Security Issues in Cypherpunk Dutch Auction

Tan Yie Ern Nicholas
U1822212L
School of Computer Science and Engineering
Nanyang Technological University
Singapore
YTAN172@e.ntu.edu.sg

Truong Cong Cuong
U1820080D
School of Computer Science and Engineering
Nanyang Technological University
Singapore
CONGCUON001@e.ntu.edu.sg

Phua Jia Sheng
U1720889B
School of Computer Science and Engineering
Nanyang Technological University
Singapore
JPHUA014@e.ntu.edu.sg

***Abstract*—This paper aims to conduct an analysis on security issues present in the Cypherpunk Dutch Auction implementation. The focus will be on smart contracts, consensus layer as well as the interface and transport layer. After identification of security issues and their impact, suggested solutions will be recommended to tackle said issues.**

Keywords—blockchain, Ethereum, smart contracts, tokens, Dutch auctions, reentrancy, front running, libsubmarine, revert attack, denial of service, gas limit, block stuffing, DApps

I. INTRODUCTION .

Blockchains are a means to store data and information in a transparent and unalterable way. This technology has widespread uses, from financial transactions to cyber security. However, there are several pressing issues regarding the technology before it can gain widespread adoption.

Security. Blockchain is very well known for its high security and transparency due to its immutability. However, there are many historic cases where blockchain implementations have been hacked. Most notably, Mt.Gox, once the largest Bitcoin exchange, lost 850 thousand bitcoins worth $450 million USD and went bankrupt. Other attacks such as 51% attacks, where someone attains a majority of blockchain network and abuses it, are still problems yet to be solved. There are still many pressing issues still present today before blockchain can be adopted for mainstream consumption.

Privacy. Blockchain has some fundamental privacy problems by design. The distributed aspect of the blockchain means that each node that process transactions will need to have access to all the transaction data. For a cryptocurrency which uses blockchain, every transaction can be traced back. Some cryptocurrencies, such as bitcoin, implement a form of pseudo anonymity. However, through heuristics, it is still possible to trace the transactions to its owners.

Scalability. One of the biggest obstacles preventing blockchain technology from taking over the entire financial transaction network is the slow transaction speed. On the Ethereum blockchain network, transactions speeds are limited to 15 Transactions per second (TPS), compared to 24000 on most payment providers like Visa. This makes blockchain currently unsuitable for small transactions as it would be very computationally expensive as well as slow to process. A large reason of the slow speed is due to the computational cost from the consensus algorithms used in blockchain. For instance, Practical Byzantine Fault Tolerance (PBFT) algorithm has a $O(n^2)$ message complexity [9], causing it to scale very poorly as the network size increases. This issue of scalability needs to be resolved for widescale adoption of blockchain technology.

For our development project, we developed a decentralized application (DApp) for an initial coin offering (ICO) of our token, the CypherpunkCoin. The coin offering used the Dutch auction mechanism, where unlike the more popular English auction, the price of the coin started at the highest level and decrease until it reached the market determined price or the minimum reserved price that we had set. This ensures that as many coins as possible will be distributed, increasing liquidity of the cryptocurrency. The backbone of our Dutch auction mechanism was designed to use the Ethereum blockchain network.

To make use of the Ethereum blockchain, two smart contracts were written in Solidity. The token contract, CypherpunkCoin is first deployed onto the network. The client can then identify the address of the token contract and connect to the contract, listening for changes. Through a method in the token contract, the auction contract will then be deployed, allowing the contract address of the auction to be available via a method from the token contract. Once the client identifies the auction address, listeners will be setup to listen to events from the auction address, to start the interface and committing with the auction contract. The client can calculate the amount of time remaining, the estimated tokens staked by the user as well as the current price. There is also a button to call the tokens to be released from the auction contract once it is over. The client was written in react, and was meant to interface with Metamask, a chrome browser extension which was used as a wallet.

In this paper, we will be presenting our analysis of the security issues in our development project and discuss some solutions that we have come up with to solve the security problems that we have encountered.

