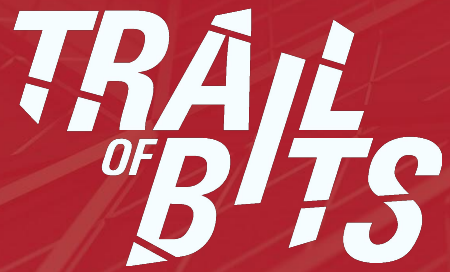

Smart Contract Program Analysis

whoami

- **Originally: security engineer at Trail of Bits**
  - Focuses: cryptanalysis, program analysis, mathy stuff
  - Went to school for math, liked RE
- **Suddenly: blockchain bubble!**
  - Wrote a fuzzer/property based testing framework (more later!)
  - Now audit smart contracts + dev tools
    - Still a good bit of the old stuff though

whatisthis

- **Domain info**
  - EVM
  - Solidity
- **Static Analysis**
- **Dynamic Analysis**
  - Fuzzing
  - Symbolic Execution
- **Questions**

Domain Info

Ethereum

- A state machine, in consensus
- Two types of entities
  - People
  - “Contracts”
- Either make “transactions”
- Contracts run EVM code when transacted with
- Thus, “World Computer”

EVM

The Ethereum Virtual Machine

- **Fairly ordinary Von Neumann machine**
- **Pay per operation**
- **Stack-based (ugh)**
- **Contract-local (but can see the whole chain)**
- **Storage and memory**
- **Has a notion of “reversion”**
  - This transaction is messed up, don’t let it change stuff
- **Terribly designed**
  - Example: higher memory addresses cost more to access

Interaction with the EVM

- Like hitting an API/ABI
- Defined set of function names and types
- Submit “ABI-encoded calldata” in a transaction
- This means no stdin
- Most patterns of calldata are meaningless
- Interaction looks like a web service, not a binary

Wanna know more?

- The Yellow paper is canonical
- The Jello paper might be better
- Best: download ethersplay and read some asm

Solidity

The solidity language

- **Solidity is the EVM lingua franca**
  - Other languages exist (barely)
- **It's terrible and beyond redemption**
  - Too many flaws for 45 min, let alone this section
- **Imagine javascript but less sensible**
- **Compiler has bizarre bugs somewhat frequently**
  - Also just bad as a compiler
  - Computes 1 as 0x100 ** 0 sometimes
- **Essentially every serious contract is written in this**

Seriously, it's bad

- `for (var i = 0; i < a.length; i ++) { a[i] = i; }`
  - Infinite loop if a is larger than 255 elements
  - i infers to byte, which is max 255
  - Overflows silently
  - Of course there's no warning here
  - i still takes up 256 bits, that's the minimum
- Stuff like this is everywhere in real code

Static Analysis

Right now

- **Currently, pretty easy**
- **Trail of Bits wrote a proprietary python script**
- **More or less a parser with grep bits**
  - Iterate over functions, variables
  - Look for patterns
  - Do two things have the same name?
  - Are there uninitialized variables?
- **No rich model, mostly heuristic**
- **Finds tons of IRL bugs**
- **Currently proprietary :(**

The future

- **Working on a lifter**
  - Convert stack machine -> SSA
  - Finally get something you can really reverse
- **Working on deeper binja integration**
  - Use existing IL for richer analysis
  - Also you can reverse faster
- **At some point, they might get a real compiler**
  - Clang-style warnings supersede most of this
  - Clang-tidy could probably do it all
  - Not holding my breath though

Dynamic Analysis

Dynamic Analysis: Surprisingly Easy

- Theory: solidity developers run code less
- You can't just execute, you must transact
- Ergo: running their code more finds bugs
- There exist debuggers
  - I like hevm from dapphub
- There exist a plethora of symbolic execution tools
- There exists exactly one fuzzer

Symbolic Execution

Symbolic execution: What?

- **“Executing every possible path at once”**
- **Different notion of value**
  - Normal execution: values are concrete (x = 5)
  - Symbolic execution: values can be symbolic (x > 3, x < 6, x != 4)
    - Values accumulate constraints during execution
- **Different notion of control flow**
  - Forking on a symbolic value executes both paths in parallel
  - Execution is a tree, not a line
- **Typically solve for eventual values with a SAT solver**

Applied to EVM

- **Transactions are symbolic**
  - Tons of symbolic data
  - Usually executors use heuristics
- **Very fragile (must totally reimplement EVM)**
- **Often heuristic-heavy**
  - Many (e.g. Oyente) assume all storage is symbolic
  - Leads to many false positives
- **Great degree of assurance though**
- **Next steps: reliability and usability by mortals**

Fuzzing

Fuzzing

- **EVM programs have no input!**
  - Traditional fuzzers generate/mutate input
  - On EVM, random input is nonsensical
- **Fuzzing needs an ABI and grammar**
- **EVM programs never segfault!**
  - Traditional fuzzers know when they've found a bug
  - On EVM, more specification needed
- **Thus, a fuzzer is just**
  - ABI parser/tool to specify ABI
  - Tool to generate random type occupants
    - Random strings, random addresses, random arrays of ints, etc.
  - Some bug specification
  - VM implementation

Notes from the real world

- **Almost no programs need fancy tools**
  - Wrote deep VM introspection
  - Wrote coverage guidance
  - Wrote fancy state machine models
  - None of it really mattered
- **You can share a fuzzer setup easily!**
  - If two programs share an ABI, they fuzz the same
  - Standardized ABI means massive scalability
- **Need more than just inputs for many real programs**
  - Temporal logic
  - Complex multi-contract setup
  - Many actors of different privilege levels
  - Contracts can change mid-fuzz!
  - All of these are next steps

Questions?

JP Smith

Security Engineer

jp@trailofbits.com, @japesinator

www.trailofbits.com