The Art of The Scam
Demystifying Honeypots in Ethereum Smart Contracts

Christof Ferreira Torres, Mathis Steichen and Radu State
Ethereum
Crash Course
Ethereum Blockchain
Ethereum Accounts

Externally Owned Account
- Address
- Balance

Contract Account
- Address
- Balance
- Code
- Storage
Ethereum Smart Contracts

Developer

Solidity Compiler

Smart Contract

```solidity
pragma solidity ^0.5.11;

contract HelloWorld {
    function hello() public pure returns(string memory) {
        return 'Hello World!';
    }
}
```
Ethereum Virtual Machine

- Turing complete
- Register-less, 256-bit, stack-based VM

- Over 100 instructions:
  - Stack instructions: PUSH, SWAP, ...
  - Arithmetic instructions: ADD, SUB, MUL, ...
  - Memory instructions: SLOAD, SSTORE, ...
  - Control-flow instructions: JUMP, JUMPI, ...
  - Contract instructions: CALL, SELFDESTRUCT, ...
  - Error handling instructions: REVERT, INVALID, ...
Exploiting
Smart Contracts
Attacks on Smart Contracts

A hacker stole $31M of Ether — how it happened, and what it means for Ethereum

THIS WAS REAL MONEY, A QUARTER OF A BILLION DOLLARS, THEIR MONEY, MEANT TO BUILD A BETTER VERSION OF THE WORLD, AND EVERY CENT WAS AT RISK

A $50 MILLION HACK JUST SHOWN THAT THE DAO WAS ALL TOO HUMAN
TEETHER: Gnawing at Ethereum to Automatically Exploit Smart Contracts

Johannes Krupp
CISPA, Saarland University,
Saarland Informatics Campus

Christian Rossow
CISPA, Saarland University,
Saarland Informatics Campus

Abstract
Cryptocurrencies like Bitcoin not only provide a decentralized currency, but also provide a programmatic way to process transactions. Ethereum, the second largest cryptocurrency next to Bitcoin, is the first to provide a Turing-complete language to specify transaction processing, thereby enabling so-called smart contracts. This provides an opportune setting for attackers, as security vulnerabilities are tightly intertwined with financial gain. In this paper, we consider the problem of automatic vulnerability identification and exploit generation for smart contracts. We develop a generic definition of vulnerable contracts and use this to build TEETHER, a tool that allows creating an exploit for a contract given only its binary bytecode. We perform a large-scale analysis of all 38,757 unique Ethereum contracts, 815 out of which our tool finds working exploits for—completely automated.

Ethereum heavily extends the way consensus protocols handle transactions: While Bitcoin allows to specify simple checks that are to be performed when processing a transaction, Ethereum allows these rules to be specified in a Turing-complete language. This makes Ethereum the number one platform for so-called smart contracts.

A smart contract can be seen quite literally as a contract that has been formalized in code. As such, smart contracts can for example be used to implement fundraising schemes that automatically refund contributions unless a certain amount is raised in a given time, or shared wallets that require transactions to be approved of by
Limitations of a Reactive Approach

- Attackers are required to scan millions of smart contracts to find bugs.
- Finding exploitable bugs in smart contracts is becoming more challenging.

“Why should I spend time looking for victims, if I can just let the victims come to me?”
Smart Contract Honeypots
What are Smart Contract Honeypots?

- Smart contracts that look vulnerable but actually are not.

- **Idea:**

  Make users believe that they can exploit a smart contract by sending funds to it, while in reality only the smart contract creator is able to retrieve them.
Multiplicator Honeypot

```
5   contract MultiplicatorX3
6   {
7       address public Owner = msg.sender;
8   
9       function() public payable{
10          
11       function withdraw() payable public
12          {
13              require(msg.sender == Owner);
14              Owner.transfer(this.balance);
15          }
16      }
17   
18      function Command(address adr,bytes data)
19          payable public
20          {
21              require(msg.sender == Owner);
22              adr.call.value(msg.value)(data);
23          }
24      
25      function multiply(address adr)
26          public payable
27          {
28              if(msg.value>=this.balance)
29              {
30                  adr.transfer(this.balance+msg.value);
31              }
32          }
33      }
34   
35   }  
36   
```

!! Balance = Previous Balance + Transaction Value !!
Multiplicator Honeypot

Sponsored: W3 - 1st place at World Blockchain Summit

Contract Overview
Balance:
Ether Value:
Transactions:

