Binary analysis:
Reverse Engineering

Yue Duan
Outline

- Reverse engineering basics
- Research papers:
  - Automatic Reverse Engineering of Program Data Structures from Binary Execution
  - Howard: A Dynamic Excavator for Reverse Engineering Data Structures
Reverse Engineering Basics

- Compilation process:
  - source code to machine code
Reverse Engineering Basics

- Differences between source code and machine code
  - Source code
    - Human readable
    - Contain high-level semantics (e.g., data structure info, function boundary info)
  - Machine code (stripped binary)
    - Impossible to read
    - High-level semantic information is gone
Reverse Engineering Basics

- Reverse engineering
  - To recover missing information
    - code info
      - disassemble => assembly code
      - decompile => source code
    - function info
      - function recovery => function boundary
    - data info
      - data structure recovery => data structures
Reverse Engineering Basics

- Ultimate goal

```asm
push %ebp
mov %esp,%ebp
sub $0x18,%esp
mov 0x8(%ebp),%eax
lea -0x98(%ebp),%ecx
mov %eax,%edx
mov $0x8c,%eax
mov %eax,0x8(%esp)
mov %edx,0x4(%esp)
mov %ecx,(%esp)
call 0x29
mov 0x8(%ebp),%eax
leave
ret
nop
nop
```

```c
struct employee {
    char name [128];
    int year;
    int month;
    int day;
};
struct employee*
foo (struct employee* src)
{
    struct employee dst;
    // init dst
```
Reverse Engineering Basics

- **Example** 6de: 48 83 ec 30
  - 6de: code address
  - 48: long mode
  - 83:
    - 83 /5 ib: SUB r/m16,imm8
    - 83 /5 ib: SUB r/m32,imm8
  - ec:
    - could mean various things depending on the opcode
    - here it means %rsp
  - 30: 0x30

- **Problems?**
  - 55 48 89 e5 48 83 ec 30 89 7d dc
Reverse Engineering Basics

```c
int main(int argc, char **argv) {
    char *buf1R1;
    char *buf2R1;
    char *buf1R2;
    char buf3[100];

    memset(buf3, 0, 100);
    buf1R1 = (char *) malloc(BUFSIZE2);
    buf2R1 = (char *) malloc(BUFSIZE2);

    free(buf1R1);
    free(buf2R1);

    buf1R2 = (char *) malloc(BUFSIZE1);
    strcpy(buf1R2, argv[1]);

    free(buf2R1);
    free(buf1R1);
}
```
Automatic Reverse Engineering of Program Data Structures from Binary Execution

Zhiqiang Lin, Xiangyu Zhang, Dongyan Xu

NDSS 2010
Problem definition

- Recover data structure specifications
  - syntactic
    - layout
    - offset
    - size
  - semantic
    - types
Motivation

- Security applications
  - vulnerability discovery
  - program signature
  - etc

```c
1
2 void main(int argc, char* argv[])
3 {
4     char tempname[1024];
5     strcpy(tempname, argv[1]);
6 }
```

```
$ ./a.out aaa\n'
```

```
struct A
{
    int a;
    char b;
    int c;
    float d;
};
```

```
struct B
{
    int h;
    char i;
    int j;
    float k;
};
```

```
$ ./a.out aaaaaaaaaaaaaaaaa...a\n'
```

```c
char [4]
unused[1020]
stack_frame_pointer
return_address
char *
input_t
```
System overview

- Recover data structure specifications
  - syntactic
    - layout
    - offset
    - size
  - semantic
    - types
System overview

Application binary -> Type Resolution Points

Data flow tracking, Type Resolution

Data Structure Specification

REWARDS
Reverse Engineering Work for Automatic Revelation of Data Structures
System design

Key idea

- find calls to a well-known function (like a system call)
- types of all the arguments are known
- label these memory locations accordingly
- propagate this info backwards and forwards through the execution of the program
  - whenever labeled data is copied, the label is also assigned to the destination
System design
System design

call 0x8048110<getpid>
return pid_t to %eax

mov %eax, 0x8049124

pid_t pid_t pid_t
Type resolution point 1

- **Syscalls**
  - syscall number
    - syscall_enter: type parameter passing registers (i.e., ebx, ecx, edx, esi)
    - syscall_exit: type return value (eax)

```
<getpid>
36 8048110:  mov  $0x14,%eax
37 8048115:  int  $0x80
38 8048117:  ret
```
Type resolution point 2

