AUDITORIA SMART CONTRACTS (HTTPS://BLOG.COINFABRIK.COM/CATEGORY/AUDITORIA-SMART-CONTRACTS/)

Timvi Smart Contract Audit

Mariana Soffer (https://blog.coinfabrik.com/author/mariana-soffel/)

November 26, 2019 (https://blog.coinfabrik.com/smart-contracts/smart-contract-audit-smart-contracts/timvi-smart-contract-audit/)

Contents

Introduction

CoinFabrik was asked to audit the contracts for the Timvi project. Firstly, we will provide a summary of our discoveries and secondly, we will show the details of our findings.

Summary

The contracts audited are from the Timvi repository at https://github.com/TimviOfficial/Timvi (https://github.com/TimviOfficial/Timvi). The audit is based on the commit 9324706d1160996f7847e1989c8567168261382e.

The audited contracts are:

- Logic (https://github.com/TimviOfficial/Timvi/blob/3e42482d8ec11bc4b4f934816ae9649a6017b7d9/contracts/Logic.sol) ERC-721 Non-Fungible TBox Token contract. The main logic contract.
- TimviSettings (https://github.com/TimviOfficial/Timvi/blob/3e42482d8ec11bc4b4f934816ae9649a6017b7d9/contracts/TimviSettings.sol) Settings store.
- TimviToken (https://github.com/TimviOfficial/Timvi/blob/3e42482d8ec11bc4b4f934816ae9649a6017b7d9/contracts/TimviToken.sol) ERC-20 Timvi stablecoin.
- PriceGetter (https://github.com/TimviOfficial/Timvi/blob/3e42482d8ec11bc4b4f934816ae9649a6017b7d9/oracle-contract/PriceGetter.sol) ETH/USD price oracle contract (using Oraclize).
- ExchangeService (https://github.com/TimviOfficial/Timvi/blob/3e42482d8ec11bc4b4f934816ae9649a6017b7d9/contracts/services/leverage-exchange/ExchangeService.sol) exchange ETH to TMV according to the system's rate.
- LeverageService (https://github.com/TimviOfficial/Timvi/blob/3e42482d8ec11bc4b4f934816ae9649a6017b7d9/contracts/services/leverage-exchange/LeverageService.sol) receive ETH for a collateral in ETH.
- BondService (https://github.com/TimviOfficial/Timvi/blob/3e42482d8ec11bc4b4f934816ae9649a6017b7d9/contracts/services/bond/BondService.sol) withdraw and sell TMV to get ETH.
- Gate (https://github.com/TimviOfficial/Timvi/blob/3e42482d8ec11bc4b4f934816ae9649a6017b7d9/contracts/services/gate/Gate.sol) This service allows to exchange TMVs for ETH.

As they are important to the system architecture and security of the solution, we have also audited a python script and dockerfile which provides the ethereum price. Solidity code referenced these files in an immutable manner. They are stored using IPFS and have the hash QmVXihTAKo3mwEHBMeM4EDG8MbWHFTvxoigdFAgmTneWra.

Url-requests.py (https://gateway.ipfs.io/ipfs/QmVXihTAKo3mwEHBMeM4EDG8MbWHFTvxoigdFAgmTneWra)

Dockerfile (https://gateway.ipfs.io/ipfs/QmVXihTAKo3mwEHBMeM4EDG8MbWHFTvxoigdFAgmTneWra)

The following analyses were performed:

- Misuse of the different call methods: call.value(), send() and transfer().
- Integer rounding errors, overflow, underflow and related usage of SafeMath functions.
- Old compiler version pragmas.
- Race conditions such as reentrancy attacks or front running.
- Misuse of block timestamps, assuming anything other than them being strictly increasing.
- Contract softlocking attacks (DoS).
- Potential gas cost of functions being over the gas limit.
- Missing function qualifiers and their misuse.
- Fallback functions with a higher gas cost than the one that a transfer or send call allows.
- Fraudulent or erroneous code.
- Code and contract interaction complexity.

- Wrong or missing error handling.
- Overuse of transfers in a single transaction instead of using withdrawal patterns.
- Insufficient analysis of the input function requirements.

