

Smart Contract Security: Lessons Learned with Real-World Attacks

Chuan Tian, Qiyao Ma, Pu Guo

May 2023

1 Introduction

Ensuring the security of smart contracts presents numerous challenges. Even a minor coding error could result in significant financial losses for a smart contract that holds a substantial amount of funds. As a result, it is meaningful to enhance the security of smart contracts and create a more reliable Ethereum blockchain. By improving the security of smart contracts, users can have more confidence in the technology and its potential applications, which could help to drive broader adoption and growth in the blockchain industry.

This article provides an in-depth analysis of various types of attacks using code examples, highlights the mistakes made by contract developers in the past, and serves as a guide for preventing future developers from repeating those same errors. With the immutability of smart contracts, it becomes crucial to identify security vulnerabilities before deploying the contract onto the blockchain. Therefore, this article stresses the importance of conducting thorough security checks before the deployment process to ensure the safety and integrity of the smart contract.

The team also creates a GitHub project containing the code snippet and instructions on how to simulate the overflow attack, the link is as follows: <https://github.com/puguo/Overflow-Attack-Example>

2 Reentrancy Attack

A vulnerability found in smart contracts called reentrancy attack enables attackers to carry out a function more than once, even before the previous call has finished executing. The vulnerable contract may be continuously drained by an attacker until it becomes insolvent. This is typically due to the vulnerable contract's inability to verify the exploiter's new balance promptly. In a reentrancy attack, the attacker first deposits tokens into a vulnerable contract and then initiates a withdrawal. Interestingly, the attacker intentionally prevents the contract from being able to receive tokens. As a result, when the vulnerable contract attempts to send tokens, the exploiter contract is unable to receive them, resulting in a mismatch that triggers the fallback function, which allows for the reception of Ether during anomalous occurrences. However, the attacker's contract contains additional manipulative code that calls the victim contract to send Ether repeatedly. This is how reentrancy functions within smart contracts. To further illustrate the concept of reentrancy, let us delve into custom scenarios.

2.1 Explaining Reentrancy with Custom Solidity Contracts

```
import 'TheBank.sol';

contract TheAttacker {

    TheBank public theBank;
    mapping(address => uint) public balances;
```

```

    constructor(address _thebankAddress) {
        theBank = TheBank(_thebankAddress);
    }

    //a fallback function
    receive() external payable {
        if (address(theBank).balance >= 1 ether) {
            theBank.withdrawall();
        }
    }

    function attack() external payable {
        require(msg.value >= 1 ether);
        theBank.deposit{value: 1 ether}();
        theBank.withdrawall();
    }

    function getBalances() public view returns (uint) {
        return address(this).balance;
    }
}

contract TheBank {
    mapping(address => uint) theBalances;

    function deposit() public payable {
        require(msg.value >= 1 ether, "cannot deposit below 1 ether");
        theBalances[msg.sender] += msg.value;
    }

    function withdrawall() public {
        require(
            theBalances[msg.sender] >= 1 ether,
            "must have at least one ether"
        );
        uint bankStatement = theBalances[msg.sender];
        (bool success, ) = msg.sender.call{value: bankStatement}("");
        require(success, "transaction failed");
        theBalances[msg.sender] -= 0;
    }

    //returns the total balance of the funds in the bank.
    function totalBalance() public view returns (uint) {
        return address(this).balance;
    }

    function improve_withdrawall() public {
        require(theBalances[msg.sender] > 0);
        theBalances[msg.sender] = 0;
        (bool success, ) = msg.sender.call{value: theBalances[msg.sender]}("");
        require(success);
    }

    uint256 private _status; // Reentrancy lock

```

```

// Reentrancy lock
modifier Lock() {
    // On the first call to Lock(), _status will be 0
    require(_status == 0, "Lock: reentrant call");
    // Any subsequent calls to Lock will fail
    _status = 1;
    -;
    // Call finished, restore _status to 0
    _status = 0;
}

// Withdraw all ether for msg.sender
function withdraw() public Lock {
    require(theBalances[msg.sender] > 0);
    theBalances[msg.sender] = 0;
    (bool success,) = msg.sender.call{value: theBalances[msg.sender]}("");
    require(success);
}
}

```

The `attack()` function stores the hacker’s “investment” in The Bank and initiates the attack by calling The Bank contract’s `withdraw()` function.

