securing

www.securing.biz

Smart Contracts Security Review for Logium

Report

document version: 1.1

author: Damian Rusinek (SecuRing)

test time period: 2022-04-19 – 2022-04-22

retest time period: 2022-04-28 – 2022-04-28

report date: 2022-04-28

TABLE OF CONTENTS

1. EXECUTIVE SUMMARY

1.1. Summary of retest results (2022-04-28)

- The risk related to **one** vulnerability with **medium** impact was accepted.
  - The team decided to accept the risk regarding this issue because the protocol protects users from flash loans, but the "multiple-block price manipulation" is also feasible on traditional market if the attacker has enough assets.
- The vulnerability with **low** impact on risk was fixed.
- **Four** out of **nine** recommendations have been implemented and one was partially implemented.
- Additionally, the bug related to pool price calculation has been fixed in commit *7f68958c8d225a07ff8ec048e122aabaeb65a7d5*. Due to this bug, we recommend the setup for testing locally (not as a mainnet fork), because it will be deterministic and will quickly allow the team to check the unit test coverage and find out uncovered functions.
- **Note (2022-05-02)**: The commits' dates are later than the report dates because they were updated after the initial test and retest. The testing team verified whether there were no changes between the initial and updated commits.

1.2. Summary of test results

- There are **no** vulnerabilities with **high** impact on risk found.
- There is **one** vulnerability with **medium** impact on risk found. Its potential consequence is:
  - Manipulation of asset's price when resolving the bet and creating a favorable situation.
- There is **one** vulnerability with **low** impact on risk found. Its potential consequence is:
  - Undefined calculations based on zero addresses.
- Additionally, **nine** recommendations have been proposed that do not have any direct risk impact. However, it is suggested to implement them due to good security practices.

2. PROJECT OVERVIEW

2.1. Project description

The analyzed system provides speculation system with financial leverage for arbitrary assets from the Uniswap V3 pools. The platform allows users to invest in either long or short positions by issuing offers (called *tickets*) that can be bought by *traders*.
Tickets are not directly issued on-chain, but they are submitted (with *issuer's* signature) to the off-chain system called *EL-DWA*. The system publishes the list of accepted *tickets* and *traders* can call a LogiumCore contract to take a *ticket* and make a bet.

The protocol distinguishes roles such as *owner*, *issuer*, *and trader*:

- ***owner*** - role with the highest privileges responsible for providing bet implementations,
- ***issuer*** – the user issuing a ticket with arbitrary amount and time frame,
- ***trader*** – the user deciding to purchase arbitrary, listed ticket.

An example usage scenario:

1. *Issuer* emits "UP option" ticket for ETH/USDC pair through *EL-DWA* system.
2. *Trader* buys emitted option executing *takeTicket* function (LogiumCore.sol#L224).
3. The price of the *ticket's* asset exceeds the strike price in the *ticket's* exercise window.
4. *Trader* executes *exercise* function (LogiumBinaryBet.sol#L168) and gains profit.

2.2. Retest scope

Retests focused on vulnerabilities found in previous tests. The object being analysed were changes in selected smart contracts accessible in the following:

- https://github.com/logium-org/logium-contracts
- Commit ID: *ac063014d07465c1288e474d56bb132ad0b0c9dc*

2.3. Target in scope

The object being analysed were selected smart contracts accessible in the following:

- GitHub repository: https://github.com/logium-org/logium-contracts
- Commit ID: *7876942689a50bfb45177796ef2c675cc5fafa6d*

The contracts reviewed were the following:

```
contracts/
├── CallTool.sol
├── interfaces
```

```
│   ├── ILogiumBinaryBet.sol
│   ├── ILogiumCore.sol
│   ├── logiumBinaryBet
│   │   ├── ILogiumBinaryBetCore.sol
│   │   ├── ILogiumBinaryBetIssuer.sol
│   │   ├── ILogiumBinaryBetState.sol
│   │   └── ILogiumBinaryBetTrade.sol
│   └── logiumCore
│       ├── ILogiumCoreIssuer.sol
│       ├── ILogiumCoreOwner.sol
│       ├── ILogiumCoreState.sol
│       └── ILogiumCoreTrader.sol
├── libraries
│   ├── Constants.mainnet.sol
│   ├── Constants.rinkeby.sol
│   ├── Constants.sol
│   ├── Market.sol
│   ├── OracleLibrary.sol
│   ├── RatioMath.sol
│   ├── TicketBinaryBet.sol
│   └── Ticket.sol
├── LogiumBinaryBet.sol
└── LogiumCore.sol
```

2.4. Threat analysis

This section summarized the potential threats that were identified during initial threat modeling. The audit was mainly focused on, but not limited to, finding security issues that be exploited to achieve these threats.

