Ethereum Smart Contract Security Analysis

Kazi Alom, Lay Jain, Srijon Mukherjee

The Ethereum blockchain offers smart contracts as a way of handling financial promises. These are implemented as Turing-complete programs allowing a smart contract creator to execute arbitrary code in handling transactions. These smart contracts are available for public viewing and are susceptible to being exploited. We aim to do a security analysis on common smart contracts and explore what kinds of critical vulnerabilities may exist.

***Index Terms*—security analysis, ethereum, smart contract**

I. Foundations

Blockchain technology's greatest strength is allowing the decentralization of historically centralized services. Bitcoin [4] has a market cap of almost $200 billion and sees a daily trading volume of about $100 billion. Besides Bitcoin, there are countless other "alt-coins" that each have their own platform and offer a type of service. Ethereum [1] is one platform that allows digital assets be directly controlled by a piece of code called a smart contract. Smart contracts essentially allow systems that automatically move digital assets according to pre-defined rules. Decentralized autonomous organizations (DAOs) are long-term smart contracts that are controlled by shareholders. The DAO [3], a crowdfunded venture capital fund that existed as a smart contract, had over $50 million stolen due to an exploited bug in the smart contract language Solidity. In fact, this incident is the reason why Ethereum was hard-forked and why we see another alt coin "Ethereum Classic". Other high profile smart contracts have also been targeted and lost their funds to these types of attacks. As the usage of smart contracts becomes more prevalent, it is critical to address bugs in smart contract implementations and in Solidity itself.

Smart contract vulnerabilities can be classified [6] as blockchain vulnerabilities, Solidity vulnerabilities, and software security vulnerabilities. The Transaction Ordering Dependency problem is one blockchain vulnerability which involves a new block on the chain containing multiple transactions invoking the same contract. There is no certainty in the state of the contract when either individual transaction invokes the contract because the order is not known until after the block is mined. The Timestamp dependency problem says that a smart contract using timestamps of blocks can potentially be manipulated by malicious miners who modify their local system's time.

Solidity compiler vulnerabilities are bugs within the smart contract's high level language and how it generates the Ethereum Virtual Machine (EVM) bytecode. There have not been many CVE's published on this which makes the EVM an excellent candidate for fuzzing.

Software security vulnerabilities are those due to erroneus smart contract code and improper Solidity practice. The DAO attack was victim to a re-entrancing attack where the contract uses an unsafe function call.value() that allows the attacker to take advantage of a callback to recursively withdraw funds from a victim contract. Other usual security vulnerabilities such as buffer overruns and integer over/underflows also fall under software bugs.

The parity multisig attack was one incident where $50 million of Ethereum was frozen due to an attacker calling an unprotected function and gaining ownership of a public smart contract library that was imported by other contracts. The attacker invoked the kill function that removed the contract from the blockchain. This could have been mitigated by using a private modifier on the function that allowed the attacker to escalate privileges.

Ethereum smart contracts rely on "gas", which is additional transaction fee paid to the miner for borrowing their computation ability on the blockchain. If this gas runs out before an operation is over, the callee contract throws an exception and these must be properly checked by the caller contract. This applys to all other exceptions that can be raised such as when exceeding stack capacity, or any other unknown system error occurs. These also fall under software security vulnerabilities that can be mitigated by safe code practice.

II. Common Vulnerabilities

As mentioned, there exist a variety of smart contract weaknesses [7] due to software errors, EVM exceptions, blockchain errors, etc. It helps to look at some Solidity code examples and understand how exactly the exploits would be triggered. The following samples demonstrate vulnerabilities besides crypto implementation flaws, obsolete function usages, and typical programming malpractice.

Public access to critical private data: When a contract uses the `private` modifier on a function or field, it does not mean that these variables cannot be read. Any attacker can look at the transactions related to this contract on the public blockchain to figure out the state of all variables. In the following example, an attacker can simply calculate what number they need to trigger `selectWinner` into transferring the contract balance to his address by first observing the value at `numbers[0]`. This would be seen the arguments to `play` in a previous transaction. One way to resolve this for a "game" like

the contract below is to use a commitment scheme that hides values using hashes until both players have picked numbers.

```
contract Test{
 uint[2] private numbers;
 address[2] private addrs;
 uint idx = 0;

 function play(uint number) public payable {
     require(msg.value == 1 ether,
         'must be called with 1 ether');
     numbers[idx] = number;
     addrs[idx] = msg.sender;
     idx++;
     if (idx == 2) selectWinner();
 }

 function selectWinner() private {
     uint n = (numbers[0] * numbers[1]) % 2;
     (bool success,) =
      addrs[n].call.value(address(this).balance);
     require(success, "transfer failed");
     delete numbers;
     delete addrs;
     idx = 0;
 }
}
```

