PUBLIC

Code Assessment

of the Automation Consultancy Smart Contracts

March 30, 2022

Produced for

by

Contents

1 Executive Summary

Dear Chris,

Thank you for trusting us to help Oazo Apps Limited with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Automation Consultancy according to Scope to support you in forming an opinion on their security risks.

Oazo Apps Limited implements an automated management solution for Maker's collateralized debt positions. Users manage command triggers which are executed by bots. In the current implementation, users can automatically close a vault position should the collateralization go below a certain threshold. All in all, no high severity issues were uncovered. All the issues have been addressed.

The most critical subjects covered in our audit are functional correctness and access control. Security regarding all the aforementioned subjects is high.

The general subjects covered are upgradability, unit testing and gas efficiency. Security regarding all the aforementioned subjects is high. The specification provided was comprehensive.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1
• Code Corrected	1
Low-Severity Findings	9
• Code Corrected	8
• Specification Changed	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Automation Consultancy repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	7 March 2022	db2fcc8856b6204fd9c069a149120285bb7e9d23	Initial Version
2	28 March 2022	6dbc5a77d5e7e811fc03880163d94948c8f4b0dd	Version with fixes

For the solidity smart contracts, the compiler version `0.8.13` was chosen. The contracts in scope are all the contracts under the `contracts` directory excluding the `tests` directory.

2.1.1 Excluded from scope

All the contracts not mentioned in scope. Specifically, the system interacts with the Maker core system. All the interactions are assumed to work as intented. Moreover, the system under review delegates calls to the multiply-proxy-action. This is also assumed to work as intended. Finally, parts of the implementation are also part of the multiply-proxy-action. These also considered to be safe and work as intended.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Oazo Apps Limited offers an implementation of automated management for the Oasis.app. Users can perform a set of automated actions to manage their positions using a decentralized keeper network. The Oasis system is set to interact with the Maker ecosystem. Thus, a user adds a trigger for a specific action (also known as command) to a specific collateralized debt position (CDP). If at any point the conditions of the action are met, an authorized keeper (also known as caller) executes the command. In the current implementation, there is only one command available. That is, the automated closing of a position should the collateralization ratio drops below a threshold. The current implementation only supports vaults that have been created by the `CdpManager`.

The system consists of the following contracts:

ServiceRegistry

This contract serves as a single source of truth for the whole implementation. It maps the name of each service to the corresponding address of the contract that implements it. `ServiceRegistry` is controlled by its owner. The owner can transfer the ownership, change the required delay (explained later), add and remove trusted addresses, add, update and remove smart contracts and cancel pending actions. In order to counter a possible key compromise, `ServiceRegistry` implements a delay mechanism. To perform

an action, the owner must call a function of the contract twice, one to commit to the action and one to actually execute it given that enough time has elapsed. For canceling a commited action or removing an address from the registry no delay is required.

AutomationExecutor

This is the contract an authorized caller should call in order to execute a command. A caller is whitelisted by the owner of this contract. Apart from that, a caller can swap the tokens this contract holds from or to DAI by calling `swap`. The main purpose of this functionality is to exchange DAI, which is drawn from the vaults to compensate for the caller's gas, into ETH which is used to bribe the miners. The exchange will happen through the 1inch aggregator. The owner of the contract can set the exchange contract, add and remove callers from the whitelist, unwrap WETH the contract holds to ETH, and withdraw to their address tokens that remain in the contract including native ETH (`withdraw`). When a caller executes a call, they are allowed to send money to bribe the miner. Moreover, the callers are partially reimbursed for the transaction by sending themselves some of the ETH the contract holds. Note that they are reimbursed only for the actual execution of the command and the gas costs for the miner's bribe. Any user can deposit ETH directly to the contract. The system assumes there is always enough money stored in the contract to cover for bribe expenses even in periods of high congestion in the network.

AutomationBot

This is the contract with which the end-users, owners of a CDP, should interact. The contract is a delegate for the DSProxy of each user, thus, some of its external functions are executed in the context of the Proxy. The end-user calls `addTrigger` which then calls `addRecord` and adds a hash of all the details of the trigger together with the unique id of the CDP. Moreover, by calling `addTrigger` a user approves the `AutomationBot` contract to execute actions (commands in this particular case) on behalf of them on their CDPs. A user can replace one trigger by adding another one, remove a trigger, directly add or remove a record, and remove the approval to the automation bot contract. Note that `addTrigger` is executed in the context of the DSProxy while `addRecord` is executed in the context of the `AutomationBot` after querying its address from the ServiceRegistry. This means that there could be a case where the call a different version of the contract is used as a delegate and a different one is returned by the registry. Finally `AutomationBot` exposes the `execute` function. This is the function the `AutomationExecutor` should call in order to execute an automated action/command. `execute` will do the following steps:

- The data the caller wants to use to execute the command is sanitized.
- Some extra debt in DAI from the CDP is withdrawn in order to cover the gas costs for the caller. Note that there is a minimum amount of debt a CDP can create. This means that if the debt withdrawn is under the minimum allowed the transaction will fail.
- A check is performed of whether the conditions for the command to execute are met.
- The command is executed.
- The results of the command are sanitized.

