2012 IEEE Sixth International Conference on Software Security and Reliability

CRAX: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations

Shih-Kun Huang*†,Min-Hsiang Huang†,Po-Yen Huang†,Chung-Wei Lai†,Han-Lin Lu†,Wai-Meng Leong†
*Information Technology Service Center,†Department of Computer Science
National Chiao Tung University
Hsinchu, Taiwan
{skhuang,mhhuang,huangpy,laicw,luhl,start03629.cs95}@cs.nctu.edu.tw

***Abstract*—We present a simple framework capable of automatically generating attacks that exploit control flow hijacking vulnerabilities. We analyze given software crashes and perform symbolic execution in concolic mode, using a whole system environment model. The framework uses an end-to-end approach to generate exploits for various applications, including 16 medium scale benchmark programs, and several large scale applications, such as Mplayer (a media player), Unrar (an archiver) and Foxit(a pdf reader), with stack/heap overflow, off-by-one overflow, use of uninitialized variable, format string vulnerabilities. Notably, these applications have been typically regarded as fuzzing preys, but still require a manual process with security knowledge to produce mitigation-hardened exploits. Using our system to produce exploits is a fully automated and straightforward process for crashed software without source. We produce the exploits within six minutes for medium scale of programs, and as long as 80 minutes for mplayer (about 500,000 LOC), after constraint reductions. Our results demonstrate that the link between software bugs and security vulnerabilities can be automatically bridged.**

***Index Terms*—automatic exploit generation; symbolic execution; taint analysis; software crash analysis; bug forensics.**

I. Introduction

Crafting exploits for control flow hijacking is typically regarded as a manual process requiring security knowledge [17]. However, based on recent advances of symbolic execution, several prototype approaches to automatically generating exploits have been proposed [2, 8, 13] . Exploits or other types of attacks, e.g., SQL injection and XSS [10], have been developed for the purposes of auditing web application security, IDS signature generation and attack preventions, and have been explored as research topics on dynamic taint analysis and symbolic execution. Another type of exploiting applications is the verification and validation purpose for hosting reliable and secure software in the App store, or software marketplace.

Our motivation of this work is straightforward. Crashes of software including web applications are inevitable. Given a large number of crashes, we need a systematic approach to judge whether they are exploitable crashes. In the crash report analysis [11], crashes are analyzed by BitBlaze[16] and compared with !exploitable [1] to prove that exploitable crashes can be diagnosed in a more precise way though still with limitations, e.g., possible false positives or requiring manual efforts. Moreover, crash analysis plays an important role to prioritize the bug fixing process [9]. A proven exploitable crash is surely the top priority bug to fix.

Dynamic taint analysis and forward symbolic execution have been the primary techniques in security fields[14]. Q [13] is regarded as a recent promising success to generate mitigation-hardened ($W \oplus X$ and ASLR[18]) exploits by feeding concrete execution trace and triggering a tainted instruction pointer. The vulnerable path is diverted by concolic execution and exploit constraints to manipulate the instruction pointer, and the constraints are combined with return-oriented programming (ROP)[15] payload and solved by a decision procedure, STP [7].

Our objective is similar to Q, but we target at a more generalized threat model: all currently tainted threat models can be viewed as a specific continuation threat. For example, control flow hijacking attacks divert the input into an attacker manipulated continuation. The continuation results in execution of arbitrary code. The SQL and command injection is a kind of tainted input flowing into the SQL server or introducing a ”shell” command execution, and thus can be viewed as a continuation into SQL control or shell control. The cross-site scripting is a reflection of web pages, inserting an explicit continuation to execute arbitrary Java script, impersonating as originating from the original Web server. If the continuation is symbolic, that is, a concolic execution to reach the invoked site of the continuation and a symbolic expression to describe the continuation, we can generate practical attacks to exploit the continuation. Software crash can be viewed as a tainted continuation. Furthermore, if the tainted continuation is found to be symbolic, an exploit can be automatically generated. We have successfully produced exploits from software crashes for control flow hijacking attacks from large applications, including Mplayer, Unrar, Foxit pdf reader and AEG [2] benchmarks. All processes are end-to-end, built on top of environment model of S^2E[6], symbolic virtual machine of KLEE[5], and processor emulator of QEMU[3].

Our framework, called CRAX, is to act as a backend of static/dynamic program analyzers, bug finders, fuzzers, and crash report database. Given these ”crashes” from the frontends and the program binary, CRAX can automatically generate attacks, practical mitigation-hardened exploits.

978-0-7695-4742-8/12 $26.00 © 2012 IEEE
DOI 10.1109/SERE.2012.20

A. Contributions

The primary contributions and impacts of our work are as follows:

- Dedicate to automatic exploit generation for large software systems without source code. Concolic execution ideas have been proposed for exploit generation since 2009. Due to the rapid development of symbolic computation, processor emulation and environment model supports, automatic exploit generation has become an integration work from existing systems. However, to the best of our knowledge, we have not found a practical integration of exploit generation work that can produce exploits from large applications, such as Mplayer, and Foxit pdf reader. We are the first to demonstrate such capability though completing the hacking process formerly regarded as a manual process. A similar scale of work is the Catchconv[12], which performs metafuzzing, taking the Mplayer as a prey. However, it only acts as a fuzzer and succeeds to produce Mplayer crashes in about the comparable numbers of zzuf[19]. In contrast, CRAX takes the crash from Mplayer (constituting of more than 500,000 lines of code) and produce exploits. Other successes of CRAX include Foxit pdf reader, and mplayer. We have automated the process of exploit writing work.
- Prioritize crashes to be fixed. Currently, many sources of crash report from various of bug analyzers and random fuzzers are available. Too many crashes need to be fixed, and there is a pressing need to determine their priorities [9]. We have preliminarily examined the crash database of Bugzilla for the Mozilla project and found that many of the crashes can be our seed input to produce exploits. Our tool would be the first screen gateway to prioritize the bug fixing order.