Detailed findings

Critical severity

Oracle python script crashes

The file provided indirectly through IPFS provides the Ether price by taking the average from multiple sources, which is later used by Oraclize to send the result to the contracts. The following line gives a runtime error when executed, crashing the script and consequently preventing the execution of the function ethUsdPrice from the contract PriceGetter.sol, making all contracts fail.

```
    return int(prices[0] * 100000)
```

We recommend fixing this bug as it's imperative for the correct contract functionality.

A possible solution is:

```
            return int(prices[0])*100000
```

But we think that the proper code to accomplish what we understand is your intention would be:

```
            return float(prices[0])
```

Update: Since the Timvi project now uses Chainlink as its main oracle solution (https://github.com/TimviOfficial/Timvi/commit/5b259227cc2660eaa71ed5a6faebcbff5d828959) *this particular issue isn't present anymore.*

Using API Keys in public script:

Using api keys in price providers is a security risk because anyone can see and use these keys in an abusive manner causing the price provider temporarily to suspend the key making the request sent by the script fail.

{'path': 'https://api.etherscan.io/api?module=stats&action=ethprice&apikey=91DFNHV3CJDJE12PG4DD66FUZEK71TC6NW', 'headers': {}},

An attacker may wait for several price providers not using API keys to be offline and then begin the attack. If enough of them fail, it will cause the error described above.

You may consider using the url data source (https://docs.provable.xyz/#data-sources-url) in a nested (https://docs.provable.xyz/#data-sources-nested) query since all the script does is perform https requests. It would avoid this kind of problems, be less expensive (computation 0.50$ vs URL 0.01$) (https://docs.provable.xyz/#pricing) and it could be possible to use android authenticity proof with this data source.

Update: Since the Timvi project now uses Chainlink as its main oracle solution (https://github.com/TimviOfficial/Timvi/commit/5b259227cc2660eaa71ed5a6faebcbff5d828959) this particular issue isn't present anymore.

Usage of tx.origin for financial instrument ownership

Using tx.origin for the creation of bid, bond and ask instruments results in those being owned by the external account that initiated the transaction:

```
        Bid memory _bid = Bid(
            tx.origin,
            msg.value,
            _percent
        );
```

Afterwards, every privilege check that restricts who is able to manipulate these instruments compares the aforementioned address with msg.sender, the caller address.

The problem is that these two are not necessarily the same. Not every account being used by users needs to be an external account. Some accounts are instantiated as contracts such as multi-signature wallets. If a multi-signature wallet was used, the instrument will be owned by the address of the user who initiated the multi-signature transaction, but will be checked against the address of the wallet itself, which does not coincide.

We asked the developers and they responded that the use of tx.origin was made to allow contract middleware to be used when creating these instruments. While the intention is now clearer, the problem described above still applies.

One possible solution could be to warn your users not to use multi signature wallets or other type of contract that would wish to interact with this project. Another solution is for the middleware to propagate the msg.sender using an additional function parameter for these functions. This obviously relies on the user to trust the middleware to correctly pass the msg.sender address, but the middleware needs to be trusted by the user regardless of this issue.

Update: This was fixed in commit: d589cd10963e627da340a49335ed60360f4e26d0 (https://github.com/TimviOfficial/Timvi/commit/d589cd10963e627da340a49335ed60360f4e26d0)

Medium severity

Wrong oraclize implementation

According to the oraclize documentation (https://docs.provable.xyz/#ethereum-quick-start-authenticity-proofs) you should specify an authenticity proof type. Since the default authenticity proof type is none using proofStorage won't have any effect because there is nothing to store. So the code:

```
oraclize_setProof(proofStorage_IPFS);
```

Should in this case also use TLSNotary which is the only available option (https://docs.provable.xyz/#data-sources) for 'computation' data source.

```
provable_setProof(proofType_TLSNotary | proofStorage_IPFS);
```

On the other hand, we also recommend to handle the 'proof' parameter sent to the callback function accordingly, otherwise data may be compromised.

