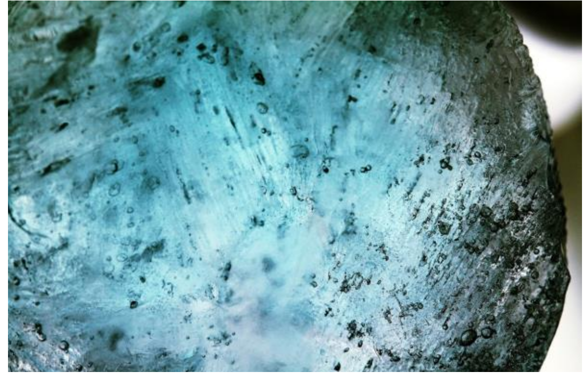
Exploit Generation from Software Failures

Shih-Kun Huang
Information Technology Service Center
National Chiao Tung University
Hsinchu, Taiwan
skhuang@cs.nctu.edu.tw

Han-Lin Lu
Department of Computer Science
National Chiao Tung University
Hsinchu, Taiwan
luhl@cs.nctu.edu.tw



***Abstract*—We normally monitor and observe failures to measure the reliability and quality of a system. On the contrary, the failures are manipulated in the debugging process for fixing the faults or by attackers for unauthorized access of the system. We review several issues to determine if the failures (especially the software crash) are reachable and controllable by an attacker. This kind of efforts is called exploitation and can be a measurement of the trustworthiness of a failed system.**

Keywords—automatic exploit generation; control flow hijaction

I. INTRODUCTION

The failure exploitation methods are to manifest the room for security breaches from the observed failures. The motivation of this type of work is rooted from generating attack inputs to compromise the system and prioritize the bug fixing order. Since failure of software is inevitable and if there are a large number of failures, we need a systematic approach to judge whether they are exploitable. In Miller et al.'s crash report analysis, the authors analyze crashes by BitBlaze [1]. Compared with !exploitable [2], the results show that exploitable crashes could be diagnosed in a more accurate way. Moreover, crash analysis plays an important role to prioritize the bug fixing process [3]. A proven exploitable crash should be the top priority bug to fix. A general review and recent advances are described in [4]. Research insight about exploit generation is analyzed in [5]. Recent work has proved feasibility for common linux and windows applications [6-8], Microsoft office [9, 10] and web applications [11].

Software crash is a special case of control-flow hijacking by the exception handler, raised by the protection hardware. If the program accesses an invalid address (program counter, or memory data access), the memory protection hardware will be signaled. It is due to an incorrect memory update of run-time context or data pointers. If the update is derived from user inputs, run-time context (especially program counter or called instruction pointer and frame pointer) and pointers can be manipulated. If we manipulate the run-time context by deriving the user input, this exploitation process is called failure hijacking. For example, the instruction pointer (or program counter) IP can be related to the failure input with a set of constraints as follows:

$$IP = F(failure\text{-}input)$$

We are able to control the value of IP by resolving the above constraint. The function with path condition is constructed by a failure input feeding to a concolic execution[12].

A failure is the observable event that violates predefined specification. Software crash is a kind of failure that raised from the execution environment, such as run time protection added by the compiler or address protection by the memory management unit. However, many types of failures may not be easily detected unless a predefined specification is enforced by a violation checker. To our problem setting, if the failure is to be controlled by an attacker, we should have tagged the source of the attacker input and monitor the potential outcome to see if the tagged input will eventually influence the failure. There are several types of failures, some of which will raise run-time exceptions, and some of which won't. We view exploit as the manipulation of the software. Exploit generation process is to find input that will control the software. For example, a program written in C is listed in the following:

```
int f(int x) {
  int y = x + 10;
  if (y > 0)
    return y;
  else
    return x;
}
```

If we want to obtain f(x) = 100, what is the value of x? Since in the function f(x), two possible conditions must be explored:

(1) y >0 and x+10 > 0 which is called a path condition and we add a constraint of x+10 = 100. The solution is x = 90.
(2) y <=0 and x + 10 <= 0 and we add another constraint of x = 100. No solution is found for x <= -10 and x = 100.