B. Paper Organization

The paper is organized as follows. Sections II and III describe our method and implementation, respectively. Experimental results are reported in Section IV. Section V presents related work. Finally, Section VI concludes our paper.

II. Method

We propose a new automated exploit generation method based on S^2E with path selection optimization to speed up the process of exploit generation. Concolic-mode simulation explores a potential vulnerable path directly, and code selection filters some complex and unrelated library functions that do not affect exploit generation in order to reduce the overhead of SMT solvers.

A. The Weakness of AEG

Both our method and AEG (Automatic Exploit Generation) [2] detect vulnerabilities by symbolic execution and then collects run-time information from concrete execution with the test case generated by the previous step. However, AEG collects run-time information and computes exploits only when vulnerability is triggered. The generated exploit may fail to work due to the propagation distance between the vulnerable site and the triggered exploit site. AEG cannot guarantee that the program under test arrives at the triggered exploit site successfully if the exploits are not revised accordingly. Considering Listing 1, the buffer overflow vulnerability occurs at line 4 where strcpy() function is located, but the exploit is triggered at line 6 where the function returns. However, the exploiting string is reversed at line 5 and therefore, fails to work when the function returns. The process is shown in Figure 1.

Listing 1. An example code for AEG

```
void test(char src[30])
{
    char des[10];
    strcpy(des, src); /* buffer overflow vulnerability */
    reverse(des);
}                     /* control flow is hijacked */
```

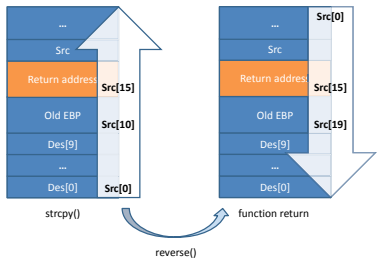

Fig. 1. The memory layout before and after reverse() is executed

B. Our Method

1) Detection of Symbolic Program Counter- the EIP Register in x86 machines: Since the EIP register contains the address of next instruction to be executed, to control the register is a very common final target of all control-hijacking attacks. Thus monitoring the state of EIP register is a comprehensive and easy way to tackle different kinds of control-flow hijacking vulnerabilities. While symbolic execution explores paths and taints memory, an exploit will be triggered when the EIP register is updated with the symbolic data. The exploit generation will search memory to find usable memory regions to inject shellcode and NOP sled, and redirect EIP register to shellcode. The process of detection of the symbolic EIP register and exploit generation is shown in Figure 2.

2) Shellcode Injection: To inject a shellcode, the first thing is to find all memory blocks that are symbolic and large enough to hold payload. Even if a symbolic block consists of many different variables, it could still be used to inject a shellcode as long as the block is contiguous. However, it is very difficult to manually analyze source code to find a contiguous memory region that is tainted by user input and combined with variables. In addition, since compiler often changes the order or allocated size of variables for optimization, it is difficult to find a shellcode buffer manually. We

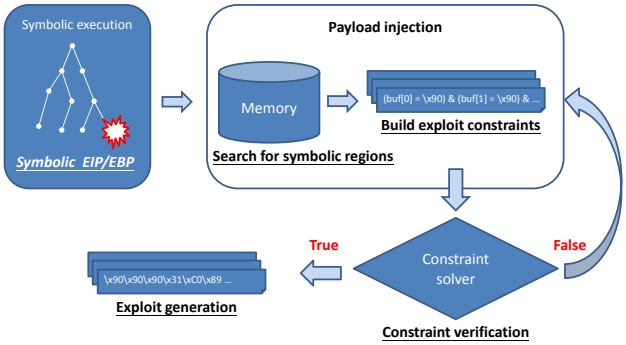

Fig. 2. The process of our exploit generation

automate this process by searching the maximum contiguous symbolic memory systematically.

3) NOP Sled and Exploit Generation: When the location of a shellcode is determined, *NOP sled* will try to insert a sequence of *NOP* instructions, which do nothings, as many as possible in front of the shellcode closely. This padding helps exploits against the inaccurate position of shellcode among different systems, or to extend the entry point of shellocde. Finally, the EIP register corrupted by symbolic data will point to the middle of NOP padding. All exploit constraints, including shellcode, NOP sled, and EIP register constraints, are passed to an SMT solver with path conditions to determine whether the exploit is feasible or not. If it is not feasible, the exploit generation goes back to the step of shellocde injection to change the location of shellcode until an exploit being generated or no more usable symbolic buffer.

C. Detection of Symbolic Pointer

In addition to the EIP register, corrupted pointers may change the control flow indirectly. Particularly, a symbolic data is assigned to a symbolic pointer means that arbitrary data can be written to arbitrary addresses. When a symbolic pointer dereference is detected, the target of writing operation will be redirected to sensitive data, such as return addresses, .dtors section, and GOT, to update EIP register indirectly. Otherwise, if the pointer operation is a reading operation or a writing operation but cannot point to sensitive data, the target is redirected to read from a symbolic data or write to a concrete data to perform tainted data propagation. Considering Listing 2, an off-by-one overflow vulnerability will corrupt the *ptr* pointer and as a result, the value of *buf[0]* may write to arbitrary addresses. Even if this vulnerability cannot corrupt return addresses directly, the symbolic pointer can taint EIP register indirectly and hijack the control of a program.

Listing 2. An example code for pointer corruption

```
void test(int *input)
{
  int *ptr = array;
  int array[10];
  int i;

  for(i=0 ; i<=10 ; i++)
    array[i] = *(input + i);

  *ptr = array[0];
}
```

D. Path Selection

1) Concolic-mode Simulation: If an input data crashes a program, the execution path the crash input exploring is very likely exploitable. Exploring the suspicious path directly is more effective than searching all paths. Concolic testing is a kind of symbolic execution, and it explores one path at a time. Concolic testing stores and updates concrete values and symbolic expressions simultaneously. It uses the concrete values to help symbolic execution determine which branch path will be explored, and uses the symbolic expressions to collect the branch conditions whenever a path is determined to travel at branches.

