Security Analysis of the Dutch Auction of WFCoin

MAHTOLIA RONAN
U2023144J
School of Computer Science and Engineering
Nanyang Technological University
ronan001@e.ntu.edu.sg

PATHAK SIDDHANT
U2023715K
School of Computer Science and Engineering
Nanyang Technological University
siddhant005@e.ntu.edu.sg

Abstract*— The advent of Bitcoin brought about a paradigm shift in the financial landscape, introducing decentralization through its groundbreaking blockchain technology. A significant challenge arising from the privacy-centric nature of blockchain is the potential for security breaches. While providing a layer of anonymity through hashed addresses, it introduces complexities in tracking and retrieving stolen assets in the aftermath of a smart contract attack. A dedicated website documenting large-scale crypto attacks highlights the alarming frequency at which millions of dollars are pilfered, underscoring the urgency of addressing security concerns within the blockchain ecosystem. This term paper goes in depth analyzing the security issues with the Dutch Auction of the WFCoin. It discusses the potential flaws and vulnerabilities with the smart contracts and how to solve them. It also discusses the potential impact of such vulnerabilities. This study contributes to the broader understanding of security considerations in blockchain-based dApps, emphasizing the critical importance of robust code and secure smart contract development.

Keywords—Blockchain, Dutch Auction, ERC20, smart contracts, Ethereum, reentrancy attack, overflow error, front-running attack

I. INTRODUCTION

The introduction of bitcoin was revolutionary because of its decentralized nature. It as based on an online immutable ledger called the blockchain that was secure because transactions were validated by consensus [1]. It incentivized fair play and removed control from any single entity. The ledger is called a block, and they are connected, forming a chain. It runs on a P2P network and all nodes store a copy of the blockchain. If someone tries to tamper it by adding or removing certain transactions to double spend or for any other reason, their version of the chain will differ from the rest and will be rejected. This helped counter the double spend problem. One of the attractive features of the blockchain was the privacy it offered. On the chain, a user was just a hashed address. Although it was not perfectly anonymous, it is not very easy to deduce someone's identity from their wallet address. While bitcoin is able to offer decent security and privacy, it is slow and expensive. In other words, it is not scalable.

The Blockchain trilemma as coined by Vitalik Buterin refers to the three-way trade-off between decentralization (privacy), scalability and security [2]. It basically highlights the 3 main problems that plague DApps and the chain in general. This term was coined during Ethereum 1 and at the time, most chain could only achieve two of the three metrics above. Decentralization means that there is no one point of control, everything is decided by consensus. Security refers to the robustness of the chain and smart contracts such that it should be able to defend against malicious attacks. Scalability refers to the ability to handle an increasing number of transactions per second. Decentralization is the very essence of blockchain and should be upheld. Then the choice is between security and scalability. While security is of paramount importance, a chain that cannot handle many transactions per second is not very usable. Therefore, in many cases, security was often sacrificed to accommodate the other two.

However, a major problem with this approach is that blockchain technology focuses heavily on privacy. Therefore, if someone was to attack a smart contract and steal hundreds of eth (hundreds of thousands of dollars as of now), it would be virtually impossible to track down and retrieve. Because of this reason, attacks on blockchain are very lucrative since there are virtually no repercussions. Hence security must be considered when developing dApps. A website [3] displays large scale crypto attacks and almost every day, millions of dollars are stolen. This is the gravity of security in blockchain, a single exploit can lead to the loss of an enormous amount of money.

Ethereum 2.0 comes up with solution to break the trilemma and incorporate all three metrics. To increase scalability, it uses sharding where it creates 'shard chains' which can process transactions and smart contracts parallelly, hence improving processing speed and potentially decreasing gas fees. The beacon chain was also introduced. This coordinates validators in the shards and introduced the proof-of-stake. Proof-of-stake replaced proof-of-work and does not require heavy computation. Instead, it requires users to stake their eth to run the software that validates and mines new blocks. This improved security and decentralization since anyone can stake eth, but everyone has access to multiple GPUs to mine.

In this term paper, we investigate the security aspect of the project that we built. The Dutch Auction, similar to how Algorand sells Algos, is set up to sell a certain number of ERC20 tokens [4]. In the case of this project, we minted a new coin called WFCoin. It starts at a certain price and its price depreciates over time. It requires buyers to commit some eth to buy the tokens. For the auction to end, one of two conditions must be met: the auction time runs out or all tokens are sold out. In the case the auction time runs out and the tokens are not sold out, whoever has committed eth will be rewarded tokens based on the reserved price at which the auction ends. The tokens they receive is equal to the eth they staked divided by the reserved price. In the case of the tokens sold out, everyone who committed eth will be awarded tokens equal to their commitment divided by the price at which all the tokens got sold out. In an auction where we store the users' eth over a certain period of time and then distribute the tokens only at the end, security is of utmost importance. Faulty code

can lead to attackers draining eth from the contract or fraudulently claiming more tokens than they are entitled to. Therefore, the focus of this report will be on the security aspect.

