Binsec – A Binary Analysis Platform

BlackHoodie – December, 07th 2019

Binsec team:

Sébastien Bardin, Richard Bonichon, Lesly-Ann Daniel, Robin David, Adel Djoudi, Benjamin Farinier, Josselin Feist, Guillaume Girol, Matthieu Lemerre, Grégoire Menguy, Manh-Dung Nguyen, Olivier Nicole, Mathilde Ollivier, Frédéric Recoules, Yaëlle Vinçont (Ella).

https://binsec.github.io
https://github.com/binsec/binsec

Why Binary-Level Analysis?

Need Code Analysis

- Bug-Finding *(e.g. find RTE)*
- Verif. *(e.g. assert no RTE)*
- Reverse-Engineering

At Binary Level

- Source code is not available
  - closed-source library
  - legacy source code
  - malware
  - CTF
- Don't trust compilers!

```c
void fun(int i, int j){}
int bat() { printf("Bat"); }
int man() { printf("Man"); }
int main() {
    fun(bat(), man());
}
```

Result

- `clang-5.0`: "BatMan"
- `gcc-5.1`: "ManBat"

Binary Code is Difficult to Analyze

- No types (only registers and memory)
- No high level CFG (no `for` or `while` loops)
- Data dependencies are not explicit (memory operations)
- Large code size

→ *Manual analysis is tedious!*

Binsec can help you!

Goal: Automatic analysis of binary code based on formal methods.

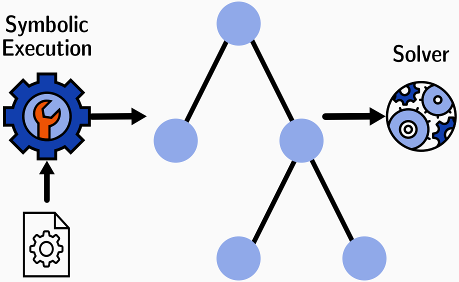
In this talk: focus on Symbolic Execution

Binary-Analysis

Symbolic Execution & Binsec

Symbolic Execution

- Scales better than other semantic binary-level analysis
- Widely used in intensive testing and security analysis
- Leading technique for BF
- Precise (no false alarm)



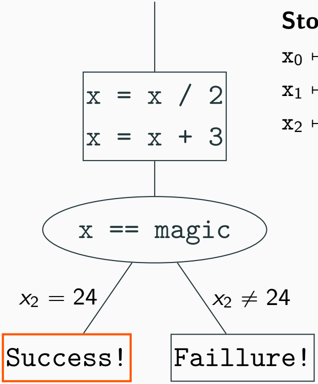
The KeY Project

BINSEC

Symbolic Execution

```
uint32_t magic = 24;

void foo(uint32_t x) {
  x = x / 2 + 3;
  if(x == magic)
    printf("Success!");
  else
    printf("Faillure!");
  return;
}
```



```
x = x / 2
x = x + 3
```

`x == magic`

$x_2 = 24$

$x_2 \neq 24$

`Success!`

`Faillure!`

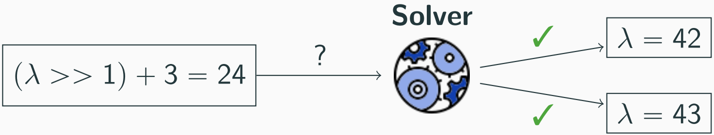
Store

$\mathtt{x}_0 \mapsto \lambda$

$\mathtt{x}_1 \mapsto \mathtt{x}_0 >> 1$

$\mathtt{x}_2 \mapsto \mathtt{x}_1 + 3$

How to reach “Success!”?



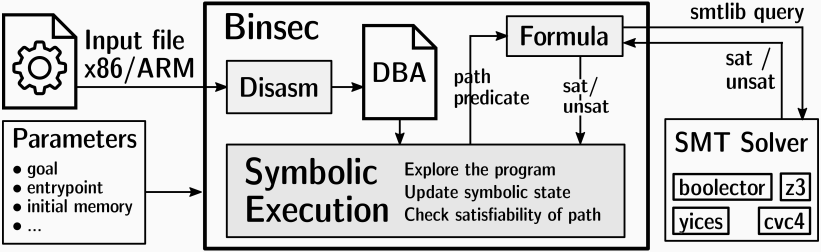
Solver

$(\lambda >> 1) + 3 = 24 \xrightarrow{?}$

✓ $\lambda = 42$

✓ $\lambda = 43$

Symbolic Execution & Binsec



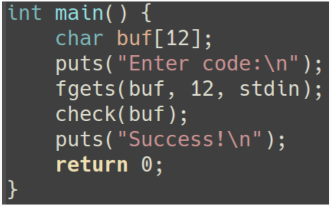
50k lines of OCaml

Decoder for x86, ARMv7, RISC-V.