In contrast to implementing concolic testing on S^2E, simulating the behavior of concolic testing on S^2E is an easier and flexible task. Whenever a branch is encountered, S^2E does not access the concrete value of variables but adds *input constraints* to limit the values of all symbolic variables to the values of original input, which are constants. Figure 3 shows an example that executes the program under test with an argument string "ABCDEF" and the input constraints are built to restrict those values of the argument. Because each symbolic variable could be only one possible value after adding the input constraints, it simulates the concrete values to choose one path to explore. This method does not modify the memory model of S^2E and based on symbolic execution to provide the ability of concolic testing.

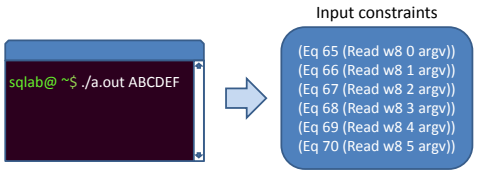

Fig. 3. An example of input constraints

With fuzzer tools to identify an input crashing the program under test, concolic-mode simulation determines whether the path is exploitable rapidly because it focuses on only one path. Combining fuzzer tools with concolic-mode simulation provides a powerful technique for exploit generation as illustrated in Figure 4.

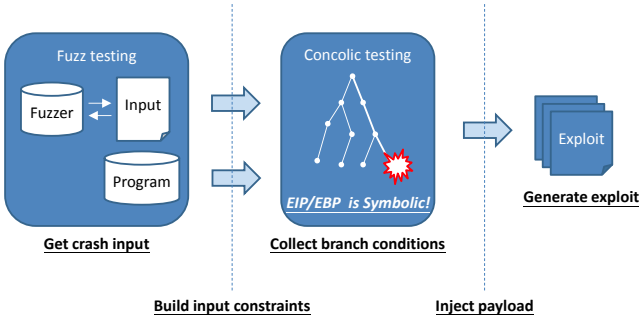

Fig. 4. The process of exploit generation from concolic-mode simulation with fuzz testing

2) Code Selection: Since S^{2}E performs symbolic execution on the entire operating system, path explosion becomes an issue when the symbolic data are passed to library or kernel. On the other hand, the constraints induced by the library or kernel are usually complex and huge, and constraint solvers often get stuck in solving them. For example, if the first argument of fopen() function, which is a path of the file to be opened, is symbolic, the constraint solvers will get a time-out error or hang in S^{2}E. But, those paths in the library or kernel are often irrelevant to exploit generation. In order to avoid exploring those irrelevant paths, those library functions should run on concrete execution.

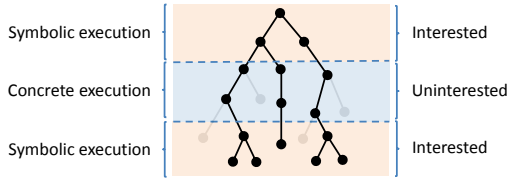

Fig. 5. An execution tree with code selection

When the symbolic arguments are changed to concrete values and then passed to those irrelevant functions, only one path will be explored. Figure 5 shows that the executions are changed into concrete runs after the entry of the irrelevant part and persist in concrete states until the program returns to symbolic execution. In order to ensure the return values of those irrelevant functions are correct, the concrete values passed to functions must be generated according to current path conditions by constraint solvers. When the functions return, those data changed to concrete must be restored to the original symbolic data to keep the symbolic execution.

III. Implementation

In order to implement automated exploit generation, many pieces of run-time information must be collected and many constraints must be built to reason out an exploit. Therefore, the memory model in S^{2}E is an important key to implementing our methods.

Many protections are implemented on compilers or operating systems in real-world systems. In addition to return-to-memory exploits, we also implement other two types of exploits, i.e., return-to-libc and jump-to-register exploits, to bypass some protections so that our exploit generation can be useful in real-world systems.

A. Detection of Continuation based Symbolic Registers

In QEMU, the *CPUX86State* structure, is used to simulate the states of x86 CPU, and all register references in a guest operating system will be turned into memory references on this structure. When S^{2}E is started, this structure is divided into two parts, i.e., *CpuRegistersState* and *CpuSystemState*, and they are stored separately. The CpuRegistersState is a symbolic area where stores all the data in front of EIP register in CPUX86State structure, such as general-purpose registers, but the CpuSystemState part is a concrete-only area that stores the other data including the EIP register.

S^{2}E translates every guest instruction into TCG IRs, and then translates those TCG IRs into host instructions or LLVM IRs. For example, the *ret* instruction is separated into more detailed operations, and the operation of updating EIP register is converted to a *store* instruction. In QEMU, all memory access operations are handled by a *softmmu* model in order to map the guest addresses to host addresses. Whenever accessing a memory data, S^{2}E checks whether the value of data is symbolic or not in the softmmu model. If the value is symbolic, S^{2}E will rerun this translated block from the current instruction in KLEE to perform symbolic execution. To detect EIP register corruption, S^{2}E must check whether the writing target is the location of EIP register and whether the source value is symbolic data whenever KLEE perfrom a *store* memory operation on symbolic execution.

When the EIP register is updated by symbolic data, the expression of symbolic data must be recorded because it describes which variable and which part of the symbolic data will update EIP register. For example, given an expression that represents a 32-bit symbolic data at the first element of an array named *buf* denoted as

$$(ReadLSB\ w32\ 0\ buf),$$

we can build a constraint to control the value of symbolic data, e.g., a constraint limiting the 32-bit data to zero as shown by

$$(Eq\ 0\ (ReadLSB\ w32\ 0\ buf)).$$

The current continuation has been set by the data, with constraints described by symbolic expressions. Next, we inject shellcode into memory to determine where EIP register should point to.

