Paul HERNAULT
phernault@quarkslab.com

Fuzzing binaries using Dynamic Instrumentation

French-Japan cybersecurity workshop Kyoto - April 23-25, 2019

quarkslab
SECURING EVERY BIT OF YOUR DATA

Presentation

Quarkslab

Presentation

acid@kyoto:~$ whoami

- Paul HERNAULT, engineer at **Quarkslab**
- **Vulnerability research**, Fuzzing, instrumentation

Quarkslab

- French cybersecurity company (30~ engineers)
- Focused on
  - **Vulnerability research**, offensive security, systems analysis
- Services, **Research**, Products

Vulnerability research at Quarkslab

What are we looking at?

- Desktop (Windows/Linux/macOS)
- Mobile (Android/iOS/Trustzones)
- Embedded systems (Routers, media (STB), IoT, cars (ECU))
- ...

We love to find vulnerabilities

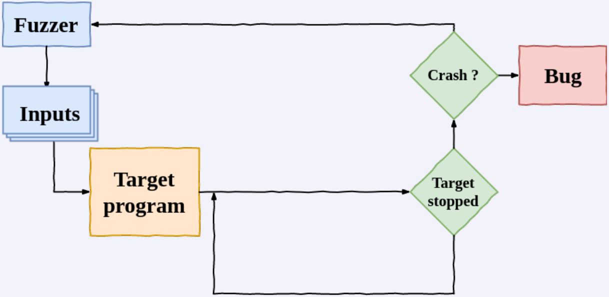
- Code review
- Reverse engineering
- **Fuzzing** (This talk!)

Fuzzing at Quarkslab

Fuzzing: definition

Fuzzing (fuzz testing) is an automated technique used to discover errors and security loopholes in programs. It works by inputting **“random”** *data to the target, in an attempt to make it crash.*

Fuzzing: in picture



Fuzzing at Quarkslab

Why fuzzing?

- Pros
  - **Efficient** way to find vulns in code
  - Fuzzing as a background task (fire and forget)
- Cons
  - Requires tailored tools
  - Not exhaustive

Our take at Fuzzing

Reuse existing tools

- Small company, building a fuzzer is **time-consuming**
- **Reuse** and adapt **existing frameworks**
  - Lots of open-source fuzzers

Our needs

- **Smart** (guided fuzzer)
- **Multi-platform** (Windows, macOS, Linux)
- **Multi-architecture** (x86, x86_64, ARM, ARM64)
- **Binary fuzzing**

Review and Benchmarks

Looking for the perfect fuzzer

- Review of the code
- Benchmarking fuzzers
    - **Speed** / **Efficiency**
- State of the tool
    - Support for **multi-[platform|architecture]** and **binary fuzzing**
    - Development state

A note on benchmarking fuzzers

- Benchmarking is **hard**
    - **Lacks references** (LogicBombs, LAVA)
    - **Results differ** from one run to another (due to randomness)
    - Hard to **simulate real-world programs**
- Interpretation of results may require in-depth analysis

The perfect fuzzer

There is no perfect fuzzer. . .

- No fuzzer fulfills all of our needs
    - Lacks **architecture supports** (AFL)
    - Lacks **binary fuzzing** (Honggfuzz)
    - **Not efficient** (Radamsa)
    - . . .

So what?

- Build upon one of them
- Tweak them to fit our needs
- We tried both AFL and **Honggfuzz**

Fuzzing binaries with AFL

About AFL

- We tried AFL for ˜1 year
    - Good results but. . .
- Coding glue is not enough. We need to modify the fuzzer itself
    - AFL lacks modern features
    - AFL is not maintained, not modular nor flexible

Why switching?

- We needed something more flexible
    - State of the Art on Fuzzing concluded **Honggfuzz** was better suited for us
- *Our experience on AFL helped a lot on Honggfuzz/QBDI*

Our choice: Honggfuzz

Honggfuzz

- Tool developed by **Robert Swiecki** (Google) since 2010
- Supports ARM, ARM64, x86, x86_64
- Supports Linux, Android, macOS, Windows
- Modular, flexible, written in C
- **Efficient and modern** fuzzing strategies

One downside

- No binary fuzzing :(
  - *There is a mode with hardware-based features for fuzzing binaries, but it's not efficient, nor cross[architecture|platform]*
- What is the difference between source and binary fuzzing?