Commands

The logic of the automated actions is implemented in the command contracts. Each command implements a different action. A command exposes the following interface:

- `isExecutionLegal`: Checks whether the precondition of the execution of the command actually holds.
- `execute`: Executes the command.
- `isExecutionCorrect`: Checks whether the postcondition of the execution of the command actually holds.

CloseCommand:

Implements the `Command` interface.

- `isExecutionLegal`: Checks whether the collateralization ratio is below a threshold. To do so it evaluates both the debt and the collateral in DAI using the next price of the OSM. Note that when the collateralization ratio is calculated extra debt might have been created to reimburse the caller.
- `execute`: closes the position either by closing the vault. It then either sends the collateral to the user or trades the collateral to DAI and sends it to the user. The action is delegated to the Multiply Proxy Actions (MPA) contract, and, thus, executed within the context of the command.
- `isExecutionCorrect`: Checks whether the CDP has been closed, i.e., both collateral (`ink`) and debt (`art`) are 0.

McdView

This a utility contract, exposing view functionality useful to other contracts of the system. It exposes the following functions:

- `getVaultInfo`: Queries the collateral and the debt for a system.
- `getPrice`: Uses the OSM to query the current price of a particular collateral token. The price is given in 18 decimals.
- `getNextPrice`: Uses the OSM to query the next price of a particular collateral token. To query the price on-chain, one should be whitelisted by the owner of this contract. The next price is given in 18 decimals.
- `getRatio`: Returns the collateralization ratio for a particular CDP.
- `approve`: allows the owner of the contract to whitelist an address to be able to query the next price on-chain.

McdUtils

This is another utility contract. The most important function it implements is `drawDebt` which draws more debt from a CDP, if needed, in order to cover the caller's gas expenses.

2.2.1 Trust Model

The owners of the contracts are trusted entities which are expected to only make non-harmful changes to the states of the contracts they own. The callers are also trusted entities and they should not try to withdraw more debt to cover their gas fees than they actually need or make swaps that are unfavourable to the system.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1

- Missing ETH Unwrapping Code Corrected

Low-Severity Findings	9

- Accidental Approval Revocation Code Corrected
- Dead Code Code Corrected
- Missing Sanity Checks Code Corrected
- Outdated Compiler Code Corrected
- Redundant Authorization Code Corrected
- Rounding Errors Code Corrected
- Specification Discrepancies Specification Changed
- Use Safe Calls Code Corrected
- Zero Debt Vaults Code Corrected

6.1 Missing ETH Unwrapping

Design Medium Version 1 Code Corrected

The `AutomationExecutor` allows its owner to withdraw tokens or native ETH. As the Oazo team informed us, the main purpose of this function is to withdraw ETH converted from DAI. The exchange contract is not able to handle native ETH but needs its wrapped version. Hence there is a need for unwrapping functionality to be able to use native ETH. However, such functionality is not implemented.

Code corrected:

`unwrapWETH` has been implemented. It can be called only by the owner of the `AutomationExecutor` contract and calls `weth.withdraw` function.

6.2 Accidental Approval Revocation

Design Low Version 1 Code Corrected

On `removeTrigger` a user can accidentally set the `removeAllowance` variable to true. If this happens the approval to the automation bot is revoked. A user can only re-approve the automation bot indirectly by adding another trigger since there is no function to do this directly. Another option for the user is to use the revocation manager.

Code corrected

The functionality to grant approval to the automation bot has been added.

6.3 Dead Code

Design Low Version 1 Code Corrected

In `ServiceRegistry`, the functions `addTrustedAddress`, `removeTrustedAddress` and `isTrusted` manipulate the `trustedAddresses` mapping. However, this mapping is not used by the rest of the implementation. Moreover, `McdUtils.convertTo18` is also never used.

Code corrected:

The dead code has been removed.

