Security In Decentralised Applications: Decentralised Domain Registrar Security Vulnerabilities and Mitigations

Lim Han Quan
U1721115K
School of Computer Science and Engineering
Nanyang Technological University
Singapore
hlim041@e.ntu.edu.sg

Ng Kai Chin
U1721647D
School of Computer Science and Engineering
Nanyang Technological University
Singapore
kng054@e.ntu.edu.sg

Tan Zhen Yuan
U1721017F
School of Materials Science and Engineering
Nanyang Technological University
Singapore
ztan055@e.ntu.edu.sg

***Abstract*—**This paper examines common vulnerabilities and security issues in smart contracts running on the Ethereum Virtual Machine (EVM). It discusses relevant issues related to, and in the context of, a Decentralised Domain Registrar dApp. It then closely investigates the potential repercussions of such security issues within the dApp and proposes tailored solutions to each issue.

Keywords—blockchain, Ethereum, smart contracts, tokens, blind auction, decentralized domain registrar

I. Introduction

Blockchain is a distributed system with the design goals of resource sharing, distribution transparency, openness, and scalability. It has seen use in cryptocurrencies, smart contracts, healthcare, and domain names amongst many others. Consistency and replication are important in blockchain to achieve consensus, reliability, and performance. These qualities depend on the security, privacy, and scalability of blockchain – security protects crypto assets and blockchain function from attacks, privacy grants users with selective disclosure of information, and scalability focuses on the ability of the system to handle a growing amount of work by adding resources to the system.

Our dApp (Appendix C) is a domain registrar service for ".ntu" domains, which users interact with via a simple front-end website. The primary purpose of the `DomainRegistry` contract is to resolve domain names to their respective owner addresses. Domains are bidded for using a `BlindAuction` system. For each auction, `DomainRegistry` instantiates a `BlindAuction` contract. The auction process can be broken into three stages – `Bid`, `Reveal`, and `Claim`.

In the `Bid` stage, bidders submit bids which are blinded. This means that the information is hashed and is unknown to other parties until after the `Bid` stage. Each bid consists of a `value`, `fake flag`, `secret` and `deposit`. The `deposit` is the Ether sent together with the transaction. For a bid to be *valid*, the `deposit` must be greater or equal to the `value`, and the `fake flag` must be set as `false`.

In the `Reveal` Stage, bidders reveal their bids by submitting plaintext versions of their bidding information as proof. When there is a valid bid that is the current highest bid revealed, it will be stored as the `currentHighestBid`. All other bids are refunded. At the end of the `Reveal` stage, the highest bid `currentHighestBid` and its corresponding bidder `currentHighestBidder` will win the auction. If there are higher bids that are unrevealed, they do not win. Unrevealed bids after the `Reveal` stage can also not be refunded and are left in the `BlindAuction` contract.

In the `Claim` stage, only the `highestBidder` can claim the domain name. `BlindAuction` will send the `bidderAddress` to the `DomainRegistry` to permanently register the domain name under the `bidderAddress`. This information can be subsequently used to resolve the domain name to the `ownerAddress`.

This paper will focus on security vulnerabilities in the project, as well as ways to mitigate these issues, contributing to the Ethereum dApp development community a baseline for future similar decentralized domain registrar dApps.

II. Motivation and Literature Survey

A. Motivation

Security can be likened to a game of chase – attackers are constantly on the lookout for opportunities to exploit, while defenses are constantly placed up to fix vulnerabilities whether proactively or reactively. Security issues with blockchain have historically resulted in large losses, such as the DAO hack in 2016 which saw the loss more than 2M of Ether (valued at US$40-55M) after a weakness in the DAO code was exploited (Siegel, 2016). For greater usage and adoption of our dApp), it is of utmost importance to give users the peace of mind needed for them to trust the dApp sufficiently; this comes in the form of safeguarding the dApp from attacks for fair use. In an auction-based domain registrar application, naturally, security is of primary concern as users use the application to send and receive Ether. The application could be likened to a bank. Users would want the full process, from bidding, claiming of a domain name, to using the domain name to receive Ether, to be secure and hardened against attacks.

