Smart Contract Security Audit Report

Thales Markets

1. Contents

2. General Information

This report contains information about the results of the security audit of the Thales Markets (hereafter referred to as "Customer") SpeedMarkets smart contracts, conducted by Decurity in the period from 08/15/2023 to 08/18/2023.

2.1. Introduction

Tasks solved during the work are:

- Review the protocol design and the usage of 3rd party dependencies,
- Audit the contracts implementation,
- Develop the recommendations and suggestions to improve the security of the contracts.

2.2. Scope of Work

The audit scope included the following smart contract: https://github.com/thales-markets/contracts/blob/main/contracts/SpeedMarkets/MultiCollateralOnOffRamp.sol. Initial review was done for the commit 6dec917a5530db8da08cf5ec4d1324a3bf6effdf and the re-testing was done for the commit 748281eb6929ed763d6957dc4dfa05e4ebf3e323.

2.3. Threat Model

The assessment presumes actions of an intruder who might have capabilities of any role (an external user, token owner, token service owner, a contract). The centralization risks have not been considered upon the request of the Customer.

The main possible threat actors are:

- User,
- Protocol owner,
- Liquidity Token owner/contract.

The table below contains sample attacks that malicious attackers might carry out.

Table. Theoretically possible attacks

Attack	Actor
Contract code or data hijacking *Deploying a malicious contract or submitting malicious data*	Contract owner Token owner
Financial fraud *A malicious manipulation of the business logic and balances, such as a re-entrancy attack or a flash loan attack*	Anyone
Attacks on implementation *Exploiting the weaknesses in the compiler or the runtime of the smart contracts*	Anyone

2.4. Weakness Scoring

An expert evaluation scores the findings in this report, an impact of each vulnerability is calculated based on its ease of exploitation (based on the industry practice and our experience) and severity (for the considered threats).

2.5. Disclaimer

Due to the intrinsic nature of the software and vulnerabilities and the changing threat landscape, it cannot be generally guaranteed that a certain security property of a program holds.

Therefore, this report is provided “as is” and is not a guarantee that the analyzed system does not contain any other security weaknesses or vulnerabilities. Furthermore, this report is not an endorsement of the Customer’s project, nor is it an investment advice.

That being said, Decurity exercises best effort to perform their contractual obligations and follow the industry methodologies to discover as many weaknesses as possible and maximize the audit coverage using the limited resources.

3. Summary

As a result of this work, we have discovered a single high-risk exploitable security issue and a single medium-risk issue.

The other suggestions included fixing the low-risk issues and some best practices (see Security Process Improvement).

3.1. Suggestions

The table below contains the discovered issues, their risk level, and their status as of May 3, 2023.

Table. Discovered weaknesses

Issue	Contract	Risk Level	Status
onrampWithEth is vulnerable to DoS	contracts/SpeedMarkets/MultiCollateralOnOffRamp.sol	**High**	Fixed
Lack of stale price check	contracts/SpeedMarkets/MultiCollateralOnOffRamp.sol	**Medium**	Acknowledged
Lack of 0 address checks	contracts/SpeedMarkets/MultiCollateralOnOffRamp.sol	**Low**	Fixed
onrampWithEth lacks ammsSupported check	contracts/SpeedMarkets/MultiCollateralOnOffRamp.sol	**Low**	Fixed
Conjunctions used instead of nested expressions	contracts/SpeedMarkets/MultiCollateralOnOffRamp.sol	**Info**	Acknowledged
String error messages used instead of custom errors	contracts/SpeedMarkets/MultiCollateralOnOffRamp.sol	**Info**	Acknowledged

4. General Recommendations

This section contains general recommendations on how to improve overall security level.

The Findings section contains technical recommendations for each discovered issue.

4.1. Security Process Improvement

The following is a brief long-term action plan to mitigate further weaknesses and bring the product security to a higher level:

- Keep the whitepaper and documentation updated to make it consistent with the implementation and the intended use cases of the system,
- Perform regular audits for all the new contracts and updates,
- Ensure the secure off-chain storage and processing of the credentials (e.g. the privileged private keys),
- Launch a public bug bounty campaign for the contracts.

5. Findings

5.1. onrampWithEth is vulnerable to DoS

Risk Level: High

Status: Fixed in the commit 748281eb6929ed763d6957dc4dfa05e4ebf3e323.

Contracts:

- contracts/SpeedMarkets/MultiCollateralOnOffRamp.sol

Location: Lines: `110`. Function: `onrampWithEth`.

Description:

`onrampWithEth` function performs the following check at line 119: `require(IERC20Upgradeable(WETH9).balanceOf(address(this)) == amount);` . This check asserts that WETH balance of the contract equals to the provided amount. A malicious user can abuse this check via sending some WETH to the contract. This will always break the check, what will result in a DOS.