II. Motivation and Literature Survey

A. Motivation

Out of the three issues most relevant to blockchain, namely security, privacy and scalability, we chose to focus on security as we feel that it is the most relevant issue with respect to our project. This is because there are several security issues that could affect the fairness and integrity of the distribution of our project token. For example, blockstuffing attacks could block legitimate bidders from participating in the auction. On the other hand, issues like privacy and scalability are results of the nature of the blockchain platform that CypherpunkCoin is deployed on (Ethereum) and cannot be easily solved by ourselves. For example, solving the scalability issue would have to depend on the increasing the transaction throughput of the Ethereum blockchain itself, which would depend on scalability solutions like sharding deployed in future Ethereum Improvement Protocol(EIP)s, or even Ethereum 2.0. Likewise, the privacy issues surrounding this project are a result of the open nature of the Ethereum blockchain itself, and while solvable, is not an issue as pressing as the security concerns.

B. Literature Survey

Because we want to examine all the aspects of our DApp, we take into consideration different layers: smart contract, consensus and front-end/ transport layers.

Regarding smart contracts and consensus layer, we started off by reading the best practices for Ethereum smart contracts which gives more details about common kinds of attacks such as reentrancy attack, front-running attacks, etc. [3] Besides, to identify common concerns among people when developing Auction application, security reports of Auctionity, DutchX were also examined. We also deep dive into solutions for these attacks from sources [3],[4],[5]

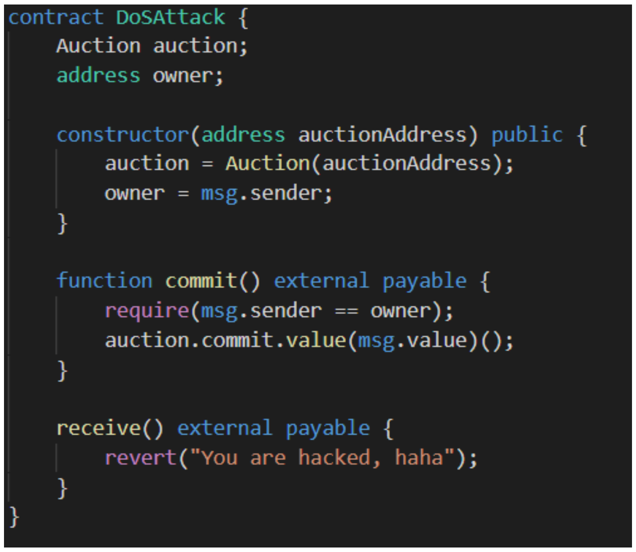
As for the frontend/transport layer, we first explored the possible security vulnerabilities in the front end. We used client-side rendering, and one of the biggest issues in client-side rendering is Cross Site Scripting (XSS) attacks. As most of our information and account access are via Metamask, this allowed our DApp to be largely safe from such attack vectors. However, the internet proved to be large source of attacks on blockchain DApps, such as DNS hijacking from BGP routers [2]. We identified that this attack could be a possible attack option for our DApp.

III. Observations and Analysis

A. Denial of Services with (Unexpected) revert attack

Regarding smart contracts, we implemented many methods to safeguard against attacks. One of them is defining the states in the contract to ensure the flow of the contract is in a correct direction. Another is the usage of SafeMath library to prevent possible overflow and underflow errors for operations. However, despite these preventive measures, our contract is not resistant to DoS with (Unexpected) revert [3].

Indeed, one can create a smart contract like below to implement such an attack.

