valid.network

Valid Network | The Reentrancy Strikes Again — The Case of Lendf.Me

Valid Network May 25, 2020

6-8 minutos

DeFi or decentralized finance is a growing sector in the blockchain and cryptocurrency space that defines an ecosystem of decentralized applications providing financial services with no governing authority.

Lendf.me is a DeFi app utilizing smart contracts in order to provide instant, decentralized lending. The platform suffered an attack causing the loss of $25m in cryptocurrency on the day of April 19, 2020.

Thus, joining the list of other DeFi protocols exploited recently:

Synthetix hack — 37M sETH stolen

bZx hack — $900k stolen

Lendf.me's current vulnerability is a unique instance of the reentrancy bug. Reentrancy is a well-known issue in the field of computing, referring to the ability of a subroutine to be interrupted in the middle of its execution and (then safely) be called again.

Other reentrancy bugs have been exploited in the past, causing massive damage, including:

The DAO hack — $150m stolen

Spank Chain hack — $30k stolen

The Lendf.me attack took place on the Ethereum main-net smart contract named MoneyMarket, which implements the core logic of the Lendf.me app.

Understanding the Lendf.me vulnerability

In order to best understand the underlying cause of the vulnerability, we should consider the contents of the various functions in the MoneyMarket contract.

First, we will consider the *MoneyMarket.supply()* function (line 1508).

```
1508    function supply(address asset, uint amount) public returns (uint) {
1509        if (paused) {
1510            return fail(Error.CONTRACT_PAUSED, FailureInfo.SUPPLY_CONTRACT_PAUSED);
1511        }
1512
1513        Market storage market = markets[asset];
1514        Balance storage balance = supplyBalances[msg.sender][asset];
1515
1516        SupplyLocalVars memory localResults; // Holds all our uint calculation results
1517        Error err; // Re-used for every function call that includes an Error in its return value(s).
1518        uint rateCalculationResultCode; // Used for 2 interest rate calculation calls
1519
1520        // Fail if market not supported
1521        if (!market.isSupported) {
1522            return fail(Error.MARKET_NOT_SUPPORTED, FailureInfo.SUPPLY_MARKET_NOT_SUPPORTED);
1523        }
1524
1525        // Fail gracefully if asset is not approved or has insufficient balance
1526        err = checkTransferIn(asset, msg.sender, amount);
1527        if (err != Error.NO_ERROR) {
1528            return fail(err, FailureInfo.SUPPLY_TRANSFER_IN_NOT_POSSIBLE);
1529        }
1530
1531        // We calculate the newSupplyIndex, user's supplyCurrent and supplyUpdated for the asset
1532        (err, localResults.newSupplyIndex) = calculateInterestIndex(market.supplyIndex, market.supplyRateMantissa, mark
1533        if (err != Error.NO_ERROR) {
1534            return fail(err, FailureInfo.SUPPLY_NEW_SUPPLY_INDEX_CALCULATION_FAILED);
1535        }
1536
1537        (err, localResults.userSupplyCurrent) = calculateBalance(balance.principal, balance.interestIndex, localResults
1538        if (err != Error.NO_ERROR) {
1539            return fail(err, FailureInfo.SUPPLY_ACCUMULATED_BALANCE_CALCULATION_FAILED);
1540        }
1541
1542        (err, localResults.userSupplyUpdated) = add(localResults.userSupplyCurrent, amount);
1543        if (err != Error.NO_ERROR) {
1544            return fail(err, FailureInfo.SUPPLY_NEW_TOTAL_BALANCE_CALCULATION_FAILED);
1545        }
1546
```

```
1547    // We calculate the protocol's totalSupply by subtracting the user's prior checkpointed balance, adding user's
1548    (err, localResults.newTotalSupply) = addThenSub(market.totalSupply, localResults.userSupplyUpdated, balance.pri
1549    if (err != Error.NO_ERROR) {
1550        return fail(err, FailureInfo.SUPPLY_NEW_TOTAL_SUPPLY_CALCULATION_FAILED);
1551    }
```

```
1578    /////////////////////////
1579    // EFFECTS & INTERACTIONS
1580    // (No safe failures beyond this point)
1581
1582    // We ERC-20 transfer the asset into the protocol (note: pre-conditions already checked above)
1583    err = doTransferIn(asset, msg.sender, amount);
1584    if (err != Error.NO_ERROR) {
1585        // This is safe since it's our first interaction and it didn't do anything if it failed
1586        return fail(err, FailureInfo.SUPPLY_TRANSFER_IN_FAILED);
1587    }
1588
1589    // Save market updates
1590    market.blockNumber = getBlockNumber();
1591    market.totalSupply =  localResults.newTotalSupply;
1592    market.supplyRateMantissa = localResults.newSupplyRateMantissa;
1593    market.supplyIndex = localResults.newSupplyIndex;
1594    market.borrowRateMantissa = localResults.newBorrowRateMantissa;
1595    market.borrowIndex = localResults.newBorrowIndex;
1596
1597    // Save user updates
1598    localResults.startingBalance = balance.principal; // save for use in `SupplyReceived` event
1599    balance.principal = localResults.userSupplyUpdated;
1600    balance.interestIndex = localResults.newSupplyIndex;
1601
1602    emit SupplyReceived(msg.sender, asset, amount, localResults.startingBalance, localResults.userSupplyUpdated);
1603
1604    return uint(Error.NO_ERROR); // success
1605 }
```

```
400  function doTransferIn(address asset, address from, uint amount) internal returns (Error) {
401      EIP20NonStandardInterface token = EIP20NonStandardInterface(asset);
402
403      bool result;
404
405      token.transferFrom(from, address(this), amount);
406
```

The main purpose of the *MoneyMarket.supply()* function is to handle token deposits. The function takes two arguments, the asset (the asset that the user wishes to deposit), and the amount (the number of tokens he wishes to deposit).