Ethereum applies an account-based model so addresses are more easily linkable, a key consideration in protecting privacy. Additionally, privacy issues in Ethereum are increasingly commonly attributed to its users as well, rather than inherent issues in dApps. For example, instead of limiting usage of their accounts and refraining from publishing identifiable information publicly, users commonly publicize their Ethereum Name Service (ENS) names on their social media profiles due to a lack of knowledge on best practices to preserve privacy; B´eres et al (2020) were able to link over 1.1 million transactions to more than 4,200 addresses. There is hence an increasing need for passive protection of user privacy than to rely merely on active efforts from users. However, mitigations in this domain largely require modifications to layers below the contract layer for Ethereum, or require external services to work in tandem with our decentralized domain registrar, which is beyond the scope of this project.

Ethereum suffers a bottleneck in scalability due to every node in the network having to process each transaction; nodes verify the work of miners, act as checks to malicious behavior, and keep a copy of the current network state. The scalability trilemma claims that we cannot achieve a blockchain with the following all three properties simultaneously: Decentralization, High transaction throughput, Security (Ethereum Wiki, n.d.). However, as a proof-of-concept application, which is presumably only meant for use within NTU, scalability is not an urgent issue.

As such, we focus on security as the area of main concern in this paper, with analysis and mitigations involving only the contract layer of the Ethereum blockchain.

B. *Literature Survey*

While security attacks can be mounted on the different strata of blockchains, we will only be analyzing contract layer vulnerabilities, limited to those that affect dApps similar to our decentralized domain registrar.

1) Reentrancy Attacks

Reentrancy occurs due to race conditions – In Ethereum, a contract function may be called rapidly multiple times, the function may not complete execution before it is called again. If the execution flow of the function depends on a state change caused by a previous invocation, the second invocation my not execute as intended if this race condition is not accounted for, e.g. withdrawing token balances multiple times before the balance is set to zero, as seen in the example below (Foster, 2018):

```
mapping (address => uint) private userBalances;

function withdrawBalance() public {
    uint amountToWithdraw =
userBalances[msg.sender];
    msg.sender.transfer(amountToWithdraw);
    userBalances[msg.sender] = 0;
}
```

2) Frontrunning Attacks

Frontrunning attacks are conducted by miners, who could reorder transactions. These attacks take on forms that fall into three main categories: displacement, insertion, and suppression.

Displacement occurs during auctions, when attackers add the minimum necessary to the current highest bid to just outbid it, thereby displacing the original highest bidder as the current highest bidder. Insertion occurs when miners insert transactions which were not originally present into blocks. Suppression, also known as block-stuffing, occurs when miners fill up blocks with transactions or irrelevant data to suppress a transaction.

Frontrunning attacks are likely to happen due to the competitive nature of auctions.

Historically, frontrunning attacks have occurred in decentralized exchanges (DEX), such as during initial coin offerings (ICO), where attackers eliminate central parties and substitute them with new decentralized untrusted partners with the same functionality, and reordered transactions. This is the central focus of the research paper *Flash Boys 2.0* by Daian et al (2019), which studied frontrunning, transaction reordering and consensus instability in DEX. Such actions have resulted in trade rates being worse than when traders send in transactions, gasPrice rocketing during ICO, and prioritization of certain transactions over others.