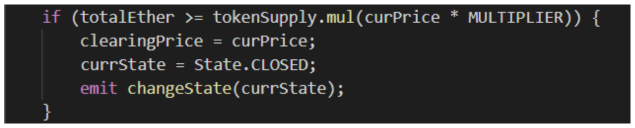


```
contract DoSAttack {
    Auction auction;
    address owner;

    constructor(address auctionAddress) public {
        auction = Auction(auctionAddress);
        owner = msg.sender;
    }

    function commit() external payable {
        require(msg.sender == owner);
        auction.commit.value(msg.value)();
    }

    receive() external payable {
        revert("You are hacked, haha");
    }
}
```

Figure 1: DoS Attack

This *DoSAttack* contract has two main functions, one is **commit()**, which is for the contract to commit to the Auction by using the owner's sent Ethereum. Another is a **receive()** function, which is to revert on any call to the contract with some Ether (but no information) such as calls made via send() or transfer().

To execute the attack, the attacker will try to make the *DoSAttack* contract become the last bidder to pay ether, making the demand larger than the supply according to the price at that time. This can be done via **commit** method and will cause the state of the contract to become *CLOSED.*



```
    if (totalEther >= tokenSupply.mul(curPrice * MULTIPLIER)) {
        clearingPrice = curPrice;
        currState = State.CLOSED;
        emit changeState(currState);
    }
```

Figure 2: Contract is closed when total Ether committed is enough to buy all the supplied tokens at current price

Afterwards, when anyone calls **releaseTokens()** of this contract, the Auction contract will pay back the redundant amount of Ether back to the *DoSAttack* contract. However, as mentioned before, the **receive()** method of this contract will revert and make the attempt to allocate the tokens to bidders obsolete. Besides, **releaseTokens()** method is also the only way to give back Ether back to the *CypherpunkCoin* contract. However, this attack makes this transfer impossible.

B. Gas Limit DoS on the Network via Unbounded Operations

As one may notice, in our **releaseTokens()** method, a loop through all the commitments is executed to distribute the tokens to the bidders of those commitments. However, this raises the concerns that because the gas executed for this method depends on the number of commitments, one attacker could attempt to increase the number of commitments to the auction such that the **releaseTokens()** methods will consume an amount of gas larger than the Block Gas Limit.

Such concern is valid given our observation from experimenting in Truffle test environments. Indeed, when we executed two attacks, one with 30 commitments and another with 300 commitments, the auctions containing these number of commitments consume 480,542 and 4,248,873 gases for releaseTokens method respectively. These numbers indicate that at about 1000 commitments, this method can take up to 12,500,000 gas, which is the current block gas limit on Ethereum [6]. (Note that gas cost increases linearly with the number of commitments) Therefore, with an estimated cost of 122,655 gases for a commitment (another result of our Truffle experiment), one mining pool can try to include only their commitment transactions with minimal amount (e.g. 1 Wei) in 10 consecutive blocks (1000 transactions /(12,500,000 gas limit per blocks/122,655 gas per transaction) ~ 10 blocks) to launch the attack, making the auction irrelevant.

C. *Token distribution domination through front-running*

Let's assume our capped supply for this auction is **S**. A pool wants to get **X Cypherpunk** coins and he only wants to use less than **Y ETH** for this attack. Therefore, the price the pool wishes the auction to close at is less than **P = Y/X.** The pool could do this in two ways:

1) If the duration of the auction is short, when auction starts, he can commit Y ETH first and use their substantial computational power to mine blocks without including bidding transactions from others to prevent the auction to close before price P. After the price reaches value P, it stops and allows the network to function normally again.

2) If the time for the price to drop to P seems to be too long and it is not economically advantageous for the pool to execute such an attack, it can actively observe the market. Indeed, if they have high connectivity in the network, they can analyze their mempool to see whether there exists a set of unconfirmed committing transactions that make the committed ether larger than S*P – Y at any time. If this is true, they can commit Y ETH before this set of transactions (and remove other committing transactions from the blocks via selfish mining before the price reaches P in case the price has not reached P yet) to ensure the auction ends at price less than P with their Y ETH committed. Otherwise, if such situation never happens, they could happily commit Y Ether when the price reaches P. Please note that the reason for the number S*P – Y is if the committed money is larger than this amount at some point and then the pool commits Y, there are two cases:

a. Case 1: The auction ends at price P' > P, then the pool gets Y/P' < Y/P = X tokens as planned.

b. Case 2: The auction ends at price P' <= P, then other bidders always get more than (S*P – Y) / P' >= (S*P – Y) /P = S – Y/P = S – X tokens, leaving the pool with less than S – (S – X) = X tokens.