DoS by uncontrolled gas consumption: All smart contracts consume gas depending on how many instructions must be computed based on what function was called. This gas price and amount is specified in the transaction sent to the contract. Miners are incentivized by higher gas prices thus transactions with high gas prices are usually completed quicker. In Ethereum, the sum of all transactions in a block cannot exceed a gas limit threshold. If a contract like the one below maintains an array of unbound size, it can lead to a denial-of-service condition where the function that loops across the array's values may exceed the block gas limit. This means that it is critical that developers do not loop over arrays that are expected to grow over time.

```
contract Test {
    address[] addr_list;

    function addAddress() public {
        addr_list.push(msg.sender);
    }

    function transferAddresses() public payable {
        for(uint i=0; i < addr_list.length; i++) {
            // doing ANYTHING is a DOS risk
        }
    }
}
```

Arbitrary data write: Similar to how in a standard buffer overflow an attacker can overwrite critical data such as a function return address, there exists a similar problem where anyone may overwrite the owner of a contract. This would then allow them to call privileged functions such as a withdrawal that validates the sender address of a transaction against the contract owner. Developers should ensure that there are no out-of-bounds accesses where writes to one data structure might corrupt another data structure in the address space.

```
contract Test {
    address public owner;
    uint256[] map;

    function Test() public {
        owner = msg.sender;
    }

    function set(uint256 key, uint256 value) public {
        map[key] = value;
    }

    function get(uint256 key)
        public view returns (uint256) {
        return map[key];
    }

    function withdraw() public {
        require(msg.sender == owner);
        msg.sender.transfer(address(this).balance);
    }
}
```

Insufficient entropy for random values: One popular use of smart contracts may be in gambling applications, which typically rely on pseudorandom number generators to pick winners. Due to the state of a contract being public on the blockchain, having a source of randomness is non-trivial. For instance, relying on the timestamp as a random value does not work because a miner can arbitrarily set their local machine times. One way to solve this is to use a commitment scheme to commit a guess and answer before revealing.

```
contract Test {
  uint8 answer;

  function init_challenge() public payable {
      require(msg.value == 1 ether);
      // hash(previous blocks hash | timestamp)
      answer = uint8(keccak256(
          block.blockhash(block.number - 1), now));
  }

  function guess(uint8 g) public payable{
      require(msg.value == 1 ether);

      if (g == answer) {
          msg.sender.transfer(address(this).balance);
      }
  }
}
```

Race condition: By nature of being on a blockchain, some transactions that call the same contract may occur in the same block and thus their ordering depends on the miner who completes it. One simple example is if there was a contract that awarded tokens to the first person to guess a correct value. Let's say you figured out the answer and send your transaction onto the network. In the time

it takes for the block to get mined, an adversary sees your transaction and copies the answer you found and sends their own transaction, except they provide a much higher gas value. Rational miners would complete the adversary's transaction before yours, letting the adversary win the money instead. To mitigate this situation, the contract can store the salted hash of the answer and then compare it against the salted hash of your guess.

```
contract Test {
    address public owner;
    uint public prize;
    bool finished;

    function Test() public {
        owner = msg.sender;
        finished = false;
    }

    function setPrize() public payable {
        require(!finished);
        require(msg.sender == owner);
        owner.transfer(prize);
        prize = msg.value;
    }

    function claimPrize(uint guess) {
        require(!finished);
        require(guess == keccak256("secret"));
        msg.sender.transfer(prize);
        finished = true;
    }
}
```

DoS through improper exception handling: Contracts that perform external calls such as sending a payment to another address may be susceptible to a denial-of-service condition if this call occurs in a loop. In other words, it is bad to attempt multiple payments in one transaction because if one of them fails, the rest of the payments will not happen and the funds are effectively frozen until the exception somehow does not occur anymore.

```
contract Test {
 address[] refundAddresses;
 mapping (address => uint) public refunds;

 function Test() {
  refundAddresses.push(0xdeadbeef);
  refundAddresses.push(0xd15ea5e);
 }

 function refundAll() public {
  for (uint x; x < refundAddresses.length; x++) {
  // exception on x=0 causes refundAddresses[1]
  // to never get a refund
      require(refundAddresses[x].send(
         refunds[refundAddresses[x]]));
  }
 }
}
```

