Software testing and concolic execution

Jonathan Salwan

LSE Summer Week 2013

July 18, 2013
Who I am: Jonathan Salwan

Where I work: Sysdream

What is my job: R&D
Software testing
From Wikipedia: Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test.
Bug impact

- $100 Billion per year in Europe
- Rocket Arianne V : $370 Million
- Therac-25 (Radiotherapy) : People died...
Certifications

- **ISO/IEC 9126**: Software engineering - Product quality

- **SGS**: Certification services from SGS demonstrate that your products, processes, systems or services are compliant with national and international regulations and standards.

- **ED-12C/DO-178C**: Software Considerations in Airborne Systems and Equipment Certification
## Software testing statistics

<table>
<thead>
<tr>
<th></th>
<th>Fast</th>
<th>Intelligent</th>
<th>Code coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual test</td>
<td>KO</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>Automatic test</td>
<td>OK</td>
<td>KO</td>
<td>KO</td>
</tr>
<tr>
<td>Formal proof</td>
<td>KO</td>
<td>OK</td>
<td>OK</td>
</tr>
</tbody>
</table>
Bugs hunting
To find bugs, we have several methodologies.

- White box
- Black box
- Pattern matching
- Dumb fuzzing
- In-memory fuzzing
- ...

Bugs hunting

Jonathan Salwan
Software testing and concolic execution
PHP 5.3.6 - Stack buffer overflow in socket_connect (CVE-2011-1938)

```c
PHP_FUNCTION(socket_connect)
{
    [...]
    struct sockaddr_un s_un;
    [...]
    case AF_UNIX:
        memset(&s_un, 0, sizeof(struct sockaddr_un));
        s_un.sun_family = AF_UNIX;
        memcpy(&s_un.sun_path, addr, addr_len);
        retval = connect(p->bsd_socket, (struct sockaddr *) &s_un,
             (socklen_t) XtOffsetOf(struct sockaddr_un, sun_path) + addr_len);
    break;
}
Most vulnerabilities are found in private softwares thanks to black box fuzzing

- Same idea than white box fuzzing
- Need to skill++ in assembly
- Really more time consuming than white box fuzzing
Pattern matching

Jonathan Salwan
Software testing and concolic execution

```
mov    rax, [rbp+var_20]
mov    rax, [rax+8]
mov    [rbp+var_8], rax
mov    rax, [rbp+var_8]
mov    rsi, rax
mov    edi, offset format ;"%s"
mov    eax, 0
call   _printf
```
Dumb fuzzing

The idea is to fuzz the program with semi-random data (based on a specification of the fileformat/protocol/whatever)

1. Focus a specific RFC (Ex: http, ftp, pdf, png...)
2. Send semi-random data based on the RFC’s fields.
Dumb fuzzing - http server
In-memory fuzzing

The idea of this method is to instrument directly the target application’s code to fuzz it. Here are the different steps:

1. Break before and after the target function
2. Save the context execution
3. Send semi-random data
4. Restore the execution context previously saved
5. Repeat until a crash is triggered
In-memory fuzzing - Call graph
In-memory fuzzing - Concept
Manual vs automatic testing

With the classical automatic tests it’s difficult to detect some bugs:

- Info leaks
- All overflows without crashes
- Design errors
- ...
Jonathan Salwan
Software testing and concolic execution
Concrete execution

The concrete execution is the execution of a real program.
Symbolic execution

The symbolic execution is used to determine a time $T$ all conditions necessary to take the branch or not.
Symbolic execution - Example

