gadget instructions by limiting the number of basic blocks of each gadget. We use a shorter gadget to prevent the side effects among the gadgets. fop-gadgets identifies the gadgets by analyzing the source code of the program, of course, we can also identify the gadgets by analyzing the binary code of the program. In the following, we introduce algorithms of dispatcher gadget and writing parameter register gadget so that people can better understand the function-oriented programming(FOP) attack.

Algorithm 1 depicts a identification algorithm of the dispatcher gadget. The input is a vulnerability program that developed in C lanuage. And the output are the dispatcher gadgets set, the number of dispatcher gadgets, the global variables in each dispatcher gadget, and the number of global variables. We analyze each function in the program. We first recognize all loops in the function. For each loop, we scan each instruction in the loop body. If the instruction is a call instruction and a function is called by a function pointer, the function is a dispatcher gadget. For each dispatcher gadget, we show global variables and the number of global variables. We provide the global variable information so that attackers can make full use of dispatcher gadgets. We call these dispatcher gadgets as candidate dispatcher gadgets. An attacker needs to select the most appropriate dispatcher gadget from the candidate dispatcher gadgets.

Algorithm 2 depicts a identification algorithm of the writing parameter register gadget. The input is a vulnerability program. The output are the writing parameter register gadget set, the number of writing parameter register gadgets, the global variables in each writing parameter register gadget, and the number of global variables. We analyze each function in the program. For each function, we scan each instruction in the function body. If the instruction is a call instruction, we will get the callee function. For each instruction in the callee function body, if it is a call instruction, it proves the current call does not fit the definition of W-P-R-G and we need to keep finding. For each FOP gadget, we show global variables and the numbers of global variables and gadgets.

We can use the tool fop-gadgets to statically identify all

```
Input: $P$:- the vulnerable program.
Output: $D$:- dispatcher gadget set;
$NUMS$:- the number of dispatcher gadgets;
$GVAL$:- global variables in each dispatcher gadget;
$GVNUMS$:- the number of global variables in each dispatcher gadget.
$D = \emptyset$; $NUMS = 0$; $GVAL = \emptyset$; $GVNUMS = 0$;
while $f$ = get_function(P) do
    $gadgets = \emptyset$;
    for loop = get_loop(f) do
        for instruction = get_instruction(loop) do
            if is_call_instruction(instruction) then
                if is_function_pointer_call(instruction) then
                    $gadgets = gadgets \bigcup \{f\}$;
                    $GVAL, GVNUMS$ = find_global_variables($f$);
                    break;
                end
            end
        end
        if $gadgets$ != $\emptyset$ then
            $D = D \bigcup gadgets$;
            $NUMS = NUMS+1$;
            break;
        end
    end
end
```
Algorithm 1: Dispatcher gadget identification algorithm

potential gadgets. We just finished the first step to identify all the gadgets. Then, we need to choose appropriate gadgets from a large number of gadgets, this step requires the attacker to manually complete. We firstly have to pick out a dispatcher gadget which is reachable in accordance with the normal control flow of the program and we are able to control the data in the dispatcher gadget through a memory vulnerability. Then, according to the attack intention, we need to pick out the function gadgets that can be used to complete the attack. And these function gadgets are able to achieve attack semantics through mutual cooperation. In the third step, we need to construct the gadget chain. We implement the attack semantics by performing function gadgets in a certain execution order. We construct the gadget chain according to the order that the gadgets are executed. Before building the gadget chain, we need to understand the layout of the program in memory, and get the address of each gadget. This can be achieved through memory leaks. Finally, we make use of memory vulnerabilities, such as buffer overflow vulnerability or any writing memory vulnerability, to insert the gadget chain into memory. When the program executes and reaches dispatcher gadget, we start to attack. After the dispatcher gadget dispatches all the function gadgets, we achieve the malicious attack.