The key threats were identified from particular perspectives as follows:

1. General (apply to all of the roles mentioned)
    - Bypassing the business logic of the system.
    - Theft of users' funds.
    - Errors in arithmetic operation.
    - Possibility to block the contract.
    - Possibility to lock users' funds in the contract.
    - Incorrect access control for roles used by the contracts.
    - Withdraw more than have deposited.
    - Deposit collateral without the transfer.
    - Price manipulation to win the bet.
2. Owner
    - Too much power in relation to the declared one.
    - Unintentional loss of the ability to governance the system.
    - Bypass issuer's signature and deploys malicious bet.
    - Instant block of legitimate implementation and its active tickets.
    - Creation of malicious contract that steal user funds.

3. Issuer
    - Usage of a malicious bet (custom implementation).
    - Issuance of undercollateralized bet.
    - Claim of tokens from bet after losing.
4. Trader
    - Bypass of issuer's signature and deployment of malicious bet (spoofing the issuer's ticket).
    - Exercising the same bet twice to get more token after win.
    - Exercising before/after exercise window.
    - Taking the expired ticket.
    - Re-initializing the bet.
5. User
    - Bypass of issuer's signature and withdrawing all their tokens.
    - Bypass of issuer's signature and invalidating their tickets.

2.5. Testing overview

The security review of the Logium smart contracts were meant to verify whether the proper security mechanisms were in place to prevent users from abusing the contracts' business logic and to detect the vulnerabilities which could cause financial losses to the client or its customers.

Security tests were performed using the following methods:

- Source code review,
- Automated tests through various tools (static analysis),
- Q&A sessions with the client's representatives which allowed to gain knowledge about the project and the technical details behind the platform.

2.6. Basic information

Testing team	Damian Rusinek Paweł Kuryłowicz Jakub Zmysłowski
Testing time period	2022-04-19 – 2022-04-22
Retesting time period	2022-04-28- 2022-04-28
Report date	2022-04-28
Version	1.1

2.7. Disclaimer

The security review described in this report is not a security warranty.
It does not guarantee the security or correctness of reviewed smart contracts. Securing smart contract platforms is a multi-stage process, starting from threat modeling, through development based on best security practices, security reviews and formal verification, ending with constant monitoring and incident response.

Therefore, we strongly recommend implementing security mechanisms at all stages of development and maintenance.

3. SUMMARY OF IDENTIFIED VULNERABILITIES

3.1. Risk classification

Vulnerabilities	
Risk impact	**Description**
Critical	Vulnerabilities that affect the security of the entire system, all users or can lead to a significant financial loss (e.g., tokens). It is recommended to take immediate mitigating actions or limit the possibility of vulnerability exploitation.
High	Vulnerabilities that can lead to escalation of privileges, a denial of service or significant business logic abuse. It is recommended to take mitigating actions as soon as possible.
Medium	Vulnerabilities that affect the security of more than one user or the system, but might require specific privileges, or custom prerequisites. The mitigating actions should be taken after eliminating the vulnerabilities with critical and high risk impact.
Low	Vulnerabilities that affect the security of individual users or require strong custom prerequisites. The mitigating actions should be taken after eliminating the vulnerabilities with critical, high, and medium risk impact.
Recommendations	
Methods of increasing the security of the system by implementing good security practices or eliminating weaknesses. No direct risk impact has been identified. The decision whether to take mitigating actions should be made by the client.	

3.2. Identified vulnerabilities

Vulnerability	Risk impact	Retest 2022-04-28
4.1 Price based on previous block	**Medium**	**Accepted the risk**
4.2 Undefined return variables	**Low**	**Fixed**
Recommendations		
5.1 Consider using the latest solidity version		**Not implemented**
5.2 Consider using custom errors		**Not implemented**
5.3 Consider propagating the return data from functionCall		**Not implemented**
5.4 Correct comments		**Implemented**
5.5 Project relays on draft-EIP712 contract implementation		**Not implemented**
5.6 Remove unused data		**Implemented**
5.7 Validate the Uniswap V3 pool (on-chain and off-chain)		**Partially implemented**
5.8 Consider adding zero address verification		**Implemented**
5.9 Consider setting tighter restrictions for protocol parameters		**Implemented**

4. VULNERABILITIES

4.1. Price based on previous block

Risk impact	Medium	SCSVS V14
Vulnerable contract	OracleLibrary	
Exploitation conditions	Pool inflation in the previous block.	
Exploitation results	Manipulation of asset's price when resolving the bet and creating a favorable situation.	
Remediation	• Use TWAP oracle with window length based on the bet window length to increase the cost of attack (more blocks to be manipulated), • Monitor selected pools (liquidity, observations) and warn users about the potential risk related to the pool.	