The `receive()` function contains malicious code that checks if there is any remaining ETH in The Bank contract, and then calls The Bank’s `withdraw()` function. The `withdraw()` function of The Bank contract does not update the account balance because the transaction that sends ETH using `msg.sender.call` from The Bank is not yet complete. The transaction keeps executing because the hacker’s `receive()` function keeps calling `withdraw()`. All balances in The Bank contract should be emptied before updating the `balances` variable in The Bank.

Once the ETH balance of The Bank contract is emptied, the `receive()` function will no longer execute the `withdraw()` function, and the execution of the `receive()` function will complete. At this point, the hacker’s account balance will be set to 0, and The Bank will have no remaining ETH.

2.2 Real-World Example

In May 2022, Bistroot adopted ERC-777 for their bridged token. Unfortunately, this token was exploited through the `emergencyWithdraw()` function in their staking contract, which had a well-known vulnerability. This allowed users to withdraw all their staked tokens at once, but the balance update only occurred after the transfer was completed. This enabled attackers to repeatedly call the function using the `tokensReceived()` hook, draining the contract by reusing the same user balance.

2.3 Preventative Techniques

1. The simplest way is to move the line `balance[msg.sender] = 0;` in the `withdraw()` function of the Bank contract above the `(bool success,) = msg.sender.call{value: balance[msg.sender]}("");` line. The modified function is as follows:

```

function improve_withdrawall() public {
    require(theBalances[msg.sender] > 0);
    theBalances[msg.sender] = 0;
    (bool success,) = msg.sender.call{value: theBalances[msg.sender]}("");
    require(success);
}

```

2. Using a locking mechanism, add a modifier function to prevent reentrancy attacks. The modifier function contains a default variable `_status`, which is set to 0 at the beginning. When the `withdraw()` function is called for the first time, it checks whether `_status` is 0. If it is, `_status` is set to 1, indicating that the function is currently being called and has not yet finished. When the attacking contract tries to attack, `_status` will still be 1 and the function has not finished yet, causing the attack contract call to fail and the reentrancy attack to be prevented.

```
uint256 private _status; // Reentrancy lock

// Reentrancy lock
modifier Lock() {
    // On the first call to Lock(), _status will be 0
    require(_status == 0, "Lock: reentrant call");
    // Any subsequent calls to Lock will fail
    _status = 1;
    _;
    // Call finished, restore _status to 0
    _status = 0;
}

// Withdraw all ether for msg.sender
function withdraw() public Lock {
    require(theBalances[msg.sender] > 0);
    theBalances[msg.sender] = 0;
    (bool success,) = msg.sender.call{value: theBalances[msg.sender]}("");
    require(success);
}
```

3. To prevent a reentrancy attack, it is recommended to set gas limits when using the `send()` or `transfer()` functions, as these functions allow only up to 2300 gas units per transaction. In contrast, a call function does not have a gas limit and can execute complex multi-contract transactions by forwarding its gas. However, this lack of limit also makes call functions vulnerable to reentrancy attacks. By setting gas limits, there won't be enough gas available for the function to be recursively called, preventing attackers from exploiting funds. It's important to note that gas costs are dependent on Ethereum's opcodes, which are subject to change, so this method should not be solely relied upon as a security strategy.

3 Denial of Service

Denial of Service (DoS) attacks essentially involve intentional and malicious actions by users, aimed at causing a contract to be rendered non-functional for a certain duration, or even permanently, thereby impeding the normal functioning of the system.

3.1 Explaining Reentrancy with Custom Solidity Contracts

```
contract VulnerableContract {
    uint[] public numbers;
    //allows users to add numbers to an array
    function addNumber(uint num) public {
        numbers.push(num);
    }
    function processNumbers() public {
        uint i = 0;
        while (i < numbers.length) {
            // do something with numbers[i]
        }
    }
}
```

```

        i++;
    }
}

```

It should be observed that the loop present in this contract iterates over an array which can be maliciously inflated. This implies that a malicious actor can repeatedly invoke the `addNumber` function, thereby augmenting the size of the numbers array to an enormous extent. In theory, this can be executed to a level where the amount of gas needed to run the for loop surpasses the block gas limit, ultimately causing the `processNumbers` function to be non-operational.

3.2 Preventative Techniques

Contracts are occasionally designed in a way that necessitates sending ether to a specific address or waiting for an external source to provide input in order to advance to a new state. Such design patterns can potentially result in DoS attacks if the external call fails or is obstructed due to external factors. To mitigate such risks, it is advised that contracts refrain from iterating over data structures that can be manipulated by external users in an artificial manner.