V. Proof of FOP Exploit

In order to prove that FOP attack is feasible, we count the number of various gadgets in some C programs and in the Linux x64 environment, we make use of ProFTPD1.3.0 server to achieve a FOP attack.

```
**Input**: $P$:- the vulnerable program.
**Output**: $D$:- writing parameter register gadget set;
$NUMS$:- the number of writing parameter register gadget;
$GVAL$:- global variables in each writing parameter register gadget;
$GVNUMS$:- the number of global variables in each writing parameter register gadget.
$D = \emptyset$; $NUMS = 0$; $GVAL = \emptyset$; $GVNUMS = 0$;
while $f = get\_function(P)$ do
    $gadgets = \emptyset$;
    for $instruction = get\_instruction(f)$ do
        if $is\_call\_instruction(instruction)$ then
            $callee\_f$ = get_callee_function($instruction$);
            $gadgets = \{f\}$;
            $GVAL, GVNUMS$ = find_global_variables($f$);
            for $callee\_ins = get\_instruction(callee\_f)$ do
                if $is\_call\_instruction(callee\_ins)$ then
                    $gadgets = \emptyset$;
                    $GVAL = \emptyset$;
                    $GVNUMS = 0$;
                    break;
                end
            end
        end
    end
    if $gadgets\ != \emptyset$ then
        $D = D \bigcup gadgets$;
        $NUMS = NUMS$+1;
    end
end
```

Algorithm 2: Writing parameter register gadget identification algorithm

We select C programs from the CVE database. And we make use of the tool fop-gadgets to analyze these C programs. Table II shows the number of various gadgets in these C programs. The column 1 presents the name of a C program, and the column 2 presents the CVE number. The column 2-7 denote the number of various gadgets. The last line presents the total number of gadgets in each type. From Table II, we can see that there are a lot of gadgets for each gadget type, and these gadgets provide the possibility for attackers to achieve FOP attacks. According to the definition of the dispatcher gadget, it is the most difficult for a attacker to find a dispatcher gadget. But from the Table II, we can see that there is more than one dispatcher gadget in each C program. So there are enough gadgets for attackers to utilize in the C program.

In the Linux x64 environment, we make use of ProFTPD1.3.0 server to achieve a FOP attack. In the following, we firstly introduce the vulnerability type of the ProFTPD1.3.0 server , and then we describe how to trigger this remote buffer overflow vulnerability, and finally we introduce how to achieve FOP attacks.

ProFTPD1.3.0 is an open source FTP server software. Fig.11 shows a stack-based remote buffer overflow vulnerability in the ProFTPD1.3.0 function *sreplace( )*[15]. The function *sreplace( )* is located in the *src/support.c* file. This vulnerability allows a remote attacker to implement a denial of service attack or execute arbitrary machine code. We can make use of this remote buffer overflow vulnerability to achieve a FOP attack.

```
char *sreplace(pool *p, char *s, ...) {
  va_list args;
  char *m,*r,*src = s,*cp;
  char **mptr,**rptr;
  char *marr[33],*rarr[33];
  char buf[PR_TUNABLE_PATH_MAX] = {'\0'}, *pbuf = NULL;
  size_t mlen = 0, rlen = 0, blen;
  int dyn = TRUE;
  cp = buf;
  .......
  while (*src) {
    for (mptr = marr, rptr = rarr; *mptr; mptr++, rptr++) {
      ......
      if (strncmp(src, *mptr, mlen) == 0)
        sstrncpy(cp, *rptr, blen - strlen(pbuf));
    .......
    }
    ......
  }
}
```

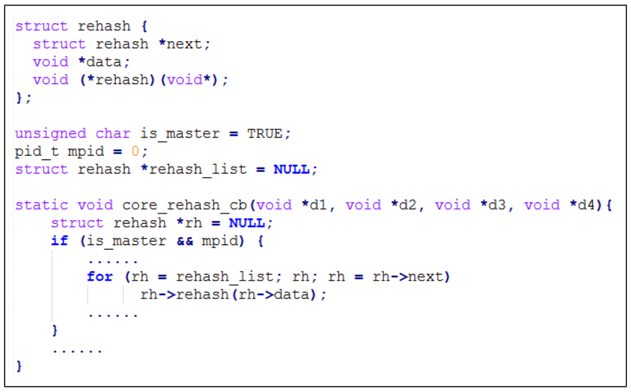
Fig. 11: Remote buffer overflow vulnerability

In ProFTPD1.3.0 server, if there is a *.message* file in a directory, when we firstly enter this directory, the server will read the contents of *.message* file into memory and display the contents of the file to the user. In the above process, the function *sreplace( )* is executed. The function *sreplace( )* will generate a remote buffer overflow vulnerability. So, we firstly need to create a writable directory, and then upload a *.message* file to this writable directory, and finally enter this directory to trigger the remote buffer overflow vulnerability through the CWD command.