Integer under/overflow: This type of vulnerability happens when an arithmetic operation results in a value greater than the maximum or smaller than the minimum for a specific type. For instance, trying to store 2^8 in a `uint8` type would actually store a 0 because the value wraps around. This is mitigated by using a safe math library that checks for overflowing

```
contract Test {
    uint public number = 1;

    function sub(uint guess) public {
        number -= guess;
    }
}
```

Unprotected self-destruct: This is a simple improper access control vulnerabilitiy but it is significant because it is the reason why the parity multisig attack occurred. An attacker was able to escalate privileges and self-destruct a library imported by many other contracts freezing all the funds in those contracts. In order to defend against this vulnerability, a developer should implement proper access controls if needing this functionality at all.

```
contract Test {
    function suicide() {
        selfdestruct(msg.sender);
    }
}
```

Re-entrancy: This is a relatively common and devastating attack where calling an external contract allows an attacker to take over control flow by recursively calling back into the contract before the initial call's appropriate state changes occur. The DAO attack was victim to reentrancy where an attacker recursively withdrew funds using a single call. Similarly, in the example below, the condition that allows a caller to trigger a transfer depends on a state that is not changed until after the external call occurs. The `msg.sender.call.value(amount)` allows an attacker to specify a function they can call back into after the transfer happens but before the necessary state change. This vulnerability is mitigated by performing all of a contract's state changes before doing an external call.

```
contract Test {
    mapping (address => uint) public credit;

    function donate(address to) payable public {
        credit[to] += msg.value;
    }

    function withdraw(uint amount) public {
        if (credit[msg.sender] >= amount) {
            require(msg.sender.call.value(amount)());
            // this state change occurs after call
            credit[msg.sender] -= amount;
        }
    }
}
```

III. Initial Approach

The current plan was to use Manticore [5], a symbolic execution engine that supports EVM bytecode (and other

architectures) to maximize code coverage on smart contracts sampled from Etherscan [2]. Manticore can simulate the EVM as well as simulate an entire "Ethereum world" where a contract may call on other contracts. One advantage is that this analysis can be done without any real Ethereum coin because Manticore can set arbitrary amounts for contracts and wallets you define in the API.

The objective was to identify devastating vulnerabilities we have seen before such as the re-entrancing attack on the DAO that allows an attacker to steal funds. Manticore would provide inputs that will hit various branches in smart contracts and the next step is to apply a fuzzer to user-controlled inputs in order to generate corner cases. In other words, we want to cause as many exceptions we can along every branch in the code and log them for further analysis. The collected smart contracts will also be analyzed for semantic bugs after reading up on the latest version of Solidity and recommended practice.

One difficulty in the approach may be dealing with all the versions of Solidity being used in the wild. Solidity is constantly being patched which causes smart contract writers to be slow in keeping up. Manticore fully supports Solidity versions pre-0.5.0 so we will have to scrape contracts using Solidity 0.4 and older.

IV. Fuzzing with Echidna

V. Honeypot Contracts

VI. Mass Scanning with Pakala

VII. Conclusion

Ultimately, we hope to shed some light on the current unfortunate state of Ethereum smart contract security and potentially report bugs, making cryptocurrency a little safer. Next steps from here may be directly fuzzing the Solidity compiler with randomly generated smart contract code and also fuzzing the EVM runtime as previously mentioned.

References

[1] V. Buterin. Ethereum white paper: A next generation smart contract & decentralized application platform. 2008. "[Online; accessed 5-April-2020]".

[2] Etherscan. Etherscan. https://etherscan.io/aboutus, 2015. [Online; accessed 6-April-2020].

[3] A. Madeira. The dao, the hack, the soft fork and the hard fork. 2019. "[Online; accessed 5-April-2020]".

[4] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008. "[Online; accessed 5-April-2020]".

[5] T. of Bits. Manticore. https://github.com/trailofbits/manticore, 2019. [Online; accessed 6-April-2020].

[6] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss. Security analysis methods on ethereum smart contract vulnerabilities — a survey. 2020. "[Online; accessed 5-April-2020]".

[7] S. C. Security. Smart contract weakness classification and test cases. https://swcregistry.io/, 2019. [Online; accessed 6-April-2020].