II. Literature Survey

Some of the most common yet devastating attacks include re-entrancy attacks, attacks on default visibility, arithmetic overflow or underflow, front running attacks, etc. [5]. Re-entrancy attack involves attacking a contract using another. The attacker contract deposits some eth to the victim then when claiming the eth back, the attacker puts a claim call in the attacking contract's receive or fallback function. This repeatedly claims eth from the victim contract before it has a chance to update the balances. This can be used to drain all the eth in a contract and is the source of the infamous DAO attack that drained 5.6% of all eth in circulation at the time [6]. Default visibility in smart contracts is public. If developers do not take care of this and let functions that should be private or internal be public, attackers can take advantage and gain control of some parts of the contract and may be able to become the owner of the contract which would mean that all the past and future funds to the contract would belong to the attacker. Overflow as the name suggests, involves assigning a value that is too big for a certain variable. Underflow is when the value is too small for the variable, for example trying to assign a negative value to an unsigned integer variable results in underflow. There are an astounding 20 cases where such errors can occur [7]. This goes to show that it is rather easy to miss when the code may have such vulnerabilities. Since the value wraps around when overflowing, a malicious attack can cause some variables like balance to become 0 by overflowing and wreaking havoc in the contract.

In a case-study conducted on blockchain games, it was pointed out that out of 1,311 smart contracts, only 11.63% were bug free [8]. This implies that 90% of the smart contracts could have been exploited or attacked through some bug. It also stated that 14.04% of the smart contracts had overflow/underflow vulnerabilities, and in 35.43% of the contracts, exception states were not handled. This brings to light how much security in blockchain is ignored. Given the ramifications of a successful attack, such security flaws must not be ignored.

III. Observation and Analysis

The most common error as stated by reference [8], is overflow/underflow errors. Indeed we came across this error in our project, especially underflow errors. This error occurred when we tried to calculate the current price of the token. We defined a variable called discountRate that we multiplied into the elapsedTime variable to calculate the depreciation in value. We then subtracted the start price by this depreciation to get the current value. However, simply multiplying the elapsed time and the discount was problematic if someone tried to call the function after the Auction ended. In this case, the depreciation became greater than the current price and threw an underflow error. To counter this, we simply added an if statement stating that if the elapsed time is greater than the auction time, then set the elapsed time equal to the auction time. While this covers most of the usual cases, there might be some edge cases that need exception handling which was not implemented.

Another flaw that could have happened, was re-entrancy attack. Like most other victim contract, we first sent the tokens to the buyers and then updated their balance. While this seemed like the normal way to do things, it is the reason why re-entrancy attacks become possible. However, since we implemented our ERC20 token by inheriting Openzeppelin's ERC20 contract [9], we used their transfer function which prevented re-entrancy attacks.

A possible security concern is that we overrode Openzeppelin's transfer function. The way their function works is that, to transfer tokens from one address to another, you first need to approve the address that is trying to transfer the tokens, to 'spend' the tokens. This adds a layer of security to make sure that the transfer can't be executed by a malicious contract. However, this implementation of approval was causing us some trouble when implementing the code. Therefore, we overrode their external transfer function so that we could work around the approval mechanism. Of course, since this becomes a security issue, we countered it by adding the only owner modifier to the transfer function. To make sure out Auction contract could call the transfer in the Token contract, we had to then make the Auction contract the owner of the Token Contract. This meant that we had to implement a change owner function (of course it had the only owner modifier). While in theory, this should be safe, there might be security issues unaccounted for. Moreover, while it is decentralized, the fact that the owner of the Token contract has the ability to transfer ownership and the owner can transfer tokens at their will, introduces a sense of centrality. It is possible to fraudulently create a contract and let people deposit eth but transfer the wrong amount of tokens if a malicious party (must be the one deploying the contract) wishes to. It makes the contract less trustworthy.