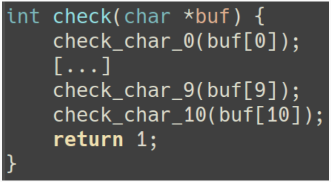
More than just SE: Disassember, DSE, simplifications

Use Case: Manticore CTF

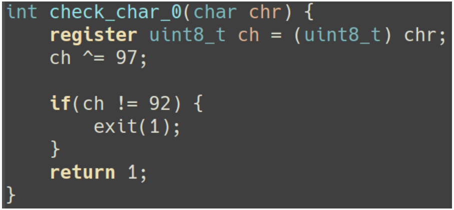
Manticore CTF



```
int main() {
    char buf[12];
    puts("Enter code:\n");
    fgets(buf, 12, stdin);
    check(buf);
    puts("Success!\n");
    return 0;
}
```



```
int check(char *buf) {
    check_char_0(buf[0]);
    [...]
    check_char_9(buf[9]);
    check_char_10(buf[10]);
    return 1;
}
```



```
int check_char_0(char chr) {
    register uint8_t ch = (uint8_t) chr;
    ch ^= 97;

    if(ch != 92) {
        exit(1);
    }
    return 1;
}
```

Result

buf [0] ˆ 97 = 92

buf [0] = 1100001 ˆ 1011100

buf [0] = 0111101

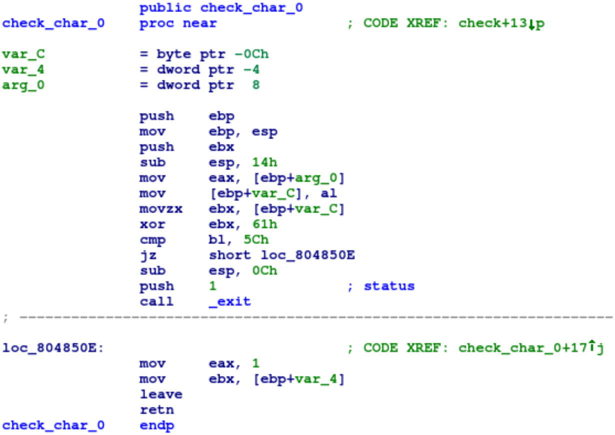
buf [0] = '='

https://blog.trailofbits.com/2017/05/15/magic-with-manticore/

Problem: I am a Lazy Person!

Need to reverse all 11 characters

We don't have the source!



```
                public check_char_0
check_char_0    proc near               ; CODE XREF: check+13↓p

var_C           = byte ptr -0Ch
var_4           = dword ptr -4
arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                push    ebx
                sub     esp, 14h
                mov     eax, [ebp+arg_0]
                mov     [ebp+var_C], al
                movzx   ebx, [ebp+var_C]
                xor     ebx, 61h
                cmp     bl, 5Ch
                jz      short loc_804850E
                sub     esp, 0Ch
                push    1               ; status
                call    _exit
; ---------------------------------------------------------------------

loc_804850E:                            ; CODE XREF: check_char_0+17↑j
                mov     eax, 1
                mov     ebx, [ebp+var_4]
                leave
                retn
check_char_0    endp
```

Binsec Can Help

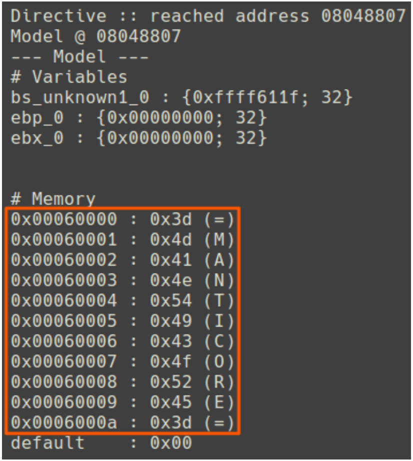
BINSEC

Configuration

```
file = manticore
entrypoint = check
reach = x08048807 #end of check
cut = x080483C0   #exit
solver = boolector
```

Initial Memory

```
esp := [xffff5000..xffff8000];
@[esp+4,4] := x00060000; #buf[]
```



```
Directive :: reached address 08048807
Model @ 08048807
--- Model ---
# Variables
bs_unknown1_0 : {0xffff611f; 32}
ebp_0 : {0x00000000; 32}
ebx_0 : {0x00000000; 32}


# Memory
0x00060000 : 0x3d (=)
0x00060001 : 0x4d (M)
0x00060002 : 0x41 (A)
0x00060003 : 0x4e (N)
0x00060004 : 0x54 (T)
0x00060005 : 0x49 (I)
0x00060006 : 0x43 (C)
0x00060007 : 0x4f (O)
0x00060008 : 0x52 (R)
0x00060009 : 0x45 (E)
0x0006000a : 0x3d (=)
default    : 0x00
```

Conclusion

Binsec vs. Other Tools

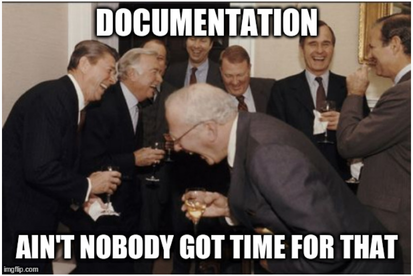
Other Tools: angr, triton, manticore, etc.

Pros of Binsec:

- Research tool, built with formal methods in mind
- Principled and generic core engine.

Cons of Binsec:

- work in progress,
- don't look for doc!



We Also Use Binsec to do Useful Stuff

- Symbolic *deobfuscation* with and application to X-Tunnel malware (Robin),
- Verification of *absence of privilege escalation* in an OS (Olivier),
- Verification of *constant-time cryptographic* implementations (Lesly-Ann),
- Automatic bug-finding using *fuzzing* guided by *symbolic analysis* (Yaëlle & Manh-Dung),
- Certified decompilation (Frédéric).

Conclusion

- Binary analysis is important but difficult
- Symbolic execution can automate the analysis
- Symbolic Execution is you friend for solving CTFs :)
- Can also be used for Bug-Finding & Verification