3) Unexpected Reverts

Unexpected reverts exploit the ability of child contract functions to propagate errors to external parent contract calls, as well as the ambiguity of entity types residing at addresses. In this vulnerability, a target contract assumes that Ether transfer to an address will succeed (or not cause a revert) when it is entirely possible that at the address resides a malicious contract containing a fallback function that will throw a revert. This revert is propagated to the target contract which halts the execution of the caller function. Should crucial state change or operations be required to occur after the transfer in the caller function, the contract may become paralyzed and deny service to subsequent users.

```
contract InsecureAuction {
    address currentLeader;
    uint highestBid;

    function bid() payable {
        require(msg.value > highestBid);

        require(currentLeader.transfer(highestBid));

        currentLeader = msg.sender;
        highestBid = msg.value;
    }
}
```

In the above simple example from ConsenSys Diligence (2020), the attacker may have bid using a proxy contract to obtain the position of `currentLeader`. Then in subsequent calls to bid by other users, if the malicious proxy contract contains a fallback function that reverts, a crucial state change to update the `currentLeader` can no longer occur since the revert from the transfer call is propagated up to `bid`, effectively paralyzing the contract and denying service to other users. Other variants include having such transfer calls in loops which may cause the loop execution to stall (ConsenSys Diligence, 2020) due to an unexpected revert.

The `HYIP` contract, a Ponzi scheme, contained the unexpected revert vulnerability in its `performPayouts` function (Chen et al, 2019), where the function loops over its investors during payouts. If there exists a malicious investor, the investor could indefinitely stall the payout operation with an unexpected revert with a similar mechanism described for the simple auction example, then demanding a ransom to stop the attack.

4) Attacks involving Gas

Gas is an important feature of the Ethereum Blockchain. It can be conceptualized as a execution fee that quantifies the amount of computation required for a transaction. Individual blocks have a limit on the gas that it can contain. Transactions that exceed this gas limit will fail with an *out-of-gas* exception, resulting in the whole transaction failing. Also, Ethereum wallets typically set a gas limit for users.

There are 2 types of attacks concerning gas that concern our dApp: exploitation of the gas limit, and insufficient gas griefing (ConsenSys Diligence, 2020).

Exploitation of the gas limit can result in Denial of Service (DoS), which falls under CWE-400: Uncontrolled Resource Consumption (MITRE, 2020). Gas limit DoS often exploits unbounded operations. For example, if there is no limit on how many times a certain section of code can be executed (e.g. a loop potentially executing an unbounded number of times), then an attacker can cause the code execution to be sufficient long such that the gas required for that computation exceeds the limit, and thus fails.

An example of this is in the *GovernMental* smart contract, which is designed primarily as a Ponzi scheme (Governmental, 2016). Participants join a scheme by sending Ether. When there are no new participants for 12 hours, the last participant gets the whole pot. *GovernMental* stores the list of users in an array, but the flawed function to 'reset' the array after each pot merely sets each element to '0' (instead of initializing a new array). Once the users array grows to a certain length, the reset function would be too computationally expensive to execute in a single transaction, thus creating an *out-of-gas* exception and resulting in the contract being stuck in that state forever (Bartoletti et al, 2017). An attacker could sabotage the smart contract by creating a large number of users to bloat the participant array.

Another variation of Gas limit DoS is via block stuffing. An attacker can stop other transactions from being executed on the Ethereum blockchain by added computationally intensive transactions which have a high gas price (ConsenSys Diligence, 2020). These expensive transactions will be prioritized by miners. If these transactions take up the remaining gas limit of the next block to be mined, the attacker can potentially deny any other transactions to be executed for the next block. An example is the *FOMO3D* prize, which was worth millions of dollars. The game rules are very similar to *GovernMental's*. It was successfully claimed via such a block stuffing attack. The winner used a block stuffing attack to reduce the number of transactions executed, thereby increasing his chances of winning (SECBIT, 2018). However, mounting such an attack is costly, and in the context of domain name registrar dApps, it is highly unlikely that the domain names are valuable enough of a commodity to incentivize such an attack.

Another category of attacks involving gas is insufficient gas griefing attacks. The prerequisite is for the vulnerable function to contain a call to another function ('sub-call'). This was first reported as a vulnerability for the *Gnosis Safe v.1* contract (Solidified Technologies, 2019). The vulnerability occurs when the transaction is given insufficient gas to execute the sub-call, but still sufficient to execute the rest of the original function.