Yet another possible attack is the front running attack. The Auction should function on a first come first serve basis. However, it is possible that during the sale of the last few tokens, a malicious party pays a high gas fee to try and snatch the token from an honest bidder. The problem with this kind of attack is as follows. Assuming a buyer tries to buy the last 5 tokens for $50. They use the auction site an place a buy request with a normal gas fee. The contract verifies that there are still some tokens left to sell and allows the transaction. While it is being processed, a malicious party sees this and places a buy with a higher gas price. Since the first transaction has not been approved yet, the token contract still had the tokens. The malicious request will get processed first because of the higher gas fee essentially taking the honest buyer's tokens. In the worst case, while claiming, some honest buyer will not get their tokens because the malicious party withdrew their tokens beforehand. The best case would be that the buyer who tried to get the last few tokens could also be rejected wasting their gas since the token contract would have already sold out all the tokens. In this case, there would be loss of eth but only gas fee and inconvenience to the user.

There is always the possibility of other code vulnerabilities that have gone unnoticed. After all, no one writes code that can be attacked intentionally. Attacks will always come from overlooked security points. As with any software

development, there are many ways to exploit code that often go unnoticed and unaccounted for. Especially when it comes to developers without many years of experience like ourselves, code vulnerabilities could be rife in our smart contract.

IV. SOLUTIONS TO VULNERABILITIES

After some more literature review, we learnt more about the code vulnerabilities and how to counter them. When implementing the contracts, our main priority was getting it to work and adding quality of life features for our users. This meant writing more code that got a little complex which could be the entry point for attacks and vulnerabilities. Here we thoroughly go through each of identified vulnerabilities and how to counter them.

A. *Overflow/Underflow errors*

The best and easiest way to handle overflow and underflow errors is to import and use the SafeMath smart contract by OpenZeppelin. It defines four functions for addition, subtraction, multiplication and division. As mwntioned in reference [7], there are 20 possible ways to suffer an overflow or underflow error, therefore it is imperative to use the aforementioned functions to carry out any kind of arithmetic operations. Developers may be tempted to use their own implementations or use if statements to try and defend againt such errors. In other cases they migh ignore it completely as it is tedious to import then use functions for basic arithmetic operations. However, this contract is an open source standard and should be followed.

```
// unsafe operation (solidity v0.7)
currentPrice = startPrice - discount;

// using SafeMath
currentPrice = startPrice.sub(discount)
```

Figure 1: Using SafeMath operations

B. *Front Running Attacks*

One way to avoid this kind of attack could be to use a double check on and off-chain. When a buyer commits some eth, we could keep track of all the variables in our website. Therefore, our website would know how many tokens are left and what their current price would be. We could then bar any kind of malicious activity before it gets sent to the chain. However, this would introduce centrality to our project, and it would fail to stop attackers sending transactions to the chain directly, without using our website.

```
// adding gasLimit as an extra parameter
await DutchAuction.connect(address1).buy({
  value: ethers.parseEther("1.0"),
  gasLimit: gasLimit
});
```

Figure 2: Gas limit Parameter added

One other way to prevent front running attacks would be to limit the gas fees. Our project had implemented a gas fee calculator hence it would be possible to predict the gas fees required by a user to send a transaction to our contract. By setting an upper limit on the gas fees, we prevent someone from trying to send a transaction with a very high gas fee to maliciously attack the contract.

However, the best and most reliable way to prevent front running attacks is by utilizing the submarine send approach [10]. In this approach, when a user sends a transaction on the chain, information is obscured in a such a way that miners and regular users cannot really take advantage of it. It uses encryption for safety and once the transaction is added to the block, the user can choose to reveal the information. The transaction sent by the user is hidden in a large anonymity from everybody except the user. The submarine address is indistinguishable from a fresh address (an address that has not been used yet) [11].

C. *Re-entrancy attack*

Reentrancy attacks enable an attacker to drain all the funds from a contract fraudulently [12]. For reentrancy attacks to work, contracts must meet a certain condition. The first condition is that when the contract is transferring funds outside, it should use the .call function to call the address it is sending funds to. Then it can add the value using curly brackets. The second condition is that the contract must update the balance or the state that updates the fact that the funds have been sent, after the calling the address. If these two conditions are satisfied, an attack contract can deposit a minimal amount of eth into the victim contract and then claim it. It will also put a claim call in its receive or fallback function. Due to this, then the victim calls the attacker to transfer funds, control flow goes to the receive function of the attacker where claim is called again. Since the update of the balance happens after the call, the attacker recursively claims funds until the victim contract is entirely drained of its funds. In the implementation of ERC20.sol by open zeppelin, the transfer function does not use the call function to transfer tokens. Therefore, the receive or fallback of the attacker is not invoked and does not run on transfer. Hence it is resistant to reentrancy attacks and no special measure need to be taken.

```
// unsafe transfer
msg.sender.call{value: balance}("");

// safe transfer by OpenZeppelin
tokenContract.transfer(msg.sender, tokenBits);
```