We firstly used the tool fop-gadgets to analyze the ProFTPD1.3.0 server source code. We found a lot of matching dispatcher gadgets and function gadgets. The number of various FOP gadgets is shown in Table II. The purpose of the FOP attack is to execute arbitrary machine code. Fig.12 is a D-G that we have selected. The D-G is located in the *src/main.c* file. The structure *rehash* includes three members, the function pointer *rehash*, the void type pointer *data*, and the variable *next* pointer that points to the structure *rehash*. *is_master* and *mpid* are two global variables. And *rehash_list* is a pointer that points to the structure *rehash*. The function *core_rehash_cb( )* is a D-G. The function *core_rehash_cb( )* has a loop, and each loop calls the function pointer *rehash* and uses the element *data* in the structure *rehash* as the function parameter. Fig.13, Fig.14 and Fig.15 depict three function gadgets. These function gadgets can read datas from the specific file, and then write datas into the specific memory. Another function gadget is library function *system()*. This FOP attack used a dispatcher gadget and four function gadgets. As shown in Fig.16, we constructed the gadget chain by counterfeiting the *rehash_list* linked list. Through the first three function gadgets, we can write a command or the name of executable file into the memory, and use it as the parameter of the function *system()*. This FOP attack can execute arbitrary machine code.

TABLE II: The Number of Various FOP Gadgets

NAME	CVE	D-G	W-M-G	ARITHMETIC-G	LOGIC-G	W-P-R-G	C-B-G	TotalGadgets
OpenSSL	CVE-2016-7052	108	358	425	304	117	417	1729
Wget	CVE-2016-4971	15	121	115	50	23	82	406
Wireshark	CVE-2014-2299	160	29426	6865	5421	20453	1111	63436
nginx	CVE-2013-2028	78	111	177	201	6	103	676
ProFTPD	CVE-2006-5815	35	166	62	125	23	88	499
Total		396	30182	7644	6101	20622	1801	66746

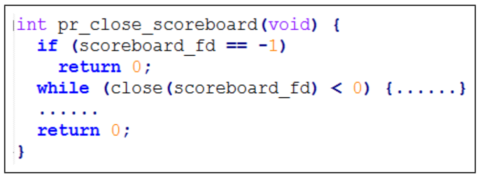


```
struct rehash {
  struct rehash *next;
  void *data;
  void (*rehash)(void*);
};

unsigned char is_master = TRUE;
pid_t mpid = 0;
struct rehash *rehash_list = NULL;

static void core_rehash_cb(void *d1, void *d2, void *d3, void *d4){
    struct rehash *rh = NULL;
    if (is_master && mpid) {
        ......
        for (rh = rehash_list; rh; rh = rh->next)
            rh->rehash(rh->data);
        ......
    }
    ......
}
```

Fig. 12: Depict a dispatcher gadget

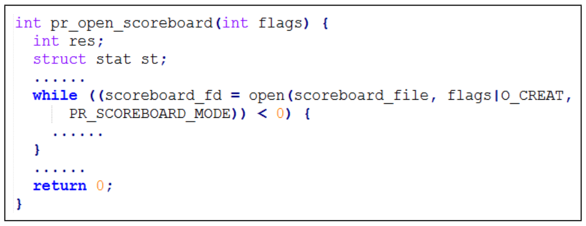


```
int pr_close_scoreboard(void) {
  if (scoreboard_fd == -1)
    return 0;
  while (close(scoreboard_fd) < 0) {......}
  ......
  return 0;
}
```

Fig. 13: The function $pr\_close\_scoreboard()$

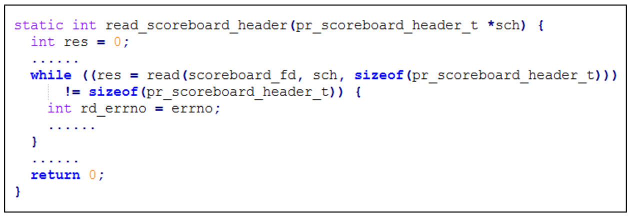
We used Python language to write an attack script. We firstly established a connection with the ProFTPD1.3.0 server, and then triggered the remote buffer overflow vulnerability through the above-mentioned method. By repeatedly triggering the ProFTPD1.3.0 server remote buffer overflow vulnerability, we controlled the global variables *is_master* and *mpid* so that the dispatcher gadget(D-G) was normally executed and was able to successfully dispatch function gadgets. And we successfully inserted the gadget chain into the program memory. ProFTPD1.3.0 server performed D-G in accordance with normal execution flow of the program. We made use of the FOP attack to execute arbitrary machine code.

