2012 Second International Conference on Instrumentation & Measurement, Computer, Communication and Control

A Directed Fuzzing Based on the Dynamic Symbolic Execution and Extended Program Behavior Model

Zhe Chen, Shize Guo, Damao Fu
North electronic equipments research institute
Beijing, China
cyshmilu@yahoo.com.cn

Abstract—**This paper presents a new automated directed fuzzing technique. First, the behavior information is extracted from the original complex Control Flow Graph (CFG) by using the dynamic symbolic execution. Then, the case theory is used to establish the access control model for the access objects. Subsequently, to describe some access properties of the objects while a program is running, we present a control flow based Extended Program Behavior model with Finite-State Machine controlled parameters (EPBFSM) by adding constraints to the control flow model. Finally, the new fuzzed inputs are generated by resolving the constraints resulting from the EPBFSM. By combining the program behavior with the security model, we can find not only the possible path-aware vulnerabilities but also the possible access control objects-aware vulnerabilities.**

Keywords-directed fuzzing; control flow graph; dynamic symbolic execution; access control model; extended program behavior model; finite-state machine

I. INTRODUCTION

Over the past few years, Fuzz testing techniques have been used to discover hundreds of vulnerabilities in an extensive range of software. Fuzzing is often employed as an automatic black box testing methodology that uses a fuzzer to randomly generate or mutate sample inputs. For the most part, it is a brute force technique which is simple and effective in exposing errors in the field of testing input parsing components. However, the black box testing technique has following drawbacks. Firstly, for the deeper semantic errors in program, the probability of producing valid random inputs which satisfy the basic syntactic constraints may be low. Secondly, as for the deep program paths which can be reached only when many conditions are required to be simultaneously true, the probability of exposing such type of incorrect behavior is low. In recent years, many researchers have attempted to give guidance to intelligent fuzzing so as to gain higher code coverage and path coverage, though intelligent fuzzing entails manual intervention. A typical guidance is the manually-written language grammar [1][2] that helps fuzzing tools generate legal and illegal input values. However, due to the burdensome manual work in building guidance, very few real applications under test can be fed with well-structured inputs. Apart from the black box testing technique, white box testing is a kind of source code analysis technology also called structural testing. The common test criteria of white box testing are code coverage, statement coverage and branch coverage. Due to the fact that all source codes are available, all possible code paths can be checked for potential vulnerabilities. However, it will lead to false positives. Furthermore, the source code is not always available. To describe some access properties of the objects while a program is running and enhance the ability of the software vulnerability analysis, this letter first establishes the access control model for the access objects using the case theory, and then presents a control flow based Extended Program Behavior model with Finite-State Machine controlled parameters (EPBFSM) by adding constraints to the control flow model.

The rest of the paper is organized as follows. In section 2, we summarize the theoretic fundamental of dynamic symbolic execution and describe how to extract the behavior information from the original complex CFG by the use of dynamic symbolic execution. Section 3 first establishes the access control model to the access objects using the case theory. Then it presents a control flow based extended program behavior mode with parameters based on the FSM (EPBFSM) by adding constraints to control flow model which can describe some access properties of the objects while a program is running and enhance the ability of the software vulnerability analysis. Section 4 depicts the experimental results. The paper concludes in section 5.

II. THE DYNAMIC SYMBOLIC EXECUTION

Symbolic execution was studied in the 1970's. However, thirty years ago, effective solutions and tools were absent for several insurmountable technical challenges such as the incomputable and undecidable problems. Recently, symbolic execution have been developing a lot thanks to the powerful modern computers, symbolic execution engines and constraint solvers, the practical software model checkers and so on. Nowadays, dynamic symbolic execution is one of the hot aspects in the area of the computer security research.

Dynamic symbolic execution executes the subject program both concretely and symbolically. It collects constraints on a seed input obtained from predicates in branch statements along the execution and will derive new inputs from previous path constraints so as to guide next executions towards new program paths. Dynamic symbolic execution is now widely used in the area of software vulnerability research due to its two obvious properties. One is that it does not lead to false positives because of the enough run-time information afforded by the executing program. The other is that it has a high code coverage because the new inputs generated by the solver are

978-0-7695-4935-4/12 $26.00 © 2012 IEEE
DOI 10.1109/IMCCC.2012.382

able to cover new codes. Below, we provide a simple example to illustrate how to generate tests automatically by means of dynamic symbolic execution.