Status 2022-04-28:
Accepted the risk
The team decided to accept the risk regarding this issue because the protocol protects users from flash loans, but the "multiple-block price manipulation" is also feasible on traditional market if the attacker has enough assets.

Vulnerability description:
The protocol calculates ticks for a given Uniswap V3 pool using OracleLibrary. However, the price is retrieved from the single block (or two blocks if there were swaps in the current block), thus it is susceptible to manipulations.

Test case:
The *consult* function (OracleLibrary.sol#L18) calculates tick for the pool. If the last oracle observation is not from the current block, the protocol assumes that there were no swaps and use the spot price (OracleLibrary.sol#L37), which comes from the previous block (assuming there was a swap in previous block). On the other hand, when there are swaps in the current block, the protocol calculates the tick basing on the current cumulative tick and the cumulative tick from the previous block.
In both cases, the tick can be manipulated by malicious user.

References:
SCSVS V14: Decentralized Finance
https://securing.github.io/SCSVS/1.2/0x23-V14-Decentralized-Finance.html

4.2. Undefined return variables

Risk impact	Low	SCSVS V8
Vulnerable contract	Market	
Exploitation conditions	Using pool that has no predefined base token (USDC, DAI, USDT, WETH).	
Exploitation results	Undefined calculations based on zero addresses.	
Remediation	Revert if the predefined token is not found in the pool.	

Status 2022-04-28:
Fixed
The revert statement has been added (Market.sol#L118).
Commit ID: *ac063014d07465c1288e474d56bb132ad0b0c9dc*

Vulnerability description:
The protocol uses USDC, WETH, DAI and USDT as base tokens. If the chosen Uniswap pool uses different base asset, the zero address is returned, what leads to undefined calculations.

Test case:
The *getPair* function (Market.sol#L100) returns address of both base and quote asset. If the pool does not use USDC, WETH, DAI or USDT, the *ifOneSort* helper returns zero addresses (Market.sol#L135).

References:
SCSVS V8: Business logic
https://securing.github.io/SCSVS/1.1/0x17-V8-Business-Logic.html

5. RECOMMENDATIONS

5.1. Consider using the latest solidity version

Status 2022-04-28:
Not implemented

Description:
Use the latest stable version (0.8.13) for testing.

How to implement:
Use a specific version of Solidity compiler (latest stable):

```
pragma solidity 0.8.13;
```

References:
Floating Pragma
https://swcregistry.io/docs/SWC-103
SCSVS V1: Architecture, design, and threat modelling
https://securing.github.io/SCSVS/1.1/0x10-V1-Architecture-Design-Threat-modelling.html

5.2. Consider using custom errors

Status 2022-04-28:
Not implemented

Description:

Custom errors are more gas efficient than comments added to require statement.

How to implement:
Consider using custom errors to reduce your gas consumption.

References:
Custom errors
https://blog.soliditylang.org/2021/04/21/custom-errors/

5.3. Consider propagating the return data from functionCall

Status 2022-04-28:
Not implemented

Description:
The current code for the *multicall* function (CallTool.sol#L14) does not return a value to its caller. Although this does not directly impact the risk, it is against best security practices as the OpenZeppelin library is not used as intended.

How to implement:
Consider handling data returned by *functionCall* function (CallTool.sol#L19).

References:
SCSVS V11: Code clarity
https://securing.github.io/SCSVS/1.2/0x20-V11-Code-Clarity.html

5.4. Correct comments

Status 2022-04-28:
Implemented
Commit ID: *ac063014d07465c1288e474d56bb132ad0b0c9dc*

Description:
The *Payload* struct (Ticket.sol#L22) has the following comment:

```
/// - deadline - unix secs timestamp, ticket is only valid for taking if
blocktime is <= deadline
```

This is not a correct statement because the function *takeTicket* (LogiumCore.sol#L237) check if *deadline* parameter is **greater** (and not greater or equal) than current block timestamp:

```
require(payload.deadline > block.timestamp, "Ticket expired");
```

The *getPair* function (Market.sol#L100) has the following comment:

```
/// warning: returns 0x0, 0x0 for pool that is not vs USDC or WETH!
```

This is not a correct statement because the function *ifOneSort* (Market.sol#L120) returns zero address for pool without more base assets: USDC, WETH, DAI or USDT.

How to implement:
Correct comments to reflect the current state of the code.

References:
SCSVS V11: Code clarity
https://securing.github.io/SCSVS/1.2/0x20-V11-Code-Clarity.html

5.5. Project relays on draft-EIP712 contract implementation

Status 2022-04-28:
Not implemented

Description:

The LogiumCore contract relays on *draft-EIP712* contract (LogiumCore.sol#L15). Contracts in draft versions might not be audited and optimized.

