PUBLIC

Audit Report

of the DEFI Automation V2 Smart Contracts

August 25, 2023

Produced for

by

CHAINSECURITY

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help DIGE with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Automation V2 according to Scope to support you in forming an opinion on their security risks.

DIGE implements the second version of the automation bot which opens the system to protocols other than Maker and introduces new kinds of triggers and grouped validation mechanics.

The most critical subjects covered in our audit are functional correctness, access control, and non-custodiality. Functional correctness and access control are good. Non-custodiality is good. However, due to several issues arising from administrator powers, see Execution data is not validated and Execution reentrancy may be possible, and the proxy action contracts being out-of-scope, there may be unforeseeable consequences for non-custodiality.

The general subjects covered are upgradeability, unit testing, documentation and error handling.

In summary, we find that the codebase provides a good level of security. However, there may unforeseeable consequences given the reasons above. In case the administrators are trusted, the codebase provides a good 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

s

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	2
• Code Corrected	2
High -Severity Findings	3
• Code Corrected	3
Medium -Severity Findings	4
• Code Corrected	2
• Risk Accepted	2
Low -Severity Findings	19
• Code Corrected	14
• Code Partially Corrected	1
• Risk Accepted	4

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 V2 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	19 Dec 2022	c2fb8685f9a62a19a6122e2016d9c6f33beac124	Initial Version
2	25 Jan 2023	cb48baeb0ab7a9122c245950d4b2237313bfb716	Second Version
3	21 Mar 2023	352be89d3a595b4fc1bdc49a6bcc7a8cf0317dea	Third Version
4	19 Apr 2023	0e1e8d70c5196487c311004daab91719861aef27	Fourth Version
5	20 Apr 2023	2f4dfa3fbf8580229f40c20a4a13c006c689e403	Fifth Version

For the solidity smart contracts, the compiler version `0.8.13` was chosen.

The following files were in scope:

```
contracts/AutomationBot.sol
contracts/AutomationBotStorage.sol
contracts/AutomationExecutor.sol
contracts/McdUtils.sol
contracts/McdView.sol
contracts/ServiceRegistry.sol
contracts/adapters/AAVEAdapter.sol
contracts/adapters/DPMAdapter.sol
contracts/adapters/MakerAdapter.sol
contracts/commands/AutoTakeProfitCommand.sol
contracts/commands/BaseMPACommand.sol
contracts/commands/BasicBuyCommand.sol
contracts/commands/BasicSellCommand.sol
contracts/commands/CloseCommand.sol
contracts/validators/ConstantMultipleValidator.sol
```

In ~~Version 4~~, the following file renamings occured:

```
AutoTakeProfitCommand.sol -> MakerAutoTakeProfitCommandV2.sol
BasicBuyCommand.sol -> MakerBasicBuyCommandV2.sol
BasicSellCommand.sol -> MakerBasicSellCommandV2.sol
CloseCommand.sol -> MakerStopLossCommandV2.sol
```

Additionally, AutomationBotStorage.sol was removed.

In ~~Version 5~~, the following file renamings occured:

```
BotLike.sol -> IAutomationBot.sol
MakerAdapter.sol -> MakerSecurityAdapter.sol & MakerExecutableAdapter.sol
```

Note that the Maker Adapter has been split into two files.

2.1.1 Excluded from scope

All other files are out of scope.

Further, the multiply proxy actions contract is fully out of scope. For the commands, it is expected that it performs the necessary computations. For the core, it is expected that any usage of multiply proxy actions will not lead to reentrancy vectors.

Similarly, the Aave proxy actions contract is fully out of scope (usage in Aave adapter).

2.2 System Overview

This system overview describes the last received version (Version 3) 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.

DIGE offers a new version of the Automation Bot architecture which introduces more flexibility in terms of protocols that it can interact with. Users can register a set of automated actions to manage their positions using a decentralized keeper network.

2.2.1 Service Registry

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 actions available to the owner are adding new addresses under hashes that have never been used and removing addresses under the existing hashes. 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, once to commit to the action and once to execute it given that enough time has elapsed. For canceling a committed action or removing an address from the registry no delay is required.

2.2.2 Automation Executor

The Automation Executor is the only contract that can call `AutomationBot.execute()`. Hence, it provides a function `execute()` to perform the keeper logic. Besides executing a command, it reimburses the gas spent by the caller and further allows bribing the miner of the block. Note that only whitelisted callers can call `execute()`.

