CSE5095-010: Blockchain Technology

Lecture 8

Ghada Almashaqbeh

UConn - Fall 2020

Outline

- Security issues in Ethereum (or smart contract platforms in general).
  - Veirifer Dilemma.
  - Reentrancy attack.
  - Transaction ordering dependence.
  - Block timestamp dependence.
  - Mishalndeld expcetions.

Writing a Secure Contract Code

- Writing a secure code is hard.
  - It needs extensive testing to cover all paths an attacker may utilize while interacting with the code.
  - Attackers are financially motivated to hack smart contracts.
    - Bugs literally cost money.
  - Lack of understanding how the underlying network or cryptocurrency system works may lead to writing buggy code.
- And remember, you cannot patch a buggy smart contract code.
  - So it is like recovering from a code security vulnerability is almost impossible.

Is it All About Code?

- Ethereum is an ecosystem that allows deploying smart contracts as a way to build new applications and services.
  - It has security issues Like any other large-scale blockchain-based system.
    - DoS, tendency for centralization, 51% attack, Eclipse/Goldfinger attacks, etc.
- Securing these services requires:
  - Extensive threat modeling.
  - Understanding how Ethereum/blockchain-based systems work.
  - Security by design; integrating countermeasures into the application design.
  - Then, develop a smart contract that implements the design.

Security Issues

- In this lecture, we will focus on the last step, smart contract security issues.
  - Threat modeling will be covered separately.
  - Covered issues:
    - Reentrancy vulnerability.
    - Transaction ordering dependence.
    - Timestamp dependence.
    - Mishandled exceptions.
- We will also cover one security risk that is of a large concern in Ethereum.
  - Verifier Dilemma.

Verifier Dilemma I

- A potential security threat that may arise due to complexity of computations a smart contract implements.
- Upon receiving a newly mined block, a miner is supposed to verify the validity of each transaction in this block before accepting it.
  - In Ethereum, this means re-executing all transactions that call functions from smart contracts and check the new EVM state.
  - Sometimes the contract code is complex and requires significant amount of time to execute.
- However, malicious miners may not verify the correctness of the transactions.
  - They so that to save time and start working on the proof-of-work race before honest miners.

Verifier Dilemma II

- This leaves the honest miners with dilemma of whether to validate blocks received from others or not.
  - Validate -- malicious miners may win the mining race faster, hence, risk losing the mining rewards.
  - Not validate -- may lead to adopting an invalid blocks in the blockchain.
- Can this be solved by saying the majority of the computing power is honest?
- This dilemma applies also to other cryptocurrencies.
  - Risk is higher when non-trivial computation is needed to verify transactions.

Verifier Dilemma - Potential Solutions

- Simplify the scripting language, hence, allow for simple, fast to verify scripts.
  - This limits the flexibility and supported functionality of the systems.
- Design correctness proofs that have fast verification time.
  - Verifying the computation does not require re-executing the whole computation.
  - Non-trivial to come up with such proof systems.
  - Such proof systems may introduce additional assumptions like a trusted setup, and may degrade the efficiency of the prover.

Reentrancy Vulnerability I

- The vulnerability that was exploited in The DAO incident.
- Happens when a contract calls a function from another contract.
  - The state of the caller contract is not updated until the called function is finished.
- An attacker may exploit the intermediate state before the final update.
- Usually it happens by defining a fallback function in an external contract (a function that is called if no funcion match is found).
  - This function will be called by the attacked contract.
  - The body of this function is the code that exploits the intermediate state of the attacked contract.
- Usually it is used to drain the currency in the contract account.

Reentrancy Vulnerability II

- For example, assume we have a contract that allows a party to withdraw its balance and then zeros the balance.
    - This contract allows the caller to specify an address to send the withdrawn currency to.
    - An attacker, may craft a contract and ask to send the money to the contract address instead of an EOA address.
    - The fallback function in the crafted contract calls the withdraw balance function several times. Will go through since zeroing the balance comes after finishing the call.
    - This allows the attacker to withdraw all the contract money instead of her balance only.
- This is the vulnerability exploited in The DAO attack.