B. Exploit Generation

1) Memory Model in S^{2}E: In S^{2}E, memory consists of *MemoryObject* objects and the actual contents of these objects are stored in *ObjectState* objects. In an object of *ObejectState*, symbolic data are stored separately from concrete data. The expressions of symbolic data are stored in an array that consists of *Expr* objects and pointers named *knownSymbolics* pointing to them. The concrete data are stored in an array of *uint8_t* and pointed by a pointer named *concreteStore*. In each *ObejectState* object, a *BitArray* object named *concreteMask* is used to record the state of each byte, i.e. the byte is concrete or symbolic.

2) Finding Symbolic Memory Blocks: The default size of the storage in an *ObjecetState* object is 128 bytes. To find continuous symbolic data in a memory region, the value of *concreteMask* structures must be checked sequentially object by object. An object can be skipped easily whenever the values of its *concreteMask* structure are all ones, otherwise the locations of every zero in *concreteMask* structure must be recorded to compute the continuous size. For the symbolic blocks crossing different objects, it is necessary to check whether the current symbolic block is connected with the last symbolic block in the last checked object. The above procedure is shown in Algorithm 1.

Algorithm 1: Searching for symbolic blocks

Input: Objects : All *ObjectState* objects to be searched.
Output: V : A set of address and size.

```
1  foreach obj ∈ Objects do
2      if isNotAllConcrete() then
3          size ← 0
4          for i ← 0 to 127 do
5              if isByteSymbolic(i) then
6                  size ← size + 1
7              else if size ≠ 0 then
8                  address ← getAddress(i)
9                  if V → isConnect(address,size) then
10                     V → updateLastItem(size); /* A
                           part of the last block
                           */
11                 else
12                     V → addNewItem(address,size)
                           /* An independent block
                           */
13                 size ← 0
```

It is also required to determine the search range of memory regions. In Linux memory layout, the stack starts from the top at address 0xbfffffff and grows downward. It is easy to search stack region from this address downward,but heap and data segment are not necessarily located at a fixed address for different programs. Therefore, those starting locations need to be obtained dynamically. According to the ELF executable layout, the top of executable files is the program header, which records the all segment information. The program header can be analyzed at address 0x08048000, which is the location where binary is loaded at, to get the location and size of data segment. On the other hand, because heap region is behind data segment and grow upward, the based address of heap can be obtained by adding the starting address and size of data segment.

3) Shellcode Injection: In order to determine whether shellcode can be stored in the potential buffers found by the previous step, each symbolic expression of a symbolic block needs to be read to build constraints that restrict each byte of symbolic data to a byte of shellcode sequentially byte by byte. Below is an example showing the constraints that inject shellocde into an array named *buf*:

$$
\begin{aligned}
&(Eq\ 31\ (Read\ w8\ 0\ buf))\\
&(Eq\ C0\ (Read\ w8\ 1\ buf))\\
&(Eq\ 89\ (Read\ w8\ 2\ buf))\\
&(Eq\ C2\ (Read\ w8\ 3\ buf))\\
&\vdots
\end{aligned}
$$

Next, the shellcode constraints are passed to an SMT solver with path conditions to validate their feasibility.

The best location of shellcode is selected by having the NOP sled as large as possible. Therefore, all the symbolic blocks are sorted by size, and shellcode is first injected from the end of the largest symbolic block. In addition to building the shellcode constraints, a new constraint needs to be added to ensure the EIP register can point to the range between the starting address of shellcode and the top of the symbolic block. Even if the EIP register cannot point to the starting location of shellcode precisely, it may be feasible because NOP sled will extend the entry point later. If all of those constraints are infeasible, the location of shellcode injection is shifted by one byte forward to try a new location iteratively.

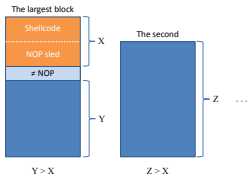

Fig. 6. The process of searching symbolic blocks

In addition, shellcode will keep to be injected to the current block or next blocks when those sizes are larger than the sum of the shellcode size and the current longest NOP sled size. For example, consider Figure 6, the sum of shellcode size

and current NOP size is X, but it is smaller than Y and Z, so shellcode and NOP sled will keep to be injected in next blocks and the current block. The algorithm is shown in Algorithm 2.

Algorithm 2: Injecting shellcode

Input: V : A set of address and size of symbolic blocks. Shellcode : A shellocde string. PC : Path conditions.

Output: ShellcodeAddress : The starting location of shellcode injection. MaxNopSize : The max size of NOP sled.

```
1  foreach v ∈ V do
2      if size_v ≥ strlen(Shellcode) then
3          address ← address_v + size_v - strlen(Shellcode)
4          MaxNopSize ← -1
5          while address ≥ address_v do
6              c1 ← injectShellcodeAt(address) /* Build
                   shellcode constraints */
7              c2 ← eipBetween(address, address_v)
                   /* Build eip constraints */
8              if Verify(PC ∧ c1 ∧ c2) then
9                  nopSize ← NOPSled(address, address_v)
10                 if nopSize > MaxNopsize then
11                     MaxNopSize ← NopSize
12                     ShellcodeAddress ← address
13                 if (address - address_v) > strlen(shellcode)
                   + MaxNopSize) then
14                     address ← address - nopSize
15                 else
16                     break
17             else
18                 address ← address - 1
```

4) NOP Sled: NOP sled aims to generate the more reliable exploits that increase chances of success. The method is to insert NOP instructions in front of the shllecode as many as possible, and make EIP register point to the range. For efficiency, binary search-like algorithm is used to determine the longest length of NOP sled rather than insert NOP instructions byte by byte. Whenever binary search finds a range that EIP register can point to, NOP instructions will be tried to fill this range sequentially to check whether both conditions are feasible simultaneously. If it is infeasible, the range is reduced, otherwise extend, and so on. The process is shown in Figure 7.

