Fuzzing Techniques

Various ideas on surprising behaviour

Andreas Molzer (@HeroicKatora)

Who am I?

Hi, I'm Andreas

 @HeroicKatora on Github and Discord

Libraries: `oxide-auth`, `image`, `static-alloc`, `ethox`

What is fuzzing?

Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program.—Wikipedia

- As a library: `afl`, `hongfuzz`, `libfuzzer`.

```
1 afl::fuzz!(|data: &[u8]|{
2   let _ = fuzz_target(&data);
3 });
```

- As an integrated tool:
  `cargo install {cargo-fuzz,afl}`[0]

[0] Require `nightly` to run.

- `cargo fuzz init`: Setup a skeleton in `fuzz`
- In `fuzz/fuzz_targets/*.rs`: The test target binaries
- `cargo fuzz run <target>`: Compile and run a target
- In `fuzz/corpus/<target>`: The generated 'test cases'
- `cargo fuzz cmin <target>`: Compress to unique test cases

How does it work with decent efficiency?

1. Choose one of several weighted inputs
2. Mutate it randomly
3. Observe execution paths
4. Adjust weights, repeat until crash

Tip: fuzzing for 32-bit systems

Find pointer-width dependent bugs:

```
rustup target add i686-unknown-linux-gnu
cargo fuzz run --target i686-unknown-linux-gnu -s none <target>
```

Or build your own

A video series on building your own fuzzer by Gamozo Labs:

`https://gamozolabs.github.io/2020/07/12/fuzz_week_2020.html`

Includes: A RISC-V emulator to fuzz arbitrary binaries

Write fuzzer based unit tests with `QuickCheck` and `proptest`.

```
proptest! {
    #[test]
    fn numbers_are_valid_json(ref s in "[0-9]+") {
        parse_json(s).unwrap()
    }
}
```

Fails, because leading 0 is not allowed:

```
thread 'numbers_are_valid_json' panicked at [..]
        Error("invalid number", line: 1, column: 2);
        minimal failing input: ref s = "00"
        successes: 1
        local rejects: 0
        global rejects: 0
', src/main.rs:11:1
```

Property based testing

```
1 proptest! {
2     #[test]
3     fn numbers_are_valid_json(ref s in "[0-9]+") {
4         if s.len() <= 1 || !s.starts_with("0") {
5             parse_json(s).unwrap();
6         }
7     }
8 }
```

Can a `Vec` simulate a stack? Let's check.

```
1 use libfuzzer_sys::arbitrary::Arbitrary;
2
3 #[derive(Arbitrary, Debug)]
4 enum StackOp {
5     Push(Unit),
6     Pop,
7 }
```

Model checking

```
libfuzzer_sys::fuzz_target!(|ops: Vec<StackOp>| {
    let mut stack = vec![];
    let mut should_len = 0usize;
    for op in ops {
        match op {
            StackOp::Push(n) => {
                stack.push(n);
                should_len += 1;
            }
            StackOp::Pop => {
                let _ = stack.pop();
                should_len = should_len.saturating_sub(1);
            }
        }
    }
    assert_eq!(stack.len(), should_len);
});
```

Fuzzing as a refactoring tool

Imagine you've written a parser for `a(ba)*b`.

Is someone else's implementation of `(ab)+` equivalent?

In reality: A compression standard, an archive, etc.

Check that the old and new versions agree on output:

```
fuzz_target!(|data: &[u8]| {
    let old: Result<_, _> = reference::decode(data);
    let new: Result<_, _> = ours::decode(data);
    assert_eq!(old, new); // Panics on mismatch.
});
```

Check that new version is more generic than the old:

```rust
fuzz_target!(|data: &[u8]| {
    if let Ok(old) = reference::decode(data) {
        let new = ours::decode(data).unwrap();
        assert_eq!(old, new);
    } else {
        let _ = ours::decode(data);
    }
});
```

Found a bug but no (usable) regression test?

1. Use test case minimization to reduce test file size.
2. Synthesize a new one:

```
if let Ok(_) = catch_unwind(old_parser) {
    return; // Old parser must crash.
}
if let Err(_) = catch_unwind(new_parser) {
    return; // New parser mustn't.
}
panic!("Expected")
```

To remember

- Fuzzing is a technique, not a single tool.
- Energy is cheap, use the machine time you have.
- Contemporary octa-core CPUs go for $<$ €300