How to implement:
Consider using stable implementation of EIPs.

References:
SCSVS V11: Code clarity
https://securing.github.io/SCSVS/1.2/0x20-V11-Code-Clarity.html
[OpenZeppelin] Contract draft-EIP712.sol
https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/cryptography/draft-EIP712.sol

5.6. Remove unused data

Status 2022-04-28:
Implemented
The *multicallUnsafe* function in CallTool contract now counts the failed calls using previously unused *if* statement.
Commit ID: *ac063014d07465c1288e474d56bb132ad0b0c9dc*

Description:
There are pieces of code that are not needed and never used:

- CallTool.sol#L35

```
if (_success) {} //ignore success or failure
```

- LogiumCore.sol#L134

```
require(from != address(0x0), "Invalid signature");
```

How to implement:
Remove unnecessary code.

References:
SCSVS V11: Code clarity
https://securing.github.io/SCSVS/1.2/0x20-V11-Code-Clarity.html

5.7. Validate the Uniswap V3 pool (on-chain and off-chain)

Status 2022-04-28:
Partially implemented
The dev team informed security reviewers that the *EL-DWA* application (bet marketplace) validates the pools off-chain and the protocol does not take risk from using unofficial marketplaces.
Commit ID: *ac063014d07465c1288e474d56bb132ad0b0c9dc*

Description:
The protocol does not validate (on-chain) whether the ticket's pool is the pool deployed by Uniswap's factory. The malicious pools could return invalid values and control the business logic of resolving the bet.

How to implement:
We recommend performing such validation on-chain and off-chain.

- It is non-trivial to be implemented on-chain, because Uniswap's function *getPoolKey* (https://docs.uniswap.org/protocol/reference/periphery/libraries/PoolAddress#getpoolkey) requires not only token addresses but also *fee* parameter. It would require adding *fee* parameter in *TicketBinaryBet.Details* structure.
- The off-chain validation should be performed by *EL-DWA* system before submitting the *ticket* or before taking the *ticket*, depending on the level of restrictions.
- Another option is to perform the validation only off-chain and warn users in case of non-official pool but still allow to use them.

References:
V4: Communications
https://securing.github.io/SCSVS/1.2/0x13-V4-Communications.html

5.8. Consider adding zero address verification

Status 2022-04-28:
Implemented
Commit ID: *ac063014d07465c1288e474d56bb132ad0b0c9dc*

Description:
The constructor of LogiumBinaryBet (LogiumBinaryBet.sol#L73) does not check whether the addresses from parameters are zero addresses. The zero address in *_feeCollector* could lead to loss of fees.

How to implement:
At the *require* statements that make sure the addresses are not zero.

References:
SCSVS V11: Code clarity
https://securing.github.io/SCSVS/1.2/0x20-V11-Code-Clarity.html

5.9. Consider setting tighter restrictions for protocol parameters

Status 2022-04-28:
Implemented
Commit ID: *ac063014d07465c1288e474d56bb132ad0b0c9dc*

Description:
The function *issue* of LogiumBinaryBet (LogiumBinaryBet.sol#L117-L118) checks whether the *issuerWinFee* and *traderWinFee* parameters are not greater or equal to 100%.

How to implement:
Set lower maximum fee percent (e.g., 10%).

References:
SCSVS V11: Code clarity
https://securing.github.io/SCSVS/1.2/0x20-V11-Code-Clarity.html

6. TERMINOLOGY

This section explains the terms that are related to the methodology used in this report.

Threat

Any circumstance or event with the potential to adversely impact organizational operations (including mission, functions, image, or reputation), organizational assets, or individuals through an information system via unauthorized access, destruction, disclosure, modification of information, and/or denial of service.[1]

Vulnerability

Weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source.[1]

Risk

The level of impact on organizational operations (including mission, functions, image, or reputation), organizational assets, or individuals resulting from the operation of an information system given the potential impact of a threat and the likelihood of that threat occurring.[1]

The risk impact can be estimated based on the complexity of exploitation conditions (representing the likelihood) and the severity of exploitation results.

		Complexity of exploitation conditions		
		Simple	Moderate	Complex
Severity of exploitation results	Major	Critical	High	Medium
	Moderate	High	Medium	Low
	Minor	Medium	Low	Low

[1] NIST FIPS PUB 200: Minimum Security Requirements for Federal Information and Information Systems. Gaithersburg, MD: Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology.

7. CONTACT

Person responsible for providing explanations:

Damian Rusinek
e-mail: damian.rusinek@securing.pl
tel.: +48 12 425 25 75

securing
http://www.securing.pl
e-mail: info@securing.pl
Kalwaryjska 65/6
30-504 Kraków
tel./fax.: +48 (12) 425 25