BLOCKSEC

Security Audit
Report for DeFiCard Contracts

Date: June 3, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Chapter 1 Introduction **1**

1.1 About the Audit Target . . . 1
1.2 Disclaimer . . . 1
1.3 Procedure of Auditing . . . 2
1.3.1 Security Issues . . . 2
1.3.2 Additional Recommendation . . . 3
1.4 Security Model . . . 3

Chapter 2 Findings **4**

2.1 Security Issue . . . 5
2.1.1 Lack of input token validation for Morpho withdrawals . . . 5
2.1.2 Stale spent amount in function `getAccountDailyLimitByCard()` . . . 6
2.1.3 Lack of delay restriction on function `cancelOperation()` . . . 6
2.1.4 Inconsistent timelock enforcement for selector risk level changes . . . 7
2.2 Recommendation . . . 9
2.2.1 Align `maxBorrow` calculation with Morpho's health check implementation . . . 9
2.2.2 Add validation for settlement configuration . . . 10
2.2.3 Adopt accurate variable naming . . . 11
2.2.4 Add validation during action execution . . . 12
2.2.5 Support share-based withdrawal in the Morpho handler . . . 12
2.2.6 Add allowance revocation logic . . . 14
2.2.7 Align interface and event with implementation . . . 14
2.2.8 Add validations on action execution results . . . 15
2.3 Note . . . 16
2.3.1 Potential centralization risks . . . 16
2.3.2 Proxy deployment and implementation binding should be atomic . . . 17
2.3.3 Assumptions on external protocols . . . 17
2.3.4 Authorize the handler before specific Morpho operations . . . 18
2.3.5 Trusted off-chain logic . . . 19
2.3.6 Assumptions on whitelisted handlers . . . 19
2.3.7 Ensure order resolution during settlement replacement . . . 19
2.3.8 Storage layout consistency across core versions . . . 20
2.3.9 Refund mechanism relies on off-chain logic . . . 20
2.3.10 Weird ERC20 tokens . . . 20
2.3.11 Cancel stale operations . . . 21

Report Manifest

Item	Description
Client	bitgetwallet
Target	DeFiCard Contracts

Version History

Version	Date	Description
1.0	June 3, 2026	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-06-04 02:34:36+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About the Audit Target

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The audit target (hereinafter referred to as the Target) of this audit is the code repository [1] of DeFiCard Contracts of bitgetwallet.

The DeFiCard Contracts project introduces an on-chain payment card infrastructure that connects card-based payment workflows with ERC20 token settlement and optional DeFi protocol actions. Its core business logic covers authorisation, settlement, and refund flows, allowing payment amounts to be escrowed, released to settlement recipients, cancelled back to users, or refunded through designated refund callers. The architecture is built around `DeFiCardRouter`, an entry contract that routes calls through enabled core implementations such as `DeFiCardCoreV1` via fallback delegatecall. Each card is backed by a dedicated `DeFiCardSettlement` contract, and a handler integrates with DeFi protocols so payment flows can optionally use DeFi actions.

Note this audit only focuses on the smart contracts in the following directories/files:

- contracts

Other files are not within the scope of the audit. Additionally, all dependencies of the Target are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
defi-card	`Version 1`	`7734e794219472ac73027799ca2ffaf3b756556a`
	`Version 2`	`314505ea7b85bbaf65a23af70b04ab7066f875dd`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset.

[1] https://github.com/bitgetwallet/defi-card

Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the Target, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of the Target.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan the Target with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of the Target and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation
* Error-prone randomness
* Oracle security
* Business logic correctness
* Semantic and functional consistency
* Emergency mechanism

* Economic and incentive impact

1.3.2 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found **four** potential security issues. Besides, we have **eight** recommendations and **eleven** notes.

- Low Risk: 4
- Recommendation: 8
- Note: 11

ID	Severity	Description	Category	Status
1	Low	Lack of input token validation for Morpho withdrawals	Security Issue	Fixed
2	Low	Stale spent amount in function `getAccountDailyLimitByCard()`	Security Issue	Fixed
3	Low	Lack of delay restriction on function `cancelOperation()`	Security Issue	Confirmed
4	Low	Inconsistent timelock enforcement for selector risk level changes	Security Issue	Confirmed
5	-	Align `maxBorrow` calculation with Morpho's health check implementation	Recommendation	Confirmed
6	-	Add validation for settlement configuration	Recommendation	Fixed
7	-	Adopt accurate variable naming	Recommendation	Confirmed
8	-	Add validation during action execution	Recommendation	Fixed
9	-	Support share-based withdrawal in the Morpho handler	Recommendation	Confirmed
10	-	Add allowance revocation logic	Recommendation	Confirmed
11	-	Align interface and event with implementation	Recommendation	Fixed
12	-	Add validations on action execution results	Recommendation	Confirmed
13	-	Potential centralization risks	Note	-
14	-	Proxy deployment and implementation binding should be atomic	Note	-
15	-	Assumptions on external protocols	Note	-
16	-	Authorize the handler before specific Morpho operations	Note	-
17	-	Trusted off-chain logic	Note	-
18	-	Assumptions on whitelisted handlers	Note	-
19	-	Ensure order resolution during settlement replacement	Note	-
20	-	Storage layout consistency across core versions	Note	-