Update: Since the Timvi project now uses Chainlink as its main oracle solution (https://github.com/TimviOfficial/Timvi/commit/5b259227cc2660eaa71ed5a6faebcbff5d828959) *this particular issue isn't present anymore.*

Minor severity

Failing Ether transfer can deny multiFill service function

The function multiFill allows the administrator of the contracts to fulfill multiple orders by id in a single transaction:

```
    function multiFill(uint256[] calldata _ids) external onlyAdmin() payable {
        emit Funded(msg.value);
        for (uint256 i = 0; i < _ids.length; i++) {
            uint256 id = _ids[i];
         require(orders[id].owner != address(0), "Order doesn't exist");
            uint256 tmv = orders[id].amount;
            uint256 eth = tmv2eth(tmv);
            require(address(this).balance >= eth, "Not enough funds");
            address payable owner = orders[id].owner;
            delete orders[id];
            IToken(settings.tmvAddress()).transfer(timviWallet, tmv);
            owner.transfer(eth);
            emit OrderFilledPool(id, owner, tmv, eth);
        }
    }
```

The problem with this function is that Solidity's transfer function can fail, as the amount of transfer's gas is limited and may not be enough to fulfill a transaction to the user if it's using a contract to interact with. For example, the user may have an expensive fallback function on the receiver end. If this happens, the entire transaction will be rolled back, and no order will get fulfilled. Moreover, the spent gas will be lost, and the admin will need to call the function again removing the conflicting address. Since the addresses that come after the failing address were not checked, they may contain another address that also fails, making this iterative process very costly for the administrator.

We recommend using the send function instead, which allows to check if the transaction was successful without rolling back. This allows you to fulfil and delete only the order on success, and continue processing the remaining addresses. The conflicting addresses that failed can be filtered by using events the administrator may catch off chain.

Update: This was fixed in commit 0a6e3d1277ecb1a460563de495b570bf0a2ced61 (https://github.com/TimviOfficial/Timvi/commit/0a6e3d1277ecb1a460563de495b570bf0a2ced61)

Logical mistake in function changeYearFee

The function changeYearFee changes the yearly fee to one that is passed as a parameter. The intention is to make the change only if the new fee is different. But instead of retrieving the old fee for comparison, it retrieves the expiration date, which is completely unrelated:

```
    function changeYearFee(uint256 _id, uint256 _yearFee) internal {
        uint256 _oldYearFee = bonds[_id].expiration;
        if (_oldYearFee != _yearFee) {
            require(_yearFee <= 10000, "Fee out of range");
            bonds[_id].yearFee = _yearFee;
        }}
```

Both integers fall into different ranges, they will never be equal and this function will always execute the body of the condition. Because of that, the impact of this particular bug is low. However, it should be fixed as it's a logical error. We recommend replacing the first line of the function with something similar to the following:

```
uint256 _oldYearFee = bonds[_id].yearFee;
```

Update: This was fixed in commit 2bd5a7fd1fba883560fc8625c67b7b7c9cea51b1 (https://github.com/TimviOfficial/Timvi/commit/2bd5a7fd1fba883560fc8625c67b7b7c9cea51b1)

Enhancements

Avoid duplicate code

There are many instances of code duplication in the project. Code duplication is not usually recommended, it can lead to bugs if is not handled correctly. It also makes the overall project harder to understand as there are more lines of code available to parse.

For example, you have multiple modifiers that do checks, but in many instances you have the modifier content hardcoded at the beginning of the function instead:

```
    function changeOwner(uint256 _id, uint256 _deposit, uint256 _expiration, uint256 _yearFe
e)
        external
        payable
    {
        require(bonds[_id].owner == msg.sender && bonds[_id].emitter == address(0), "You are
not the owner ot bond is matched");
        changeDeposit(_id, _deposit);
        changeExpiration(_id, _expiration);
        changeYearFee(_id, _yearFee);
        emit BondChanged(_id, _deposit, 0, _expiration, _yearFee, msg.sender);
    }
```

There are also contract files that are very similar and only differ in a couple of lines. As is the case with ExchangeService.sol and LeverageService.sol.