Definition 1: Let $V=\{ v_1,v_2,\ldots,v_N \}$ be the set of nodes, where v_i ($1\le i\le N$) is the basic block of the program. Specifically, $s\in V$ is defined as the entry basic block of the program, while $f\in V$ is defined as the exit basic block of the program.

Definition 2: Let $E=\{<v_i,v_j>|v_i,v_j\in V\}$ be the set of directed edges between nodes, where $<v_i,v_j>$ denotes the control transfer from node v_i to v_j, and it is associated with a predicate that represents the condition of control transfer from v_i to v_j.

Definition 3: Let $G=(V,E,s,f)$, where E is the Control Flow Graph (CFG) of the program.

Definition 4: A path p is defined as a sequence of nodes in a control flow graph.

Let's denote

$$p = v_{p_1} v_{p_2} \cdots v_{p_{m+1}}$$

where

$$s = v_{p_1} \text{ and } f = v_{p_{m+1}}$$

Also p can be denoted as

$$p = e_{p_1} e_{p_2} \cdots e_{p_m}$$

For each execution path p, we can construct a path constraint c step by step in order to characterize the input values for which the program executes along p. All the variables appearing in c are the expression of input parameter value. The constraint is expressed in the theories T decided by the logic system. A constraint solver is an automated theorem prover which can judge whether the path constraint c is satisfied or not. If the constraint c is satisfied, the assignment of all input parameters appearing in the constraint c can be obtained by the solver. Then the solutions to c can tell us the inputs that satisfy the program towards p.

Example 1. Consider the following C function.

```
1: void foo(int x, int y)
2: {
3:    int counter=0;                /*s1*/
4:    while (x<10)                  /*s2*/
5:      x++;                        /*s3*/
6:    while (y<20)                  /*s4*/
7:      y++;                        /*s5*/
8:    if (x = = 10)                 /*s6*/
9:      counter++;                  /*s7*/
10:   if (y = = 20)                 /*s8*/
11:     counter++;                  /*s9*/
12:   if (counter= = 2)             /*s10*/
13:     error();                    /*s11*/
14: }
```

The control flow graph of Example 1 can be illustrated in Fig.1. The path feasibility problem is undecidable in general. But if we restrict the constraint of the path (e.g., only with Boolean and linear arithmetic constraints), the problem becomes solvable.

At first, the input parameters are generated randomly. The dynamic instrumentation tools such as CIL[3,4] (C Intermediate Language), Pin[5,6] and Valgrind[7,8] can capture the inputs in run-time. Then the concrete inputs will be symbolized, i.e., x, y will be mapped to symbols "input(0)" and "input(1)" respectively. If the comparisons in Statements 4 and 6 are true and the comparisons in Statements 5 and 7 are false, we can obtain the path constraint for the first run c= <input(0)<10, input(1)>=20>.

We must calculate a solution to a different path constraint to let the program execute a different path. For example, we can negate the last predicates of the current path constraint c and get a new constraint <input(0)<10, input(1)<20>. Now a new test case obtained by the solution can make the comparisons in Statement 6 true. By repeating this process, all the feasible program paths can be executed. Finally, the test case x<10 and y<20 solved from the path constraint <input(0)<10, input(1)<20> is able to trigger the error in Statement 13.

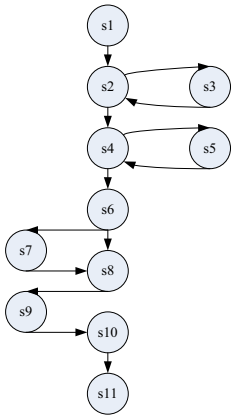

Figure 1. The control flow graph of Example 1

Obviously, dynamic symbolic execution is smarter than white box testing. The problems resulting from extern complex arithmetic functions can be alleviated by using dynamic symbolic execution due to the help of run-time information of program. Let us consider the above example again. In the first run, we cannot generate the input values x and y to trigger the error, but we can collect the calculation of y dynamically. In the next run, we can set the new input y that is smaller than 20 by fixing the former y, so as to force the execution towards the "then" branch.