The final solution is x =90 to obtain f(x) = 100. The above process is called symbolic execution since we treat x and y as symbolic variables and don't assume any concrete values as the

values of x and y. A special case of symbolic execution is to build the first set of path conditions according to an initial input. The variables are still treated as symbolic. If the evaluated results of any path condition is false, the negation is added to the path condition. Otherwise, the original path condition is added. For example, with the initial input of 100 for a concolic execution, f(x) is expressed as x+10 > 0 and f(x)=x+10. If we feed an initial input that will trigger a failure in the software by a concolic execution, we will obtain a set of path conditions that is a precise symbolic model of the failure. For example, if we have a Microsoft office RTF file that will crash the Microsoft Office Word software, we can feed the RTF file as the initial input by concolic execution of Word 2010, and obtain a precise symbolic model like:

Path conditions: $C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge ...$
EIP = $F(I_0,I_1,I_2,...)$
EBP = $F'(I_0,I_1,I_2,...)$
ESP = $F''(I_0,I_1,I_2,...)$

Where C_i is the path condition and I_i is the input to be manipulated.

The following of Fig.1 is a partial listing of the EIP constraint after concolic execution with a failure as the initial input.

```
(Concat w20 (Extract w8 0 (Add w20 ffffffa9 (Or w20 (And w20 (Add w20 (Or w20 (ZExt w20 (Extract w8 0 (Shl w20 (ZExt w20 (Extract w8 0 (Add w20 ffffffd0 (Or w20 (ZExt w20 (Read w8 9 eip)) 7c810000)))) 4))) 207200) (Or w20 (Or w20 (ZExt w20 (Read w8 a eip)) 7c810000) 20)) ff) 7c810000))) (Concat w18 (Extract w8 0 (Add w20 ffffffa9 (Or w20 (And w20 (Add w20 (Or w20 (ZExt w20 (Extract w8 0 (Shl w20 (ZExt w20 (Extract w8 0 (Add w20 ffffffd0 (Or w20 (ZExt w20 (Read w8 7 eip)) 7c810000)))) 4))) 207200) (Or w20 (Or w20 (ZExt w20 (Read w8 8 eip)) 7c810000) 20)) ff) 7c810000))) (Concat w10 (Extract w8 0 (Add w20 ffffffa9 (Or w20 (And w20 (Add w20 (Or w20 (ZExt w20 (Extract w8 0 (Shl w20 (ZExt w20 (Extract w8 0 (Add w20 ffffffd0 (Or w20 (ZExt w20 (Read w8 5 eip)) 7c810000)))) 4))) 207200) (Or w20 (Or w20 (ZExt w20 (Read w8 6 eip)) 7c810000) 20)) ff) 7c810000))) (Extract w8 0 (Add w20 ffffffd0 (Or w20 (And w20 (Add w20 (Or w20 (ZExt w20 (Extract w8 0 (Shl w20 (ZExt w20 (Extract w8 0 (Add w20 ffffffa9 (Or w20 (Or w20 (ZExt w20 (Read w8 3 eip)) 7c810000) 20))))
....
```

Fig. 1. The Microsoft Word EIP Constraint with a Failure Input

We are able to control the value of EIP, EBP, or ESP by solving the above constraints. The solution of $I_0,I_1,..,I_n$ are the exploit input of the failure. Failures of stack overflow and uninitialized uses can be modeled in the above similar way. Situations like format string and heap corruption is treated by introducing pseudo symbolic variables for assuming the variable referred by the pointer that is symbolic is probably symbolic. To resolve the pseudo symbolic variables, we first obtain a solution assuming pseudo symbolic variables are symbolic. By searching the memory contents that meet the solutions of the pseudo symbolic variable, we can resolve the values of the symbolic pointers that refer to the pseudo symbolic variables.

II. EXPLOITS WITH SHELLCODE AND ANTI-MITIGATIONS

To launch a practical manipulations of the failures, a set of malicious commands called shell code must be supplied. For example, to execute arbitrary code, the memory location of MEM[X] is injected with the malicious code $D_0,D_1,...$ and the EIP is resolved with the value of X by solving the constraints:

Path conditions: $C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge ...$
MEM[X] = $D_0 = F(I_0,I_1,I_2,...)$
MEM[X+1]= $D_1 = F'(I_0,I_1,I_2,...)$
...
EIP = X = $F''(I_0,I_1,I_2,...)$

The concolic execution is performed under the CRAX framework[10] and depicted in Fig. 2.