This strategy also works in case the attacker is not a mining pool. In fact, instead of mining their own blocks to compete with other miners, they generate sufficient transactions with very high gas price to attract the miners to only include their transactions in their blocks.

There are many impacts that such an attack can have the token ecosystem:

1. As many projects use their issued tokens to vote on the key decisions towards software development, having one party controlling a big number of tokens makes the project easily affected by the party.

2. Some tokens are being used as Utility tokens, where one party controlling large number of tokens may lower usage of the tokens.

A similar attack already happened with Status.im ICO when the mining pool F2Pool transferred 100 Ether to 30 fresh addresses before Status.im ICO started. After this ICO started, the pool used their mining power (which accounts for 23% of the mining hash rate in Ethereum at that time) to include 31 transactions from the fresh addresses to the ICO smart contract and prevent others from sending transactions to the ICO. [8]

D. *DNS hijacking in Transport layer*

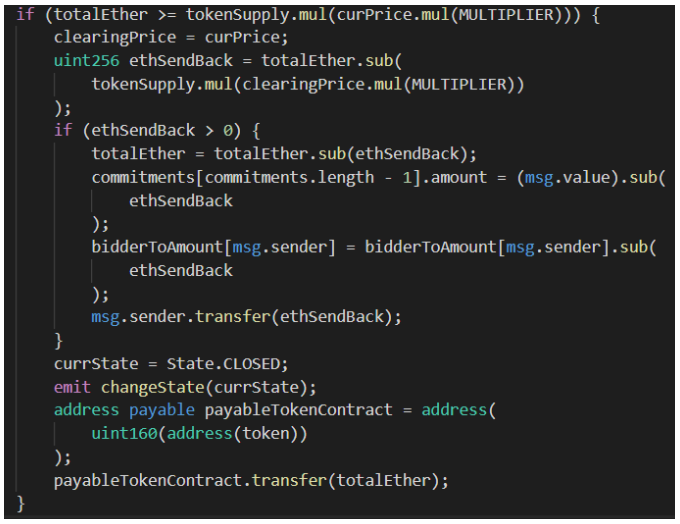
In our development project, we purely focused on maximizing security of our smart contract, as there are ways to interact with the contract from the command line. However, we did not build any protections for the internet. Once the app is loaded, there is no interaction with server, hence our client is less vulnerable to attacks on frequent interactions between client and servers. That said, there is still valid concern about attack on the first interaction between server and client when the web app is loaded. For example, there are DNS hijack attacks, where attackers can take control and redirect users to phishing sites. In DApp, it is possible that attackers redirect our domain address to their phishing server and users unknowing purchase from their site, sending their Ethereum tokens to unknown contracts. This is possible by taking control of BGP (Border Gateway Protocol) routers, which are essentially the highways of the internet. As they are designed for trusted dynamic changes such as outages, it can be hard to ensure that users are on the correct page and fall prey to such attacks.

IV. PROPOSED SOLUTIONS

A. *Solutions to Denial of Services Attack*

We made two changes to prevent the attacks.