Remediation:

Consider replacing == with >= to avoid DOS vulnerability.

5.2. Lack of stale price check

Risk Level: Medium

Status: Acknowledged: we trust Chainlink oracles for BTC and ETH to always be fresh.

Contracts:

- contracts/SpeedMarkets/MultiCollateralOnOffRamp.sol

Location: Lines: `209, 223`. Function: `getMinimumReceived, getMaximumReceived`.

Description:

`getMinimumReceived` and `getMaximumReceived` functions perform `priceFeed.rateForCurrency` calls. `priceFeed.rateForCurrency` returns current rate, however it doesn't return update time. This makes update time ignored by this functions, which may result in a stale price data from ChainLink oracle.

Remediation:

Consider retrieving update time and checking that it has been updated at least one hour ago.

5.3. Lack of 0 address checks

Risk Level: Low

Status: Fixed in the commit 748281eb6929ed763d6957dc4dfa05e4ebf3e323.

Contracts:

- contracts/SpeedMarkets/MultiCollateralOnOffRamp.sol

Location: Function: `initialize`.

Description:

The following functions lack 0 address checks:

- `initialize` at line 67. Lacks zero address check for `_sUSD` parameter.
- `setSupportedCollateral` at line 249. Lacks zero address check for `collateral` parameter.
- `setSupportedAMM` at line 255. Lacks zero address check for `amm` parameter.
- `setWETH` at line 261. Lacks zero address check for `_weth` parameter.
- `setSUSD` at line 267. Lacks zero address check for `_usd` parameter.
- `setSwapRouter` at line 273. Lacks zero address check for `_router` parameter.
- `setPriceFeed` at line 279. Lacks zero address check for `_priceFeed` parameter.
- `setPriceFeedKeyPerAsset` at line 285. Lacks zero address check for `asset` parameter.
- `setPathPerCollateral` at line 291. Lacks zero address check for `asset` parameter.

Remediation:

Consider adding 0 address checks

5.4. onrampWithEth lacks ammsSupported check

Risk Level: Low

Status: Fixed in the commit 748281eb6929ed763d6957dc4dfa05e4ebf3e323.

Contracts:

- contracts/SpeedMarkets/MultiCollateralOnOffRamp.sol

Location: Function: `onrampWithEth`.

Description:

`onrampWithEth` function lacks `ammsSupported[msg.sender]` check. Lack of this check makes it callable by anyone. Since `onramp` function implements such check we consider this as an issue.

Remediation:

Consider adding the following check `require(ammsSupported[msg.sender], "Unsupported caller");` to the function.

5.5. Conjunctions used instead of nested expressions

Risk Level: Info

Status: Acknowledged: we prefer this way of writing the code for readability and the difference in L2 gas spent is not a factor here.

Description:

There're multiple occasions where the logical && operators are used instead of the nested expressions:

```
90 } else if (curveOnrampEnabled && (collateral == usdc || collateral == dai
|| collateral == usdt))

122 require(curveIndex > 0 && curveOnrampEnabled, "unsupported collateral");
```

Using nested is cheaper than using && multiple check combinations. There are more advantages, such as easier to read code and better coverage reports.

Remediation:

Consider using nested `if` expressions and multiple `require` statements.

5.6. String error messages used instead of custom errors

Risk Level: Info

Status: Acknowledged: we prefer semantically easier to read error messages and the difference in L2 gas spent is not a factor here.

Contracts:

- contracts/SpeedMarkets/MultiCollateralOnOffRamp.sol

Description:

The contracts make use of the `require()` to emit an error. While this is a perfectly valid way to handle errors in Solidity, it is not always the most efficient.

```
81: require(collateralSupported[collateral], "Unsupported collateral");
82: require(ammsSupported[msg.sender], "Unsupported caller");
105: require(msg.value > 0, "Can not exchange 0 ETH");
106: require(msg.value >= amount, "Amount ETH has to be larger than specified
amount");
107: require(curveIndex > 0 && curveOnrampEnabled, "unsupported collateral");
132: require(amountOut <= getMaximumReceived(collateral, collateralQuote),
"Amount above max allowed peg slippage");
156: require(amountOut <= getMaximumReceived(tokenIn, amountIn), "Amount above
max allowed peg slippage");
185: require(amountOut <= getMaximumReceived(tokenIn, amountIn), "Amount above
max allowed peg slippage");
```

Remediation:

Consider using custom errors as they are more gas efficient while allowing developers to describe the error in detail using NatSpec.

6. Appendix

6.1. About us

The Decurity team consists of experienced hackers who have been doing application security assessments and penetration testing for over a decade.

During the recent years, we've gained expertise in the blockchain field and have conducted numerous audits for both centralized and decentralized projects: exchanges, protocols, and blockchain nodes.

Our efforts have helped to protect hundreds of millions of dollars and make web3 a safer place.