Below is an adaption of an example by Sandford (n.d.):

```
contract Executor {
    function execute(address callee, bytes calldata
data, uint256 gas) external {
        (bool success, bytes memory returnData) =
callee.call.gas(gas)(data);
        // do something
    }
}
```

There is no guarantee for `callee` to receive sufficient gas from the execute function. Nevertheless, due to EIP-150 (EIP-150, 2020) which states that callers are always left with at least 1/64 of the gas left, the `execute` function could still be left with enough gas to complete its execution. Since there is no requirement for the success of the `callee` call, this potentially means that the call can be skipped.

The most common solutions to insufficient gas griefing attacks are requiring the original function (in this case, `execute`) to provide enough gas through a `require` statement (ConsenSys Diligence, 2020):

```
require(gasleft() >= gas);
```

However, this solution is not sufficient because the statement itself will consume some gas, and also that 1/64 of the gas left will remain in the original call.

One of the most complete solutions currently checks for the gas amount after the sub-call has been made, and ensures that the gas left is more than 1/63rd of the gas used in that sub-call transaction (Sanford, n.d.). This ensures that the `callee` call did not use up all of the gas available to it (i.e. 63/64 of the gas left) – and thus had sufficient gas to execute. The solution code snippet adapted from Sanford (n.d.) is as follows:

```
callee.call.gas(txGas)(data); // call
assert(gasleft() > txGas / 63); // "insufficient
gas"
```

III. Observations and Analysis

Upon inspection, our dApp had some defenses already in place for the vulnerabilities found during the literature survey.

For example, reentrancy attacks, which could have been mounted by repeatedly calling the `reveal` function to withdraw deposits multiple times, was patched by storing the bids to validate in memory and flushing the bids stored for the bidder before performing the validation (which required the calling of another function, `checkIfHighestBid`). This followed the *checks-effects-interactions* pattern (Marx, 2019).

However, some vulnerabilities remain unpatched. The mechanism of these vulnerabilities (in the context of our dApp) will thus be further explained in detail in this section. The

solutions for each of these vulnerabilities will be discussed later in the paper.

These unpatched vulnerabilities involve frontrunning attack during the `Bid` stage, a DoS caused by an unexpected revert attack, and gas-related attacks involving using insufficient gas and exceeding gas limits.

A. *Frontrunning Attacks in case of Bid Stage of BlindAuction Contract*

Our dApp blinds and obfuscates bids by hashing (using *Keccak256*), including the possibility of fake bids and allowing invalid bids (`deposit` < `value`). Despite these strategies, there are two key pieces of information which are still known: the destination address of the bids, which correspond to a known `BlindAuction` addresses, and the amount of Ether sent in the transaction (i.e. the `deposit` of the bid).

As such, malicious actors could still carry out frontrunning attacks. For displacement, an attacker could outbid the highest `deposit` sent, even if the highest `deposit` might belong to an invalid bid. Likewise, insertion and suppression attacks can be carried out using the `deposit` values as reference.

Thus, the blinded bids only make the attacks less efficient and potentially more expensive (i.e. where the highest `deposit` is an invalid bid) than in a traditional open auction, but importantly do not prevent the possibility of frontrunning attacks.

B. *Unexpected Revert DoS Attack in Balance Withdrawals*

Balance withdrawals are a common design requirement in many smart contracts. In the case of a decentralized domain registrar, deposits are withheld for a bid should the bid be seen as a potential winning bid of the auction. This deposit is only returned if another bid is revealed to be of higher value, hence there is a need for funds transfer back to the bid owners.

This transfer, unless handled properly, opens the contract to a DoS attack caused by an unexpected revert. If state changes need to be made after the transfer, such as updating the highest bidder, this unexpected revert stalls further execution of the contract function, effectively paralyzing the contract and causing a DoS on subsequent users of the contract, preventing further advancement of the auction. This vulnerability is made possible when an auction contract assumes an EOA or wallet is the caller of this fund transfer function, when in reality no such assertion can be made as contracts can perform such contract calls too.