Whitelisted callers can swap any token to BNB using `swap()` on Pancake V3. The main purpose is to exchange the arbitrary payment tokens, that were received on command execution, to BNB so that the callers can be reimbursed and the miners can be bribed. Note that `swap()` can only be called by the whitelisted addresses.

Note that it is assumed that any new version (new deployment losing all triggers) of the Automation Bot will be accompanied with a redeployment of the entire system.

Also note that it has some `public` getters for querying Pancake Swap V3 60 second TWAP prices. These areintended to be used as helpers and not suitable for on-chain usage.

2.2.3 Automation Bot

The AutomationBot is the main entry point for users. Note that it consists of proxy action functionality (delegate-called by the DSProxy or a DPM proxy) and core functionality (called directly). Note that generally all state is kept in a second contract AutomationBotStorage whose functions are only callable by the AutomationBot.

To add a group of triggers, users delegate-call into `addTriggers()` which first validates whether the triggers in the group match a given group type. However, an exception to this validation is the `SINGLE_TRIGGER_GROUP_TYPE` group type which has no validation. Then, permissions needed for the command execution are given to the AutomationBot by delegatecalling into the `permit()` function of the adapter matching the trigger type. Next, the triggers are added iteratively to the AutomationBot with a call to `addRecord()`. Once all triggers are added, the group is published through a call to the function `emitGroupDetails()` of the AutomationBot.

The call to `addRecord()` validates the trigger data and the caller, and stores a hash of the trigger's data along with the command address and a `continuous` flag. Note that the flag allows specifying whether a trigger should be removed after execution or whether it should be executable multiple times. Further, note that triggers can be replaced.

Note that `addTriggers()` is a wrapper around `addRecord()`. Hence, any validation and group logic is assumed to be fully optional.

To remove triggers, users delegate-call into `removeTriggers()` which first calls `removeRecord()` on the AutomationBot for each trigger to remove. First, the caller is validated using the adapter for the command. Second, the provided data is validated against the hash created when adding a trigger. Last, the trigger is removed.

Finally, AutomationBot exposes the `execute()` function which only the AutomationExecutor can call to execute a command. Its execution can be outlined as follows:

1. Validates the provided trigger data against its stored hash.
2. Ensures that the execution is allowed with `ICommand.isExecutionLegal()`.
3. Retrieves the user-payment with a delegatecall to the `IAdapter.getCoverage()` function of the execution adapter. Note that this is all defined by the keeper.
4. Gives the executed command the permissions with a delegate-call to `IAdapter.permit()` function of the regular adapter.
5. Calls `ICommand.execute()`.
6. Removes the trigger if `!continuous`.
7. Removes the permissions (similarly to 4.).
8. Validates the execution using `ICommand.isExecutionCorrect()`.

2.2.3.1 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:

- `isTriggerDataValid`: Checks whether the trigger data provided by the user matches the command.
- `isExecutionLegal`: Checks whether the pre-condition of the execution of the command holds.
- `execute`: Executes the command.
- `isExecutionCorrect`: Checks whether the post-condition of the execution of the command holds.

BaseMPACommand is the base contract for MPA (Multiple Proxy Actions) targeted command contracts. It implements some basic validation functions, a `getVaultDebt()` function that is used to query the

debt of a particular CDP and the `executeMPAMethod()` that delegates the call to the MPA with the provided execution data.

Four command implementations inherit from the BaseMPACommand:

- **BasicBuyCommand:**

  Increases the leverage of a CDP by decreasing its collateralization ratio.

  Pre-execution conditions:

  1. The next collateralization ratio of the CDP is greater than the execution collateralization ratio
  2. The next price of the collateral is smaller than the maximum buying price specified by the user
  3. The block base fee is smaller than the maximum base fee specified by the user

  Post-execution condition: The collateralization ratio of the CDP is within the user-defined target interval

- **BasicSellCommand:**

  Decreases the leverage ratio of a CDP by increasing its collateralization ratio.

  Pre-execution conditions:

  1. The next collateralization ratio of the CDP is smaller than the execution collateralization ratio
  2. The next price of the collateral is greater than the minimum buying price specified by the user
  3. The block base fee is smaller than the maximum base fee specified by the user or the position is liquidatable

  Post-execution condition: The collateralization ratio of the CDP is within the user-defined target interval

- **AutoTakeProfitCommand:**

  Takes profit from a CDP by closing it and withdrawing the collateral.