Update: This was enhanced in commit a89a9a6e3b5b88b4b57a816e590fb95d34590281 (https://github.com/TimviOfficial/Timvi/commit/a89a9a6e3b5b88b4b57a816e590fb95d34590281)

Conclusion

The contracts are straightforward and have an adequate amount of documentation. We found multiple issues that would prevent the contract correct functionality. Even though the project has unit tests, some of the problems we found won't show with current implemented testing. For example an external functionality like the python oracle script or the multi-signature wallet

problem won't show and will require a more comprehensive testing strategy. We believe a project of this magnitude would benefit from having a testing phase with a wider coverage and more user interactivity. Once changes are implemented we would suggest conducting a second audit.

Disclaimer: This audit report is not a security warranty, investment advice, or an approval of the Timvi project since CoinFabrik has not reviewed its platform. Moreover, it does not provide a smart contract code faultlessness guarantee.

Related Posts

(https://blog.coinfabrik.com/smart-contracts/smart-contract-audit-smart-contracts/money-on-chain-gas-cost/)

Money on Chain Gas Cost (https://blog.coinfabrik.com/smart-contracts/smart-contract-audit-smart-contracts/money-on-chain-gas-cost/)

The purpose of this document is to offer different alternatives to reduce gas cost in…

(https://blog.coinfabrik.com/smart-contracts/smart-contract-audit-smart-contracts/smart-contract-auditing-news/)

Smart Contract Auditing News (https://blog.coinfabrik.com/smart-contracts/smart-contract-audit-smart-contracts/smart-contract-auditing-news/)

Every month several important smart contract audits are performed by blockchain security companies like us.…

(https://blog.coinfabrik.com/auditoria-smart-

Money on Chain Security Audit I (https://blog.coinfabrik.com/auditoria-smart-

(https://blog.coinfabrik.com/auditoria-smart-contracts/money-on-chain-security-audit-i/)

This is the second of a series of four audits we performed for MOC: Second…

(https://blog.coinfabrik.com/smart-contracts/smart-contract-audit-smart-contracts/nahmii-security-audit/)

Nahmii Security Audit (https://blog.coinfabrik.com/smart-contracts/smart-contract-audit-smart-contracts/nahmii-security-audit/)

CoinFabrik was asked to audit the contracts for the Nahmii Token project. Firstly, we will…

(https://blog.coinfabrik.com/smart-contracts/smart-contract-audit-smart-contracts/coinfabrik-audit-process/)

Smart Contract Audit Process (https://blog.coinfabrik.com/smart-contracts/smart-contract-audit-smart-contracts/coinfabrik-audit-process/)

A security audit is a process in which a client subjects his or her smart…

(https://blog.coinfabrik.com/auditoria-smart-contracts/auditing-with-securify/)

Auditing with Securify (https://blog.coinfabrik.com/auditoria-smart-contracts/auditing-with-securify/)

After our articles Smart Contract Auditing: Human vs. Machine and Auditing Solidity code with Slither…

(https://blog.coinfabrik.com/smart-contracts/smart-contract-

EasyPool Smart Contract Security Audit v2 (https://blog.coinfabrik.com/smart-contracts/smart-contract-

audit-smart-contracts/easypool-smart-contract-security-audit-v2/)

CoinFabrik has been hired to audit the EasyPool smart contracts. We start this report writing…

(https://blog.coinfabrik.com/smart-contracts/smart-contract-audit-smart-contracts/arcadierx-security-audit/)

ArcadierX Security Audit (https://blog.coinfabrik.com/smart-contracts/smart-contract-audit-smart-contracts/arcadierx-security-audit/)

CoinFabrik was asked to audit the contracts for the ArcadierX project. Firstly, we will provide…

(https://blog.coinfabrik.com/smart-contracts/smart-contract-audit-smart-contracts/stasis-token-smart-contract-audit/)

Stasis Token Smart Contract Audit (https://blog.coinfabrik.com/smart-contracts/smart-contract-audit-smart-contracts/stasis-token-smart-contract-audit/)

Coinfabrik has been hired to audit the smart contracts for Stasis sale, the Stable Euro…