In this function-oriented programming(FOP) attack, we used a dispatcher gadget and four function gadgets, and successfully achieved a FOP attack. This FOP attack is relatively simple. There are enough gadgets for attackers to achieve a FOP attack. So if the attacker wants to achieve more complex attacks, it is also possible. More complex attacks require more gadgets and the gadget chain will be more complicated. Here, we just realized a simple FOP attack so that people can better understand the FOP attack.



```
int pr_open_scoreboard(int flags) {
  int res;
  struct stat st;
  ......
  while ((scoreboard_fd = open(scoreboard_file, flags|O_CREAT,
      PR_SCOREBOARD_MODE)) < 0) {
    ......
  }
  ......
  return 0;
}
```

Fig. 14: The function $pr\_open\_scoreboard()$



```
static int read_scoreboard_header(pr_scoreboard_header_t *sch) {
  int res = 0;
  ......
  while ((res = read(scoreboard_fd, sch, sizeof(pr_scoreboard_header_t)))
      != sizeof(pr_scoreboard_header_t)) {
    int rd_errno = errno;
    ......
  }
  ......
  return 0;
}
```

Fig. 15: The function $read\_scoreboard\_header()$

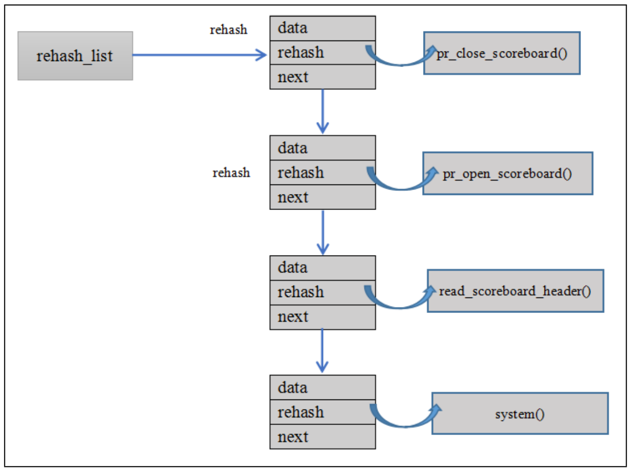


Fig. 16: Counterfeiting the rehash_list linked list

VI. Related Work

ROP[3] attack can bypass DEP[1] and coarse-grained CFI, and shadow stack technology can prevent ROP attack. There are many variants of ROP attack[12], [13], [14], such as JOP[4] attack, they all make use of short instruction sequences as gadgets. Compared with ROP and its variants, FOP use a whole function as the gadget. FOP attack is similar to

Return-into-libc[2] and COOP[5] attack, but there are crucial differences among them. Return-into-libc attack jumps to the library function to execute malicious code through the return instruction. So shadow stack can prevent Return-into-libc attack. FOP attack does not modify return address and makes use of functions of the source code as gadgets. So FOP attack can bypass shadow stack. COOP attack uses C++ virtual functions as gadgets, and it needs to modify virtual function table. Virtual table verification(VTV) technology can defend against COOP attacks. Compared with COOP attack, Fop attack is aimed at C program, and every C function could become a gadget. Types and algorithms of gadgets are different, and we provide a tool fop-gadgets to find FOP gadgets. FOP attack model needs to take advantage of function pointers, and the CPI[7] technique is mainly to protect the code pointer. But CPI can be broken[24], and attackers can get the address of the safe area. If we can break the CPI, we can control the sensitive pointers through the memory vulnerabilities. So we can also introduce FOP malicious attacks.

VII. Conclusion

In this paper, we introduce function-oriented programming (FOP), a new code reuse attack technique. FOP attack technology can bypass many existing defense mechanisms, such as coarse-grained CFI, stack canaries, shadow stacks, Control Flow Guard, Return Flow Guard, Indirect Function-Call Checks and Reuse Attack Protector. We provide a tool fop-gadgets to find FOP gadgets. And we provide the FOP attack model and algorithms of various gadgets. We have counted the number of various gadgets in some C programs from CVE and in the Linux x64 environment, we make use of ProFTPD1.3.0 server to achieve a FOP attack. We have proved that FOP attack is feasible. The FOP attack is a potential threat. We hope that our research result will be of great concern. We believe that more researchers can develop more efficient, more practical, and more secure defense mechanisms to prevent code reuse attacks.