  Pre-execution conditions:

  1. The CDP is not empty
  2. The next price of the collateral is greater than the user-defined execution price
  3. The block base fee is smaller than the maximum base fee specified by the user or the position is liquidatable

  Post-execution condition: The CDP is closed and its collateral and debt value are equal to zero

- **CloseCommand:**

  Closes a CDP.

  Pre-execution conditions:

  1. The CDP is not empty
  2. The next collateralization ratio of the CDP is greater than the user-specified stop-loss collateralization ratio

  Post-execution condition: The CDP is closed and its collateral and debt value are equal to zero

2.2.3.2 Adapters

Adapters are actions specific to a command which include:

- `permit()`: Gives the necessary permissions. Needs to be delegate-called by an address that can give out permissions.
- `getCoverage()`: Collects the payment.
- `canCall()`: Defines whether or not an operator can perform an action.

Note that there are two kinds of adapters, the regular ones, and the executable ones. Regular ones implement the access control while executable ones handle the protocol-specific coverage. Note that an adapter may be both, regular and executable.

The DPMAdapter is a regular adapter. It replicates the caller validation of the modifier `authAndWhitelisted` defined in the DPM's AccountImplementation contract. Note that this validation reproduces using `AccountGuard.canCall()`. The `permit()` function calls `permit()`. `getCoverage()` is not implemented.

The AaveAdapter is an executable adapter. It implements `getCoverage()` which delegatecalls into the AaveProxyActions contract's function `drawDebt()`. It is assumed that this borrows a token on Aave and transfers it to a recipient.

The Maker adapter is a regular and an executable adapter. `canCall()` replicates the `cdpAllowed` modifier of the CDPManager while `permit()` wraps the call to `cdpAllow()` on the CDPManager.

2.2.4 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.

2.2.5 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.

2.2.6 Trust Model

In the system, DIGE expect to guarantee their users that these invariants hold:

1. Creating a trigger record is an agreement. Never in the future should the addresses that are interacted with change.
2. No actions should be able to modify existing triggers, unless the trigger belongs to the actor.
3. A trigger execution should only be able to act on a position in the way the trigger defines it.
4. When a user defines a setting off-chain, there shouldn't be a way to maliciously change the expected outcome of this setting.

However, DIGE uses centralized actors to maintain the system:

- Keepers
    - Can draw any amount of coverage on users' position
    - Can provide any paths for token swaps which might lead to slippage or loss of tokens if chosen maliciously.
    - Can provide any execution data for triggers, which could lead to undesired actions on the users' positions.
- Owner of the AutomationExecutor
    - Can add and remove any keepers to the AutomationExecutor
    - Can transfer the ownership of the AutomationExecutor
- Owner of the ServiceRegistry
    - Can add any non-existing service address to the registry
    - Can remove any existing service from the registry
    - Can transfer the ownership of the registry
    - Can change the required delay
    - Can cancel pending actions

With the model in mind, all of these actors should be trusted addresses because of their abilities to act on user positions.

Note that this audit does not consider them as such so as to assess the potential consequence of a malicious behavior.

2.2.7 Changes in Version 4

- The AutomationBotStorage contract has been removed. The state now resides in AutomationBot.
- Adapters are now called with `call` instead of `delegatecall`. Hence, the AutomationBot does not hold the permissions anymore but the adapters do.
- The users can now set a maximum coverage amount in trigger data. That is validated now additionally in the `getCoverage` functions and in the ConstantMultipleValidator.

2.2.8 Changes in Version 5

The executable adapter only receives permissions when getting the coverage now. Thus, the maker adapter had to be split into separate contracts.

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, ChainSecuritys 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:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- 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	2

- Execution Data Is Not Validated Risk Accepted
- Execution Reentrancy May Be Possible Risk Accepted

Low -Severity Findings	5

- Future Debt Validation Does Not Take Bounds Into Account Code Partially Corrected Risk Accepted
- Negative gasRefund Possible Risk Accepted
- Overflow When Computing the Used Gas Risk Accepted
- Too Low execCollRatio Risk Accepted
- Too Low slLevel Risk Accepted

5.1 Execution Data Is Not Validated

Security Medium Version 3 Risk Accepted

CS-OazoAutomV2-001

The executors always provide some execution data for the execution of the trigger. However, this data should correspond to the trigger's intention, but is not validated for the most part. Note that it still is limited in its ability to act maliciously because of the post-execution checks.

Risk accepted:

DIGE accepts the risk.

5.2 Execution Reentrancy May Be Possible

Security Medium Version 3 Risk Accepted

CS-OazoAutomV2-002

Commands delegate calls to action smart contracts to act on the user's position. However, some of these action smart contracts can contain logic that would permit other external contracts.

