Benchmarks and Vulnerabilities of DeFi

Kuldeep Meena (2019CS10490)

May 29, 2021

0.1 Introduction

DeFi or Decentralised Finance is an open, permissionless, and highly interoperable protocol stack built on public smart contract platforms, such as the Ethereum blockchain. Instead of relying on intermediaries of traditional financial instruments like brokerages, exchanges, or banks it depends on open protocols and DApps.The agreement between two parties are enforced by code that executes all the transactions in a way that is verified by all the nodes in a blockchain and this is secure. The design is such that there is no central authority.
Smart Contracts are central to a DeFi infrastructure.Smart contract is a code in a high level language that is executed on blockchain to complete a transction automatically and is verified by large number of validators. Since the smart contracts are to be verified by large number of validators the throughput is often slow as compared to the system of servient client architecture where we trust the server or a central authority but because of validations the transaction is much more secure.

0.2 Benchmarks of DeFi

- Transparency: Since DeFi is a blockchain bases infrastructure so transactions are publicly observable and smart contract code can be analysed on chain.
- Accessibility: (Working on unable to find good resources)

0.3 Security vulnerabilities of DeFi

1. **Smart Contract Vulnerability**: Smart Contract Vulnerability risk is the risk caused due to an incorrect code of smart contract or an attacker using well know attack vectors to alter functionalities of a smart code. The most famous of such smart contract attacks is 'The DAO' attack which used the re-entrancy vulnerability and the cost to about 60 million dollars of ETH.
   Some Smart Contract Vulnerability Risks are as follows:
   (a) **Re-Entrancy Vulnerability**: The vulnerability occurs because of incorrect code. The vulnerability occurs if a function to be executed can call another function(maybe from a different contract) in the mids of its execution and the function called can re-invoke the original

function without completing the instructions of the parent function that are responsible for critical state change. For example: If ETH in a contract are send before state change it might be possible that the target adddress is itself a contract that might call for requesting ETH again causing a cycle without state change and possiblity of extracting huge amounts of ETH. The 'DAO' attack utilised this vulnerability and became the worst attack so far.

```
1
2  // INSECURE
3  mapping (address => uint) private userBalances;
4
5  function withdrawBalance() public {
6      uint amountToWithdraw = userBalances[msg.sender];
7      (bool, success ) = msg.sender.call.value(
           amountToWithdraw)(""); // At this point, the
           caller's code is executed, and can call
           withdrawBalance again
8      require(success);
9      userBalances[msg.sender] = 0;
10 }
```

In the aove example there is a transfer of control from withdrawBalance() function to msg.sender.call.value(amount)(). The attackers fallback function might call withdrawBalance() again,thus leading to ETH extraction until the stack size limit is not reached.

(b) **Unhandled Exceptions Vulnerability**: A smart contract does not work independently and often calls another contracts' function to complete the transaction functionality, calls are made by sending istructions with a refrence to the contract.If an exception is raised it results in termination of contract and reverting to original state. But some low level operations in Solidity such as **send** doesn't throw an exception but instead return a boolean to mark success or failure. If the return value is unchecked in the caller function it might continue execution even when the payment failed causing inconsistencies.

```
1
2  function pay(employee){
3      if(!emp_paid[employee]){
4          employee.send(salaries[employee]);
5          emp_paid[employee] = true;
6      }
7  }
```

In the above example since the retun vlue of send is unchecked even if a send fails employee being paid will be marked true for the and hence cause inconsistencies.

(c) **Integer Underflow/Overflow Vulnerability**: The risk is common to not only to Solidity or Smart Contracts but to most programming languages and is the most common vulnerability in most Smart Contracts.The Solidity compiler does not trigger any error flag to resolve the code with integer overflow/underflow problem.If an integer variable is assigned to a value larger than the number for the assigned range for example in the cause uint8 max value is 255, it resets to 0; if the variable assigned to a value less than the range, it

would be reset to the top value of the range.If a loop counter were to overflow, creating an infinite loop, the funds of a contract could become completely frozen. This can be exploited by an attacker if he has a way of incrementing the number of iterations of the loop, for example, by registering enough users to trigger an overflow. 'Proof of Weak Hands' is the famous hack that utilised the risk. The risk is more in case of smart contracts where values goes to 0 and changing it to negative might make it the largest number.

```
1
2  mapping (address => uint256) public balanceOf;
3
4  // INSECURE
5  function transfer(address _to, uint256 _value) {
6      /* Check if sender has balance */
7      require(balanceOf[msg.sender] >= _value);
8      /* Add and subtract new balances */
9      balanceOf[msg.sender] -= _value;
10     balanceOf[_to] += _value;
11 }
```