In the context of our dApp, tight coupling between this deposit transfer logic and the logic to check for the highest bid exposes the application to such an attack. The functions involved are `reveal` and `checkIfHighestBid`. `reveal` handles the checking of bid validity, as well as refunds for unsuccessful bids, while `checkIfHighestBid` handles highest bid value checking and highest bidder updating [Appendix A and B].

Given the current implementation, an exploit for the identified vulnerability can be carried out as such to prevent refunds to the highest bidder in the event of an ousting of the previous highest bidder:

1. A malicious bidder Alice would set up a proxy contract with 3 functions – a bid and reveal function to proxy a wallet calling the auction contract, and a fallback function in which its only logic is to throw an error.
2. Alice uses the proxy contract to place and reveal her bids. Suppose she places the highest bid at some point in the reveal stage of the auction, allowing her to be placed as the `highestBidder`.
3. Mallory, an honest bidder, reveals a higher valid bid, this causes the contract to run logic to refund Mallory's deposit for the highest bid in `checkIfHighestBid`.
4. Since Alice has been interacting with the auction contract with a proxy contract, the auction contract attempts to transfer the deposit back to the proxy, calling the fallback function and inducing a throw. This causes the `checkIfHighestBid` function to halt execution and consequently prevents the refunds transfer logic at the end of the `reveal` function from being executed, blocking Mallory's refund of her other bids or any excess deposit (i.e. `deposit` − `value`) for that winning bid. Mallory may also not call for a subsequent reveal since records of all her blinded bids would have been cleared, causing her to lose her deposits, other than that of the winning bid, forever.

C. *Gas-Based Attacks in case of Application as a Whole*

1) *Gas limit DoS via unbounded operations*

Similar to the *GovernMental* (Governmental, 2016) contract, `DomainRegistry` keeps 2 ever-growing mappings: `records` and `ownerToDomain`. An attacker can start auctions and claim domains for many obscure domain names. This will greatly increase the size of the mappings stored in the `DomainRegistry`. Even if there is no malicious actor, these mappings will still grow steadily over time because there is no expiry for domains.

If enough domain names are claimed, the gas cost of executing `getRegisteredDomains` and `getOwnedDomains` functions will exceed the maximum gas limit and fail since these 2 functions iterate over the mappings. The unbounded size of these mappings thus implies an unbounded execution time for the operation of these 2 functions.

Eventually, this may prevent the front-end website from retrieving the state of `DomainRegistry` (namely, the domain names and owners) and will fail to function as intended should these executions require a gas cost above the gas limit.

2) *Griefing via insufficient gas*

There are a few functions in `BlindAuction` in which execution is delegated to sub-call, and thus might be vulnerable to insufficient gas griefing. However, not all cases are exploitable in practice. In `reveal()`, there is the following command at the end of the function:

```
msg.sender.transfer(refund);
```

But it is illogical for an attacker to exploit this as providing insufficient gas to execute this sub-call would only prevent themselves from obtaining their refunds. Another such sub-call is in `checkIfHighestBid`. The code snippet is shown below:

```
1  if (highestBidder != address(0)) {
2   // Refund the previously highest bidder.
3    address payable previousHighestBidder
4        = highestBidder;
5    uint previousHighestBid = highestBid;
6
7    highestBid = value;
8    highestBidder = bidder;
9
10   previousHighestBidder.transfer
11       (previousHighestBid);
12  }
13  ...
14  return true;
```

In lines 10-11, the sub-call to `previousHighestBidder` does not benefit the sender in any way, and thus incentivizes attacks to avoid execution of that command. By providing insufficient gas to execute lines 10-11, but while still having sufficient left to execute the rest of the function (i.e. line 14), an attacker can launch a griefing attack.

IV. Proposed Solutions

A. Solution to Frontrunning Attacks

The current implementation of our dApp prevents to some extent the displacement variant of frontrunning attacks by way of hiding valid bids amongst potentially invalid bids. However, the intent of each transaction can still be partially deduced by its transaction value, as well as its target address. Thus, this information leak still exposes the vulnerability of the application to other variants of frontrunning attacks.