Figure 3: Using OpenZeppelin transfers

D. *Default Visibility*

```
// add interval keyword to ensure no outsider can call
function _refund(uint amount, address addr) internal
```

Figure 4: Encapsulation for refund procedure

This flaw could let attackers drain all the funds from our contract if we were not careful. We defined a refund function to refund the users if their transaction was unsuccessful or they bought out the tokens but paid extra. Solidity does not allow payable functions to be private or internal, therefore, we defined two functions, one was public and payable and it called the internal function where the eth was transferred. If we had not taken care to make sure that the functions were

not callable by external users, they could have just transferred all the funds in the contract to themselves.

E. General Vulnerabilities in Code

In our attempt to make our code robust, we came up with some scenarios and wrote test cases to test our contract's execution in those scenarios. We used Chai [13] to assert that the result should be equal to what we expected it to be. We broadly divided our test cases into 5 broad categories. The first was basic functionality where we tested the basic function calls like getPrice(), buy(), claim(), etc. This was mainly to ensure our code didn't break when we changed it. The second was in the case that the Auction time elapsed and we had to burn the remaining tokens. We added a few test cases here and each test case had multiple asserts to ensure that everything was as planned. The third was if the tokens get sold out, we had a few test cases here to test the balances and the ending price, etc. The fourth was for testing the revert and require statements in out contract. For example, we tried to buy tokens despite them being sold out or trying to claim before the auction finished. The contract simply had to revert with the particular message we had set. Lastly, the fifth was an attack contract where we wrote a reentrancy attack and tried to fraudulently drain tokens from the contract. As expected, it did not work for the aforementioned reason.

```
// testing buy tokens after sold out
await expect(
  DutchAuction.connect(owner).buy({
    value: ethers.parseEther("0.1")
  })
).to_be_revertedWith("TOKENS SOLD OUT")
```

Figure 5: Testing for possible overspending

In total we had 14 test cases to test the various scenarios that we could come up with. However, 14 test cases is not a substantial amount. Thorough and proper testing should be much more meticulous. However, due to time constraints, we were unable to come up with more test cases and implement them. Conduction a more detailed literature review would have equipped us with the knowledge required to implement more test cases that could ensure our contract was more robust and iron out bugs and vulnerabilities. It goes without saying that the lack of experience in developing blockchain applications also played a part in our lack of testing. We should have tried various ways to try to attack overflow/underflow vulnerabilities since SafeMath was not being used. We also should have tried front running attacks to see how it could affect our Auction contract. In addition to this, tests for the function visibility should have been written. We assumed that they wouldn't be possible since in theory it was correct. However, this was not the best approach and tests should have been written even if it was obvious that the attack would fail.

Test Case	***Category of test***	***Result***
Deployment & Initialization of token contract.	Basic Functionality	Passed
Getting the Price on deployment.	Basic Functionality	Passed
Getting Token Balance on deployment.	Basic Functionality	Passed
Committing some amount of eth.	Basic Functionality	Passed
Checking total supply of eth in the case no tokens are bought.	Auction time elapsed	Passed
Checking the case where there is a single buy, and the rest is unsold.	Auction time elapsed	Passed
Checking balances and total Supply in the case there are 2 buyers	Auction time elapsed	Passed
Testing the case where all tokens are bought out at the start	Tokens Sold Out	Passed
Testing the case where the tokens are sold out during the auction	Tokens Sold Out	Passed
Checking revert when trying to buy less than 1 token	Errors Test	Passed
Checking revert when trying to claim before the auction ends	Errors Test	Passed
Checking revert when trying to buy tokens after the auction ends	Errors Test	Passed
Checking revert when trying to buy tokens after they are sold out	Errors Test	Passed
Trying re-entrancy attack on the Auction	Attack test	Passed

Table 1: Test cases we implemented and their results.

CONCLUSION

In this paper, we have solely focused on the security aspect of our project. There were some decentralization issues, but they were not discussed in detail since it is out of the scope of this paper. We discussed various types of attacks and vulnerabilities that our contracts might have had and how to counter them. We also went into detail as to how those attacks occur and what could be the potential implications of leaving such vulnerabilities in our code especially since our code would be open to the public to view and try to attack. While we delved deep into the vulnerabilities in our smart contract, that is not the source all possible attacks.