After the longest length of NOP sled is obtained, the next step is to make EIP register point to the middle of NOP sled as close as possible. Because the number of NOP sled may be large, the constraint solver is used to reason out the suitable location that EIP register points to. To help a constraint solver to compute the address as close the middle of NOP sled as possible, a constraint is added to limit the range. First, the range is a point in the middle of NOP sled, and the constraints are passed to a constraint solver to get the solution. If it is infeasible, the range is extended twice each time, and so on. This process can always get a solution, because the previous step guarantees that the EIP register can point to the range of NOP sled. The process as shown in Figure 8.

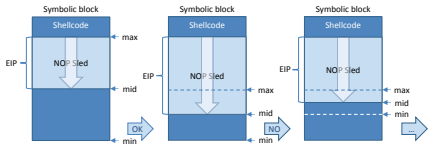

Fig. 7. The process of NOP sled

Finally, when the staring address of shellcode, the size of NOP sled and the location where EIP register points to all are determined and feasible, the constraint solver will solve the final path conditions to generate the exploit that performs the malicious task in the shellocde.

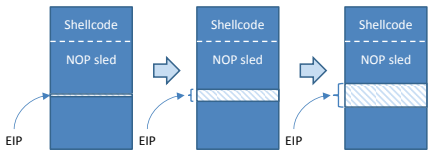

Fig. 8. The process of determining where EIP register point to

C. Other Types of Exploit

1) Return-to-libc: A return-to-libc attack is a technique to bypass non-executable memory regions, such as W⊕X protection. It redirects control flow to functions in C runtime library, such as system(), and injects the arguments of the function into stack manually to fake the behavior of function callers. Because runtime library is always executable and loaded by operating systems, a return-to-libc attack can perform malicious tasks by executing library code and bypass executable space protection. Considering Figure 9, function callers have to push arguments and return address into stack when calling functions. It does not really matter where the libc function call returns to, but the arguments are the key to perform the tasks we are interested.

TABLE I
THE DIFFERENCES BETWEEN RETURN-TO-MEMORY AND RETURN-TO-LIBC EXPLOIT

Exploit	Shellocde Injection	NOP Sled
Return-to-memory	shellocde	NOP instruction(0x90)
Return-to-libc	"/bin/sh"	whitespace(0x20)

Taking system("/bin/sh") for example, which will open a shell, the only one argument is a pointer that points to the string "/bin/sh" as shown in Figure 9. The process of return-to-libc exploit generation is very similar to return-to-memory.

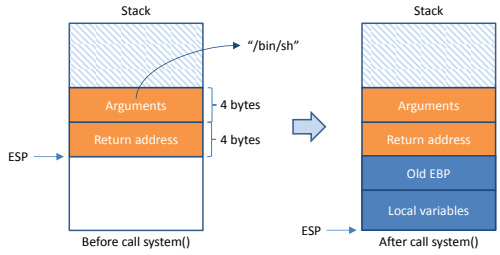

Fig. 9. The process of return-to-libc exploit generation

As shown in Table I, the steps of shellcode injection and NOP sled are still required to return-to-libc exploit generation. But, shellcode injection injects the string "/bin/sh" instead of a shellcode, and NOP sled fills whitespace characters rather than NOP instructions.

2) Jump-to-register: Stack is the most common memory region for shellocode injection, but ASLR randomizes the base address of stack so that control flow is very difficult to jump to shellcode accurately. A large NOP sled may bypass ALSR, but it is not always feasible. A jump-to-register attack is a technique to bypass ASLR. It uses a register that points to a shellcode as a trampoline to execute the malicious tasks. For example, EAX register is usually used to store the return value of functions. Strcpy() function returns a pointer points to the location of buffer, and EAX register is often used to store the address. If a "call %eax" instruction can be found in code segment, which is very common, and shellcode can be injected into the buffer EAX register points to, control flow will be redirected to execute this instruction and jump to shellcode.

In addition, a jump-to-esp attack is also a common and reliable technique without NOP sled and guessing stack offset in Windows and old version of Linux. Because return addresses are always popped to make ESP register points to their next address when functions return, shellcode can be injected behind the return address and uses ESP register as a trampoline. If a "jmp %esp" instruction can be found in code segment, a jump-to-esp exploit can be generated to bypass ASLR. The process as shown in Figure 10.

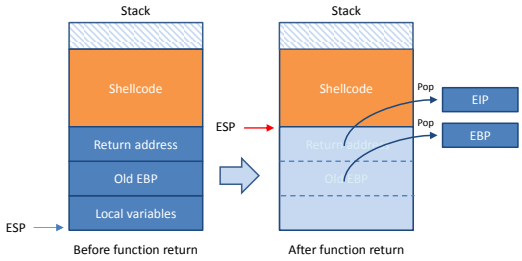

Fig. 10. The process of jump-to-register exploit generation

In order to generate jump-to-register exploits, code segment must be searched to find the related instructions such as "call %eax" and "jmp %esp". If the related instructions are found and the memory region that register points to is symbolic, shellcode will be injected into the location, and EIP register will be redirected to execute the related instruction. In addition, if there is not any usable instructions in code segment, data segment may be searched to find a two-byte symbolic data to inject the related instruction because data segment is unaffected by ASLR. For example, "jmp %esp" instruction is 0xffe4 and "call %eax" instruction is 0xffd0.