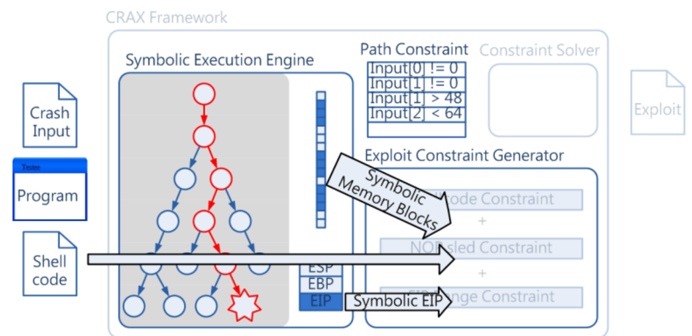


Fig. 2. Concolic Execution for Constructing Failure Constraints

Given a crash input, the target program, and the shell code, the exploit can be produced.

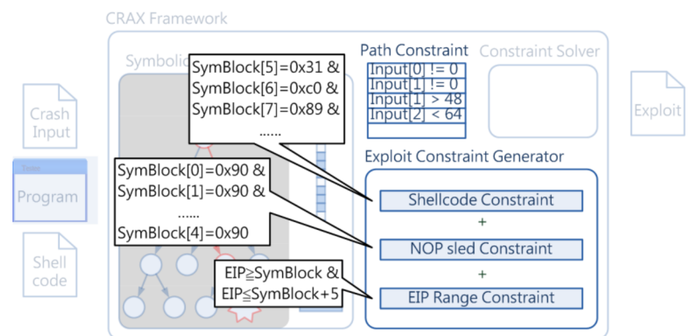


Fig. 3. The Generated Failure Constraints

Other types of attacks such as SQL injection is to find potential manipulations to the query strings to the SQL server. We can also build concolic constraints of web applications by feeding failure inputs. If the query to the SQL server is found to be symbolic, arbitrary SQL injection attacks may be constructed. If the output as the HTML response is symbolic, arbitrary Javascript code is very likely constructed as Cross site script (XSS) attacks. The generation is listed in Fig. 4. We extend these types of exploitation as follows.

a. SQL injection
Path conditions: $C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge ...$
SQL query = Q = $F(I_0,I_1,I_2,...)$

b. Cross site scripting

Path conditions: $C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge ...$

HTML Output response = R = $F(I_0,I_1,I_2,...)$

c. Command injection

Path conditions: $C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge ...$

string to the system() function = S = $F(I_0,I_1,I_2,...)$

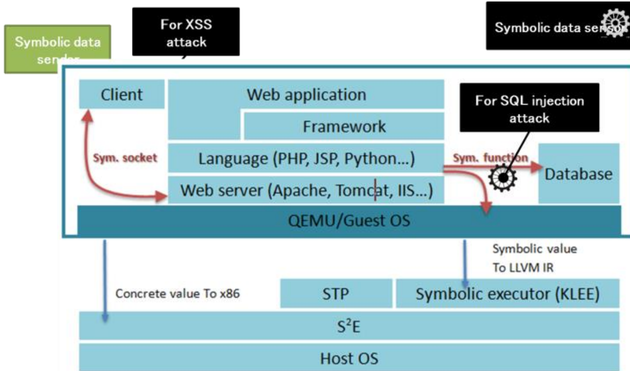


Fig. 4. Concolic execution of Web applications

Even if we can control the instruction pointer, several guards in front of the failures must be escaped. The first guard is the path constraint to reach the failure site without mitigations (surviving security attacks).

Many systems will be with protections such as data execution prevention (DEP) and address space layout randomization (ASLR)[13]. In such a system, executable code may not be injected and a return-oriented programming (ROP)[14] payload built through the application code must be constructed. The exploit constraint is changed into:

Path conditions: $C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge ...$

Search ROP Gadgets in $m_0,m_1,m_2,....$ locations of the application code

STACK[X] = $M_0 = F(I_0,I_1,I_2,...)$

STACK[X+1]= $M_1 = F(I_0,I_1,I_2,...)$

…

The R_1 is the location containing the instruction of “ret” of the code and the starting of the gadget.