For many programs, source code is not available. To disassemble the binary, in this paper, we use IDA Pro[9]. IDA Pro is so powerful that it can provide comprehensive information about targets of control flow instructions. Based on

the CFG for the entire component, the next step is to isolate those parts of the graph that are responsible for handing events. In particular, we are interested in all subgraphs of the CFG that contain the code to handle the different events. While traversing the graph, the static analysis process inspects all instructions to identify those operating system calls. To describe the behavior of the program more accurately, we utilize the information of the library, function calls, system calls, transformation functions which include encryption and decryption, compression and decompression, code packing and unpacking, checksums to research the behavior of the program. To extract the behavior information from the original complex CFG provided by the IDA Pro, we must abstract some concepts. CM (Call Module) refers to a function call or a system call module which has a relatively stable and independent sequence of functions invoking when the program runs normally. Obviously CM is made up of sequential instruction pieces sets and sequential instruction pieces relationship sets which can be obtained from the original complex CFG coming from the IDA Pro. For the sake of further obtaining the CM's behavior characteristic, we must pay more attention to the following places: the start address of the current function, the return address of the function, the address where the jump instructions jump to and the next instruction's address of the jump instructions. This paper uses the finite automaton theory [10,11] to model the program behavior in the following section.

III. THE EXTENDED PROGRAM BEHAVIOR MODEL

To extract a parameter abstraction [12] at the binary level and translate the assembly parameters into the formal parameters in the function's prototype, we make use of the dynamic symbolic execution analysis as described in Section 2. We are required to identify the important attributes such as the parameters type (input, output, input-output), the parameter location (register, stack, table) and the operation to the parameters (read, write) from a given execution trace, and finally combine this information across multiple function runs.

The traditional control flow model can only analyze those vulnerabilities concerning the change of the execution path. For the vulnerabilities invoking by the misuse of access control of the parameter, it is helpless. In this section, we first establish the access control model to the access objects using the case theory. Then we use the EPBFSM to model the program behavior. By combining the program behavior with the security model, we can not only find the possible path-aware vulnerabilities but also the possible access control objects-aware vulnerabilities.

A. The Access Control Model of Program

Any computer system includes two types of entities which we call them subject and object. The subject often refers to the process or the thread of a program. The object often refers to the memory, register, stack, heap or table that stores the information of a program. Here, we use the case theory to model the security mechanism of the subject and object of the program.

Definition 5: Let $O_0=\{o_1,o_2,\ldots,o_n\}$ be the finite set of all the objects in a program. As described in the above, here we have O_0 ={memory, register, stack, heap, table}. In addition, the power set of O_0 is denoted as $P(O_0)$.

Definition 6: Let $\boldsymbol{O}=\{O_i|O_i\in P(O_0)\}$ be the set of all the possible objects which can be accessed by the system calls or the functions of a program safely.

Definition 7: The object security domain model of a program is an algebraical system $<\boldsymbol{O},\oplus,\otimes,\Phi,O_0>$. The empty set Φ is the zero element of $\boldsymbol{O}$ and the set O_0 is the unit element of $\boldsymbol{O}$. For any $O_i,O_j\in\boldsymbol{O}$, the operations $O_i\oplus O_j$, $O_i\otimes O_j$ and the order relation $O_i\leq O$ are defined as $O_i\cup O_j$, $O_i\cap O_j$, and $O_i\subseteq O_j$ respectively. Thus the algebraical system $\boldsymbol{O}$ is a case, the order relation $\leq$ is a partial order relation, and $<\boldsymbol{O},\leq>$ is the partial set.

Definition 8: Let $\boldsymbol{S}=\{S_1,S_2,\ldots,S_n\}$ be the set of all the security levels. $\boldsymbol{S}$ is often defined by the system security administrator. Here we define $\boldsymbol{S}=\{U,C,LS,MS,TS\}$, where U means Unclassified, C means Confident, LS means LowSecuret, MS means MiddleSecuret and TS means TopSecuret.

Definition 9: The security level model of the object is an algebraical system $<\boldsymbol{S},\oplus,\otimes,low,high>$. The lowest security level is low which is the zero element of $\boldsymbol{S}$, and the highest security level is high which is the unit element of $\boldsymbol{S}$. For any $S_i,S_j\in\boldsymbol{S}$, the operations $S_i\oplus S_j$, $S_i\otimes S_j$ and the order relation $S_i<S_j$ are defined as $\max(S_i,S_j)$, $\min(S_i,S_j)$ and $i<j$ respectively. Thus the algebraical system $\boldsymbol{S}$ is a case. The order relation $<$ is simply the order relation. The instantiation here is $U<C<LS<MS<TS$. $<\boldsymbol{S},<>$ is the simply-ordered set.