4 Missing Input Validation

Missing input validation refers to a situation where an application fails to properly validate or sanitize the data entered by users before processing or storing it. This can lead to various security issues, such as injection attacks, cross-site scripting (XSS), and other vulnerabilities that can compromise the integrity and security of the application or system.

4.1 Resulting Attack Scenarios and Potential Consequences

- **Smart contract vulnerabilities:** If a smart contract deployed on a blockchain lacks proper input validation, an attacker could send carefully crafted input data to exploit the contract's functions. This could result in unintended behavior, such as unauthorized fund transfers, frozen assets, or manipulation of contract variables.
- **Sybil attacks:** A blockchain network typically relies on a consensus mechanism to validate transactions and maintain the integrity of the distributed ledger. If the consensus mechanism lacks proper input validation, an attacker could create multiple fake identities (Sybil nodes) and use them to manipulate the consensus process, potentially leading to double-spending attacks or disrupting the network.
- **Transaction manipulation:** If a blockchain implementation fails to validate transaction inputs properly, an attacker could manipulate transaction data, such as the sender, receiver, or amount. This could lead to unauthorized transfers of funds or other undesired outcomes.
- **Denial of service (DoS) attacks:** In the absence of input validation, an attacker could send large volumes of malformed or invalid transactions to the network, causing the nodes to spend excessive resources processing these transactions. This could lead to a denial of service, making the network unresponsive or slow for legitimate users.

4.2 Preventative Techniques

- **Implement robust input validation:** Ensure that all data entered into the system, such as transaction details, smart contract parameters, and node identities, are validated against predefined criteria.
- **Use secure coding practices:** Follow best practices for secure coding, such as using well-tested libraries and frameworks, peer review, and regular security audits.
- **Test and audit smart contracts:** Thoroughly test and audit smart contracts for vulnerabilities related to input validation, and deploy only after addressing any identified issues.

- Monitor and analyze network activity: Continuously monitor the blockchain network for signs of unusual or malicious activity, and implement measures to respond to and mitigate potential attacks.

4.3 Real-World Example

Missing quantity validation: In March 2022, the NFT Marketplace Treasure DAO `buyItem()` allowed attackers to purchase NFTs without payment by specifying a `quantity` of 0. The total was calculated by multiplying the per-item price with the quantity, then a `transferFrom()` with the resulting amount of zero did not revert, and from this, the protocol assumed that payment must have been successful.

```
function buyItem(
    address _nftAddress,
    uint256 _tokenId,
    address _owner,
    uint256 _quantity
)
    external
    nonReentrant
    isListed(_nftAddress, _tokenId, _owner)
    validListing(_nftAddress, _tokenId, _owner)
{
    require(_msgSender() != _owner, "Cannot buy your own item");
    require(_quantity > 0, "Cannot buy zero");
```

The last line `require(_quantity > 0, "Cannot buy zero");` is missing, which results in exploiter calls `buyItem()` with zero quantity, pays 0, still receives NFT.

How the attack works:

1. Assume NFT address is set to '0x123' and the token ID is set to '1'. Also assume that the seller's address is '0x456'.
2. Exploiter calls `buyItem()` function with zero quantity `buyItem(0x123, 1, 0x456, 0);`
3. The function executes without throwing an error The exploiter receives the NFT without paying for it.

5 Arbitrary/Unchecked External Calls

Arbitrary or unchecked external calls refer to a situation in which a smart contract interacts with another contract or external service without proper validation or restrictions. This can lead to potential security risks and vulnerabilities, particularly in the context of blockchain and decentralized applications built on platforms like Ethereum.

5.1 Resulting Attack Scenarios and Potential Consequences

- Reentrancy attacks: If a smart contract calls an external contract without proper checks, an attacker could exploit the external contract to re-enter the original smart contract and execute its functions multiple times before the initial call completes. This can lead to unintended behaviour, such as the withdrawal of more funds than intended.
- Denial of service (DoS) attacks: An attacker could create a malicious contract that consumes excessive gas or resources when called, causing the calling contract to run out of gas and fail, effectively leading to a DoS attack.

- Unpredictable behaviour: Unchecked external calls can lead to unpredictable behaviour if the called contract is updated, compromised, or behaves in an unexpected manner.
- Exposure to third-party vulnerabilities: If a smart contract relies on an external contract or service with its own vulnerabilities, those vulnerabilities may also impact the calling contract, leading to potential attacks or exploits.