Understanding modern fuzzers internals

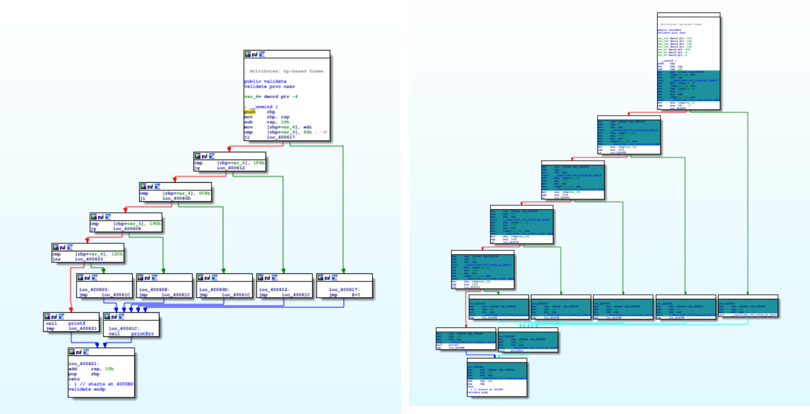
Source-based fuzzing

- Most fuzzers provide their **tweaked compiler**
    - e.g. AFL: **afl-gcc**, Honggfuzz: **hfuzz**
- Adds instrumentation code at various locations during compilation
    - Tracks coverage (basic blocks)
    - Tracks specific instructions (comparisons, divisions)
    - Tracks function calls

What is the use of instrumentation?

- Determines if an input is interesting (good coverage? reaches deep blocks?)
- Updates a **corpus** of **inputs**
- Mutates the **corpus** to discover the binary, and bugs

Fuzzers static instrumentation



```
; Attributes: bp-based frame

public validate
validate proc near

var_4= dword ptr -4

; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     [rbp+var_4], edi
cmp     [rbp+var_4], 64h ; 'd'
jl      loc_400617
```

```
cmp     [rbp+var_4], 1F4h
jg      loc_400612
```

```
cmp     [rbp+var_4], 0C8h
jl      loc_40060D
```

```
cmp     [rbp+var_4], 190h
jg      loc_400608
```

```
cmp     [rbp+var_4], 12Ch
jnz     loc_400603
```

```
loc_400603:
jmp     loc_40061C
```

```
loc_400608:
jmp     loc_40061C
```

```
loc_40060D:
jmp     loc_40061C
```

```
loc_400612:
jmp     loc_40061C
```

```
loc_400617:
jmp     $+5
```

```
call    printX
jmp     loc_400621
```

```
loc_40061C:
call    printErr
```

```
loc_400621:
add     rsp, 10h
pop     rbp
retn
; } // starts at 4005B0
validate endp
```

Figure: Compilation [without—with] instrumentation

How to use instrumentation

Callbacks and bitmap

- There are instrumentation callbacks, used to update a bitmap
- Bitmap is shared between the monitoring process and the target

Simplified bitmap update

- Called on every basic block entry

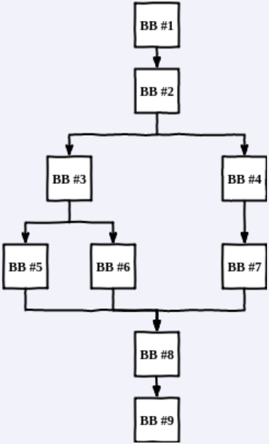
```
int bitmap[ARBITRARY_SIZE]; // shared memory
void basicBlockCallback(){
    bitmap[H(currentInstructionPointer)]++;
}
```

What is it used for

- Keep track of reached basic block (and number of time)
- If bitmap is updated, the input is added to the corpus