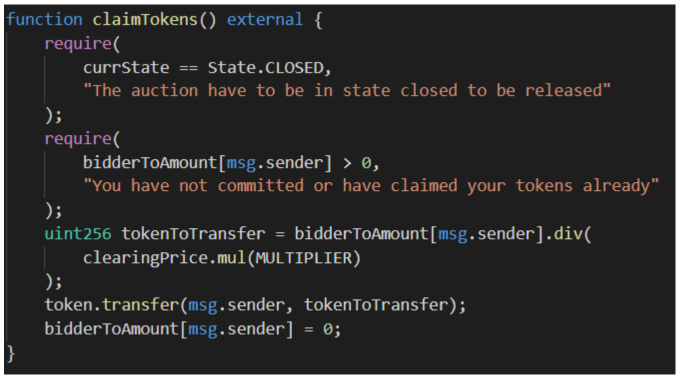
Firstly, we moved the logic for refund for last bidder up to the committing phase. By doing this, if the refund to the attacker fails, it only costs the gas spent by them and does not change the state of the Auction contract or other bidders' commitments.



```
if (totalEther >= tokenSupply.mul(curPrice.mul(MULTIPLIER))) {
    clearingPrice = curPrice;
    uint256 ethSendBack = totalEther.sub(
        tokenSupply.mul(clearingPrice.mul(MULTIPLIER))
    );
    if (ethSendBack > 0) {
        totalEther = totalEther.sub(ethSendBack);
        commitments[commitments.length - 1].amount = (msg.value).sub(
            ethSendBack
        );
        bidderToAmount[msg.sender] = bidderToAmount[msg.sender].sub(
            ethSendBack
        );
        msg.sender.transfer(ethSendBack);
    }
    currState = State.CLOSED;
    emit changeState(currState);
    address payable payableTokenContract = address(
        uint160(address(token))
    );
    payableTokenContract.transfer(totalEther);
}
```

Figure 3: The refund of tokens happens right after demand exceed supply

Secondly, to avoid GasLimit DoS, the release of tokens now is executed individually by *claimTokens()* method instead as a whole like before.



```
function claimTokens() external {
    require(
        currState == State.CLOSED,
        "The auction have to be in state closed to be released"
    );
    require(
        bidderToAmount[msg.sender] > 0,
        "You have not committed or have claimed your tokens already"
    );
    uint256 tokenToTransfer = bidderToAmount[msg.sender].div(
        clearingPrice.mul(MULTIPLIER)
    );
    token.transfer(msg.sender, tokenToTransfer);
    bidderToAmount[msg.sender] = 0;
}
```

Figure 4: Tokens can now be claimed individually

These corrections have been implemented and could be found in file contracts/AuctionAgainstAttack.sol in this link: https://github.com/sphades/CZ4153-Dutch-Contract/tree/main

B. *Proposal for commit-reveal scheme to solve Token Domination*

For the first attack, we can avoid it by pushing up the starting price or reducing the downward rate of the price, making the duration for the price to drop to P much longer. This increases the cost of blockstuffing at the start and diminishes the attacker's incentive to perform the attack.

For the second attack, we could not pre-define the length from the time the attack happens till when the price reaches P. The timing of the attack really depends on risk aversion of the miner/ non-miner and how the market behaves.

If the only type of attacker is a non-miner, this can be easily resolved. Because the attacker will try to raise the gas price for their commitment to gain advantage, a post by team OpenZeppelin [4] suggests that we may add a modifier called *gasThrottle* to the method *commit*, which will check whether gas price of the transaction is less than a number called MAX_GAS_PRICE. This would allow other bidders to try to send the transaction with the gas price close to MAX_GAS_PRICE without worrying about someone else pushing up the gas price.

However, the market's attackers also include miners and require a more comprehensive solution. The fact that the miner could perform the attack is because they get access to two information: the value of the commitments and how much ether that the auction has up till now. Therefore, a possible solution is to keep this information private. Based on the LibSumarine scheme [5], we propose the following commit-reveal scheme which totally changes the whole workflow of the auction. It comprises of 5 phases as follows:

1. Prepare: Bidders prepare their own commit address and transaction unlock.
2. Commit: Bidders send their money to the prepared fresh commit address in the defined commit duration.
3. Reveal and unlock: the bidder gives proof to the Auction contract that they commit to the commit address in the specified block range. Besides, he broadcasts the unlock transaction to release Ether from this commit address to the Auction contract. (reveal and unlock can happen in any order)
4. Prepare for finalizing: Based on all the reveal and unlock events emitted by the Auction contract, the owner of the auction sets the transaction that ends the auction. Besides, the clearing price, the last bidder and refunding amount to this bidder are also decided.
5. Finalize: the bidder calls the finalize method of the contract, either receives tokens if their commitment is in the commitment period and before the transaction ending the auction specified in step 4, or receives Ether back.

