teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts

Johannes Krupp, Christian Rossow
In a nutshell
Ethereum

johannes.krupp@cispa.saarland
Smart Contracts

- Ethereum Virtual Machine (EVM) bytecode
- executed on incoming transaction
- otherwise like regular account:
  - address
  - balance
- use cases:
  - crowdfunding schemes
  - shared wallets
  - games
  - ...
Smart Contracts

- Ethereum Virtual Machine (EVM) bytecode
- executed on incoming transaction
- otherwise like regular account:
  - address
  - balance

may contain bugs

cannot be updated

goal: find & exploit bugs
Smart Contract Exploitation
Transactions

- **from** sender
- **to** recipient
- **value** transferred amount
  - may also be zero
- **gas** „transaction fee“
- **data** input data
  - may be empty
Ethereum Virtual Machine (EVM)

- stack machine
- 256 bit wordsize
- ~70 instructions
  - arithmetic
  - logic
  - control flow
  - blockchain interaction
Challenges

- control flow graph recovery
Control Flow Instructions

- **JUMP**
  - unconditional jump
  - jump to `target`

- **JUMPI**
  - conditional jump
  - jump to `target` if `test` is non-zero

- **JUMPDEST**
  - marks valid jump target
  - no op
CFG Recovery

600934600757005b565b00

JUMP  JUMPI  JUMPDEST
CFG Recovery

0  6009  PUSH1 09
2  34    CALLVALUE
3  6007  PUSH1 07
5  57    JUMPI
6  00    STOP
7  5b    JUMPDEST
8  56    JUMP
9  5b    JUMPDEST
a  00    STOP
CFG Recovery

0: PUSH1 09, CALLVALUE, PUSH1 07, JUMPI

6: STOP

7: JUMPDEST, JUMP

9: JUMPDEST, STOP
CFG Recovery

0. PUSH1 09
   CALLVALUE
   PUSH1 07
   JUMPI

6. STOP

7. JUMPDEST
   JUMP

9. JUMPDEST
   STOP
CFG Recovery

0  PUSH1 09
   CALLVALUE
   PUSH1 07
   JUMPI

6  STOP

7  JUMPDEST
   JUMP

9  JUMPDEST
   STOP
Challenges

- control flow graph recovery

how can we get money from a contract?
Ethereum Virtual Machine (EVM)

- stack machine
- 256 bit wordsize
- ~70 instructions
  - arithmetic
  - logic
  - control flow
  - blockchain interaction
Critical Instructions

- **CALL**
  - regular transaction
  - transfer `value` to `to`

- **SELFDESTRUCT**
  - contract destruction
  - transfer funds to `recipient`

- **CALLCODE / DELEGATECALL**
  - execute code of `target`
  - „code injection“

must execute one of these
1. locate critical instructions
2. compute backward slices of argument(s)
3. filter for attacker controlled slices
1. locate critical instructions
2. compute backward slices of argument(s)
3. filter for attacker controlled slices
1. locate critical instructions
2. compute backward slices of argument(s)
3. filter for attacker controlled slices
4. generate path through a slice
Exploit Generation – General Approach

1. locate critical instructions
2. compute backward slices of argument(s)
3. filter for attacker controlled slices
4. generate path through a slice
5. execute path symbolically
   - collect path constraints
6. use constraint solver
   - unsatisfiable: generate next path
   - satisfiable: done

\[ x > 0 \]
\[ y > 0 \]
Challenges

- control flow graph recovery
- contract state
- SHA3 instruction
contract Stateful{
  bool vulnerable = false;

  function exploit(address attacker){
    require(vulnerable);
    attacker.transfer(this.balance);
  }

  function makeVulnerable(){
    vulnerable = true;
  }
}

state at bytecode level?
EVM – Memory Model

- **Stack**
  - stack
  - 256 bit words
  - volatile

- **Memory**
  - array
  - byte-addressable
  - volatile

- **Storage**
  - map/dictionary
  - 256 bit keys, 256 bit values
  - persistent

state change = storage change
Storage Instructions

- **SLOAD**
  - load value for **key**

- **SSSTORE**
  - store **value** at **key**
1. locate **SSTORE** instructions
2. compute backward slices of argument(s)
3. generate path through a slice
4. execute path symbolically
   - collect path constraints $C$
   - collect storage reads $R$ & writes $W$

$$C = \{x \leq 0\}$$

$$R = \emptyset$$

$$W = \{k\}$$
Path Stitching

- combine $n$ state changing paths + 1 critical path

\[
\begin{align*}
\mathcal{C} &= \{x \leq 0\} \\
\mathcal{R} &= \emptyset \\
\mathcal{W} &= \{k\}
\end{align*}
\]

\[
\begin{align*}
\mathcal{C} &= \mathcal{C}^* \cup \mathcal{C}^* \\
&= \{x_0 \leq 0, x_1 > 0, y_1 > 0\}
\end{align*}
\]

\[
\begin{align*}
\mathcal{R} &= \mathcal{R} \setminus \mathcal{W} \cup \mathcal{R} = \emptyset \\
\mathcal{W} &= \mathcal{W} \cup \mathcal{W} = \{k\}
\end{align*}
\]
Challenges