Definition 10: Let $\boldsymbol{L}=\boldsymbol{O}\times\boldsymbol{S}=\{(O_i,S_i)|O_i\in\boldsymbol{O},S_i\in\boldsymbol{S}\}$, where $\boldsymbol{L}$ is the direct product of the set $\boldsymbol{O}$ and $\boldsymbol{S}$.

Definition 11: The multi-level security domain model of the program is an algebraical system $<\boldsymbol{L},\oplus,\otimes,(\Phi,low),(O_0,high)>$. The zero element of $\boldsymbol{L}$ is (Φ,low) and the unit element of $\boldsymbol{L}$ is $(O_0,high)$. For any $(O_i,S_i),(O_j,S_j)\in\boldsymbol{L}$, the operations $\oplus$, $\otimes$ and the order relation $\leq$ are defined as $(O_i,S_i)\oplus(O_j,S_j)=(O_i\cup O_j,\max(S_i,S_j))$, $(O_i,S_i)\otimes(O_j,S_j)=(O_i\cap O_j,\min(S_i,S_j))$ and $(O_i,S_i)\leq(O_j,S_j)\rightarrow(O_i\leq O_j,S_i<S_j)$ respectively. Thus the algebraical system $\boldsymbol{L}$ is a case.

Definition 12: The access control model of program is denoted as $\boldsymbol{PAC}=(\boldsymbol{E},\boldsymbol{O},\boldsymbol{A},\boldsymbol{L},f)$. Here $\boldsymbol{E}$ is the set of subject which refers to a library, system call, a function, a process and a thread. $\boldsymbol{O}$ is the set of object which refers to memory, register, stack, heap and table. $\boldsymbol{A}$ is defined as $\boldsymbol{E}\times\boldsymbol{O}\rightarrow\{\Phi,\{read\},\{write\},\{read,write\}\}$ which is the access mode function from the subject to the object. $\boldsymbol{L}$ is the multi-level security domain model of the program according to ***Definition 11***. f is defined as $\boldsymbol{E}\cup\boldsymbol{O}\rightarrow\boldsymbol{L}$ that is the function from the subject or object to the multi-level security. The security level of the subject is a ternary denoted as $(L_{min},L_{cur},L_{max})$, where $L_{min},L_{cur},L_{max}\in\boldsymbol{L}$, L_{min} means the minimal security level of the subject, L_{max} is the maximal security level of the subject, and L_{cur} is the current security level admitted by the system. Obviously, we have $L_{min}\leq L_{cur}\leq L_{max}$. The security level of the object is denoted as $L_{obj}\in\boldsymbol{L}$.

B. The proposed EPBFSM

In this subsection, we give the definitions of proposed EPBFSM as follows.

Definition 13: An EPBFSM is defined as a septenary M = (***I***, ***S***, ***PAC***, **y**, *T*, *s*, *Q*). Here, ***I***={Entry, Exit} is the set of the input event symbols. Entry implies the entrance of a system call or a function. Exit means the exit of a system call or a function. ***S*** is the set of states. ***PAC*** is the vector of state parameter as described in ***Definition 12***. **y** is the vector of the input parameter. **y**=(img_id, offset, object), where img_id is the execution file or the share file symbol when the input event occurs, offset is the opposite value of the system call or the function in the file, and object is the access object that the input event will visit. *T* is the set of transition functions. *s* is the initial state. *Q* is set of the final state.

For any $t \in T$, t is a quadruple (s_t, *event*, q_t, P_t), where s_t is the current state, *event* is the input event symbol, q_t is the next state, and P_t is the predicate on the valuation of ***PAC***. In general, the input parameter vector is denoted as P_t(***PAC***, **y**). The predicate operation can cause the state transition from s_i to s_j if and only if P_{ij}(***PAC***, **y**)=***TRUE***. With regard to the input parameter **y**=(img_id, offset, object), there are 3 cases:

Case 1: The system call or function identified by the (img_id, offset) is reading a memory. In this case, if P_{ij}(***PAC***, **y**)=(***PAC***.$L_{obj} \le$ ***PAC***.L_{max}) $\wedge$(***PAC***.***A***(**y**.(img_id, offset),**y**.object)=={read})=***TRUE***, the state will be transited from s_i to s_j.