5.2 Preventative Techniques

- Use proper checks and restrictions: Implement proper checks and restrictions when calling external contracts, such as using the `transfer` function instead of `call.value()` for transferring Ether in Ethereum-based smart contracts.
- Implement mutexes or reentrancy guards: Use mutexes or reentrancy guards to prevent reentrancy attacks by ensuring that a contract function cannot be called again until the first call is completed.
- Verify external contracts: Verify and audit the external contracts or services that a smart contract interacts with to ensure their security and integrity.
- Use established patterns and best practices: Follow established design patterns and best practices for handling external calls in smart contracts, such as the use of the “Checks-Effects-Interaction” pattern.
- Keep contracts modular and isolated: Design smart contracts to be modular and isolated, reducing the potential impact of vulnerabilities in external contracts or services.

5.3 Real-World Example

5.3.1 Fortress’s `submit()` Function

In May 2022, Fortress’s `submit()` function was supposed to accept price updates only from certain verified accounts with enough “power”-tokens to do so. But according to a code comment, this feature wasn’t turned on yet as they were waiting to have proper “DPoS”. What the function did instead simply counted the amount of provided signatures and ensured that those signers were unique. At no point, it actually checked who the signer in question was, so the attacker could easily satisfy these checks and submit their own price data. Additionally, the attacker created and voted for a malicious governance proposal that was unnoticed for three days until he could execute it.

5.3.2 LI.FI’s smart contract

On March 20th, 2022, an attacker exploited LI.FI’s smart contract, specifically the swapping feature which allows users to perform swaps before bridging. Instead of actually swapping, they were able to call token contracts directly in the context of the contract. And as a result of the exploit, anyone who gave infinite approval to the contract was vulnerable.

The attacker started by passing a legitimate swap of a small amount followed by multiple calls directly to various token contracts. Specifically, they called the ‘`transferFrom`’ method which allowed the attacker to transfer funds from users’ wallets that had previously given infinite approval to our contract for that specific token.

```

function swap(bytes32 transactionId, SwapData calldata _swapData) internal {
    uint256 fromAmount = _swapData.fromAmount;
    uint256 toAmount = LibAsset.getOwnBalance(_swapData.receivingAssetId);
    address fromAssetId = _swapData.sendingAssetId;
    if (!LibAsset.isNativeAsset(fromAssetId) && LibAsset.getOwnBalance(fromAssetId) < fromAmount) {
        LibAsset.transferFromERC20(_swapData.sendingAssetId, msg.sender, address(this), fromAmount);
    }

    if (!LibAsset.isNativeAsset(fromAssetId)) {
        LibAsset.approveERC20(IERC20(fromAssetId), _swapData.approveTo, fromAmount);
    }

    // solhint-disable-next-line avoid-low-level-calls
    (bool success, bytes memory res) = _swapData.callTo.call{ value: msg.value }(_swapData.callData);
    if (!success) {
        string memory reason = LibUtil.getRevertMsg(res);
        revert(reason);
    }

    toAmount = LibAsset.getOwnBalance(_swapData.receivingAssetId) - toAmount;
    emit AssetSwapped(
        transactionId,
        _swapData.callTo,
        _swapData.sendingAssetId,
        _swapData.receivingAssetId,
        fromAmount,
        toAmount,
        block.timestamp
    );
}

```

This worked because these calls were performed within the context of the contract, which had permission to transfer user funds. The attacker transferred these tokens into a separate wallet that he controlled. Once the transfers were completed, the small amount swapped at the beginning was bridged, and the transaction was completed.

Solution: The team disabled all contract methods that allowed swapping. Then they implemented a whitelist to only allow calls to approved DEXs. The contract was also upgraded to include this new whitelist functionality, and swaps were reenabled. On top of that, the team has disabled infinite approvals by default.

6 Arithmetic Over/Under Flows

Overflow and underflow happen when an arithmetic operation attempts to create a numeric value that is outside of the range that can be represented with a given number of digits – either higher than the maximum or lower than the minimum representable value.

In Ethereum, integer variables are stored using a fixed number of bits. When a variable exceeds the maximum value that can be represented by those bits, the variable “wraps around” and starts again from the minimum value. For example, suppose there’s an *uint8* variable that has a value of 0, subtracting 1 from this variable would result in a value of 255.