Like we discussed when trying to fend off front running attacks, we could alter our website to store the state of the smart contract, the website itself could be a source of vulnerabilities. We have not considered security in our

website because our main focus was the smart contract. Due to time constraints, we were unable to focus much on the website.

However, it should be noted that even if we were able to craft a perfectly safe smart contract and website, there are still some attacks that are out of our control. For example, a phishing attack to bait users to stake eth to an attacker's contract instead of ours. This would be out of our hands and only preventable by the users themselves. Other than phishing attacks, there are other attacks that we cannot prevent, such as attacks on Ethereum itself. Any sort of 51% attack or the Sybil attack [14] on the Main-Net itself would result catastrophic consequences and any attack would be undefendable. It would probably require a hard fork to restore the original chain just like they did after the DAO attack.

Attacks on smart contracts are fairly common and a lot of money is lost each time a successful attack is lost. Here is a list of some of worst attacks yet [1].

Network Attacked	***Date of Attack***	***Loss***
Ronin Network	March 23, 2022	$624M
Poly Network	August 10, 2021	$611M
BNB Bridge	October 06, 2022	$586M
Wormhole	February 02, 2022	$326M

Table 2: Past attacks on smart contracts

The list above shows some of the most devastating attacks that have happened yet. They cumulatively add up to 2 trillion dollars. These are major Networks that succumbed to attacks implying that it could happen to anyone. It goes to show that no matter how much security you may try to incorporate into your code, there might be some flaw that could result in losing all the funds.

REFERENCES

[1] D. Floyd, "How Bitcoin Works," *Investopedia*, May 11, 2022. https://www.investopedia.com/news/how-bitcoin-works/ (accessed Dec. 01, 2023).

[2] @pramodAIML, "Ethereum 2.0 & Blockchain Trilemma For Dummies," *CryptoWise*, Jan. 27, 2022. https://medium.com/crypto-wisdom/ethereum-2-0-blockchain-trilemma-for-dummies-60978aa8fa62 (accessed Nov. 30, 2023).

[3] "Rekt - Home," *rekt*. https://rekt.news/

[4] "Algo Auctions Overview | Algorand Foundation," *www.algorand.foundation*. https://www.algorand.foundation/algo-auction-overview (accessed Dec. 01, 2023).

[5] Hacken, "7 Most Common Smart Contract Attacks," *Hacken*, Nov. 10, 2022. https://hacken.io/discover/most-common-smart-contract-attacks/ (accessed Dec. 01, 2023).

[6] D. Siegel, "Understanding The DAO Attack," *www.coindesk.com*, Jun. 25, 2016. https://www.coindesk.com/learn/understanding-the-dao-attack/ (accessed Dec. 01, 2023).

[7] "Known Attacks - Ethereum Smart Contract Best Practices," *ethereum-contract-security-techniques-and-tips.readthedocs.io*. https://ethereum-contract-security-techniques-and-tips.readthedocs.io/en/latest/known_attacks/#integer-overflow-and-underflow (accessed Dec. 01, 2023).

[8] T. Min and W. Cai, "A Security Case Study for Blockchain Games," Jun. 2019. Accessed: Dec. 01, 2023. [Online]. Available: https://arxiv.org/pdf/1906.05538.pdf

[9] OpenZeppelin, "OpenZeppelin/openzeppelin-contracts," *GitHub*, Apr. 07, 2022. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol (accessed Dec. 01, 2023).

[10] "How to Solve the Frontrunning Vulnerability in Smart Contracts," *Hackernoon.com*, 2023. https://hackernoon.com/how-to-solve-the-frontrunning-vulnerability-in-smart-contracts (accessed Nov. 30, 2023).

[11] "Defeat Front-Running on Ethereum," *Libsubmarine.org*, 2022. https://libsubmarine.org/ (accessed Dec. 01, 2023).

[12] K. Polak, "Hack Solidity: Reentrancy Attack | HackerNoon," *hackernoon.com*, Jan. 17, 2022. https://hackernoon.com/hack-solidity-reentrancy-attack (accessed Dec. 01, 2023).

[13] "5. Testing contracts | Ethereum development environment for professionals by Nomic Foundation," *hardhat.org*. https://hardhat.org/tutorial/testing-contracts (accessed Dec. 01, 2023).

[14] "Sybil Attack," *GeeksforGeeks*, Jan. 10, 2019. https://www.geeksforgeeks.org/sybil-attack/ (accessed Dec. 01, 2023)