IV. Experimental Results

We conducted five types of experiments to evaluate our work for automatic exploit generation. The first experiment is with five different common control-flow hijacking vulnerabilities to demonstrate that our method can handle all vulnerabilities that symbolically update the EIP register and some vulnerabilities that symbolically update pointers. The second experiment is with return-to-libc and jump-to-register exploit generations to demonstrate that our method could bypass some mitigation protections in real-world systems. In the third experiment, we generated exploits for 16 real-word programs, most chosen from the benchmark of AEG, to demonstrate that our method can handle at least all cases that AEG addresses. Our method is slightly slower than AEG since we are conducting the whole system symbolic execution and AEG is on the application level. The performance has been optimized by reducing the number of constraints during the concolic execution. The fourth experiment reveals the performance speedup between the original concolic method (also the AEG uses) and the improved method. After the optimization, the performance of the whole system execution approximates to the speed of AEG. Due to this performance tuning, the speedup can achieve 50 times. Finally, we demonstrate the power of our CRAX to produce exploits of large applications, including foxit pdf reader, and Mplayer.

A. Testing Method and Environment

TABLE II
THE RESULTS OF EXPLOIT GENERATION FOR SAMPLE CODE

Vulnerability	Corrupted Data	Concolic Wall Time(sec.)	Symbolic Wall Time(sec.)
Stack buffer overflow	Return address	0.61/3.59	0.69/303.59
Heap buffer overflow	Pointer	0.24/3.16	0.25/301.73
Off-by-one overflow	EBP register	0.46/3.24	0.51/302.14
Uninitialized variable	Function pointer	0.41/3.59	0.46/303.23
Format string	Pointer	0.05/4.81	–
Average		0.35/3.67	0.47/302.67

All of the experiments were performed on a 2.66 GHz Intel Core 2 Quad CPU with 4 GB of RAM, and the host environment is Ubuntu 10.10 64-bit. The guest environment was Debian 5.0 32-bit with default settings of QEMU, which is a 266MHz Pentium II (Klamath) CPU with 128 MB of RAM.

Most of the programs under test were compiled by GCC 4.3.2 and ran on Glibc 2.7, which are the default in Debian 5.0. The other programs used GCC 3.4.6 or Glibc 2.3.2 to

generate exploits since the default version of GCC protects main functions against stack buffer overflow or performs heap hardening integrity checks to stop heap overflow.

We used an end-to-end approach to generate exploits on binary executables without modifying the source code, i.e. the source code are not required. Our approach was to fork a new process to execute the program under test and passing the symbolic data to it from outside. In Debian 5.0, ASLR is enabled by default so that the based address of stack and heap is randomized. Therefore, ASLR was disabled in our experiments for generating and testing all exploits except jump-to-register exploits.

B. Sample Code

Since our method is based on detection of symbolic EIP register instead of specific vulnerabilities, it can handle different types of vulnerabilities. In the first experiment, We designed five sample code for five different vulnerabilities and four corrupted data, and performed automated exploit generation on them. The results are shown in Table II, where the wall time is expressed by *(exploit reason time / total time)*.

In this experiment, the source input of all sample code was argument and its length was 100 characters. We compared the efficiencies of concolic-mode simulation and traditional symbolic execution. In symbolic execution, depth-first search (DFS) was used to explore a symbolic execution tree. The heap overflow code was executed on Glibc 2.3.2, because some protections that check pointer consistency have included in Glibc since version 2.3.6. In addition, this exploit generation cooperated with *libfmtb*[1] library to build format strings to exploit format string vulnerabilities.

In the five vulnerabilities considered in the first experiment, stack buffer overflow and uninitialized variable vulnerability corrupt EIP register directly, and the other three vulnerabilities taint EBP register or pointers to corrupt EIP register indirectly. As the results show, the average of total time was 3.67 seconds in concolic mode and the exploit reason time was 0.35 seconds. On average, symbolic execution spent 302.67 seconds on generating an exploit and 0.47 seconds on reasoning out it. Concolic mode was faster about 100 times than symbolic execution because it just explored only one suspicious path. In the experiments of symbolic execution, format string vulnerability got an out-of-memory error because symbolic execution attempted to explore all paths in snprintf() function, which performs a complex behavior.

C. Return-to-libc and Jump-to-register Exploits

In the second experiment, we implemented return-to-libc and jump-to-register exploit generation, and the generated exploits could bypass non-executable stacks or ASLR protection.

Since return-to-libc and jump-to-register exploit generation do not apply to all cases, i.e. only suitable for some special cases, we chose the sample code of stack buffer overflow vulnerability to conduct the experiment. Furthermore, the experiment was conducted on concolic mode.

[1]http://packetstormsecurity.org/files/26173/

TABLE III
THE RUN-TIME INFORMATION OF RERUN-TO-LIBC EXPLOIT GENERATION

Run-time Information	
EIP register	(ReadLSB w32 54 arg)
ESP register	0xbffff8e0 (value:(ReadLSB w32 58 arg))
ESP register + 4	0xbffff8e4 (value:(ReadLSB w32 62 arg))
Address of system()	0xb7ebb7a0
Potential shellcode buffers	0xbffffa8f (size:100 bytes) 0xbffff8a6 (size:100 bytes)

TABLE IV
THE RUN-TIME INFORMATION OF JUMP-TO-REGISTER EXPLOIT GENERATION

Run-time Information	
EIP register	(ReadLSB w32 54 arg)
Usable trampoline registers	EAX
EAX register	0xbffff8a6 (value:(ReadLSB w32 0 arg))
Address of "call %eax" instruction	0x0804839f
Potential shellcode buffers	0xbffffa8f (size:100) 0xbffff8a6 (size:100)

In the experiment on return-to-libc exploits, we uses system() function to execute "/bin/sh" command. The run-time information at exploit generation is shown in Table III. The locations where the ESP register pointed at and ESP register + 4, which is the address to insert argument, were all symbolic and the potential shellcode buffers were large enough to insert the string "/bin/sh". Therefore, this vulnerable program satisfied all the conditions for return-to-libc exploit generation. The time spent on exploit reason was 0.34 seconds, while the total time of this experiment was 3.25 seconds.