Following further with the paradigm of security by k-anonymity, submarine commitments (Breidenbach et al, 2018) could be utilized by bidders in the bid stage to conceal the intent of their transactions. This technique consists of 4 main stages (Prepare, Commit, Reveal and Unlock) and 3 interacting parties (the user, a submarine address, and the target contract).

A submarine address is a randomly generated address prepared by a user such that the user can transfer funds to but does not hold the private key to. This private key is held by the target contract instead. As such, the user can 'commit' funds to the address, but only the target contract can 'unlock' these funds after the user has 'revealed' it to the target contract.

In the context of our dApp, a decentralized domain registrar, there need only be some modification to the existing implementation. Blinded bid commitments will no longer be made directly to the blind auction contract. Instead, the bidder would prepare a submarine address before commitment and commit the blinded bid, along with the Ether deposit, to the submarine address. Also, the `Reveal` stage would have the `BlindAuction` contract verify the validity of bids from the submarine addresses, rather than records stored within its state. Finally, in the blind auction `Claim` phase (analogous to the Unlock stage of a submarine send), committed funds could then be withdrawn by the `BlindAuction` contract and, if the bids did not win, be refunded by the `BlindAuction` contract to respective bidders. An easy to integrate implementation for Ethereum contracts, LibSubmarine (Breidenbach et al, 2018), has been identified as ideal for modifying the current dApp implementation. Finally, to ensure a fair reveal process to prevent frontrunning attacks of the insertion variant, the winner of an auction could be randomized should there be equivalent highest bids.

Since all bids are made known only in the `reveal` stage, assuming a fair reveal process where reveal order does not determine an auction winner, frontrunning attacks of the insertion or suppression type cannot be carried out as bids made in the `bid` phase (analogous to the Commit stage of a submarine send) are indistinguishable from regular transactions on the Ethereum network. This affords no information to malicious parties to properly carry out common frontrunning techniques, such as 'out pricing' targeted transactions or block stuffing (Solmaz, 2018).

B. Solution to Unexpected Revert DoS Attack

While the identified vulnerability, in our context, can only be exploited by an attacker on a single subsequent party, the attacker may set up many such proxy contracts to exploit multiple users on the same or multiple auction contracts with properly placed bid amounts. Attempts to prevent a complete DoS were made by simply making the transfer of Ether to the previous highest bidder occur only after updating the highest bidder, thus preventing a highest bidder 'lock'. However, this still does not prevent a refund 'lock' to the new highest bidder.

To remedy the mentioned hanging vulnerability, a 'pull' fund withdrawal pattern could be implemented (Chen et al, 2019) – the blind auction contract could have its fund transfer call for the previous highest bidder be moved to a separate function, requiring the previous highest bidder to call the function on their own during the `Claim` stage. In its place in the `checkIfHighestBid` function, the following logic and data structure to store withdrawal balances for later withdrawal can be implemented:

```
// withdrawal balances data structure
mapping(address => uint) withdrawalBalances;

// logic for storing balances
withdrawalBalances[previousHighestBidder] +=
previousHighestBid;

// separate function for withdrawing balance
function withdraw() public onlyAfter(revealEnd) {
  uint value = withdrawalBalances[msg.sender];
  withdrawalBalances[msg.sender] = 0;
  msg.sender.transfer(value);
}
```

This separation prevents an attacker from using the beforementioned exploit from locking up refunds to the new highest bidder since any error thrown from the proxy contract is self-contained in the separated `withdraw` function.

C. Solution to Gas-Based Attacks

1) Gas Limit DoS via unbounded operations

The key reason for the vulnerability arises from the lack of minimum bid values during auctions, making it relatively cheap to carry out the DoS. As a workaround, `BlindAuction` could set a minimum bid value, such that it will cost a significant amount

of Ether to buy a large number of (arbitrary) domain names. The implementation for this is simply to initialize the `highestBid` variable to have the minimum bid value desired, as shown below:

```
uint public highestBid = <minimum_bid_value>;
```

Then, in the `checkIfHighestBid` function, if there is no current highest bid, the following existing command will discard any bids which are lower than the minimum:

```
if (value <= highestBid) { return false; }
```

The solution does not directly fix the vulnerability, but it makes it impractical and costly for an attacker to mount such an attack.