In this case, external contracts could contain malicious code that could reenter the core contracts to add/remove triggers, or simply temper with the user's position.

Risk accepted:

DIGE accepts the risk.

5.3 Future Debt Validation Does Not Take Bounds Into Account

Design | Low | Version 1 | Code Partially Corrected | Risk Accepted

CS-OazoAutomV2-003

In the BasicSellCommand, the post-execution debt is computed in `isExecutionLegal()` to validate that it will be bigger than the dust limit of the cdp's ilk:

```
uint256 futureDebt = (debt * nextCollRatio - debt * wad) /
    (trigger.targetCollRatio.wad() - wad);
```

Note that in this computation the target collateralization ratio is accounted as is and used to predict future debt. However, the post-execution cdp's debt might not exactly correspond to this collateralization ratio, this is the reason behind the existence of the deviation parameter.

This makes it possible for the `isExecutionLegal()` function to return true even though execution will fail afterward because the debt is smaller than the dust limit.

Code partially corrected and risk accepted:

Now, the upper bound for the collateralization ratio is used. However, this will yield the minimal value `futureDebt` could reach. Hence, the function could return `false` in some scenarios where the execution could be legal.

However, DIGE accepts the risk.

5.4 Negative `gasRefund` Possible

Correctness | Low | Version 1 | Risk Accepted

CS-OazoAutomV2-004

`gasRefund` should be used to decrease the necessary coverage according to the gas refunds. However, it is of type `int` and, hence could be negative so that the computation

```
uint256(int256(initialGasAvailable - finalGasAvailable) - gasRefund);
```

could increase the gas used instead of lowering it.

Note that with the current trust model, the executor providing this gas refund value should not be trusted.

Risk accepted:

DIGE states that no changes were made since calls come from trusted callers. However, disallowing negative numbers (e.g. by using `uint256`) could help. Hence, DIGE accepts the risk.

5.5 Overflow When Computing the Used Gas

Correctness Low Version 1 Risk Accepted

CS-OazoAutomV2-005

The cast from `int256` to `uint256` could overflow in the following code

```
uint256(int256(initialGasAvailable - finalGasAvailable) - gasRefund);
```

if `gasRefund` is greater than the computed gas used.

Risk accepted

The gas refund is limited now by `10**12`. However, this does not protect against bad `gasRefund` argument since it could still be greater than `(initialGasAvailable - finalGasAvailable)`.

5.6 Too Low `execCollRatio`

Design Low Version 1 Risk Accepted

CS-OazoAutomV2-006

In BasicSell, the `execCollRatio` could be below the liquidation ratio. Hence, the trigger validity check could allow an unexecutable trigger to be added.

Risk accepted:

DIGE replied that the data is validated with the calldata creation on the front-end.

5.7 Too Low `slLevel`

Design Low Version 1 Risk Accepted

CS-OazoAutomV2-007

In CloseCommand, the `slLevel` could be below the liquidation ratio. Hence, the trigger validity check could allow an unexecutable trigger to be added.

Risk accepted:

DIGE replied that the data is validated with the calldata creation on the front-end.

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	2

- Arbitrary Actions and Storage Manipulations Code Corrected
- Executor Could Draw an Unbounded Amount of Coverage Code Corrected

High -Severity Findings	3

- Executing a DPM Command Reverts Code Corrected
- Removing Arbitrary Triggers Possible Code Corrected
- Upgrades Break AutomationBot Code Corrected

Medium -Severity Findings	2

- Possible Reentrancy Through Execution Code Corrected
- addRecord() Does Not Unlock Code Corrected

Low -Severity Findings	14

- Wrong Event Argument Code Corrected
- Incorrect Argument for ApprovalGranted Event Code Corrected
- Bad Trigger Ids Emitted Code Corrected
- Command Validation Functions Can Revert Code Corrected
- Contracts Do Not Extend the Interfaces Code Corrected
- Group Counter Starts at 2 Code Corrected
- Left Comments Code Corrected
- Missing Validation in AutoTakeProfitCommand Code Corrected
- Redundant Approval Code Corrected
- Removing Non-Callers Emits an Event Code Corrected
- Specification Mismatches Code Corrected
- Unequal Array Lengths Code Corrected
- Unused Imports Code Corrected
- Unused Parameter, Variable and Event Code Corrected

6.1 Arbitrary Actions and Storage Manipulations

Security Critical Version 3 Code Corrected

CS-OazoAutomV2-011

The owner of the ServiceRegistry could perform arbitrary storage manipulations and actions on any position. Namely, the delegatecalls in the storage contract (in `executePermit()` and `executeCoverage()`) allows them to execute arbitrary code in the context of the storage contract. In

addition, a malicious command could be added along with its adapter so as to execute malicious actions on a user's position.