```c
int foo(int i1, int i2)
{
    int x = i1;
    int y = i2;
    if (x > 80){
        x = y * 2;
        y = 0;
        if (x == 256)
            return TRUE;
    }
    else{
        x = 0;
        y = 0;
    }
    ...
    return FALSE;
}
```
Symbolic execution - Example

Three possible paths. One path for True and two paths for False.

- return True
  
  PC: i1>80 & (i2 * 2)=256

- return False
  
  PC: i1<=80 | (i1>80 & (i2 * 2))=256
Concolic execution

Concolic execution is a technic that uses both symbolic and concrete execution to solve a constraint path.
IR and constraints solver
Valgrind is an instrumentation framework for building dynamic analysis tools.
Valgrind - VEX

VEX is the Valgrind’s intermediate language.
Valgrind - VEX sample

Instruction: **add eax, ebx**

\[
\begin{align*}
t3 &= \text{GET:} I32(0) \quad \# \text{get } \%\text{eax, a 32-bit integer (t3 = eax)} \\
t2 &= \text{GET:} I32(12) \quad \# \text{get } \%\text{ebx, a 32-bit integer (t2 = ebx)} \\
t1 &= \text{Add32}(t3, t2) \quad \# \text{t1 = addl(eax, ebx)} \\
PUT(0) &= t1 \quad \# \text{put } \%\text{eax (eax = t1)}
\end{align*}
\]
Z3 is a high-performance theorem prover developed by Microsoft.
Z3 - Example

```python
$ cat ./ex.py
from z3 import *

x = BitVec('x', 32)
s = Solver()
s.add((x ^ 0x55) + (3 - (2 * 12)) == 0x30)
print s.check()
print s.model()

$ ./ex.py
sat
[x = 16]
```
Z3 - Why?

We will use it to solve all the constraints from our VEX’s output.
Proof of concept
Last summer, with my friends Ahmed Bougacha and Pierre Collet, we worked on a concolic PoC just for fun.
Goal

Objectif : Solve this dumb crackme

char *serial = "\x30\x39\x3c\x21\x30";

int main(void)
{
    int fd, i = 0;
    char buf[260] = {0};
    char *r = buf;

    fd = open("serial.txt", O_RDONLY);
    read(fd, r, 256);
    close(fd);
    while (i < 5){
        if (((*r ^ 0x55) != *serial)
            return 0;
        r++, serial++, i++;
    }
    if (!*r)
        printf("Good boy\n");
    return 0;
}
Plan

1. Taint the user input (via Valgrind)
2. Spread the taints (via Valgrind)
3. Save all constraints (via Valgrind)
4. Solve all constraints (via Z3)
Taint syscall entries - Diagram
Taint syscall entries - in Valgrind

With valgrind we can add a **Pre** and **Post** syscall handler.
Taint syscall entries - in Valgrind

```c
static void pre_syscall(ThreadId tId, Uint syscall_number, UWord* args,
                        Uint nArgs){
}

static void post_syscall(ThreadId tId, Uint syscall_number, UWord* args,
                          Uint nArgs, SysRes res){
}

static void init(void)
{
    VG_(details_name)       ("Taminoo");
    VG_(details_version)    (NULL);
    VG_(details_description)("Taint analysis poc");
    [...]                    
    VG_(basic_tool_funcs)   (init, instrument, fini);
    [...]                    
    VG_(needs_syscall_wrapper)(pre_syscall, post_syscall);
}

VG_DETERMINE_INTERFACE_VERSION(init)
```
Spread the taints

To propagate correctly the taints, we instrument each instruction of the binary. If it is a **GET**, **LOAD**, **PUT** or **STORE** instruction we spread the taints.
The variable $a$ is tainted. When $b = a$ and $c = b$, $b$ and $c$ will also be tainted because they can be controlled via $a$.

```c
uint32_t a, b, c;