- Standard Library call
  - Types of arguments and return value
  - More useful than syscalls
    - 2016 APIs in Libc.so.6
    - 289 syscalls (2.6.15)

13 80480ce: mov %eax,0x4(%esp) <arg2>
14 80480d2: movl $0x8049128, (%esp) <arg1>
15 80480d9: call 0x80480e0 <strcpy>
Evaluation

● Experiment setup
  ○ 10 utility binaries (e.g., ls, ps, ping)

● False negative (data structures missed)
  ○ global: 70%
  ○ heap: 55%
  ○ stack: 60%
  ○ reason
    ■ dynamic analysis

● False positive (get wrong data types)
  ○ global: 3%
  ○ heap: 0%
  ○ stack: 15%
Howard: A Dynamic Excavator for Reverse Engineering Data Structures

Asia Slowinska, Traian Stancescu, Herbert Bos

NDSS 2011
Motivation

● Rewards only
  ○ recovers those data structures
    ■ appear directly or indirectly in the arguments of well-known functions
  ○ only a very small portion of all data structures
  ○ example:
    ■ internal variables
    ■ data structures in the program

● Goal:
  ○ recover more data structures!
System Design

- Why is it so difficult?

```c
1. struct employee {
2.     char name[128];
3.     int year;
4.     int month;
5.     int day;
6. }
7.
8. struct employee e;
9. e.year = 2010;
```

MISSING

- Data structures
- Semantics
System Design

- Key insight

```c
1. struct employee {
2.     char name[128];
3.     int year;
4.     int month;
5.     int day
6. };
7.
8. struct employee e;
9. e.year = 2010;
```

recovery by *memory access patterns*

Yes, data is unstructured...
But – usage is NOT!
System Design

- System overview
  - dynamic approach
  - on top of KLEE - a symbolic execution engine
System Design

- Observe how memory is used at runtime
  - detect data structures based on access pattern
  - memory access patterns provide clues about the layout of data in memory
- if A is a function frame pointer
  - then *(A+8) is likely to point to a function argument passed via the stack
System Design

- if A is an address of an array
  - then *(A+8) is likely to point to an element of this array
System Design

- Example: array recovery
  - access patterns
    - `elem = *(next++)`;
      - looking for chains of accesses in a loop
    - `elem = array[i]`;
      - looking for sets of accesses with the same base in a linear space
System Design

- Challenges 1:
  - Different memory access patterns within the same space
    - Solution:
      - Howard prefers pattern 1 over pattern 2 over pattern 3
System Design

- Challenges 2:
  - Decide which memory accesses are relevant
  - Problems caused by *memset-like* functions
  - Solution
    - Heuristic preference for non-regular accesses
## Evaluation

<table>
<thead>
<tr>
<th>Prog</th>
<th>LoC</th>
<th>Size</th>
<th>Funcs %</th>
<th>Vars %</th>
<th>How tested?</th>
<th>KLEE %</th>
</tr>
</thead>
<tbody>
<tr>
<td>wget</td>
<td>46K</td>
<td>200 KB</td>
<td>298/576 (51%)</td>
<td>1620/2905 (56%)</td>
<td>KLEE + test suite</td>
<td>24%</td>
</tr>
<tr>
<td>fortune</td>
<td>2K</td>
<td>15 KB</td>
<td>20/28 (71%)</td>
<td>87/113 (77%)</td>
<td>test suite</td>
<td>N/A</td>
</tr>
<tr>
<td>grep</td>
<td>24K</td>
<td>100 KB</td>
<td>89/179 (50%)</td>
<td>609/1082 (56%)</td>
<td>KLEE</td>
<td>46%</td>
</tr>
<tr>
<td>gzip</td>
<td>21K</td>
<td>40 KB</td>
<td>74/105 (70%)</td>
<td>352/436 (81%)</td>
<td>KLEE</td>
<td>54%</td>
</tr>
<tr>
<td>lightpd</td>
<td>21K</td>
<td>130 KB</td>
<td>199/360 (55%)</td>
<td>883/1418 (62%)</td>
<td>test suite</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Thank you!

Question?