Let's consider different scenarios for understandability reasons:

Modifying storage in AutomationBotStorage:

1. Command X is added to the ServiceRegistry, it is a no-op command and is immediately executable.
2. Adapter X is also added.
3. The owner calls `addRecord()` to add a record for the command.
4. `execute()` happens. Arbitrary code is executed during the delegatecall made in adapter X.

Executing a malicious command:

1. Command X is added to the ServiceRegistry, it is a command that can, given a position, unwind all funds to the owner's address and is immediately executable.
2. Adapter X is also added, its `canCall()` function always return `true`.
3. The owner calls `addRecord()` to add a record for the command and is allowed to do so.
4. `execute()` happens. Arbitrary code is executed during the execute made in AutomationBot and funds are stolen.

Ultimately, the service registry owner may self-destruct the contract, change any storage locations, and execute any operations on any position that the AutomationBotStorage has allowance on.

Note that there are many possibilities when adding arbitrary commands and adapter, and the examples listed above are non-exhaustive.

Code corrected:

DIGE has changed the design. Users approve adapters which are callable by the automation bot.

6.2 Executor Could Draw an Unbounded Amount of Coverage

Security Critical Version 3 Code Corrected

CS-OazoAutomV2-026

The typical execution flow of a trigger is

1. Pre-condition checks.
2. Getting the coverage with `getCoverage()` so that the user pays for the fees.
3. Execution.
4. Post-condition checks.

Note that `getCoverage()` will typically create additional debt which changes also the collateralization ratio, and that the coverage amount is not bounded upwards.

A trigger could potentially be non-executable before the execution but become executable once `getCoverage()` has been called.

This gives the executor an ability to execute some triggers on demand.

Further note that there can be other unforseeable consequences, but they are limited by thepost-execution validation.

The changes in Version 4 created an alternative for getting arbitrary coverage that severly amplifies the owners possibilities. With this new version, the executable adapters alway holds permissions for the position (if set up correctly), so that it can draw coverage when needed. Now consider the following scenario, where the governance would want to draw excessive coverage on a target position:

1. Governance adds a malicious command along with a malicious security adapter assigned to it. The malicious adapter returns `true` on `canCall` calls and the command allows execution but does not perform any operation. They also assign to the command a legit executable adapter, the one that the target position permitted to.
2. Governance adds a trigger record with the trigger data pointing to the target position, and using the malicious command.
3. Execution occurs (governance has control over the executors) and the excecution adapter is called to get coverage and can (nearly) drain the target position. The `maxCoverage` variable was passed along the malicious `triggerData` in step 2, so can be arbitrary.

Code corrected:

A user-defined maximum and a payment token has been introduced. The executable adapter is now only granted permissions when getting the coverage.

6.3 Executing a DPM Command Reverts

Design High Version 1 Code Corrected

CS-OazoAutomV2-016

A typical command execution flow of AutomationExecutor's `execute()` function is as follows (some steps are omitted for clarity):

1. Giving permissions to the command address
2. Calling the command to execute
3. Disallowing the command

Note that when the automation bot permits the command address, it delegates the call to the specific adapter. This should work because the automation bot was already permitted when the trigger was added.

In the case of the DPMAdapter, the `permit()` function starts with this line:

```
require(canCall(triggerData, msg.sender), "dpm-adapter/not-allowed-to-call");
```

This is executed in the context of the AutomationBot, so the `msg.sender` is still the AutomationExecutor, which has not been permitted by the owner of the proxy. The execution call will fail in all cases except when the owner of the proxy has manually permitted the AutomationExecutor.

Code corrected:

DIGE changed `msg.sender` to `address(this)`.

6.4 Removing Arbitrary Triggers Possible

Security High Version 1 Code Corrected

CS-OazoAutomV2-024

`addRecord()` adds a trigger and allows to replace a given trigger id and data with.

The following check is performed:

```
require(
    replacedTriggerId == 0 || adapter.canCall(replacedTriggerData, msg.sender),
    "bot/no-permissions-replace"
);
```

However, following issues arise:

1. `replacedTriggerData` is not validated to match the `replacedTriggerId` (as in `checkTriggersExistenceAndCorrectness()`).
2. The security adapter for the new command is used. However, the replaced trigger id may be another command that has another adapter.