(https://blog.coinfabrik.com/smart-contracts/etherparty-

EtherParty Smart Contract Security Audit

token-smart-contract-security-audit-coinfabrik/)

(https://blog.coinfabrik.com/smart-contracts/etherparty-token-smart-contract-security-audit-coinfabrik/)

Coinfabrik team has been hired to audit Etherparty smart contracts. Firstly, we will provide a…

(https://blog.coinfabrik.com/smart-contracts/smart-contract-audit-smart-contracts/rcn-smart-contracts-audit-v2/)

RCN Smart Contracts Audit v2

(https://blog.coinfabrik.com/smart-contracts/smart-contract-audit-smart-contracts/rcn-smart-contracts-audit-v2/)

The smart contracts that have been audited were taken from the RCN repository at: https://github.com/ripio/rcn-network/tree/v2….

(https://blog.coinfabrik.com/smart-contracts/smart-contract-audit-smart-contracts/dmtoken-token-sale-smart-contract-audit/)

DMToken Token Sale Smart Contract Audit

(https://blog.coinfabrik.com/smart-contracts/smart-contract-audit-smart-contracts/dmtoken-token-sale-smart-contract-audit/)

Coinfabrik was hired to audit the contract in terms of its security. First of all,…

Tags:

Smart Contract (https://blog.coinfabrik.com/tag/smart-contract/)

smart contract audits (https://blog.coinfabrik.com/tag/smart-contract-audits/)

stable-coins (https://blog.coinfabrik.com/tag/stable-coins/)

SHARE ON

(https://www.facebook.com/sharer/sharer.php?u=https://blog.coinfabrik.com/smart-contracts/smart-contract-audit-smart-contracts/timvi-smart-contract-audit/)
(https://twitter.com/intent/tweet?text=Timvi%20Smart%20Contract%20Audit&url=https://blog.coinfabrik.com/smart-contracts/smart-contract-audit-smart-contracts/timvi-smart-contract-audit/)
(https://pinterest.com/pin/create/button/?url=&media=https://blog.coinfabrik.com/wp-content/uploads/2019/11/timvi.png&description=Timvi+Smart+Contract+Audit)
(https://www.linkedin.com/shareArticle?mini=true&url=https://blog.coinfabrik.com/smart-contracts/smart-contract-audit-smart-contracts/timvi-smart-contract-audit/&title=Timvi%20Smart%20Contract%20Audit&source=CoinFabrik%20Blog)

← PREVIOUS ARTICLE

Money on Chain Gas Cost (https://blog.coinfabrik.com/smart-contracts/smart-contract-audit-smart-contracts/money-on-chain-gas-cost/)

NEXT ARTICLE →

Money on Chain Security Audit III (https://blog.coinfabrik.com/smart-contracts/smart-contract-audit-smart-contracts/money-on-chain-security-audit-iii/)

You may also like

(https://blog.coinfabrik.com/smart-contracts/magic-bridge-audit/)

Magic Bridge Audit (https://blog.coinfabrik.com/smart-contracts/magic-bridge-audit/)

(https://blog.coinfabrik.com/smart-contracts/smart-contract-audit-smart-contracts/mintingfactoryv2-baseupgradablemarketplace-kodav3upgradablegatedmarketplace/)

2 months ago

Smart Contract Audit (https://blog.coinfabrik.com/category/smart-contracts/smart-contract-audit-smart-contracts/)

MintingFactoryV2, BaseUpgradableMarketplace & KODAV3UpgradableGatedMarketplace (https://blog.coinfabrik.com/smart-contracts/smart-contract-audit-smart-contracts/mintingfactoryv2-baseupgradablemarketplace-kodav3upgradablegatedmarketplace/)

(https://blog.coinfabrik.com/feed/)

(https://ar.linkedin.com/company/coinfabrik)

(https://twitter.com/coinfabrik)

(https://www.youtube.com/channel/UC2GmjCr7aEz-il31kqOy9aw)

(https://www.facebook.com/CoinFabrik/)

(https://www.reddit.com/r/CoinFabrik/)

(https://github.com/coinfabrik)

Made with ♥ in Buenos Aires, Argentina.