Acknowledgment

This work is partially supported by the National Natural Science Foundation of China (No. 61602469), and the Fundamental theory and cutting edge technology Research Program of Institute of Information Engineering, CAS(Grant No.Y7Z0411105).

References

[1] Microsoft. Data Execution Prevention (DEP). https://support.microsoft.com/en-us/help/875352, 2017.

[2] Nergal. The Advanced Return-into-lib(c) Exploits: PaX Case Study. Phrack Magazine, Volume 11, Issue 0x58, 2001.

[3] H. Shacham. The geometry of innocent flesh on the bone: Return into-libc without function calls (on the x86). In Proceedings of ACM Conference on Computer and Communications Security (CCS), 2007.

[4] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: A new class of code-reuse attack. In ACM Symposium on Information, Computer and Communications Security (ASIACCS), 2011.

[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 Proceedings of the 36th IEEE Symposium on Security and Privacy, 2015.

[6] M. Abadi,M. Budiu, J. Ligatti, and U. Erlingsson. Control-Flow Integrity in Proc. the 12th ACM Conference on Computer and Communications Security(CCS05), pp.340-353, 2005.

[7] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song,Code-pointer Integrity, in Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, 2014.

[8] PaX ASLR Documentation. http://pax.grsecurity.net/docs/aslr.txt.

[9] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: An E_cient Approach to Combat a Broad Range of Memory Error Exploits. 12th USENIX Security, 2003.

[10] S. Bhatkar, R. Sekar, and D. C. DuVarney. E_cient Techniques for Comprehensive Protection from Memory Error Exploits. 14th USENIX Security, 2005.

[11] K. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. Just-In-Time Code Reuse:On the Effectiveness of Fine-Grained Address Space Layout Randomization in Proc. the 2013 IEEE Symposium on Security and Privacy.(SP13), pp. 574-588,2013.

[12] N. Carlini ,D. Wagner. ROP is still dangerous: Breaking modern defenses in Proc. the 23rd USENIX Security Symposium.(Usenix14) pp. 385-399, 2014.

[13] L. Davi, A. Sadeghi, D. Lehmann. and F. Monrose, Stitching the Gadgets On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection in Proc. of the 23rd USENIX Security Symposium.(Usenix14), pp. 401-416, 2014.

[14] S. Checkoway, L. Davi, A. Dmitrienko, and H. Shacham. Return-oriented programming without returns in Proc. the ACM Conference on Computer and Communications Security (CCS10), pp. 559-572, 2010.

[15] Stack-Based Buffer Overlfow in the sreplace Function in Proftpd, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5815.

[16] Bittau A, Belay A, Mashtizadeh A, et al. Hacking blind[C]Security and Privacy (SP), 2014 IEEE Symposium on. IEEE, 2014: 227-242.

[17] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In Symposium on Network and Distributed System Security (NDSS), 2014.

[18] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In USENIX Security Symposium, 2013.

[19] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In IEEE Symposium on Security and Privacy, 2013.

[20] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In USENIX Security Symposium, 2013.

[21] Hu, Hong and Shinde, Shweta and Adrian, Sendroiu and Zheng, Leong Chua and Saxena, Prateek and Liang, Zhenkai. Hu, Hong and Shinde, Shweta and Adrian, Sendroiu and Zheng, Leong Chua and Saxena, Prateek and Liang, Zhenkai. Security and Privacy. pp. 969-986, 2016.

[22] Liu Z, Xu B, Liang D, et al. Semantics-based memory leak detection for C programs[C] International Conference on Fuzzy Systems and Knowledge Discovery. 2015:2283-2287.

[23] Ghanavati, Mohammadreza, and A. Andrzejak. Automated memory leak diagnosis by regression testing. IEEE, International Working Conference on Source Code Analysis and Manipulation IEEE, 2015:191-200.

[24] Isaac Evans, Sam Fingeret, Julin Gonzlez, Ulziibayar Otgonbaatar,Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the point (er): On the effectiveness of code pointer integrity. In Security and Privacy (SP), 2015 IEEE Symposium on, pages 781C796. IEEE, 2015.

[25] L. Szekeres and M. Payer and T.Wei and D. Song. Sok: Eternal war in memory. In IEEE Symposium on Security and Privacy,2013.