EIP = $R_1 = F(I_0,I_1,I_2,...)$

III. THE CONSIDERATIONS OF THE ENVIRONMENT MODEL

Since the inputs to the target applications are through the operating system environment, for example, the file inputs, environment variables, or network socket, we must be able to feed inputs as symbolic through the OS environment which is called the environment model. There are two possible implementations. The first is to intercept the system calls or revise the standard library functions to mark the inputs as symbolic for concolic execution. This method is used by KLEE[15], AEG[7] and Mayhem[8]. Another solution is to use the whole system emulation like S2E[16] which is based on KLEE and QEMU. By using mmap() system call as in Fig. 5, or RAM disk, we can feed any environment input as symbolic variables to the target applications. The first implementation will be with limited supports of system functions intercepted. The second implementation will support all types of environments. To support an end-to-end approach of exploit generation, environment models of symbolic inputs must be supported. Otherwise, revisions of source is needed like the Heelan’s method [6].

```
1.  #include <sys/mman.h>
2.  int main(){
3.
4.      int fd;
5.      pid_t pid;
6.      int i;
7.      char *ptr;
8.      fd = open("./test",O_RDWR);
9.      struct stat b;
10.     fstat(fd , &b);
11.     ptr=mmap(0,b.st_size,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
12.     ptr[0] = 'a';
13.     s2e_make_symbolic(ptr,b.st_size, "buf" );
14.     if((pid = fork())==0){
15.         execv("./tested program", NULL);
16.     }else{
17.         wait();
18.         close(fd);
19.         s2e_kill_state(0, "program terminated");
20.     }
21.     return 0;
22. }
```

Fig. 5. The Symbolic File Environment by mmap()

IV. SUPPORTING BINARY PROGRAMS AND DEALING WITH LARGE INPUTS

To support binary programs of exploit generation, we must perform concolic execution over binary programs. Instrumentation over binary programs is needed. There are several concolc execution supports over binary program. Mayhem is based on PIN [17], Catchconv[18] is based on Valgrind[19] and S2E is based on QEMU. Another issue of binary programs for exploit generation is to use concrete address for symbolic memory. Conventional symbolic execution is to use abstract address and these addresses cannot be used for practical exploits. The concolic execution in S2E are treated differently in the host and the guest OS. In the guest OS, all addresses are concrete while abstract in the host OS. Since our exploits are for the guest OS, the concrete addresses meet the need for exploits of binary programs.

A. *Dealing with large inputs*

The primary steps are to crash the software and control the crash from carefully crafted inputs based on the crash input. There are two techniques to craft the inputs for easier crash manipulation: (1) search the influence over the crash by injecting special patterns of input [20]. (2) find the critical fragments of the input(called hot spot) that will influence the crash (or failure) by tainted input analysis [21]. Since the path conditions, EIP, shell code, and other constraints contain inputs as symbolic variables, the input size will influence the exploit generation process. For example, if the input size is 1024 bytes, there may be several large constraints with thousands of variables. The constraint resolution time is exponentially proportion to the size of input variables as listed in Fig. 6.

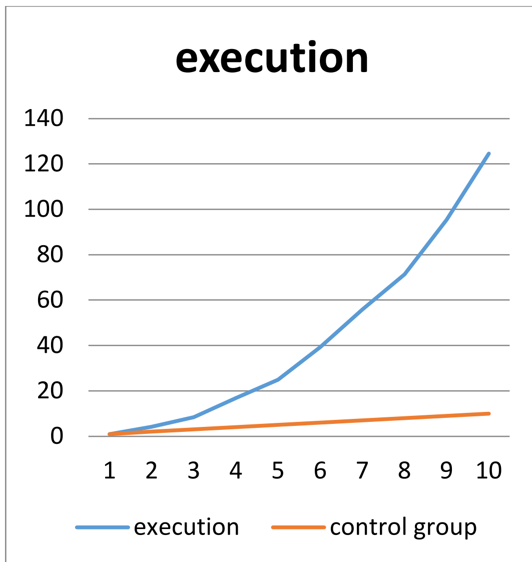


Fig. 6. The Execution time in seconds for symbolic input size from 100 to 1000 bytes

We have proposed an adaptive input selection method by dividing the input into several small size of symbolic inputs to track the influence. Table 1 shows the performance improvement of the adaptive input selection. Originally, if we use the input length of 5000, the explore time is 1388 seconds. If we divide the input into 20 bytes of small chunks, the total explore time is reduced to 11.7 seconds. The improvement is significant.