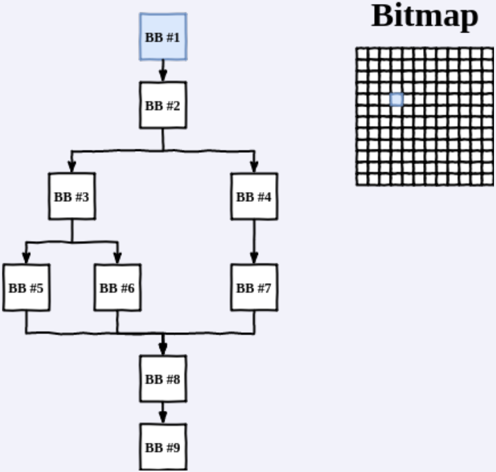
The use of the instrumentation

Updating bitmap using static instrumentation



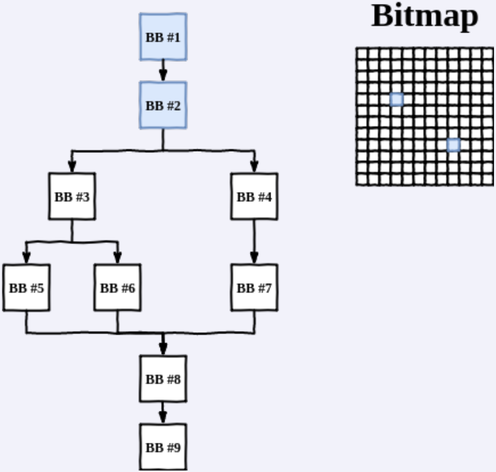
The use of the instrumentation

Updating bitmap using static instrumentation



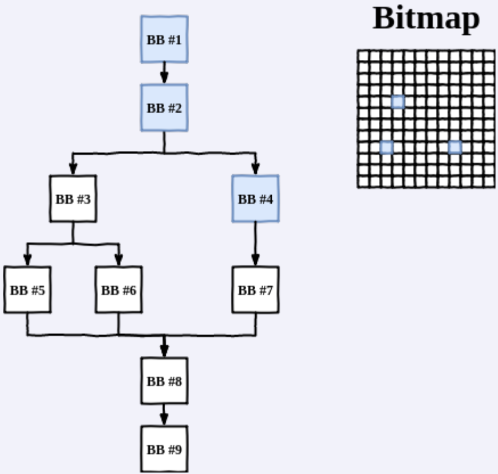
The use of the instrumentation

Updating bitmap using static instrumentation



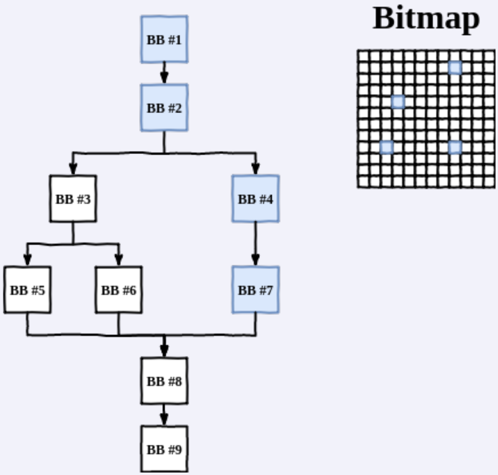
The use of the instrumentation

Updating bitmap using static instrumentation



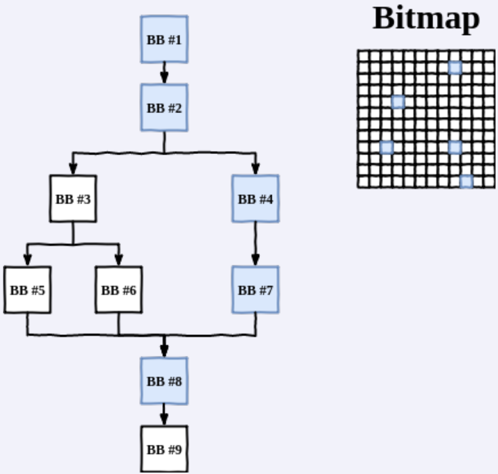
The use of the instrumentation

Updating bitmap using static instrumentation



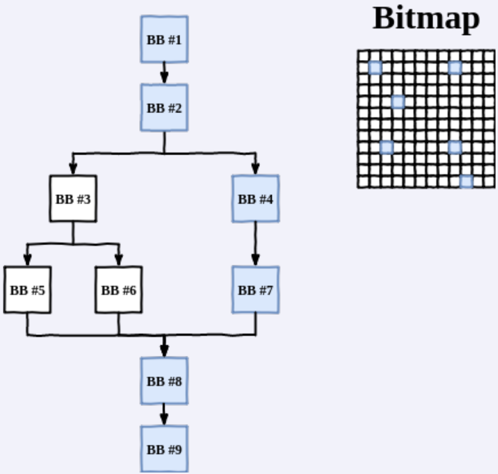
The use of the instrumentation

Updating bitmap using static instrumentation



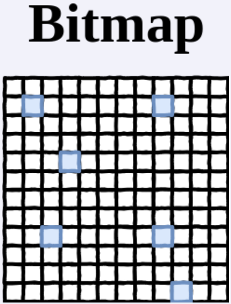
The use of the instrumentation

Updating bitmap using static instrumentation



The use of the instrumentation