Latest 4 txns

<table>
<thead>
<tr>
<th>TxHash</th>
<th>Block</th>
<th>Age</th>
<th>From</th>
<th>To</th>
<th>Value</th>
<th>[TxFee]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbd49300ebc18953d0...</td>
<td>4750052</td>
<td>331 days 18 hrs ago</td>
<td>0xb43df878996c90...</td>
<td>0x5a88d2901c68f...</td>
<td>0 Ether</td>
<td>0.000271362</td>
</tr>
<tr>
<td>0x7d3050438741eb...</td>
<td>4748394</td>
<td>331 days 21 hrs ago</td>
<td>0x22d0135701299...</td>
<td>0x5a88d2901c68f...</td>
<td>1.1 Ether</td>
<td>0.0011633</td>
</tr>
<tr>
<td>0x4e4eb3605782e5...</td>
<td>4569381</td>
<td>362 days 2 hrs ago</td>
<td>0xb3df878996c90...</td>
<td>0x5a88d2901c68f...</td>
<td>1 Ether</td>
<td>0.0000002104</td>
</tr>
<tr>
<td>0x585e1b933ab27...</td>
<td>4569361</td>
<td>362 days 2 hrs ago</td>
<td>0x4b3df878996c90...</td>
<td>Contract Creation</td>
<td>0 Ether</td>
<td>0.0000034583</td>
</tr>
</tbody>
</table>

SENT 1 ETH TO THIS CONTRACT TO GET 2 ETH

AAAAAAAAAND IT'S GONE
CryptoRoulette Honeypot

```solidity
contract CryptoRoulette {
    uint256 private secretNumber;
    uint256 public lastPlayed;
    uint256 public betPrice = 0.1 ether;
    address public ownerAddr;

    struct Game {
        address player;
        uint256 number;
    }
    Game[] public gamesPlayed;

    function CryptoRoulette() public {
        ownerAddr = msg.sender;
        shuffle();
    }

    function shuffle() internal {
        secretNumber = uint8(sha3(now, block.blockhash(block.number-1))) % 20 + 1;
    }

    function play(uint256 number) payable public {
        require(msg.value >= betPrice && number <= 20);
        game = Game(msg.sender, number);
        game.player = msg.sender; // this line
        game.number = number;
        gamesPlayed.push(game);
        if (number == secretNumber) {
            // win!
            msg.sender.transfer(this.balance);
        }
        shuffle();
        lastPlayed = now;
    }

    function kill() public {
        if (msg.sender == ownerAddr && now > lastPlayed + 1 days) {
            suicide(msg.sender);
        }
    }
}
```
Honeypot Phases

1) Deployment
2) Exploitation
3) Withdrawal
Why Do Honeypots Work?

- People actively look for exploitable smart contracts.
- Complexity of the Ethereum ecosystem.
Detecting Honeypots
Taxonomy of Honeypots

- Collected 24 honeypot smart contracts from public sources on the Internet.
- Extracted 8 different techniques, each exploiting a feature ("bug") on a particular level of Ethereum.

<table>
<thead>
<tr>
<th>Level</th>
<th>Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ethereum Virtual Machine</td>
<td>Balance Disorder</td>
</tr>
<tr>
<td></td>
<td>Inheritance Disorder</td>
</tr>
<tr>
<td>Solidity Compiler</td>
<td>Skip Empty String Literal</td>
</tr>
<tr>
<td></td>
<td>Type Deduction Overflow</td>
</tr>
<tr>
<td></td>
<td>Uninitialised Struct</td>
</tr>
<tr>
<td>Etherscan Blockchain Explorer</td>
<td>Hidden State Update</td>
</tr>
<tr>
<td></td>
<td>Hidden Transfer</td>
</tr>
<tr>
<td></td>
<td>Straw Man Contract</td>
</tr>
</tbody>
</table>

Table 1: A taxonomy of honeypot techniques in Ethereum smart contracts.
HoneyBadger Design


Z3 Bit-Vector Solver

Report

Bytecode
Symbolic Analysis

- Based on Luu et al.’s symbolic execution engine Oyente [CCS ‘16].
- Constructs control flow graph and executes every instruction symbolically.
- Our symbolic execution does not ignore infeasible paths.
- Collects meta information about:
  - Storage writes $S$
  - Infeasible basic blocks $IB$
  - Arithmetic operations $A$
  - Execution paths $P$
  - Feasible basic blocks $FB$
  - Contract calls $C$
- Discard contracts that cannot receive and transfer funds.

- **Receiving** funds:
  - \( \exists \ p \in P : \text{REVERT} \notin p \)
  - We use Z3 to verify that \( l_v > 0 \) is satisfiable under \( p \).

- **Transferring** funds:
  - Explicitly (e.g. *transfer*):
    \( \exists \ c \in C : c_v > 0 \lor c_v \text{ is symbolic} \)
  - Implicitly (i.e. *selfdestruct*):
    \( \exists \ p \in P : \text{SELFDESTRUCT} \in p \)
- Consists of several sub-components.

- Each sub-component is responsible for the detection of a particular technique.