In the above example if the banlanceOf the account to send + valye to be sent exceeds the assigned memory say 8 bits in case of uint8 then after completing transaction instead of gaining ETH the balnceOf the receiver will be zero.

(d) **Timestamp dependence**: In block chain time is asynchronous and not a synchronous digital clock. If a contract uses the block.timestamp (or now) global variable as a triggering condition for executing a critical operation (e.g. a money transfer) or as a source of randomness, it can be manipulated by a malicious miner. If a miner holds a stake on a contract, it could gain an advantage by choosing a suitable timestamp for a block it is mining.

```
1
2  function payAfter(tMins){
3      require(msg.value==10 ETH);
4      require(now!=initTime);
5      initTime = now;
6      if(now%tMins==0){
7          msg.send.transfer(this.balance)
8      }
9  }
```

Since block timestamp manipulation can be used by miners, Such a contract is potentially vulnerable.

(e) **Unrestricted Action**:Often there is check about who can make a particluar change by using the senders address,this is done to restrict the action of a particular user. Typically, only the owner of a contract should be allowed to destroy the contract or set a new owner. It is possible not only if the programmer has not performed checks for the same but also if the attacker executes aribitrary code, for example by being able to control the address of a delegated call

(f) **Locked Ether**: Funds might be locked permanently is a possiblity. One reason might be the contract is dependent on other contract which if possible to be killed by anyone using SELFDESTRUCT

which is the contract code is removed and if this was the only means for the contract to complete, it would result in funds being locked permanently. Parity wallet hack locking millions is an example of such a vulnerability.It is also possible that the contract will always run out of gas when trying to send Ether which could result in locking the contract fund.

(g) **Transaction Sate Dependence**:Attackers might utilise transaction state variables such as tx.origin which allows the contract to check the original initiator of the chain which might allow attacker to fulfill a check for the initiator and thus making contract vulnerable.

```
1
2 function transferAmt(address to,unit amt){
3     if (tx.origin != owner) { throw; }
4     to.send(amount);
5 }
```

If the actor deploys an attacking contract with the above function then since the original initiater was tx.origin check will fail and the owner will losse ETH.

2. **Transaction Ordering Dependency Vulnerability**: It is possible that multiple transactions could be present in the block that are allowed to change the state of a contract. So the reordering might be of concern as Miner mines the one with higher gas and it is possible to have case in which there is a reward with puzzle solved and a transaction then if multiple transaction then a mallacious owner might reduce the prize.

3. **Design Risk**: DeFi often utilises the code of others with its own code. So even if DeFi has high levels of risk mitigitation if it adds a new smart contract or protocol which is less secure it still poses a threat to the whole transaction,

4. **Block State Dependence Vulnerabilities**:This vulnerability utilises the block state variables that can be used to generate randomness, eg. timestamp, coinbase,number, difficulty and gaslimit.These vulnerabilities are possible as these are generated at the miners computer and thus miner can manipulate the states accordingly that favours payouts intended to it.For example, consider a lottery contract in which users purchase a ticket and one lucky user wins the jackpot. This requires randomness to choose the winner. If a miner wants to increase their chances, they could see what the contract uses as a source of entropy and try to manipulate it for their benefit.

0.4 References

1. https://www.researchgate.net/publication/344689196$_{I}dentifying_{K}ey_{N}on-Financial_{R}isks_{i}n_{D}ecentralised_{F}inance_{o}n_{E}thereum_{B}lockchain$
2.$https://www.usenix.org/system/files/sec21summer_{p}erez.pdf$
3.$https://consensys.github.io/smart-contract-best-practices/known_{a}ttacks/$
4.$https://www.atlantis-press.com/journals/ijndc/125913574/view$
5.$https://subodhvsharma.github.io/publication/ndss18/ndss18.pdf$