Reentrancy Example I

Example 9-1. EtherStore.sol

```
contract EtherStore {

    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(address => uint256) public balances;

    function depositFunds() public payable {
        balances[msg.sender] += msg.value;
    }

     function withdrawFunds (uint256 _weiToWithdraw) public {
         require(balances[msg.sender] >= _weiToWithdraw);
         // limit the withdrawal
         require(_weiToWithdraw <= withdrawalLimit);
         // limit the time allowed to withdraw
         require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
         require(msg.sender.call.value(_weiToWithdraw)());
         balances[msg.sender] -= _weiToWithdraw;
         lastWithdrawTime[msg.sender] = now;
     }
 }
```

*From Mastering Ethereum book, Chapter 9

Reentrancy Example II

Example 9-2. Attack.sol

```
import "EtherStore.sol";

contract Attack {
  EtherStore public etherStore;

  // intialize the etherStore variable with the contract address
  constructor(address _etherStoreAddress) {
      etherStore = EtherStore(_etherStoreAddress);
  }

  function attackEtherStore() public payable {
      // attack to the nearest ether
      require(msg.value >= 1 ether);
      // send eth to the depositFunds() function
      etherStore.depositFunds.value(1 ether)();
      // start the magic
      etherStore.withdrawFunds(1 ether);
  }

  function collectEther() public {
      msg.sender.transfer(this.balance);
  }

  // fallback function - where the magic happens
  function () payable {
      if (etherStore.balance > 1 ether) {
          etherStore.withdrawFunds(1 ether);
      }
  }
}
```

How to fix it?

*From Mastering Ethereum book, Chapter 9

Transaction Ordering Dependence I

- Also called race condition or front running.
- The state of the blockchain, and hence, the state of the deployed smart contracts depends on the order of executing the transactions.
    - Two transactions issued at the same time, or close time intervals, from different accounts can be executed in an arbitrary order.
        - Recall that for transactions tied to the same account, the transaction nonce is used to resolve order issues.
- An attacker may utilize this dependence to gain monetary profits.
    - Observe transactions from others and act accordingly by issuing competing transactions.
    - Network propagation delays, and other factors like transaction fees, may result in executing the attacker's transaction first.

Transaction Ordering Dependence II

- For example, consider a puzzle solving contract where Alice posts a contract rewarding for solving a puzzle.
    - Bob has solved the puzzle and issued a transaction containing the solution.
    - Alice monitors the network, once it hears about Bob’s transaction, she issues another transaction to withdraw the bounty.
    - There is a chance that Alice’s transaction will be executed first, hence, Alice obtains the puzzle solution for free.

Timestamp Dependence

- Also called block timestamp manipulation.
- Some smart contracts may use the timestamps of the blocks on the blockchain.
  - For example, use the hash of a future block and its timestamp to determine the outcome of a lottery draw.
- A miner sets the timestamp based on its local machine.
  - It can vary by up to 900 seconds and still accepted by other miners.
- Hence, a miner can set this timestamp in a way that influences the contract in the way it desires.
  - Tying this to the above example, a miner can change the timestamp in a way that produces a favorable lottery draw outcome.

Mishandled Exceptions

- This occurs in contracts with code that does not check whether a function call has succeeded or not.
  - Usually happens when invoking functions from external contracts.
- For example, let's modify our Market smart contract to allow the market owner to sell the market to someone else. So the owner address will be changed.
  - Once the original owner receives the money from the new owner, which is also using a function inside the contract, change the owner to be the new owner.
  - If the money send function fails, and the contract does not check for such failure and act accordingly, the new owner will get the market for free.

Other Vulnerabilities

- As in conventional coding:
  - Buffer overflow.
  - Input/output sanity checking.
  - The use of external services/contracts that could be insecure.
  - Buggy built-in helper functions.
  - Uninitialized pointers.
  - etc.

References

- Luu et al., "Demystifying incentives in the consensus computer." In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 706-719. ACM, 2015.
- Luu, Loi, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. "Making smart contracts smarter." In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pp. 254-269. ACM, 2016.