Case 2: The system call or function identified by the (img_id, offset) is writing a memory. In this case, If P_{ij}(***PAC***, **y**)=(***PAC***.$L_{cur} \le$ ***PAC***.L_{obj})$\wedge$(***PAC***.***A***(**y**.(img_id, offset), **y**.object)=={write})=***TRUE***, the state will be transited from s_i to s_j.

Case 3: The system call or function identified by the (img_id, offset) is executing a memory. In this case, if P_{ij}(***PAC***, **y**)=(***PAC***.L_{cur}=***PAC***.L_{obj})$\wedge$(***PAC***.***A***(**y**.(img_id, offset), **y**.object)={*read*,*write*}) = ***TRUE***, the state will be transited from s_i to s_j.

IV. THE DIRECTED XPERIMENTAL RESULTS

To evaluate the effectiveness of our directed fuzzing, we perform directed fuzzing and random fuzzing on vsftpd respectively. Table 1 shows the experimental results. The second column stands for the total number of fuzzed test inputs, the third lists the number of total errors (TE), the fourth shows the number of distinct errors (DE) responsible for these failures, the fifth gives the mean stack depth (MSD) when the failure occurred, the sixth presents the coverage rate (CR) of binary codes of the application and the last presents the false positive rate (FPR).

TABLE I. THE COPARISON OF DIRECTED AND RANDOM FUZZING

Method	Fuzzed tests	TE	DE	MSD	CR	FPR
Directed	*2472*	*117*	*6*	*25*	*0.28*	*0.036*
Random	*9587*	*183*	*2*	*17*	*0.16*	*0.024*

V. CONCLUSION

In this paper, we compare some useful testing approaches. Fuzzing offers clear advantages in automation, ease of use and its ability to find the unexpected vulnerabilities of the program. Our new directed fuzzing implements the information of the attack point-level control flow and access control model to resolve the constraints of the dynamic symbolic execution to generate new inputs. Testing is currently the most effective and widely used technique to analyze the software vulnerability. By using the dynamic symbolic execution, the access control mode of program and the extended program behavior model with parameters based on the FSM, our directed fuzzing technique promises to help developers find and eliminate deep subtle errors more quickly, efficiently, and with less effort.

REFERENCES

[1] H. C. Kim, Y. H. Choi, D. Lee, and D. H. Lee, "Practical security testing using file fuzzing," Proc. 10th Int. Conf. on Advanced Communication Technology, vol. 2, 1304-1307, February. 2008.

[2] H. C. Kim, Y. H. Choi, and D. H. Lee, "Efficient file fuzz testing using automated analysis of binary file format," Journal of Systems Architecture, vol.57, no.3, pp. 259-268, March 2010.

[3] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: intermediate language and tools for analysis and transformation of C programs, " Proc. 11th Int. Conf. on Compiler Construction, pp. 213-228, April 2002.

[4] "CIL (C Intermediate Language)", Idgay, Mattharren and Necula, http://cil.sourceforge.net/, accessed Jun. 15. 2011.

[5] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation, " Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, pp. 190-200, June 2005.

[6] "Pin: A Dynamic Binary Instrumentation Tool," http://www.pintool.org/, accessed Jun. 15. 2011.

[7] N. Nethercote, and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," SIGPLAN Notices, vol. 42, no.6, pp. 89-100, June 2007.

[8] "Valgrind," Valgrind Developers, http://valgrind.org/, accessed Jun. 15. 2011.

[9] Data Rescure, "IDA Pro: Disassembler and Debugger", http://www.hex-rays.com/idapro/, , accessed Jun. 15. 2011.

[10] A. D. Friedman, and P. R. Menon, Fault Detection in Digital Circuits, New Jersey: Prentice-Hall, 1971.

[11] Z. Kohavi, Switching and Finite Automata Theory, 2nd Ed., New York : McGraw-Hill, 1978.

[12] A. V. Aho, R. Sethi, and J. D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986.