Table IV shows the run-time information at jump-to-register exploit generation. A "call %eax" instruction was found at address 0x0804839f, and the EAX register pointed to the starting location of a symbolic region exactly. Therefore, this vulnerable program can generate jump-to-register exploit to bypass ASLR. Only 0.06 seconds were spent on reasoning out the exploit, while the total execution time was 3.16 seconds.

D. Real-world Programs

In the final part of experiments, we generated exploits for real-world programs. Because real-world programs are more large and complex than the sample code, this experiment demonstrated that our method is effective and practical in real-world applications.

We chose several programs from benchmarks of AEG and found three new vulnerable programs released in recent years to perform this experiment. The 16 real-world programs were evaluated by an end-to-end approach, i.e. all programs were tested in binary forms, and the vulnerabilities of these programs were all stack buffer overflow. Concolic-mode simulation was used to perform exploit generation on all programs, and code selection intercepted functions associating with file-related operations or pure error feedback, such as fopen() and perror(), to speed up the process. Table V shows the results of 16 real-word programs.

TABLE V
THE RESULTS OF EXPLOIT GENERATION FOR REAL-WORLD PROGRAMS

Program	Version	Input Source	Input Length	Wall Time(sec.)	Advisory ID.
aeon	0.2a	Env. Var.	550	12.36/32.03	CVE-2005-1019
iwconfig	V.26	Arguments	85	0.89/3.57	BID-8901
glftpd	1.24	Arguments	300	2.68/8.06	OSVDB-16373
ncompress	4.2.4	Arguments	1050	45.57/99.36	CVE-2001-1413
htget	0.93	Arguments	276	8.21/35.48	CVE-2004-0852
htget	0.93	Env. Var.	180	1.16/5.08	AEG's 0-day
expect	5.43	Env. Var.(HOME)	300	5.84/29.35	OSVDB-60979
expect	5.43	Env. Var.(DOTDIR)	300	6.10/29.28	AEG's 0-day
rsync	2.5.7	Env. Var.	201	2.17/9.92	CVE-2004-2093
acon	1.0.5	Env. Var.	1300	93.84/162.70	CVE-2008-1994
gif2png	2.5.3	Arguments	1080	65.24/154.67	CVE-2009-5018
hsolink	1.0.118	Arguments	1050	56.44/103.91	CVE-2010-2930
exim	4.41	Arguments	304	3.21/122.26	EDB-ID#796
aspell	0.50.5	Stdin	300	3.61/14.52	CVE-2004-0548
xserver	0.1a	Socket	104	1.16/14.35	CVE-2007-3957
xmail	1.21	Stdin	307	20.23/371.65	CVE-2005-2943

As the results show, *iwconfig* spent 3.57 seconds on exploit generation, and it was the shortest one. On the other hand, the slowest was *xmail* which spent about six minutes. According to the results, the speed was proportional to the length of program input, because the more symbolic data exist, the more code may perform on symbolic execution. In addition, the longer symbolic data will bring huge and complex constraints, and SMT solvers must spend a lot of time on constraint solving.

In order to reduce the overhead of SMT solvers and speed up the process, code selection was used to concretize arguments of irrelevant functions. In this experiment, *aeon*, *htget* and *acon* intercepted fopen(); *ncompress* intercepted __lxstat() and perror(); *gif2png* intercepted fopen() and perror(); *expect* intercepted open(); *rsync* intercepted vsnprintf(); *hsolink* intercepted system(). Those functions related with file operations were often make constraint solvers stick, and perror() function just printed error messages without return value and doesn't influence exploit generation, so we filtered these functions to speed up the process.

In this experiment, we performed exploit generation on real-world programs and produced exploits for those applications successfully. The results show that the worst total time was about six minutes to generate an exploit for real-world programs, and the quality of exploits was good because they contained the longest NOP sled to increase the chances of successful attacks.

E. Constraint Optimization and Large Applications

In the ordinary concolic execution (as used by AEG), the path constraints and the input constraint, along with the exploit constraints are combined together to be solved. We have tried to separate the constraint resolving processes, with two exclusive conditions: (1) path constraints and branch constraint, (2) input constraint and the branch constraint. After this separation, the performance speedup can achieve as high as 50 times, as shown in the Table VI and Figure 11.

After the success of the constraint reducing, we apply the exploit generation process to large programs and web applications. We exploit mplayer in 80 minutes and 255 minutes for foxit pdf reader, as shown in Table VII.

TABLE VI
THE COMPARISONS OF SPEEDUP AFTER CONSTRAINT REDUCING

Program	Version	Input Length	Old Exploring Time(sec.)	Optimization Exploring Time(sec.)	Speed Up
aeon	0.2a	550	298.12	19.67	15.10x
iwconfig	V.26	85	4.21	2.68	1.57x
glftpd	1.24	300	50.07	4.71	10.63x
ncompress	4.2.4	1050	2000.41	53.79	37.18x
htget	0.93	276	146.72	27.19	5.39x
expect	5.43	300	172.50	23.51	7.33x
rsync	2.5.7	201	210.53	7.75	27.16x
acon	1.0.5	1300	3782.50	68.86	54.93x
gif2png	2.5.3	1080	2254.87	89.43	25.21x
hsolink	1.0.118	1050	2422.07	47.47	51.02x