Users relying on the execution could be liquidated or miss out on profit scenarios.

Code corrected:

`replacedTriggerData` is now checked against the hash of the `replacedTriggerId` since Version 2. The original adapter is used since Version 3.

6.5 Upgrades Break AutomationBot

Correctness High Version 1 Code Corrected

CS-OazoAutomV2-030

The split of the storage from the AutomationBot intends to

```
[...] enable the upgradeability of AutomationBot implementation without the need for migration
for all of the triggers.
```

However, note that the AutomationBot contract is given permissions for the positions but changing the address does not transfer the permissions. Ultimately, no previous triggers can be executed.

Code corrected:

Now, the AutomationBot does not hold any permissions. However, permissions are granted to the storage contract. Thus, the AutomationBot calls functions on the storage contract that can grant commands the necessary rights.

6.6 Possible Reentrancy Through Execution

Security Medium Version 1 Code Corrected

CS-OazoAutomV2-015

When a caller executes a trigger, the automation bot permits the specific command to act on the target position/cdp for the execution call and removes this permission after it. To execute a trigger the `execute()` function is called on the command. Note that none of the implementations of this function have access control or reentrancy protection. On execution, external smart contracts that belong to third parties will be called and it can lead to reentrancy possibilities. Entering again the `execute()` function would then be possible.

Code corrected:

Now, the AutomationBot and the commands have reentrancy locks. Given that only a command at a time is granted permissions, this protects commands from being malicously executed.

6.7 `addRecord()` Does Not Unlock

Correctness Medium Version 1 Code Corrected

CS-OazoAutomV2-021

`addRecord()` adds new triggers. It calls `lock()` to ensure that `emitGroupDetails()` has the right calldata. However, note that `emitGroupDetails()`, which unlocks the lock counter, is only called if `addRecord()` is used through `addTriggers()`. Note that this is not necessarily the case. Hence, the contract could temporarily be locked by direct calls to `addRecord()` (without using the proxy actions function `addTriggers()`). Calling `addTriggers()` would revert because of the inconsistency between the emit group details and the `lockCount`.

Code corrected:

The lock is always cleared using the new function `clearLock()` in `addTriggers()` and `removeTriggers()`.

6.8 Wrong Event Argument

Correctness Low Version 4 Code Corrected

CS-OazoAutomV2-032

In the AutomationBot smart contract, both events `ApprovalGranted` and `ApprovalRemoved` are emitted with the AutomationBot address as argument, when the permission is actually granted to or removed from adapters.

Code corrected:

The correct argument is now used.

6.9 Incorrect Argument for `ApprovalGranted` Event

Correctness Low Version 2 Code Corrected

CS-OazoAutomV2-019

The event `ApprovalGranted(bytes indexed triggerData, address approvedEntity)` should pass the approved entity as second argument. In `AddTriggers()`, the approval is granted to the `AutomationBotStorage` but the argument specifies the `automationBot`.

Code corrected

The correct argument is now used.

6.10 Bad Trigger Ids Emitted

Correctness Low Version 1 Code Corrected

CS-OazoAutomV2-013

The `addTrigger()` function triggers a call to `emitGroupDetails()` with bad trigger ids.

Consider the following scenario:

1. `addTriggers()` is used.
2. `firstTriggerId` is 0 since no triggers have been added so far.
3. `addRecord()` is called.
4. `appendTriggerRecord()` is called. The trigger is added with id 1 and the trigger counter is set to 1. The call returns.
5. `addRecord()` emits an event with trigger id 1 and returns.
6. `addTriggers()` sets the local variable `triggerIds[0]` to 0.
7. An event is emitted with the wrong id.

Ultimately, events with errors are emitted.

Code corrected:

The first trigger id is now computed correctly (triggers counter + one).

6.11 Command Validation Functions Can Revert

Design Low Version 1 Code Corrected

CS-OazoAutomV2-031

Command smart contracts implement 3 different data/execution flow validation functions that return a boolean:

- `isTriggerDataValid()`
- `isExecutionLegal()`
- `isExecutionCorrect()`

All of those are wrapped in a `require()` in the automation bot so that the trigger data is valid and the pre and post-execution conditions also are.

However, some of these functions in AutoTakeProfitCommand fail on some conditions instead of just returning false:

1. If the owner of the cdp is the address 0 in `isExecutionLegal()`

2. If the trigger execution price is greater than the next price in `isTriggerDataValid()`

Code corrected:

The functions return `false` now.