The main logic flow of the *MoneyMarket.supply()* function is as follows:

First, we read the balance variable that represents the user's deposited asset balance in MoneyMarket storage (line 1514), then, *MoneyMarket.checkTransferIn()* function is invoked (line 1526). This function (externally) calls the asset contract in order to figure if the user has the number of tokens he wishes to deposit and that he approved the

MoneyMarket contract to withdraw this amount on his behalf.

Later *MoneyMarket.doTransferIn()* function is invoked (line 1583) which (externally) calls the asset contract's *transferFrom()* function (line 405) that in turn transfers the amount from the user to the MoneyMarket contract. After the return from the external call, the *MoneyMarket.supply()* function is updating the user's deposited balance (lines 1599–1600).

Let's go over the *MoneyMarket.withdraw()* function's logic briefly. In a simplified manner, this function gets the requested amount of tokens to withdraw, checks that the user holds at least this amount of tokens then transfers these tokens to the user by (externally) calling the token contract *transfer()* function.

Can you spot the vulnerability by now?

The issue here is that *MoneyMarket.supply()* function is actually updating the user's asset balance **after** the external call to *asset.transferFrom()* (lines 1599–1600), but based on a value that was read **before** the external call (line 1514), which means that the update potentially ignores any updates that were made within the external call. In many terms, we can consider this anomaly to be a "Lost Update".

But why is Lendf.me's vulnerability exploitable?

In order to understand this, we will have a look at the imBTC contract (or any other ERC-777 compliant contract)

```
860     function _transferFrom(address holder, address recipient, uint256 amount) internal returns (bool) {
861         require(recipient != address(0), "ERC777: transfer to the zero address");
862         require(holder != address(0), "ERC777: transfer from the zero address");
863
864         address spender = msg.sender;
865
866         _callTokensToSend(spender, holder, recipient, amount, "", "");
867
868         _move(spender, holder, recipient, amount, "", "");
869
870         _approve(holder, spender, _allowances[holder][spender].sub(amount));
871
872         _callTokensReceived(spender, holder, recipient, amount, "", "", false);
873
874         return true;
875     }
```

```
1044    function _callTokensToSend(
1045        address operator,
1046        address from,
1047        address to,
1048        uint256 amount,
1049        bytes memory userData,
1050        bytes memory operatorData
1051    )
1052        internal
1053    {
1054        address implementer = _erc1820.getInterfaceImplementer(from, TOKENS_SENDER_INTERFACE_HASH);
1055        if (implementer != address(0)) {
1056            IERC777Sender(implementer).tokensToSend(operator, from, to, amount, userData, operatorData);
1057        }
1058    }
1059
```

The attacker took advantage of the fact that some of the assets implement ERC-777 standard, which means that the *imBTC._callTokensToSend()* function and thus, *attackerContract.tokensToSend()* function are invoked (lines 866, 1056 respectively) before the actual transfer of value between the two parties. This way, the attacker's contract gets a chance to call *MoneyMarket.withdraw()* function **before** the invocation of MoneyMarket.supply() is finished!

Attack Strategy

The only prerequisite for attempting the exploit is for an attacker to deploy an attacker contract that holds some amount of any asset that is ERC-777 compliant, let's assume for example that the attacker holds 10 tokens of imBTC.

Now,

1. The attacker would place the first transaction that invokes *MoneyMarket.supply(asset = imBTCAddress, amount = 9)*. At this point, the attacker holds a supply of 9 imBTC in the MoneyMarket contract, and a balance of 1 imBTC in the imBTC token contract.
2. The attacker would place the second transaction that invokes *MoneyMarket.supply(asset = imBTCAddress, amount = 1)* , but now with an external call to *MoneyMarket.withdraw(asset = imBTCAddress, requestedAmount = 9)* inside the *attackerContract.tokensToSend()* callback. By the end of this transaction, the attacker's imBTC balance in the imBTC token contract is 9, but the imBTC supply in the MoneyMarket contract is 10! This unwanted state occurred as the *MoneyMarket.supply()* function increases the supply for the attacker (lines 1599–1600) it uses stale data. Therefore, the function doesn't "know" at this point, that the attacker has already withdrawn some of his supply.
3. Now, the attacker holds a deposit on the MoneyMarket contract, backed by nothing. The attacker can use this to (falsely) borrow or withdraw assets deposited by other users. Furthermore, these two steps could be potentially

performed, again and again, thus draining MoneyMarket's liquidity.

Mitigation

When writing the smart contract code,

1. Try not to update **any** storage variables after an external call.
2. If not possible, deploy some locking mechanism, like the commonly known ReentrancyGuard instead. Make sure that any pair of code paths that have a possible read/write conflict for a variable will be "reentrancy guarded". For example, in this case, deploying a reentrancy guard only for the *MoneyMarket.supply()* function would not solve the problem, it should be deployed for the *MoneyMarket.withdraw()* function as well. Valid network's automated tools can help identify locations where these guards are missing, or incorrectly implemented.

About Valid Network

Valid Network's blockchain security platform provides complete life cycle security for enterprise blockchains from initial development to active deployment and management. Based in Be'er Sheva, Israel, the company's solutions enable enterprises to innovate with blockchain faster, providing complete visibility and control over their distributed applications and smart contract governance, compliance, and security posture through advanced platform capabilities.

Secure the block with Valid Network.

Learn more: https://valid.network

Follow us: LinkedIn | Twitter | Blog