Both Solidity and EVM(Ethereum Virtual Machine) have poor defensive mechanisms to overflow/underflow, or we should say there’s nothing to help identify arithmetic overflow/underflow, not even a compiler warning for that. Newer versions of Solidity can throw errors when overflow/underflow

happens, and some modules like SafeMath can also prevent this issue, however, EVM cannot track or indicate any overflow/underflow.

6.1 Code Example

```
contract OverflowAttack {
    mapping(address => uint) public balances;

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

    function withdraw(uint _amount) public {
        require(balances[msg.sender] >= _amount);
        balances[msg.sender] -= _amount;
        msg.sender.transfer(_amount);
    }
}
```

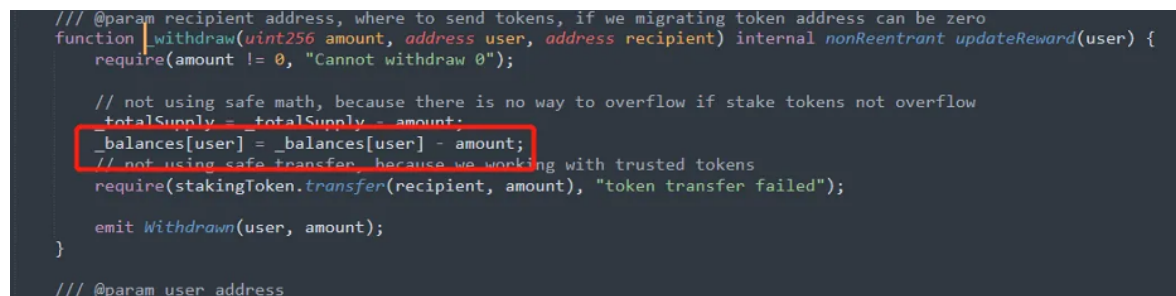
Here, the withdraw function doesn't check overflow, an attacker can give a number that is larger than the maximum value of a unit (e.g. 2^{256}) as an amount to bypass the required code, as it would circle back to 0. This enables the attacker to steal funds from the smart contract.

6.2 Overflow Attack Example

See the code snippet and instructions on <https://github.com/puguo/Overflow-Attack-Example>

6.3 Real World example

On March 21, 2022 Beijing time, Knownsec Blockchain Lab monitored a hack of the UmbNetwork rewards pool on BSC Chain and Ether with a loss of about \$700,000. The key to the vulnerability is an overflow vulnerability in the `_balance` function in the StakingRewards contract of the UmbNetwork rewards pool. The contract does not check the value of `balance` and the attacker launches an overflow attack via `amount`, emptying the pool of tokens.



```
/// @param recipient address, where to send tokens, if we migrating token address can be zero
function withdraw(uint256 amount, address user, address recipient) internal nonReentrant updateReward(user) {
    require(amount != 0, "Cannot withdraw 0");

    // not using safe math, because there is no way to overflow if stake tokens not overflow
    totalSupply = totalSupply - amount;
    _balances[user] = _balances[user] - amount;
    // not using safe transfer, because we working with trusted tokens
    require(stakingToken.transfer(recipient, amount), "token transfer failed");

    emit Withdrawn(user, amount);
}

/// @param user address
```

6.4 Preventative Techniques

Over/Under flow could be really easy to prevent, using SafeMath library or using a compiler that checks underflow or overflow could prevent this, you could also manually write code that checks this problem.

7 Front running Attack

Front running is a type of attack that occurs when an attacker exploits their knowledge of pending transactions to gain an unfair advantage in a smart contract. The attacker gains knowledge of transactions by monitoring the blockchain network for pending transactions, and they could be benefited

by quickly executing their own transaction before the pending transaction is confirmed, thereby "front running" the pending transaction.

In many cases, as the attacker receive a broadcast of the target transaction, the attacker can submit a transaction with a higher fee to front-run the original transaction.

7.1 Code Example

```
contract TokenSale {
    uint public tokenPrice = 1 ether;
    mapping(address => uint) public balances;

    function buyTokens(uint tokens) public payable {
        require(msg.value == tokens * tokenPrice);
        balances[msg.sender] += tokens;
    }

    function sellTokens(uint tokens) public {
        require(balances[msg.sender] >= tokens);
        balances[msg.sender] -= tokens;
        payable(msg.sender).transfer(tokens * tokenPrice);
    }
}
```

Suppose A wants to buy 5 tokens, and A sends a transaction. B sees the transaction, and B quickly submits his own transaction with a higher gas price before A's transaction is confirmed. Since B's transaction has a higher gas price, the miner is incentivized to include B's transaction in the next block instead of A's transaction. As a result, B's transaction is confirmed first, and B is able to buy the tokens at the current price of 1 Ether per token.