- control flow graph recovery
- contract state
- SHA3 instruction
## SHA3 Instruction

- **SHA3**
  - compute Keccak-256 hash over 
    \[ \text{memory[offset : offset + len]} \]
  - used to implement Solidity’s mapping type

```solidity
function check(bytes32 data, bytes32 check){
    require(data == "1337" && sha3(data) == check)
    //...
}
```

\[ C = \{ \text{data = "1337"}, \text{sha3(data) = check}\} \]

How to solve \( \text{sha3(data) = check} \)?
SHA3 Instruction

\[ C = \{ \text{data} = "1337", \text{sha3(data)} = \text{check} \} \]

1. remove dependent constraints
2. solve reduced set
3. compute hash values
4. replace dependent constraints
5. repeat

\[ C' = \{ \text{data} = "1337" \} \]

\[ \text{sha3(data)} \rightarrow 0x985d.. \]

\[ C' = \{ \text{data} = "1337", 0x985d.. = \text{check} \} \]
Challenges

- control flow graph recovery
- contract state
- SHA3 instruction
teEther

CFG recovery → critical instructions → path generation

constraint generation → path stitching → exploit generation
Evaluation

- contracts from blockchain
- 784,344 total
- 38,757 unique
- 30 min CFG recovery + 30 min exploit generation

![Graph showing percentages of critical paths and exploits for different blockchain functions.](attachment:image.png)
Evaluation - Exploits

- contracts from blockchain
- 784,344 total
- 38,757 unique
- 30 min CFG recovery + 30 min exploit generation
- 630 unique exploits
- 1,731 affected contracts
- 1,769 total exploits
Validation

- local test network
- three accounts:
  - target contract
  - attacker
  - "shellcode" contract
- two step validation:
  - update exploit to reflect target storage
  - replay exploit

<table>
<thead>
<tr>
<th>Function</th>
<th>Successful</th>
<th>Failed Update</th>
<th>Failed Exploit</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>1301</td>
<td>84</td>
<td>85</td>
</tr>
<tr>
<td>CALLCODE</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>DELEGATECALL</td>
<td>7</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SELFDESTRUCT</td>
<td>255</td>
<td>28</td>
<td>6</td>
</tr>
</tbody>
</table>
Validation

- local test network
- three accounts:
  - target contract
  - attacker
  - "shellcode" contract
- two steps:
  - update exploit to reflect target storage
  - replay exploit

What are the vulnerabilities?

<table>
<thead>
<tr>
<th>Function</th>
<th>Successful</th>
<th>Failed Update</th>
<th>Failed Exploit</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>1301</td>
<td>84</td>
<td>85</td>
</tr>
<tr>
<td>CALLCODE</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DELEGATECALL</td>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SELFDESTRUCT</td>
<td>255</td>
<td>28</td>
<td>6</td>
</tr>
</tbody>
</table>

Successful, Failed Update, Failed Exploit
Vulnerabilities

- reverse engineering infeasible
- source code unavailable
- OSINT: „publish & verify“ on etherscan.io
- manual analysis
Vulnerabilities

- logic bugs

```solidity
modifier onlyowner() {
    require(msg.sender != owner);
    _;
}
```
Vulnerabilities

- logic bugs
- semantic confusion

- msg.value  value of current transaction
- this.balance  balance of account
Vulnerabilities

- logic bugs
- semantic confusion
- visibility errors
Vulnerabilities – Visibility

contract Bet{
    function play() {
        if(bet1 > bet2){
            win(player1);
        } else if(bet2 > bet1){
            win(player2);
        } else{
            draw(player1, player2);
        }
    }
}
Vulnerabilities – Visibility

contract Bet{

    ...

    function win(address winner) internal {
        winner.transfer(AMOUNT_WIN);
    }

    function draw(address player1, address player2) {
        player1.transfer(AMOUNT_DRAW);
        player2.transfer(AMOUNT_DRAW);
    }

    }

    default visibility: public

    call draw(attacker, attacker)
Vulnerabilities

- logic bugs
- semantic confusion
- visibility errors
- constructor errors
Vulnerabilities – Constructor

contract Owner{
    function Owned() {
        owner = msg.sender;
    }
    ...
}

- constructor
- executed only once
- msg.sender = contract creator

contract Owned{
    function owned() {
        owner = msg.sender;
    }
    ...
}

- regular function
- can be called by anyone
- msg.sender = anyone
Vulnerabilities

- logic bugs
- semantic confusion
- visibility errors
- constructor errors

Solidity partially at fault
**Conclusion**

**Exploit Generation – General Approach**

1. locate critical instructions
2. compute backward slices of argument(s)
3. filter for attacker controlled slices
4. generate path through a slice
5. execute path symbolically
   - collect path constraints
6. use controlled values
   - `update()`
   - `set()`
   - `satisfy`

**Validation**

- local test network
- three accounts:
  - target contract
  - attacker
  - "willcode" contract
- two step validation:
  - update exploit to reflect target storage
  - replay exploit

---

@KruppJohannes

**teEther will be open sourced!**

---

johannes.krupp@cispa.saarland