2) Insufficient gas griefing

The vulnerability in the `checkIfHighestBid` function would be removed through implementing the 'pull' withdrawal pattern mentioned in *B* (to mitigate unexpected revert attacks). This removes the external address call from the function and thus does not leave any exploitable call function by insufficient gas.

V. Conclusion

This paper highlighted the security vulnerabilities, namely frontrunning, unexpected reverts, and gas-related vulnerabilities, as well as their mitigations, within the contract layer of a decentralized domain registrar dApp meant for deployment on the Ethereum network. While this paper is far from exhaustive in elucidating the multitude of vulnerabilities present in dApps similar to the one developed in this paper, it still presents a development baseline for which other similar applications can follow, since the vulnerabilities are easily remedied with the solutions presented.

However, the vulnerabilities mentioned in this paper were manually identified, this further work should be conducted in this area, possibly with more sophisticated analytical tools to automate the identification of more of such common vulnerabilities (Perez and Livshits, 2020).

VI. References

[1] Siegel, D. (2016). Understanding The DAO Attack. Retrieved from: https://www.coindesk.com/understanding-dao-hack-journalists

[2] B´eres, F., Seres, I. A., Bencz´ur, A. A. and Quintyne-Collins, M. (2020). Blockchain is Watching You: Profiling and Deanonymizing Ethereum Users. *Institute for Computer Science and Control (SZTAKI) 2020*.

[3] Ethereum Wiki (n.d.). On sharding blockchains FAQs. Retrieved from: https://eth.wiki/sharding/Sharding-FAQs

[4] Foster, J. (2018). Common attacks in Solidity and how to defend against them. Retrieved from: https://medium.com/coinmonks/common-attacks-in-solidity-and-how-to-defend-against-them-9bc3994c7c18

[5] Daian, P., Goldfeder, S., Kell, T., Li, Y., Zhao, X., Bentov, I., Breidenbach, L. and Juels, A. (2019). Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges. *arXiv preprint arXiv:1904.05234*.

[6] ConsenSys Diligence (2020). Known Attacks. Retrieved from: https://consensys.github.io/smart-contract-best-practices/known_attacks/

[7] Chen, H., Pendleton, M., Njilla, L., and Xu, S. (2019). A Survey on Ethereum Systems Security: Vulnerabilities, Attacks and Defenses. *arXiv:1908.04507*.

[8] Narayanan, A., Bonneau, J., Felten, E. W., Miller, A., and Goldfeder, S. (2016). Bitcoin and Cryptocurrency Technologies – A Comprehensive Introduction. *Princeton University Press 2016*.

[9] Bonneau, J., Miller, A., Clark, J., Narayanan, A., Kroll, J. A., and Felten, E. W.. (2015). SoK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. *IEEE Symp. on Security and Privacy 2015: 104-121*.

[10] Eyal I., and Gun Sirer, E. (2014). Majority is not enough: Bitcoin mining is vulnerable. *Financial Cryptography, 2014, pp. 436–454*.

[11] MITRE Corporation (2020). CWE-400: Uncontrolled Resource Consumption. Retrieved from: https://cwe.mitre.org/data/definitions/400.html

[12] Chen, T. et al (2020). An Adaptive Gas Cost Mechanism for Ethereum to Defend Against Under-Priced DoS Attacks. *arXiv preprint arXiv:1712.06438v2*.

[13] GovernMental (2016). Retrieved from: http://governmental.github.io/GovernMental/

[14] Bartoletti, M., Carta, S., Cimoli, T., Saia, R. (2017). Dissecting Ponzi schemes on Ethereum: identification, analysis, and impact. *arXiv preprint arXiv:1703.03779*.

[15] Solidified Technologies (2019). Bug Description - Gnosis Safe v.1. Retrieved from: https://web.solidified.io/contract/5b4769b1e6c0d80014f3ea4e/bug/5c83d86ac2dd6600116381f9

[16] Sandford, R. (n.d.). Ethereum: The Concept of Gas and its Dangers. Retrieved from: https://ronan.eth.link/blog/ethereum-gas-dangers/

[17] EIP-150 (2020). Retrieved from: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-150.md

[18] Marx, S. (2019). Stop Using Solidity's transfer() Now. Retrieved from: https://consensys.net/diligence/blog/2019/09/stop-using-solidittys-transfer-now/

[19] Adler, J. (2019). Security and Scalability in Committee-Based Blockchain Sharding. *Medium 2019.* Retrieved from: https://medium.com/@adlerjohn/security-and-scalability-in-committee-based-blockchain-sharding-58fab3901193

[20] SECBIT (2018). How the winner got Fomo3D prize — A Detailed Explanation. Retrieved from: https://medium.com/coinmonks/how-the-winner-got-fomo3d-prize-a-detailed-explanation-b30a69b7813f

[21] Buterin, V. (2016). Privacy on the Blockchain. *Ethereum Blog on Research & Development 2016*.

[22] Praitheeshan, P., Pan, L., Yu, J., Liu, J., and Doss, R. (2020). Security Analysis Methods on Ethereum Smart Contract Vulnerabilities — A Survey.

[23] Breidenbach, L., Kell, T., Gosselin, S., and Eskandari, S., (2018). Libsubmarine: Defeat frontrunning on ethereum. Retreived from : https://libsubmarine.org/

[24] Solmaz, O. (2018). The anatomy of a block stuffing attack. Retreived from: https://osolmaz.com/2018/10/18/anatomy-block-stuffing/

[25] Perez, D., and Livshits, B. (2020). Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited.

VII. APPENDICES

APPENDIX A – ORGINAL REVEAL FUNCTION IMPLEMENTATION

```
    function reveal (
        uint[] memory _values,
        bool[] memory _fake,
        bytes32[] memory _secret
    )
        public
        onlyAfter(biddingEnd)
        onlyBefore(revealEnd)
    {
        uint length = bids[msg.sender].length;
        require(_values.length == length);
        require(_fake.length == length);
        require(_secret.length == length);

        uint refund = 0;
        Bid[] memory tempBids = bids[msg.sender];
        delete bids[msg.sender];

        for (uint i = 0; i < length; i++) {
            Bid memory bidToCheck = tempBids[i];
            (uint value, bool fake, bytes32 secret) =
                    (_values[i], _fake[i], _secret[i]);
            if (bidToCheck.blindedBid != keccak256(abi.encodePacked(value, fake, secret))) {
                continue;
            }
            refund += bidToCheck.deposit;
            if (!fake && bidToCheck.deposit >= value) {
                if (checkIfHighestBid(msg.sender, value)) {
                    // if the bid is the current highest, then do not refund it.
                    refund -= value;
                }
            }
        }
        msg.sender.transfer(refund);
    }
```

APPENDIX B – ORIGINAL CHECKIFHIGHESTBID FUNCTION IMPLEMENTATION

```
    function checkIfHighestBid(address payable bidder, uint value) internal
            returns (bool success)
    {
        if (value <= highestBid) {
            return false;
        }
        if (highestBidder != address(0)) {
            address payable previousHighestBidder = highestBidder;
            uint previousHighestBid = highestBid;

            highestBid = value;
            highestBidder = bidder;

            previousHighestBidder.transfer(previousHighestBid);
        } else {
            highestBid = value;
            highestBidder = bidder;
        }

        return true;
    }
```

APPENDIX C – LINK TO CODE REPOSITORY FOR DECENTRALISED DOMAIN REGISTRAR DAPP

For reference: https://github.com/ngkc1996/ethereum-dapp