When A's transaction is eventually confirmed, A sees that the price of the tokens has increased because of B's transaction, and A has to pay a higher price to buy the tokens.

7.2 Real-World Example

In January 2022, Zora's NFT sale contract had a vulnerability that shows how infinite approvals can bite one back. To buy an NFT, users had to give the contract an allowance before calling the function to trigger the sale in a separate transaction. A malicious seller could front-run the second transaction to change the NFT's price and take all of the buyer's ERC20 tokens that were approved beforehand. Since unlimited approvals tend to be the default, that would quite possibly be all of the tokens the buyer owns.

8 Flash Loans Attack

Flash loans enable users to borrow funds without the need for any collateral. In these uncollateralized loan transactions, both borrowing and repayment must be completed within a single block. Currently, Ethereum's block time is approximately 12 seconds, allowing for the swift execution of these operations, hence living up to the name, flash loans, compared to traditional lending practices.

Due to the nature of blockchain, the transaction records within a block only become verified facts once they are packed and added to the blockchain. If a user fails to record the repayment within the same block after borrowing, the corresponding loan is automatically cancelled, rendering it as if no substantial borrowing occurred. This is why flash loans do not require collateral since, without repayment, the loan is effectively nullified, and the funds automatically roll back to their original source.

Flash loan attacks work by taking advantage of the fact that borrowed funds can be used to manipulate the price of an asset on a DeFi platform. For example, if the attacker wants to profit from a price increase, they may borrow the asset at a lower price on another platform without collateral and then sell it at the inflated price on the manipulated DeFi platform.

8.1 Real-World Example

In February 2023, the Platypus Finance project on the Avalanche network fell victim to a flash loan attack, resulting in a significant loss of \$8.5 million. Exploiting a flash loan from Aave, the attacker borrowed 44 million USDC to stake and borrow additional funds from the PlatypusTreasure contract.

By exploiting a vulnerability in the MasterPlatypusV4 contract, the attacker used the emergency-Withdraw function to withdraw the staked LP tokens without repaying the borrowed funds. This vulnerability was due to a logic issue in the emergencyWithdraw function, which failed to verify the borrower’s repayment status.

As a result, the attacker successfully carried out the attack, leveraging the flash loan to directly access the staked funds and causing a substantial financial loss of \$8.5 million for the Platypus Finance project.

8.2 Preventative Techniques

Currently, there is no universal solution to eliminate the rapid proliferation of these attacks. However, this article highlights notable steps that can be taken to combat this issue.

As these attacks exploit a singular DEX for their price feed, one effective measure is to utilize decentralized oracles that rely on multiple sources to determine the accurate price of assets. Certain decentralized oracles ensure data reliability by submitting data to the blockchain. This helps ensure that flash loan attacks aimed at manipulating asset prices would fail, as the data obtained from decentralized oracles would be trustworthy and resistant to manipulation.

9 Conclusion

This article provides an in-depth analysis of various types of attacks using code examples, highlights the mistakes made by contract developers in the past, and serves as a guide for preventing future developers from repeating those same errors. With the immutability of smart contracts, it becomes crucial to identify security vulnerabilities before deploying the contract onto the blockchain. Therefore, this article stresses the importance of conducting thorough security checks before the deployment process to ensure the safety and integrity of the smart contract.

10 References

- Amber Group. (2022, Mar 22). *Reproducing the \$APE Airdrop Flash Loan Arbitrage/Exploit*. <https://medium.com/amber-group/reproducing-the-ape-airdrop-flash-loan-arbitrage-exploit-93f79728fcf5>
- Beaver Finance. (2021, Nov 21). *DeFi Security Lecture 2 — Flash Loan Attacks*. <https://medium.com/beaver-smartcontract-security/defi-security-lecture-2-flash-loan-attacks-4aee2d3f07ca>
- Daejun, P. (2018, Mar 9). *Verified Smart Contracts*. <https://github.com/runtimeverification/verified-smart-contracts>
- Nekt. (2022, Mar 3). *Treasure DAO*. <https://rekt.news/treasure-dao-rekt/>
- Rob, B. (2021, May 7). *What is a front-running attack?*. <https://www.halborn.com/blog/post/what-is-a-front-running-attack>