- Honeypot techniques are detected via simple heuristics:
  - Ex.: Balance Disorder

$$\exists c \in C : c \in IB \land c_v = \sigma[i_a]_b + I_v$$

```solidity
1 contract MultiplicatorX3 {
2     ...
3     function multiply(address adr) payable {
4         if (msg.value >= this.balance)
5             adr.transfer(this.balance + msg.value);
6     }
7 }
```
Evaluation
We crawled 2,019,434 contracts from August 7, 2015 to October 12, 2018.

151,935 contracts are unique in terms of bytecode (7.52%).

We run HoneyBadger on the set of unique smart contracts.
- 48,487 contracts have been identified as cash flow contracts (32%).
- Our tool detected 460 unique honeypots (690 on the 2 million).
- Analysis took about 2 minutes per contract (91% code coverage).
Manual inspection of the source code for the 460 flagged contracts.

We managed to collect the source code for 323 contracts (70% of 460).

Validation shows that **282** contracts are true positives (87% precision).

<table>
<thead>
<tr>
<th>Technique</th>
<th>TP</th>
<th>FP</th>
<th>Balance Disorder</th>
<th>Inheritance Disorder</th>
<th>Skip Empty String Literal</th>
<th>Type Deduction Overflow</th>
<th>Uninitialised Struct</th>
<th>Hidden State Update</th>
<th>Hidden Transfer</th>
<th>Straw Man Contract</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20</td>
<td>0</td>
<td>41</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2: Number of true positives (TP), false positives (FP) and precision $p$ (in %) per detected honeypot technique for contracts with source code.
Honeypot Insights
Methodology

- Analyzed all transactions of the 282 true positives.

- Used simple heuristics to label addresses as:
  - **Attacker:**
    1) created the contract.
    2) first to send funds to the contract.
    3) received more funds than actually spent.
  - **Victim:** not labeled as attacker and spent more funds than actually received.

- Used this to label honeypots as:
  - **Successful:** a victim has been detected.
  - **Aborted:** the balance is zero and no victim has been detected.
  - **Active:** the balance is larger than zero and no victim has been detected.
- 71% manage to trap only one victim.
  - Users potentially look at transactions.
- Majority are successful within the first 24 hours.
  - Users quickly attempt to exploit honeypots.
- Bytecode of honeypots is vastly different even within the same technique.
- Signature-based detection methods are rather ineffective.

Table 3: Bytecode similarity (in %) per honeypot technique.
First deployment in January 2017 with highest activity in February 2018.

Honeypots are an emerging and increasing trend.
A total profit of 257.25 ether has been made through honeypots.

An accumulated profit of $90,118 at the time of withdrawal.

<table>
<thead>
<tr>
<th></th>
<th>Min.</th>
<th>Max.</th>
<th>Mean</th>
<th>Mode</th>
<th>Median</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>BD</td>
<td>0.01</td>
<td>1.13</td>
<td>0.5</td>
<td>0.11</td>
<td>0.11</td>
<td>3.5</td>
</tr>
<tr>
<td>ID</td>
<td>0.004</td>
<td>6.41</td>
<td>1.06</td>
<td>0.1</td>
<td>0.33</td>
<td>17.02</td>
</tr>
<tr>
<td>SESL</td>
<td>0.584</td>
<td>4.24</td>
<td>1.59</td>
<td>1.0</td>
<td>1.23</td>
<td>9.57</td>
</tr>
<tr>
<td>TDO</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>US</td>
<td>0.009</td>
<td>1.1</td>
<td>0.46</td>
<td>0.1</td>
<td>0.38</td>
<td>6.44</td>
</tr>
<tr>
<td>HSU</td>
<td>0.00002</td>
<td>11.96</td>
<td>1.44</td>
<td>0.1</td>
<td>1.02</td>
<td>171.22</td>
</tr>
<tr>
<td>HT</td>
<td>1.009</td>
<td>1.1</td>
<td>1.05</td>
<td>1.0</td>
<td>1.05</td>
<td>2.11</td>
</tr>
<tr>
<td>SMC</td>
<td>0.399</td>
<td>4.94</td>
<td>1.76</td>
<td>2.0</td>
<td>1.99</td>
<td>47.39</td>
</tr>
<tr>
<td>Overall</td>
<td>0.00002</td>
<td>11.96</td>
<td>1.35</td>
<td>1.0</td>
<td>1.01</td>
<td>257.25</td>
</tr>
</tbody>
</table>

Table 4: Statistics on the profitability of each honeypot technique in ether.
Conclusion
Honeypots are an emerging new type of fraud and requires further investigation.

We propose a taxonomy and a tool called *HoneyBadger*, that detects honeypots at a large scale.

We identified 690 honeypots with a precision of 87%.

We provide interesting insights: 240 victims and $90,000 profit.
All **code & data** is available on GitHub:

https://github.com/christoftorres/HoneyBadger

More information at:

https://honeybadger.uni.lu

Supported by: