UROP1000 Report 2021 Summer

Name: Yu Cheuk Hei

SID: 20522432

Supervisor: Prof. CHEUNG Shing Chi

Project Title: Building a Blockchain and Smart Contract application

ABSTRACT

As Ethereum introduces its new blockchain together with a Turing-complete programming language, users can write smart contracts and deploy decentralized applications freely. Smart contracts are a type of Ethereum account that run as automated programs on the blockchain. Due to blockchains' immutability, no amendments can be made to fix security flaws in smart contracts after deployment. A simple vulnerability may sometimes result in large amount of stolen funds. This project identifies one of the many vulnerabilities – Displacement attacks in front-running by sampling past transactions. This report provides explanation on how the vulnerability was located and how analysis was conducted. It also contains recommendations on how smart contract engineers can avoid similar mistakes in the future.

1. INTRODUCTION

The concept of front-running originates from traditional financial markets [1]. In traditional markets, brokers serve as intermediaries between investors and security exchanges and perform arbitrage trading by leveraging private information such as clients' pending orders. Similar applies to cryptocurrency markets where miners serve as intermediaries. The only difference is that there are laws regulating the traditional market but not the cryptocurrency market. Although front-running has been a pervasive issue on public blockchains for years, there are no fixes to the problem [1]. Currently, the best practice is to avoid writing smart contract codes that involve transaction ordering dependencies [1]. Some users may prefer using other platforms, such as Avalanche, that have taken front-running into consideration when designing its consensus protocol. The two major ideas in Avalanche, repeated-subsampling and transitive voting, avoided the situation where a single miner dictating the next block [2]. However, the problem of front-running remains inevitable for Ethereum users unless they switch to other platforms or Ethereum changes its consensus protocol.

In this project, I focused on one of the three taxonomies in front-running, displacement attacks. An example of displacement attacks is that an attacker steals a solution to a puzzle from the victim and submits it with a gas price much higher than the victim's transaction. Since transaction fees are one source of miners' income, there is always a larger incentive for miners to include transactions with higher gas price first. By increasing the gas price to a higher value, the attacker ensures that its transaction appears before the victim's transaction.

A recent research conducted by PhD candidates from the University of Luxembourg have already investigated the prevalence of front-running in the cryptocurrency market [3]. According to their results, 2983 displacement attacks from 49 unique attacker accounts were

identified and a total of more than 4.5M USD were involved. However, no explanation was provided on how attackers exploited the vulnerability and why these contracts were vulnerable. This project works on top of their results, analyzes the call trace of all the 2983 displacement attacks and aims to provide a simple security checklist on areas that smart contract engineers should be aware of when dealing with transaction ordering dependency issues. Further, I suspect that most of the victim transactions failed due to duplicated transaction inputs.

2. METHODOLOGY

I started by reviewing previous literatures and reproducing results obtained by their research [3]. Then, I focused on displacement attacks and tried collecting the vulnerable contracts behind each attack. My sampling method works on top of 3 heuristics.

Heuristic 1: If there are overlaps in the input of internal calls within a transaction, I focus on the target address of the inner call and investigate that contract. Otherwise, I assume that the internal calls have no relationship with the vulnerability since attackers perform displacement attacks according to victim transaction inputs. The method of detection in Part 4.2 Detecting Displacement of the Frontrunner Jones research paper [3] also identifies displacement attacks by comparing transaction inputs using bloom filters. Therefore, it is reasonable to have the same assumption when scanning through internal transaction calls.

Heuristic 2: The vulnerability only appears in functions that are invoked by the victim's transaction. There is no reason for the attack to appear if the transaction didn't even go through the vulnerable line of code. To check if a function is involved, I compare the first 8 characters of the input field, which should be a function selector, with the KECCAK256 hash of each function prototype. If they are equal, the involved function of the transaction is located.

Otherwise, none of the functions in the contract are involved and this may occur due to two reasons. The invoked function is a fallback, or the client inputted a wrong function selector when performing a low-level call.

With the two heuristics above, I collected all transaction trace with Ethereum Virtual Machine (EVM) Opcode CALL, DELEGATECALL, and STATICCALL from an archive node using call tracer and scraped related transaction details from the Etherscan API. Among all the attacked transactions, over 2000 of them had the same attacked function name, `execute`. Moreover, the contracts involved all have similar implementations despite having different contract addresses. At this point, I thought that the vulnerability lies within the `execute` function but in fact, these contracts are performing relay transactions, which are meant to forward transactions to other contracts.

I modified my sampling method and dig deeper into the transaction call stack. At this point, I proposed the third heuristic, but it was actually incorrect.

Heuristic 3: The vulnerable contract was located at the bottom of the call stack. I cannot provide a reasonable justification for this heuristic, but my original thought is that the bottom of the call stack is where the sender's ultimate goal of code execution resides. Therefore, it should contain all the important information and the purpose of the transaction.

I finally arrived with a list of attacked functions and contracts based on the 3 heuristics. There are also some transactions that I failed to locate the attacked function due to the following 2 reasons, failure in locating the function selector within the contract, and unverified contract source code.

3. ANALYSIS

Around 300 lines of code were written to locate the vulnerable functions and to scrap the vulnerable contracts from Etherscan. The main part of the code can be found in scripts/displacement.py inside the repository, https://github.com/haydenych/frontrunning-displacement. Among all the 2983 attacked transactions, the vulnerable function and contract for 2484 transactions were identified. Table 1 shows the results.

Results	No. of Transactions
Vulnerable function and contract identified.	2484
Failure in locating function selector within contract.	234
Unverified contract source code.	265
Total	2983

Table 1: Results for Displacement Attacks

I then manually read through the vulnerable contracts and focused on the attacked functions to see if I can locate the exact lines of code where the vulnerability occurs. However, since one of my three heuristics is wrong, as mentioned above, there is no guarantee that my results are 100% correct. As a result, I failed to locate the vulnerable lines of code using my own results.

With the help of a research assistant, Aaron, I managed to locate the vulnerable lines of code in 17 contracts using his results. These contracts can be grouped into 4 major categories. Table 2 shows the groupings, the number of vulnerable contracts in each category, and the number of transactions affected by these contracts. Yet, there are still 9 contracts that I failed to identify the exact location of the vulnerability. A complete list of the 26 vulnerable contracts provided can be found in Appendix A. The details of the groupings can also be found in Appendix B.

Category	Issue	No. of Contracts	No. of Transactions
1	Duplicated transaction inputs	6	1291
2	Unencrypted on-chain private data	4	7
3	Duplicated digital signatures	5	107
4	Arbitrage on decentralized exchanges	2	4

Table 2: Different Categories of Vulnerable Contracts

Category 1 identifies the issue with duplicated transaction inputs. Contracts in this category requires uniqueness of transaction input and returns an error upon receiving a second transaction with a duplicated request. Therefore, the victim transaction is always blocked since it is no longer unique after being front-ran by a similar attacker transaction. A possible fix to this issue is to check uniqueness within a single user instead of all users. An example of vulnerable code is `require(checkAndUpdateUniqueness(_wallet, _nonce, signHash), "RM: Duplicate request");` from the contract ApprovedTransfer 0x67933cccf3f6b40148271f5e7b5408fd1d5b2837.

Category 2 identifies the issue with unencrypted on-chain private data. Contracts in this category store confidential user information on public chains. Although variables may have private accessibility, they are only invisible to external contracts. As public blockchains are transparent, attackers can still manually read private variables and front-run users with the stolen credentials [4]. Possible fixes to this issue are to either store private data off-chain or encrypt it with special care. When parsing important information as function parameters, contracts should first ask users to send the hash of the input. The user should only send the original input after the transaction has been confirmed. The contract finally checks if the input matches the hash. This prevents attackers from stealing important information for their own

benefits, especially in gambling contracts. An example vulnerable code from the contract ChipTreasury 0xdc1d53dc4f8e44c2fabe22e76236bcdffab77124 can be seen below.

```
function claimChip (uint chipId, string password) public
whenNotPaused {
     require(isClaimed(chipId) == false;
     require(isChipPassword(chipId, password));
     …
}
```

Category 3 identifies the issue with duplicated digital signatures. Contracts in this category requires transactions to provide a digital signature or authentication key. As these keys can only be used once, the victim transaction is always blocked after being front-ran by a similar attacker transaction. Some digital signature algorithms may also include some randomness to make it unique. However, random generations on blockchains are extremely difficult as block.timestamp and block.blockhash can be easily manipulated by miners. A possible fix to this issue is to avoid using Digital Signature Algorithms, e.g., ECDSA that are based on random generations. An example vulnerable code from the contract AuthereumAccount 0x20af9e54a3670ef6a601bca1f1ec22b1f93cbe23 can be seen below.

```
require(_isValidAuthKey(_authKeyAddress), "LKMTA: Auth key is
invalid");
```

Category 4 identifies the issue with arbitraging on decentralized exchanges. Contracts in this category are ERC-20 standard based tokens. On an Automated Market Maker-based

Decentralized Exchange that uses Constant Product Market Maker, order of transactions may lead to arbitrage when buying or selling tokens. An attacker may also front-run a victim and burn all available liquidity such that the victim transaction fails to process due to insufficient assets. For this category, I cannot think of a method to mitigate the problem. Arbitraging still remains a prevalent issue on cryptocurrency markets. An example vulnerable code is `require(_sellAmount <= token.balanceOf(msg.sender));` from the contract BancorConverter 0x51907923c3280c24b6b69b0d217ea34cabde684d.

4. DISCUSSION

As multiple contracts may be involved in one transaction, locating the vulnerable contract is already a difficult task. Further, some transactions on the Ethereum Mainnet may contain large call stacks, which significantly increases the project difficulty. Since one of my three heuristics is incorrect, my methodology failed to identify the vulnerable functions and contracts accurately. The workload for manually checking the results is extremely large and there is currently no statistics on the percentage of correctly identified vulnerable contracts. It would be much better if dynamic analysis is used instead of static analysis. A way to do so is to build a local chain and execute the transactions in different orders to see if there are any differences.

Besides, including multithreading and mongodb when coding is also recommended. Multithreading allows multiple transactions to be analyzed concurrently and mongodb stores previously analyzed results into a database. Both improvements would save a lot of code execution time.

5. CONCLUSION

In this project, I investigated 2983 displacement attacks and identified the vulnerable lines of code in 17 contracts. I grouped these contracts into 4 categories, explained the issue in each category, and provided recommendations on how to fix the problem. I hope that this report provides some insights on how to avoid including codes that involve transaction ordering dependencies.

6. REFERENCES

[1] ConsenSys Diligence. (n.d.). Ethreum Smart Contract Best Practices – Known Attacks. [Online]. Available: https://consensys.github.io/smart-contract-best-practices/known_attacks/#front-running

[2] Avalanche. (n.d.). Avalanche Consensus. [Online]. Available: https://docs.avax.network/learn/platform-overview/avalanche-consensus

[3] C. F. Torres et al, Frontrunner Jones and the Raiders of the Dark Forest: An Empirical Study of Frontrunning on the Ethereum Blockchain, 3 Jun 2021. [Online] Available: https://arxiv.org/abs/2102.03347

[4] SmartContractSecurity. (n.d.). Unencrypted Private Data On-Chain. [Online]. Available: https://swcregistry.io/docs/SWC-136

APPENDIX A List of Vulnerable Contracts

1. ApprovedTransfer - 0x67933cccf3f6b40148271f5e7b5408fd1d5b2837
2. ApprovedTransfer - 0xc927a1e4c431babb798d705988ab036e2bdcf640
3. ApprovedTransfer - 0xcd23f51912ea8fff38815f628277731c25c7fb02
4. AuthereumAccount - 0x20af9e54a3670ef6a601bca1f1ec22b1f93cbe23
5. BancorConverter - 0x51907923c3280c24b6b69b0d217ea34cabde684d
6. CErc20 - 0x6c8c6b02e7b2be14d4fa6022dfd6d75921d90e4e
7. CEther - 0x4ddc2d193948926d02f9b1fe9e1daa0718270ed5
8. ChipTreasury - 0xdc1d53dc4f8e44c2fabe22e76236bcdffab77124
9. ERC1155Sale - 0x93f2a75d771628856f37f256da95e99ea28aafbe
10. Exchange - 0x080bf510fcbf18b91105470639e9561022937712
11. Exchange - 0x12459c951127e0c374ff9105dda097662a027093
12. GnosisSafe - 0x34cfac646f301356faa8b21e94227e3583fe3f5f
13. InstantTrade - 0xe17dbb844ba602e189889d941d1297184ce63664
14. LendingPoolCore - 0x5766067108e534419ce13f05899bc3e3f4344948
15. LiquidityConversionRates - 0x0d8c194e877af78bea7d1a7b00f593aeed7be709
16. LiquidityConversionRates - 0x6fef87bf675c8ffeaa3e713aa36d3a9358786999
17. MarbleNFTCandidate - 0x30e0130141b3f113480a5941ca180ad8c5f98612
18. MatchingMarket - 0x39755357759ce0d7f32dc8dc45414cca409ae24e
19. Mooniswap - 0x322a1e2e18fffc8d19948581897b2c49b3455240
20. PasswordEscrow - 0x11ab8e468a6e4e0f69bfd35e4e5a941043f51fd3
21. Pot - 0x197e90f9fad81970ba7976f33cbd77088e5d7cf7
22. RebalanceAuctionModule - 0xe23fb31dd2edacebf7d92720358bb92445f47fdb
23. TransferManager - 0x103675510a219bd84ce91d1bcb82ca194d665a09
24. TransferManager - 0x2b6d87f12b106e1d3fa7137494751566329d1045
25. TransferManager - 0xaf1c9fc0588f3caf654c796b3ce83d74f477419c
26. TryToGetYourMoney - 0x9478abe9244872274808d324b968c30f29e1a442

APPENDIX B Vulnerable Contract Groupings

Category	Issues and Involved Contracts
1	Duplicated transaction inputs • ApprovedTransfer - 0x67933cccf3f6b40148271f5e7b5408fd1d5b2837 • ApprovedTransfer - 0xc927a1e4c431babb798d705988ab036e2bdcf640 • ApprovedTransfer - 0xcd23f51912ea8fff38815f628277731c25c7fb02 • TransferManager - 0x103675510a219bd84ce91d1bcb82ca194d665a09 • TransferManager - 0x2b6d87f12b106e1d3fa7137494751566329d1045 • TransferManager - 0xaf1c9fc0588f3caf654c796b3ce83d74f477419c
2	Unencrypted on-chain private data • ChipTreasury - 0xdc1d53dc4f8e44c2fabe22e76236bcdffab77124 • MarbleNFTCandidate - 0x30e0130141b3f113480a5941ca180ad8c5f98612 • PasswordEscrow - 0x11ab8e468a6e4e0f69bfd35e4e5a941043f51fd3 • TryToGetYourMoney - 0x9478abe9244872274808d324b968c30f29e1a442
3	Duplicated digital signatures • AuthereumAccount - 0x20af9e54a3670ef6a601bca1f1ec22b1f93cbe23 • ERC1155Sale - 0x93f2a75d771628856f37f256da95e99ea28aafbe • Exchange - 0x12459c951127e0c374ff9105dda097662a027093 • GnosisSafe - 0x34cfac646f301356faa8b21e94227e3583fe3f5f • InstantTrade - 0xe17dbb844ba602e189889d941d1297184ce63664
4	Arbitrage on decentralized exchanges • BancorConverter - 0x51907923c3280c24b6b69b0d217ea34cabde684d • MatchingMarket - 0x39755357759ce0d7f32dc8dc45414cca409ae24e