Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited

Daniel Perez and Benjamin Livshits, Imperial College London

USENIX Security'21, August 2021
Ethereum Smart Contracts

- Programs deployed on the Ethereum blockchain
- Usually written in Solidity, compiled into EVM bytecode
- Can transfer money to other addresses (including contracts)
- Each instruction execution consumes gas
Smart Contracts: What could go wrong?

TheDAO hack (2016)

- TheDAO raised \(~$150M\) in ICO
- Soon after, it got hacked \(~$50M\)
- Price of Ether halved
- Ethereum community decided to hard-fork
- Attacker used a re-entrancy vulnerability

Parity Wallet bug (2017)

- Parity wallet library was used to manage multisig wallet contracts
- Parity wallet has been removed due to a "bug"
- Dependent contracts became unable to send funds
- Around \($280M\) frozen
Common vulnerabilities / bugs

- **Reentrancy** can allow an attacker to drain funds
- **Unhandled exceptions**: can result in lost funds
- **Dependency on destructed contract**: can result in locked funds
- **Transaction order dependency**: can allow an attacker to manipulate prices
- **Integer overflow** can result in locked fund
- **Unrestricted action** can result in an attacker stealing or locking funds
Smart Contract analysis tools

- Usually static analysis and/or symbolic execution
- Work either on Solidity or on the EVM bytecode
- Check for known vulnerabilities/patterns

Securify web interface
https://securify.chainsecurity.com
Smart Contract exploitation

According to analysis tools

- Thousands of vulnerable contracts
- Hundreds of millions USD at risk

⇒ We analyze the Ethereum blockchain to look for actual exploitation

In the real-world

- TheDAO and Parity wallet bug were impressive
- Otherwise, a few attacks here and there
**Dataset**

- Data received from the paper authors
- Received replies from 5 out of 8 of the authors we contacted
- Ether at stake computed at time of the report
- Total of around **3M ETH at stake**

<table>
<thead>
<tr>
<th>Name</th>
<th>Contracts analyzed</th>
<th>Vulnerabilities found</th>
<th>Ether at stake</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oyente</td>
<td>19,366</td>
<td>7,527</td>
<td>1,289,177</td>
</tr>
<tr>
<td>Zeus</td>
<td>1,120</td>
<td>855</td>
<td>729,376</td>
</tr>
<tr>
<td>Maian</td>
<td>NA</td>
<td>2,691</td>
<td>14.13</td>
</tr>
<tr>
<td>Securify</td>
<td>29,694</td>
<td>9,185</td>
<td>719,567</td>
</tr>
<tr>
<td>MadMax</td>
<td>91,800</td>
<td>6,039</td>
<td>1,114,692</td>
</tr>
</tbody>
</table>

Dataset overview
Detection overview

1. Retrieve all transactions
2. Retrieve execution traces for all transactions
3. Encode execution traces to Datalog
4. Query Datalog for vulnerabilities
Checking for re-entrancy

1. Record all calls

2. Check for mutually recursive calls between contracts

Datalog rules

call_flow(A, B) :- call(A, B).
call_flow(A, B) :- call(A, C), call_flow(C, B).
reentrant_call(A, B) :- call_flow(A, B),
    call_flow(B, A),
    A != B.

Vulnerable contract

```plaintext
function payMe(address account) public {  
    uint amount = getAmount(account);  
    // XXX: vulnerable  
    if (!account.send(amount))  
        throw;  
    balance[account] -= amount;  
}
```

Attacker contract

```plaintext
function () {
    victim.payMe(owner);
}
```
Checking for unhandled exceptions

1. Record all call results
   - Top value on the stack after a call

2. Check if the return value is used in a condition (JUMPI)

Datalog rules

```
influences_condition(A) :- used_in_condition(A).
influences_condition(A) :-
    depends(B, A), used_in_condition(B).

unhandled_exception(A) :-
    failed_call(A), not influences_condition(A).
```

Problematic contract

```cpp
// allows user to withdraw funds
function withdraw(address account) public {
    uint amount = getAmount(account);
    balance[account] -= amount;
    account.send(amount); // could silently fail
}
```
Checking for integer overflow

1. Infer variable types from bytecode
   - `SIGNEXTEND` means the variable is signed
   - `AND n 0xff` means it is a `uint8`

2. Compare typed and untyped results

   The approach can lead to some false-positives

```solidity
function overflow(uint fee) {
    uint amount = 100;
    amount -= fee;
    msg.sender.send(amount);
}
```

Contract vulnerable to integer overflow
## Detection results

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Vulnerable contracts</th>
<th>Total Ether at Stake</th>
<th>Contracts exploited</th>
<th>Exploited Ether</th>
<th>% of Exploited Ether</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re-entrancy</td>
<td>4,337</td>
<td>1,518,067</td>
<td>116</td>
<td>6,076</td>
<td>0.40%</td>
</tr>
<tr>
<td>Unhandled Exception</td>
<td>11,427</td>
<td>419,418</td>
<td>264</td>
<td>271.9</td>
<td>0.07%</td>
</tr>
<tr>
<td>Destructed contract</td>
<td>7,285</td>
<td>1,416,086</td>
<td>0</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Transaction order</td>
<td>1,881</td>
<td>302,679</td>
<td>54</td>
<td>297.2</td>
<td>0.01%</td>
</tr>
<tr>
<td>Integer overflow</td>
<td>2,492</td>
<td>602,980</td>
<td>62</td>
<td>1,842</td>
<td>0.31%</td>
</tr>
<tr>
<td>Unrestricted action</td>
<td>5,163</td>
<td>580,927</td>
<td>42</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>23,327</td>
<td>3,124,433</td>
<td>463</td>
<td>8,487</td>
<td>0.27%</td>
</tr>
</tbody>
</table>

In total, **at most 0.27%** of the 3M Ether at stake has been *exploited*.
Takeaways

- **Re-entrancy** seems to be the *most dangerous* issue. Good target for static analysis. Better prevention at the EVM level could be helpful.

- **Unhandled exceptions** are *handled well enough* by current tooling.

- **Dependency on destructed contract** is mostly a *non-issue*.

- **Transactions order dependency** does *not* seem to be *used to steal ETH* directly.

- **Integer overflow** is hard to prevent and consequences *hard to predict*. Also good target for static analysis.
Contracts wealth distribution

• Combined value: ~2 million ETH

• Only ~2,000 out of 23,327 contracts held Ether

• Less than 100 contracts had more than 10 Ether

• The top 6 contracts held 83% of the total Ether

Cumulative Ether held in contracts holding more than 10 ETH
Vulnerable but not exploitable

- Many cases where "vulnerable" contracts are not exploitable
- High-value contracts flagged vulnerable are typically not exploitable
- Usually limitation due to the nature of static analysis

```solidity
function removeOwner(address owner) onlyWallet {
    isOwner[owner] = false;
    // Could in Theory run out of gas
    for (uint i=0; i<owners.length - 1; i++) {
        if (owners[i] == owner) {
            owners[i] = owners[owners.length - 1];
            break;
        }
    }
    // a bit more logic
}
```

Multisig wallet with >350K ETH at address
0x7da82C7AB4771ff031b66538D2fB9b0B047f6CF9
Summary

- Analyzed 23k contracts with 3M Ether at risk
- At most 0.27% of this Ether, less than 10k ETH, was exploited
- Overall, high-value contracts seem to be secure
Thank you

For questions, please find my contact info at https://daniel.perez.sh