6.4 Missing Sanity Checks

Design Low Version 1 Code Corrected

There are multiple points in the contract where sanity checks are missing. The absence of such checks can allow users to assign invalid values to variables:

1. `AutomationBot.addRecord` does not sanitize `triggerType` and `triggerData`. Also, `triggerData` includes `slLevel` which can have invalid values that cause reverts inside `isExecutionLegal`. Should the variables store invalid data the corresponding trigger will not be able to execute at a later point in time.
2. `AutomationExecutor.(transferOwnership, setExchange)` does not sanitize the input values. Notice that there is no delayed execution implemented for this contract.
3. `ServiceRegistry.constructor` sanitizes the required delay by requiring it to be less than the maximum integer. However, any value close to the maximum integer would be valid and problematic for the system.

Code corrected:

1. `AutomationBot.addRecord` validates the `triggerData` by using `commandAddress.isTriggerDataValid`.
2. `AutomationExecutor` now sanitizes the data.
3. The maximum required delay is now set to 30 days.

6.5 Outdated Compiler

Design Low Version 1 Code Corrected

The system is compiled using solidity version `0.8.4`. However, more recent versions are available. At the time of writing `0.8.13` is the most recent version.

Code corrected:

The compiler version `0.8.13` is now used.

6.6 Redundant Authorization

Design | Low | Version 1 | Code Corrected

In `McdUtils.drawDebt`, the authorization of the AutomationBot to `daiJoin` is given every time the call is made since the following line is always executed:

```
IVat(vat).hope(daiJoin);
```

We noticed that the implementation is quite similar to https://github.com/OasisDEX/multiply-proxy-actions/blob/develop/contracts/multiply/MultiplyProxyActions.sol#L205. However, there is always a check if the authorization is needed.

Code corrected:

The current implementation only grants authorization to `daiJoin`, if it has not been given before.

6.7 Rounding Errors

Design | Low | Version 1 | Code Corrected

According to the specification (https://github.com/dapphub/ds-math), `DSMath.wmul` should be used with two Wads and `DSMath.rdiv` should be used with two Rays. However, this is not true in `McdView.getRatio` where the following snippet exists:

```
uint256 ratio = rdiv(wmul(collateral, price), debt);
```

Here, `wmul` is applied on `collateral` which is a wad and `price` which is a 9-decimal number. The result will also be a 9-decimals number. Later, `rdiv` is applied on this 9-decimal number and `debt` which is a wad. The result is a Wad instead of a Ray. Wrong usage of `DSMath` leads to rounding errors. This means that a vault is rendered closable at different levels than the users have actually set.

Code corrected:

In the current implementation price is a Wad and the problematic snippet has been rewritten to:

```
return wdiv(wmul(collateral, price), debt);
```

6.8 Specification Discrepancies

Design | Low | Version 1 | Specification Changed

There are some discrepancies between the provided specification and the actual implementation. We follow the enumeration provided in the documentation.

System Requirements & Assumptions:

ServiceRegistry:

1. The addresses are not trusted, in the sense that the `trustedAddresses` mapping is not used. 4. `removeTrustedAddress` also does not use the `delayedExecution` modifier.

AutomationBot:

5. If a user executes `addRecord` directly to add a trigger then `cdpAllow` will not be called. 13. The permission might have been revoked by the user.

Smart Contract Architecture:

AutomationExecutor:

- `swapTokenForDai` is documented but does not exist.
- `swap` is implemented but not documented.

Specification changed:

All the discrepancies in the specification have been fixed.

6.9 Use Safe Calls

Correctness Low Version 1 Code Corrected

`AutomationExecutor` exposes `swap` and `withdraw` functions. These functions, interact with ERC20 contracts by calling `ERC20.approve` and `ERC20.transfer`. However, these calls will fail, should a user try to interact with a USDT contract. For example, a user sends accidentally USDT to the `AutomationExecutor`, the amount will remain stuck there since any withdrawal by the owner will fail.

Code corrected:

`SafeERC20` library is now used. `ERC20.approve` has been replaced with `SafeERC20.safeIncreaseAllowance` and `ERC20.transfer` has been replaced with `SafeERC20.safeTransfer`.

6.10 Zero Debt Vaults

Design Low Version 1 Code Corrected

`McdUtils.getRatio` returns 0 when the debt of a vault is 0. This means than `CloseCommand.isExecutionLegal` will return true, and thus, render the vault closable. This means that a caller might try to close the a zero debt vault. When the `AutomationExecutor` calls `AutomationBot.execute`, the latter will try to withdraw extra debt (`drawDaiFromVault`) to cover its gas costs. However, the Maker system only allows users to withdraw debt that exceeds a specific limit (dust). Since the amount of extra debt withdrawn to cover the caller is small compared to the dust amount, the whole transaction will revert.

Code corrected:

An execution of the `CloseCommand` is legal as long as the collateralization ratio is not 0.