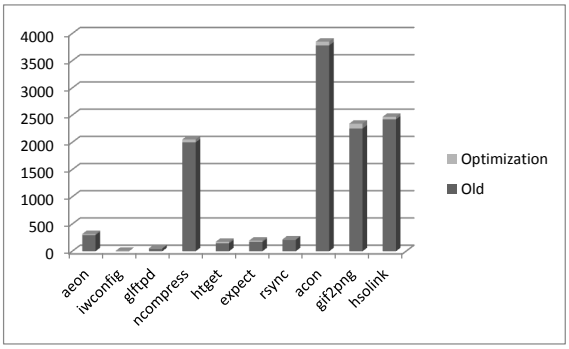

Fig. 11. The speedup after constraint reducing

TABLE VII
THE RESULTS OF EXPLOIT GENERATION FOR LARGE PROGRAMS

Program	Version	Executed Symbolic LOC	Constraint Size (bytes)	Path Exploring Time	Exploit Gen. Time
Unrar	2.90 beta 2 (linux 2.6.26)	1177301	2.91M	1388.45	2569.83
Mplayer	SVN-r33064 (Windows-XP)	1146887	3.89M	1713.76	2939.43
Foxit pdf reader	3.0 BUild 1301 (Windows-XP)	1825260	3.91M	5211.13	10094.17

V. RELATED WORK

APEG (Automatic Patch-based Exploit Generation) [4] compares the differences of a program between its buggy version and a patched version, and generates the exploits to fail the added check in the patched version program. This work needs a patched version program and is feasible only when the patch is to fix by adding input sanitization logic. In addition, most of the exploits generated by APEG are DoS (Denial-Of-Service) attacks, which just crash a program, without executing

shellcode or malicious tasks.

AEG (Automatic Exploit Generation) [2] generates exploits by two stages, which are finding bugs on symbolic execution and then collecting run-time information on concrete execution. AEG only deals with stack buffer overflow and format string vulnerability because it has to add individual safety check constraints to detect each bug. Furthermore, AEG implements an end-to-end approach for exploit generation, including symbolic files, symbolic sockets, etc., and uses return oriented programming to bypass both W⊕X and ASLR[13].

Heelan *et al.* [8] use binary instrumentation to perform taint propagation and collect runtime information. Their work generates exploits by checking whether the EIP register is corrupted by a tainted value, and also handles pointer corruption that corrupts the EIP register indirectly. Similar to our work, a crashing input is needed for taint analysis.

In addition, some work do not generate exploits explicitly, but aim to report a bug which is probably exploitable. For example, !exploitable[1] and some projects [11] of BitBlaze analyze a crash and determine whether it is exploitable.

VI. Conclusion

In this paper, we implemented an automated exploit generation framework, called CRAX, which is built on S^2E, a new platform for symbolic execution. In order to generate control flow hijacking attacks, we focus on detection of the symbolic EIP, other continuation based registers and pointers, and propose a systematic method for searching maximum contiguous symbolic memory for payload injection. Detection of symbolic registers is a comprehensive and easier way to deal with all kinds of control flow hijacking vulnerabilities.

We implemented concolic-mode simulation to perform concolic testing on symbolic execution so that switching between symbolic execution and concolic testing mode is easy without modifying the memory model. In addition, code selection help S^2E to filter irrelevant functions and thus enables symbolic execution to explore interested code more effectively and speedup the process of exploit generation.

In order to evaluate CRAX, we conducted experiments on a variety of vulnerable sample code to demonstrate that it can tackle different kinds of control flow hijacking vulnerabilities. We also experimented on real-world large programs and generated *return-to-libc* and *jump-to-register* exploits to bypass mitigations of ALSR and $W \oplus X$ in real-world software. The successes on mplayer, and foxit pdf reader show that CRAX is a feasible and powerful exploit generation tool for real environments.

References

[1] "!exploitable crash analyzer," *http://msecdbg.codeplex.com/*.

[2] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic Exploit Generation," in *Proceedings of the Network and Distributed System Security Symposium (NDSS'11)*, San Diego, California, USA, February 2011.

[3] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference*, Anaheim, CA, USA, April 2005, pp. 41–46.

[4] D. Brumley, P. Poosankam, D. X. Song, and J. Zheng, "Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy (S&P 2008)*, Oakland, California, USA, May 2008, pp. 143–157.

[5] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, San Diego, California, USA, December 2008, pp. 209–224.

[6] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: a platform for in-vivo multi-path analysis of software systems," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*, Newport Beach, CA, USA, March 2011, pp. 265–278.

[7] V. Ganesh and D. Dill, "A decision procedure for bit-vectors and arrays," in *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, Berlin, Germany, 2007, pp. 519–531.

[8] S. Heelan and D. Kroening, "Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities," *MSc Computer Science Dissertation, University of Oxford, UK*, 2009.

[9] D. Kim, X. Wang, S. Kim, A. Zeller, S. 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, no. 3, pp. 430–447, 2011.

[10] M. Martin and M. Lam, "Automatic Generation of XSS and SQL Injection Attacks with Goal-directed Model Checking," in *Proceedings of the 17th conference on Security symposium*, U. Association, Ed., 2008, pp. 31–43.

[11] C. Miller, J. Caballero, N. M. Johnson, M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "Crash Analysis using BitBlaze," in *Proceedings of the Black Hat USA 2010*, Las Vegas, US, July 2010.

[12] D. A. Molnar and D. Wagner, "Catchconv: Symbolic execution and run-time type inference for integer conversion errors," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2007-23, February 2007.

[13] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit Hardening Made Easy," in *Proceedings of the 20th USENIX Security Symposium (USENIX'11)*, San Francisco, CA, USA, August 2011.

[14] E. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proceedings of the 31st IEEE Symposium on Security and Privacy (SP 2010)*, Berleley/Oakland, California, USA, 2010, pp. 317–331.

[15] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*, Acm, Ed., 2007, pp. 552–561.

[16] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis," in *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, 2008.

[17] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerabilty discovery*. Addison-Wesley Professional, 2007.

[18] P. Team, "Pax address space layout randomization (ASLR)," 2003.

[19] ZZUF, "zzuf – multi-purpose fuzzer."