Bitmap state

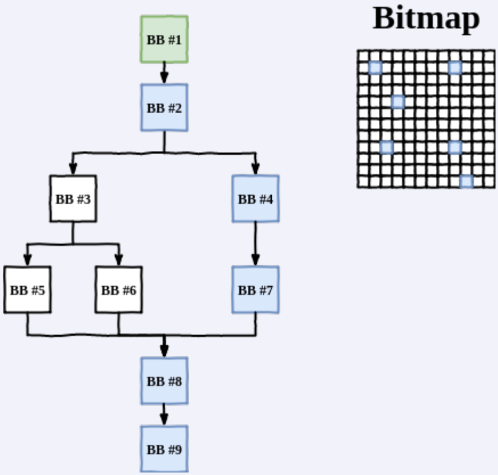


Updating bitmap using static instrumentation

- Blue input updated the bitmap -> Keep it

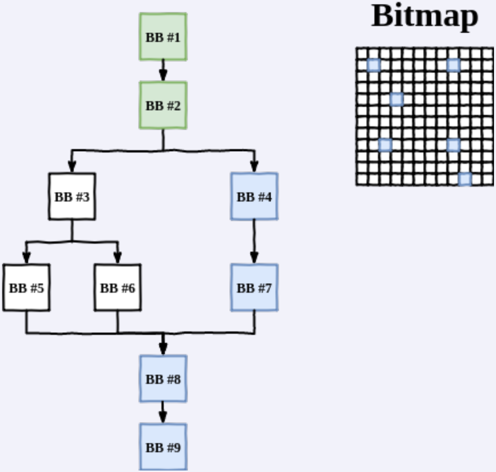
The use of the instrumentation

Updating bitmap using static instrumentation



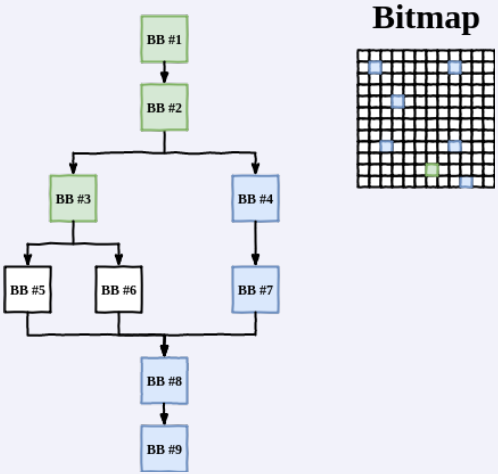
The use of the instrumentation

Updating bitmap using static instrumentation



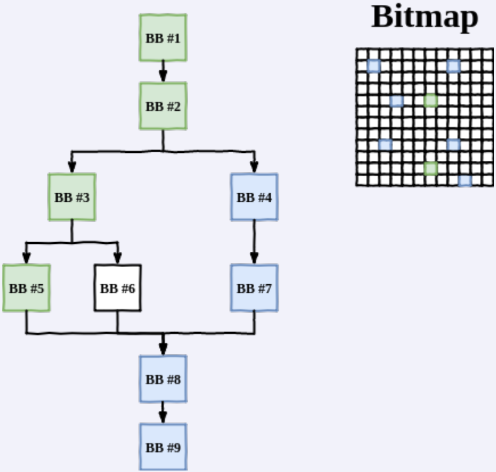
The use of the instrumentation

Updating bitmap using static instrumentation



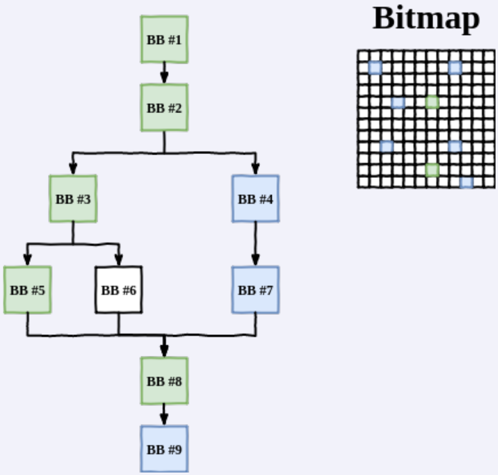
The use of the instrumentation

Updating bitmap using static instrumentation



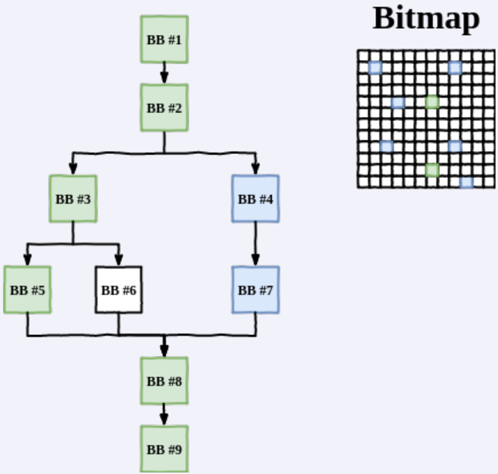
The use of the instrumentation

Updating bitmap using static instrumentation



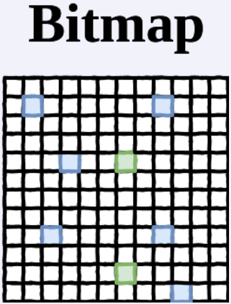
The use of the instrumentation

Updating bitmap using static instrumentation