6.12 Contracts Do Not Extend the Interfaces

Design | Low | Version 1 | Code Corrected

CS-OazoAutomV2-017

Most of the contracts interact with each other based on the interface definitions. For example, the `addTrigger()` call `BotLike.addRecord()` on the AutomationBot. However, the AutomationBot contract itself does not explicitly implement the BotLike interface. Similarly, other contracts do not implement interfaces.

Without this, there are no compile-time guarantees that the contract will be compatible with the calls to the functions that the interface defines. This can lead to potential runtime errors and exceptions that are hard to debug. It is important to explicitly define that the contracts implement the corresponding interfaces, to minimize such errors.

Code corrected:

The usage of interfaces has been improved.

6.13 Group Counter Starts at 2

Design | Low | Version 1 | Code Corrected

CS-OazoAutomV2-018

Why does the group counter start at `2` when AutomationBot emits the first `TriggerGroupAdded` event?

Code corrected:

The group counter starts now at 1.

6.14 Left Comments

Design | Low | Version 1 | Code Corrected

CS-OazoAutomV2-020

Several TODOs are left open. Additionally, other comments are left. For example,

```
or type ? do we allow execution of the same command with new contract - waht if
contract rev X is broken ? Do we force migration (can we do it)?
```

Such comments can increase complexity.

Code corrected:

The comments have been removed.

6.15 Missing Validation in AutoTakeProfitCommand

Correctness Low Version 1 Code Corrected

CS-OazoAutomV2-012

The AutoTakeProfitCommand smart contract takes care of closing a cdp when its collateral has reached a certain price. This kind of trigger should not be continuous because of the way this command works.

However, `isTriggerDataValid()` does not check wether it is or not.

Code corrected:

Now, only non-continous triggers are valid for AutoTakeProfitCommand.

6.16 Redundant Approval

Design Low Version 1 Code Corrected

CS-OazoAutomV2-022

The AutomationExecutor approves Pancake Swap's router contract twice which increases gas consumptionand adds additional complexity.

Code corrected:

The redundant approval was removed.

6.17 Removing Non-Callers Emits an Event

Design Low Version 1 Code Corrected

CS-OazoAutomV2-023

Removing non-callers emits a `CallerRemoved` event. In contrast, adding callers that are already callers skips the event emission.

Code corrected:

Only callers can now be removed.

6.18 Specification Mismatches

Correctness Low Version 1 Code Corrected

CS-OazoAutomV2-025

Note that there are multiple specification mismatches:

1. BasicSellCommand implies `trigger.execCollRatio.wad() >= nextCollRatio` which contradicts the code in the equality case.
2. The documentation describes `maxBuyPrice` instead of `minSellPrice`.
3. The documentation states that Automation Bot is a stateless contract. However, it has a state variable `lockCount`.
4. The adapter section specifies that adapters are delegatecalled in the AutomationBot context. However, the functions introduced functions `executePermit()` and `executeCoverage()` changed this behaviour since they are delegatecalled from the storage contract now.

Specification corrected

All 4 points were correct in the documentation.

6.19 Unequal Array Lengths

Design | Low | Version 1 | Code Corrected

CS-OazoAutomV2-027

In `addTriggers()`, the array parameters could have distinct lengths. The execution in such cases is unspecified. Similarly, this holds for `removeTriggers()`.

Code corrected:

All arrays are not checked against for length.

6.20 Unused Imports

Design | Low | Version 1 | Code Corrected

CS-OazoAutomV2-028

Some smart contracts have some unused imports, some examples are:

In AutomationExecutor:

1. FullMath
2. IExchange
3. ICommand

In BaseMPACommand:

1. AutomationBot

In McdView:

1. ICommand
2. BotLike

Note that this is an incomplete list of examples.

Code corrected:

All unused imports have been removed.

6.21 Unused Parameter, Variable and Event

Design Low Version 1 Code Corrected

CS-OazoAutomV2-029

1. The function `AutomationExecutor.execute()` has an unused parameter `cdpId`.
2. AutomationExecutor stores the DAI address as an immutable which is unused.
3. The AutomationBot has an event `ApprovalGranted` that is not used.
4. `AUTOMATION_BOT_STORAGE_KEY` is unused in AutomationBot.

Code corrected:

The event `ApprovalGranted` is now used and the others have been removed.