In this scheme, because the committed amount and address are well hidden thanks to k-anonymity set [7] (also, the total ether committed is anonymous in the commit period thanks to this) it is hard for miner to guess the right time to attack. Besides, although revelation and unlocking can be censored by attackers' nodes, we make the duration for this phase long enough to avoid bulk displacement/ censorship from them. The main security concern about this scheme is the owner of the contract who decides the main attributes (ending transaction, last bidder etc.) of the contract. However, because it is related to the reputation of the entity hosting this auction, we assume the owner is honest here.

An untested version of this scheme is now available at this repository:
https://github.com/cuongquangnam/DutchAuctionLibSubmarine

C. DNS hijacking in Transport layer

To address the lack of encrypted traffic communication between the server hosting the DApp and the browser, we can tackle this problem by obtaining a SSL certificate as well as making sure that our server hosting the DApp requires the use of HTTPS to allow the browser to access the DApp. Locally, we just need to modify the script to add HTTPS=true as well as adding an SSL certificate location to our start script.

As BGP is inherently vulnerable and continues to remain so, we can try to educate consumers to do due diligence to ensure that the addresses displayed on the interface matches the contract addresses used.

V. Conclusion

A chain is only as strong as the weakest link. In conclusion, this paper presented a thorough, but not exhaustive analysis on security aspects of our Dutch auction smart contract implementations. These aspects include some common attacks such as Denial of Services and much less obvious attack vectors (DNS hijacking) that other DApp developers should take note as well. Finally, we have also suggested solutions based off other work on mitigating the risks. Some of these solutions are already implemented while others are work in progress or depends on the current internet infrastructure.

References

[1] DutchX, *DutchX 1.0 Audit Report*. Accessed on: Nov 26 2020. Available: https://dutchx.readthedocs.io/en/latest/_static/docs/DutchX_1.0_Audit%20Report.pdf

[2] Brewster, T., 2020. *A $152,000 Cryptocurrency Theft Just Exploited A Huge 'Blind Spot' In Internet Security. Forbes*. Accessed on: 27 Nov 2020 Available: https://www.forbes.com/sites/thomasbrewster/2018/04/24/a-160000-ether-theft-just-exploited-a-massive-blind-spot-in-internet-security/?sh=4f5dde985e26

[3] Consensys, *Ethereum Smart Contract Best Practices*. Accessed on: Nov 25, 2020. Available: https://consensys.github.io/smart-contract-best-practices/known_attacks/

[4] OpenZeppelin, *Protecting Against Front-running and Transaction.* Accessed on: Nov 26 2020. Available: https://forum.openzeppelin.com/t/protecting-against-front-running-and-transaction-reordering/1314

[5] IC3, *LibSubmarine*. Accessed on: Nov 26 2020. Available: https://github.com/lorenzb/libsubmarine

[6] CoinDesk, *Ethereum Miners Vote to Increase Gas Limit, Causing Community Debate*. Accessed on: Nov 26 2020. Available: https://cointelegraph.com/news/ethereum-miners-vote-to-increase-gas-limit-causing-community-debate

[7] Lorenz Breidenbach, Phil Daian, Ari Juels, Florian Tramer, *To Sink Frontrunners, Send in the Submarines.* Accessed on: Nov 26 2020. Available: https://hackingdistributed.com/2017/08/28/submarine-sends/

[8] Shayan Eskandari, Seyedehmahsa Moosavi, Jeremy Clark, *SoK: Transparent Dishonesty: Front-running Attacks on Blockchain* . Accessed on: Nov 27 2020. Available: https://arxiv.org/pdf/1902.05164.pdf

[9] Arxiv.org. 2020. *Proteus: A Scalable BFT Consensus Protocol For Blockchains*. Accessed on: 27 Nov 2020. Available: https://arxiv.org/pdf/1903.04134.pdf