The use of the instrumentation

Bitmap state



Updating bitmap using static instrumentation

- Green input updated the bitmap -> keep it

The use of the instrumentation

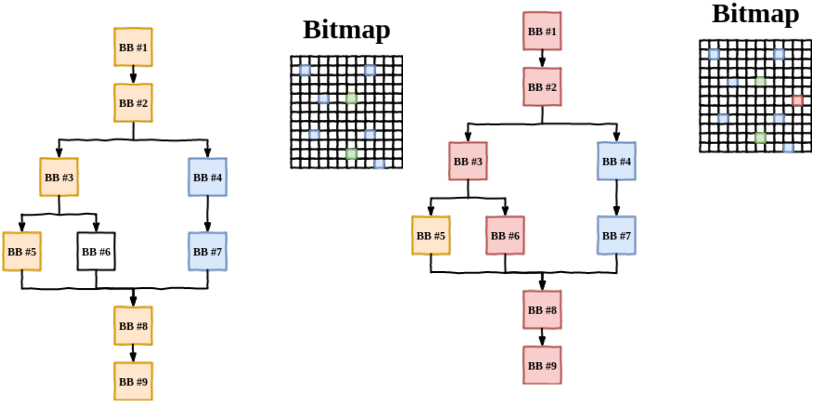


Figure: Run orange and red

The use of the instrumentation

Bitmap state

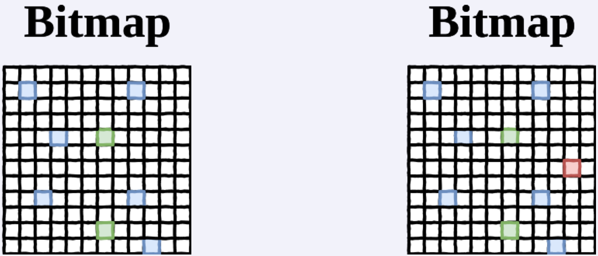


Figure: Run orange and red

Updating bitmap using static instrumentation

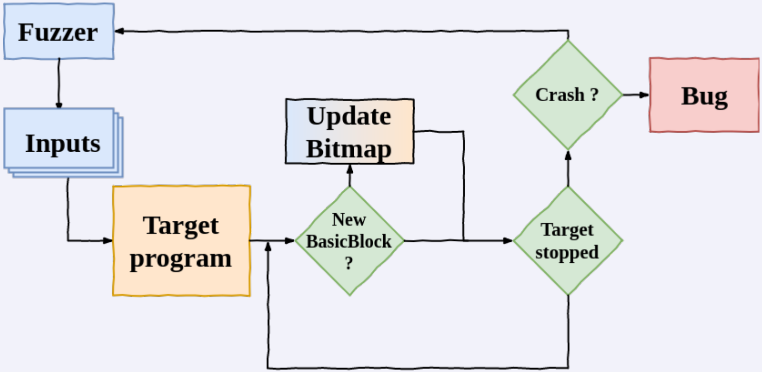
- Orange input did not update the bitmap -> drop it
- Red input updated the bitmap -> keep it

The use of the instrumentation

Guided fuzzing

- Instrumentation is used to **guide** the fuzzer

Guided fuzzing: in picture



What about closed-source binaries?

What makes it hard to fuzz binaries?

- We need to inject code
    - But we are not compiling the binary
- How can we do that?
    - Debugging (slow)
    - Emulation (slow)
    - Binary rewriting (hard)
    - Dynamic Binary Instrumentation \o/ (?)

Introduction of DBI

What is Dynamic Binary Instrumentation

Dynamic Binary Instrumentation (DBI) allows to monitor and analyze the behaviour of a binary through instrumentation code injected at runtime. The instrumentation code is injected in the stream of the normal instructions without the target program knowing.

How it works?

0. The DBI engine is injected in the target program (same address space)
1. The DBI discovers a basic block.
2. It takes instructions, patches them, and adds instrumentation code.
3. It JITs everything together, and executes it.
4. goto 1

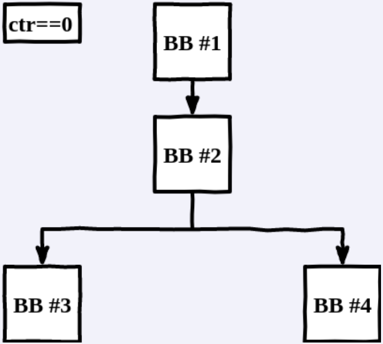
Introduction of DBI: Example

DBI usage: Counting basic blocks

- One easy usage of a DBI, is for profiling
  - Counting instructions executed
  - Counting basic blocks executed