a = atoi(user_input);
b = a; /* b is tainted */
c = b; /* c is tainted */
```
Spread the taints - in Valgrind

switch (st->tag) {
    case Ist_Store:
        INSERT_DIRTY(helper_store,
        /* dst_addr */ st->Ist.Store.addr,
        /* src_tmp */ INSERT_TMP_NUMBER(st->Ist.Store.data),
        /* size */ INSERT_EXPR_SIZE(st->Ist.Store.data));
        break;
    case Ist_Put:
        INSERT_DIRTY(helper_put,
        /* dst_reg */ mkIRExpr_HWord(st->Ist.Put.offset),
        /* src_tmp */ INSERT_TMP_NUMBER(st->Ist.Put.data),
        /* size */ INSERT_EXPR_SIZE(st->Ist.Put.data));
        break;
    case Iex_Get:
        INSERT_DIRTY(helper_get,
        /* dst_tmp */ mkIRExpr_HWord(dst),
        /* src_reg */ mkIRExpr_HWord(data->Iex.Get.offset),
        /* size */ mkIRExpr_HWord(sizeofIRType(data->Iex.Get.ty)));
        break;
    case Iex_Load:
        INSERT_DIRTY(helper_load,
        /* dst_temp */ mkIRExpr_HWord(st->Ist.WrTmp.tmp),
        /* src_addr */ st->Ist.WrTmp.data->Iex.Load.addr,
        /* size */ INSERT_TYPE_SIZE(data->Iex.Load.ty));
        break;
    [...]
}
Constraints - Output

```
==14567==
#1:8    = Read(4,0)
#2:8    = Read(4,1)
#3:8    = Read(4,2)
#4:8    = Read(4,3)
#5:32   = 8Uto32(#1:8)
#6:32   = Xor32(#5:32,0x55)
#7:8    = 32to8_0(#6:32)
#8:8    = 32to8_1(#6:32)
#9:8    = 32to8_2(#6:32)
#10:8   = 32to8_3(#6:32)
#11:32  = 8Uto32(#7:8)
#12:8   = 32to8(#11:32)
#13:1   = CmpEQ8(#12:8,0x30) = False
#14:32  = 1Uto32(#13:1)
#15:1   = 32to1(#14:32)
Jump(#15:1) = False
#6 freed
#5 freed
#14 freed
#13 freed
#12 freed
#15 freed
#11 freed
#7 freed
==14567==
```
Constraints - List

Every constraint depends of the previous constraint.
Constraints - List

1. \#13:1 = CmpEQ8(\#12:8, 0x30)
2. \#12:8 = 32to8(\#11:32)
3. \#11:32 = 8Uto32(\#7:8)
4. \#7:8 = 32to8_0(\#6:32)

5. \#6:32 = Xor32(\#5:32, 0x55)
6. \#5:32 = 8Uto32(\#1:8)
7. \#1:8 = Read(4, 0)
8. Serial.txt
Solve constraints with Z3

All the constraints are converted using the Z3 syntax
Solve constraints with Z3 - Original constraint

The first constraint is: \( \text{CmpEQ8(Xor32(Read(4,0),0x55),0x30)} \)
Solve constraints with Z3 - Z3 pattern

# First constraint in Z3 pattern
x = BitVec('x', 32)
s = Solver()
s.add((x ^ 0x55) == 0x30)
Solve constraints with Z3 - Concolic execution

Entry

\[ C_1 = \text{False} \]

Failure

Entry

\[ C_2 = \text{False} \]

Failure

Entry

\[ C_3 = \text{False} \]

Failure

Entry

\[ C_4 = \text{False} \]

Failure

Entry

\[ C_5 = \text{False} \]

Failure

Entry

\[ C_1 = \text{True} \]

Entry

\[ C_2 = \text{True} \]

Entry

\[ C_3 = \text{True} \]

Entry

\[ C_4 = \text{True} \]

Entry

\[ C_5 = \text{True} \]

Success
Solve constraints with Z3 - All constraints solved

\[
\begin{align*}
C1 &= \text{CmpEQ8(Xor32(Read(4,0),0x55),0x30)} = 'e' \\
C2 &= \text{CmpEQ8(Xor32(Read(4,1),0x55),0x39)} = 'l' \\
C3 &= \text{CmpEQ8(Xor32(Read(4,2),0x55),0x3c)} = 'i' \\
C4 &= \text{CmpEQ8(Xor32(Read(4,3),0x55),0x21)} = 't' \\
C5 &= \text{CmpEQ8(Xor32(Read(4,4),0x55),0x30)} = 'e'
\end{align*}
\]
Questions ?
Thanks!

http://sbxc.org
http://twitter.com/JonathanSalwan