6.22 The CloseCommand Does Not Inherit BaseMPACommand

Informational Version 1 Code Corrected

CS-OazoAutomV2-014

The CloseCommand smart contract does not inherit the BaseMPACommand smart contract even though it executes through the MPA.

Code corrected:

Inheritance was improved.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Floating Dependencies Version

Informational **Version 1**

CS-OazoAutomV2-009

The versions of the contract libraries in `package.json` are not fixed. Please consider the following examples:

```
"@openzeppelin/contracts": "^4.5.0"
```

The caret `^version` will accept all future minor and patch versions while fixing the major version. With new versions being pushed to the dependency registry, the compiled smart contracts can change. This may lead to incompatibilities with older compiled contracts. If the imported and parent contracts change the storage slot order or change the parameter order, the child contracts might have different storage slots or different interfaces due to inheritance.

In addition, this can lead to issues when trying to recreate the exact bytecode.

7.2 Floating Pragma

Informational **Version 1**

CS-OazoAutomV2-010

DIGE uses a floating pragma solidity ^0.8.13. Contracts should be deployed with the same compiler version and flags that have been used during testing and audit. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively, see https://swcregistry.io/docs/SWC-103 (snapshot).

7.3 Unecessary External Calls

Informational **Version 4**

CS-OazoAutomV2-008

The AutomationBot smart contract must sometimes call itself because of the delegate logic, however there are cases where this is not necessary:

1. In the `emitGroupDetails()` function
2. In the `addRecord()` function

Note that both of these function call `automationBot` like an external contract, but will not be called in a delegate context.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 AutomationBot Permission Handling

Note Version 1

It should be made clear to the user and in the documentation that the functions `addTriggers()` and `removeTriggers()` are only helper functions, and that data must be provided in certain ways so that the system works correctly.

For example, triggers will not be able to execute if a user adds multiple triggers that point to different positions/cdps through `addTriggers()` without manually granting permission to the AutomationBot for each of these (except the first one).

Also, removing multiple triggers with the `removeAllowance` flag set to true through `removeTriggers()` will only remove the allowance for the first position/cdp pointed by the trigger at index 0.

Further, a user could have cleared all of their triggers but still have some active permission on the AutomationBot for their positions.

8.2 Data for Execution

Note Version 1

The data for the execution can be freely selected by the callers. There are the following limitations:

- Post-conditions of the command must hold.
- Selectors are validated in commands.

However, users should be aware that the coverage token can be an arbitrary token (e.g. if Aave position is managed) and, hence, the execution could lead to unwanted risks (e.g. being exposed to very volatile tokens). Further, users should be aware that the execution could be configured so that re-executions could occur faster.

Ultimately, users should always consider the possibility of bad execution data. However, note that the callers are trusted addresses.

8.3 Oracle Not Suitable for On-Chain Usage

Note Version 1

AutomationExecutor implements an oracle for Pancake Swap V3 TWAP prices. This is intended to be used off-chain. Using the oracle on-chain could lead to issues.

First, the TWAP is hardcoded to 60 seconds which would allow for simple manipulations. Second, the function iterates over a list of pools and tries to choose the biggest one. It does so by selecting the one with the highest WETH balance. However, this is not a safe value for estimating actual pool size. Note that there are some further discrepancies compared to Pancake Swap V3's reference implementation of price oracle. For example, AutomationExecutor adds one to the TWAP interval array values (which Pancake Swap

does not), does not round down to negative infinity (which Pancake Swap does) and does not have a precisionof computation as high as Pancake Swap in some scenarios.

Generally, the oracle-like getters are not suitable for on-chain usage. Also, the off-chain usage must be done carefully.

8.4 `getCoverage()` Implications

Note **Version 1**

The typical execution flow of a trigger is

1. Pre-condition checks.
2. Getting the coverage with `getCoverage()` so that the user pays for the fees.
3. Execution.
4. Post-condition checks.

Note that `getCoverage()` will typically create additional debt which changes also the collateralization ratio.

Users should be aware that in the BasicBuy command, where a precondition is that the collateralization ratio must be above a certain threshold, could technically be violated after `getCoverage()`. Hence, the execute function could be executed on a vault where the collateralization ratio is below the threshold.

Further note that there can be other unforeseeable consequences. Users should be aware that the configuration requires consideration of the coverage.

In **Version 4** DIGE repeats the precondition checks after getting the coverage. Users should be aware of this and configure their positions accordingly.

8.5 `getTick` Computes the Square Root Price

Note **Version 1**

The naming of `getTick()` suggests that a tick is returned. However, it computes the square root price.