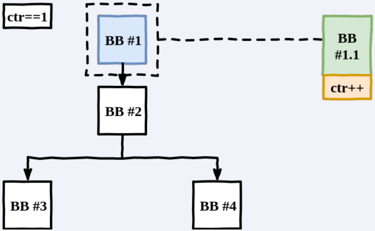
Introduction of DBI: Example

DBI usage: Counting basic blocks



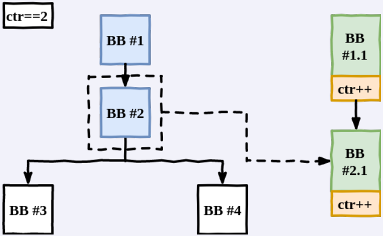
Introduction of DBI: Example

DBI usage: Counting basic blocks



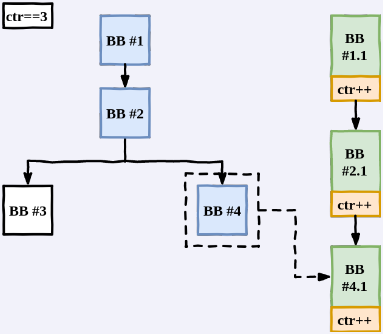
Introduction of DBI: Example

DBI usage: Counting basic blocks



Introduction of DBI: Example

DBI usage: Counting basic blocks



Introduction of DBI: QBDI

QBDI: QuarkslaB Dynamic binary Instrumentation

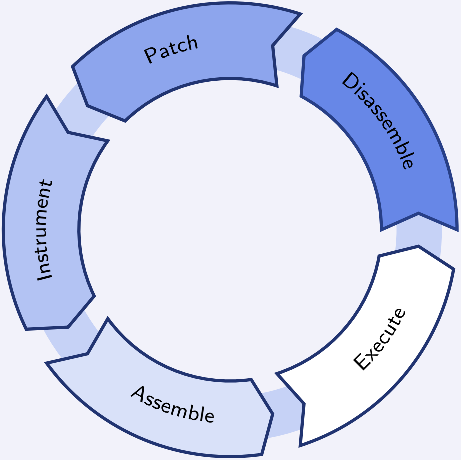
- Quarkslab has its own DBI: **QBDI**
  - *https://github.com/quarkslab/QBDI/*
- Based on LLVM
- Instruction / basic block granularity

How it works?

1. Disassemble
2. Patch
3. Instrument
4. Assemble
5. Execute

Introduction of DBI: QBDI

How it works: in picture?



Mimic Honggfuzz

QBDI callback to simulate Honggfuzz instrumentation

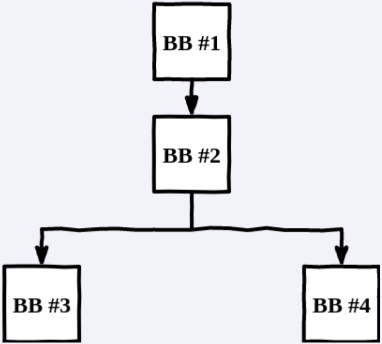
- Dynamic instrumentation allows to inject code at
    - every instruction
    - every basic block
    - specific instructions (mnemonic)

How we use QBDI

- Inject callbacks at the end of basic blocks
- Manually update the bitmap
- Fake **Honggfuzz** \o/ without modifying the source code!