Table 1. The Performance Improvement of Adaptive Input Selection

Prog.	Input Length	Explore Time	Exploit Gen. Time	Explore Time (Adaptive)	Exploit Gen. Time (Adaptive)
Unrar	5000	1388.5	2569.8	11.7	1.8
Mplayer	145	145.8	151.2	3.3	0.3

V. Conclusion

Failure Exploitation is firstly to construct a set of failure conditions by initially feeding the failure input for concolic execution. We manipulate the failure path condition in the failure exploitation process to hijack the failure for exploitation generation. The failure hijacking is to compute a pre-destined value of instruction pointer (IP) in the relation of IP=F(failure-input). The software exploitation process will be a good measurement of trustworthiness for a failed system.

References

[1] C. Miller, J. Caballero, N. M. Johnson, M. G. Kang, S. McCamant, P. Poosankam, et al., "Crash analysis with BitBlaze," at BlackHat USA, 2010.

[2] M. S. E. C. M. S. S. Team. (2009 March). !exploitable Crash Analyzer - MSEC Debugger Extensions. Available: http://msecdbg.codeplex.com/

[3] D. Kim, X. Wang, S. Kim, A. Zeller, S.-C. Cheung, and S. Park, "Which crashes should I fix first?: Predicting top crashes at an early stage to prioritize debugging efforts," Software Engineering, IEEE Transactions on, vol. 37, pp. 430-447, 2011.

[4] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," Communications of the ACM, vol. 57, pp. 74-84, 2014.

[5] J. Vanegue, S. Heelan, and R. Rolles, "SMT Solvers in Software Security," in WOOT, 2012, pp. 85-96.

[6] S. Heelan, "Automatic generation of control flow hijacking exploits for software vulnerabilities," M.Sc. thesis, University of Oxford, 2009.

[7] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic Exploit Generation," in NDSS, 2011, pp. 59-66.

[8] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in IEEE Symposium on,Security and Privacy, 2012, pp. 380-394.

[9] S. K. Huang, M. H. Huang, P. Y. Huang, C. W. Lai, H. L. Lu, and W. M. Leong, "CRAX: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations," in IEEE Sixth International Conference on Software Security and Reliability (SERE), 2012, pp. 78-87.

[10] S. K. Huang, M. H. Huang, P. Y. Huang, H. L. Lu, and C. W. Lai, "Software Crash Analysis for Automatic Exploit Generation on Binary Programs," IEEE Transactions on Reliability, vol. 63, pp. 270-289, 2014.

[11] S. K. Huang, H. L. Lu, W. M. Leong, and H. Liu, "CRAXweb: Automatic Web Application Testing and Attack Generation," in IEEE 7th International Conference on Software Security and Reliability (SERE), 2013, pp. 208-217.

[12] K. Sen, "Concolic testing," in Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, 2007, pp. 571-572.

[13] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in Proceedings of the 11th ACM conference on Computer and communications security, 2004, pp. 298-307.

[14] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," ACM Transactions on Information and System Security (TISSEC), vol. 15, p. 2, 2012.

[15] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in OSDI, 2008, pp. 209-224.

[16] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," ACM SIGARCH Computer Architecture News, vol. 39, pp. 265-278, 2011.

[17] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "PIN: a binary instrumentation tool for computer architecture research and education," in Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture, 2004, p. 22.
[18] D. A. Molnar and D. Wagner, "Catchconv: Symbolic execution and run-time type inference for integer conversion errors," Tech. Rep. UC Berkeley EECS, 2007-23, 2007.
[19] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in ACM Sigplan Notices, 2007, pp. 89-100.
[20] H. Moore, "The metasploit project," http://www.metasploit.com.
[21] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in IEEE Symposium on Security and Privacy, 2010, pp. 497-512.

Shih-Kun Huang *received his B.S. (1989), M.S. (1991) and Ph.D. (1996) in Computer Science and Information Engineering from the National Chiao Tung University, and was an assistant research fellow at the Institute of Information Science, Academia Sinica between 1996 and 2004. Currently he is the deputy director of Information Technology Service Center, and jointly with the Department of Computer Science, National Chiao Tung University. Dr. Huang's research integrates software engineering, and programming languages to study cyber security and software attacks. He is the Principal Investigator of the project on Exploit Generation from Software Crash (CRAX and CRAXweb).*

Han-Lin Lu *received the B.S. degrees in Department of Transportation Technology and Management, and M.S. degrees in Science in Computer Science and Engineering from the National Chiao-Tung University, Taiwan in 2010, and 2012 respectively. He is currently pursuing the Ph.D. degree at the Institute of Science in Computer Science and Engineering of National Chiao-Tung University. His research interests include software quality, network security, and software security.*