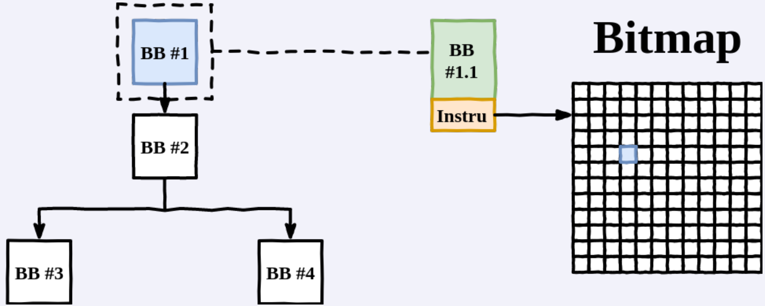
Mimic Honggfuzz

Mimic Honggfuzz: in picture



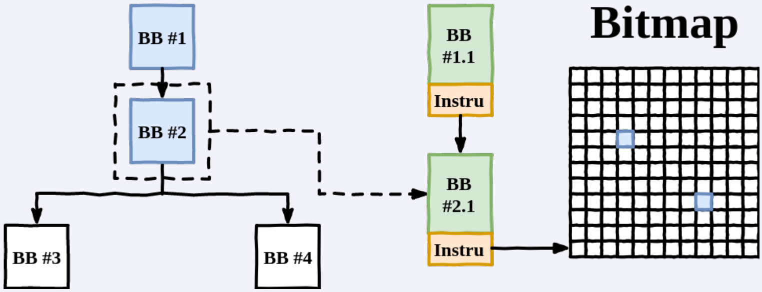
Mimic Honggfuzz

Mimic Honggfuzz: in picture



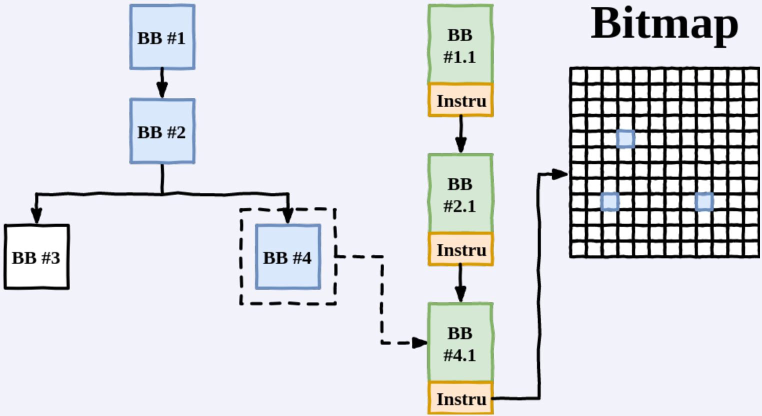
Mimic Honggfuzz

Mimic Honggfuzz: in picture



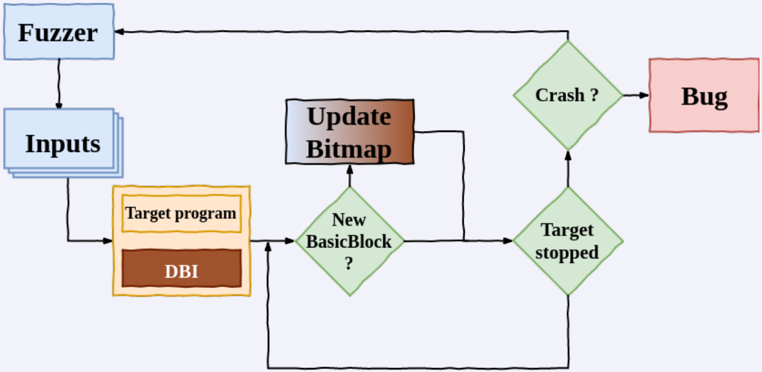
Mimic Honggfuzz

Mimic Honggfuzz: in picture



Mimic Honggfuzz

Mimic Honggfuzz: in picture



Fuzzer

Inputs

Target program

DBI

New BasicBlock ?

Update Bitmap

Target stopped

Crash ?

Bug

Honggfuzz/QBDI - Demo time

Demo - HF VS blackbox honggfuzz VS HF/QBDI

- Normal update in the first run –> HF compiled
- No update in 2nd –> clang compiled
- Updates in 3rd run –> clang compiled + preload

Honggfuzz/QBDI - Demo time

Demo - HF VS blackbox honggfuzz VS HF/QBDI

1. Static instrumentation (native) –> Compile the binary with honggfuzz-clang
2. No instrumentation (black box) –> Compile with clang
3. Runtime instrumentation with QBDI –> Compile with clang (and preload QBDI)

Results

Type of instrumentation	Honggfuzz	None	QBDI
Speed (exec/s)	880	1566	130
Coverage	Yes	No	Yes
Sources needed ?	Yes	No	No

What is to come

In the next episode of: Fuzzing at Qb

- Peformance improvement
    - Binary fuzzing has a performance cost, we need to **get faster**

Results

Type of instrumentation	Honggfuzz	None	QBDI	QBDI + FS
Speed (exec/s)	880	1566	130	864
Coverage	Yes	No	Yes	Yes
Sources needed ?	Yes	No	No	No

What is to come

In the next episode of: Fuzzing at Qb

- Peformance improvement
  - Binary fuzzing has a performance cost, we need to **get faster**
- Symbolic Execution for vulnerability research, we need **to be smarter**
  - Integrate Triton in HF/QBDI to find hard to reach vulnerabilities
- Windows support
  - Windows support is unstable/experimental
- Infrastructure setup
  - Scale up our fuzzing potential

Conclusion and questions

What we learned from this journey

1. There are no perfect fuzzers
2. Benchmarking tools (especially fuzzers) is hard
3. R&D is never lost (From AFL to Honggfuzz)
4. Fuzzing on Windows is always a pain :)

Questions

- Thanks for listening!

quarkslab

SECURING EVERY BIT OF YOUR DATA