21	-	Refund mechanism relies on off-chain logic	Note	-
22	-	Weird ERC20 tokens	Note	-
23	-	Cancel stale operations	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Lack of input token validation for Morpho withdrawals

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `DeFiCardCoreV1`, function `_executeActions()` allows the `WITHDRAW` action to carry non-zero `tokenIn` and `amountIn`, and then transfers `tokenIn` from `payer` to the handler before execution. However, in the contract `DeFiCardMorphoBlueMarketHandler`, function `_withdraw()` ignores `tokenIn` and `amountIn`, performing the withdrawal based on `tokenOut`. If an action is incorrectly constructed with non-zero `tokenIn` and `amountIn`, tokens are transferred into the handler contract unexpectedly. These tokens are not returned to `payer` and can only be recovered through the privileged function `rescueERC20()`.

```
258         } else if (action.actionType == ActionType.WITHDRAW) {
259             require(action.tokenIn != paymentToken, InvalidActionTokenIn());
260             require(action.tokenOut == paymentToken, InvalidActionTokenOut());
261             if (action.tokenIn == address(0)) {
262                 require(action.amountIn == 0, InvalidActionAmountIn());
263             } else {
264                 require(action.amountIn > 0, InvalidActionAmountIn());
265             }
266         } else if (action.actionType == ActionType.REPAY) {
267             require(action.amountIn > 0, InvalidActionAmountIn());
268             require(action.tokenIn != paymentToken && action.tokenIn != address(0),
                    InvalidActionTokenIn());
269             require(action.tokenOut == address(0), InvalidActionTokenOut());
270         }
271
272         if (action.amountIn > 0 && action.tokenIn != address(0) && action.tokenIn !=
                paymentToken) {
273             TokenHelper.safeTransferFrom(action.tokenIn, payer, action.handler, action.amountIn
                    );
274         }
```

Listing 2.1: contracts/core/DeFiCardCoreV1.sol

```
57  function _withdraw(Action calldata action, address payer) internal virtual returns (uint256
        withdrawAmountOut) {
58      MorphoMarketParams memory mp = abi.decode(action.extraData, (MorphoMarketParams));
```

```
59        uint256 balanceBefore = TokenHelper.balanceOf(action.tokenOut, payer);
60        if (action.tokenOut == mp.loanToken) {
61            IMorpho(MORPHO).withdraw(mp, action.minAmountOut, 0, payer, payer);
62        } else if (action.tokenOut == mp.collateralToken) {
63            IMorpho(MORPHO).withdrawCollateral(mp, action.minAmountOut, payer, payer);
64        } else {
65            revert InvalidMorphoMarketHandlerTokenOut();
66        }
67        _checkPositionHealth(mp, payer, MAX_BORROW_BPS);
68        withdrawAmountOut = TokenHelper.balanceOf(action.tokenOut, payer) - balanceBefore;
69        require(withdrawAmountOut >= action.minAmountOut, InsufficientOutput());
70    }
```

Listing 2.2: contracts/handlers/morphoblue/market/DeFiCardMorphoBlueMarketHandler.sol

Impact User funds unrelated to the withdrawal action may be stranded in the handler contract and require privileged recovery.

Suggestion Enforce zero `tokenIn` and `amountIn` in contract `DeFiCardMorphoBlueMarketHandler`.

2.1.2 Stale spent amount in function `getAccountDailyLimitByCard()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `DeFiCardRouter`, function `getAccountDailyLimitByCard()` returns the stored `dailySpent` directly, without checking whether a new day has begun. The `dailySpent` represents the accumulated spent amount as of the last payment, and is only reset on the first payment of a new day. As a result, the function may return a stale value if no payment has occurred within the current day.

```
263    function getAccountDailyLimitByCard(bytes32 cardId, address account) public view returns (
           uint96, uint96) {
264        Card memory card = _getCard(cardId);
265        if (card.cardId == bytes32(0)) return (0, 0);
266        StorageLib.CardSpendLimitStorage storage ss = StorageLib.getCardSpendLimitStorage();
267        StorageLib.SpendLimitState storage st = ss.settings[account][cardId];
268        uint96 effective = st.dailyLimit == 0 ? card.defaultDailyLimit : st.dailyLimit;
269        return (effective, st.dailySpent);
270    }
```

Listing 2.3: contracts/DeFiCardRouter.sol

Impact Off-chain consumers may underestimate the available daily quota between day boundaries.

Suggestion Return zero spent amount when the current time crosses into a new day.

2.1.3 Lack of delay restriction on function `cancelOperation()`

Severity Low

Status Confirmed

Introduced by Version 1

Description In contract DeFiCardTimelock, function setSelectorLevels() does not enforce that the delay assigned to function cancelOperation() must be shorter than the delay of the operations it can cancel. Consequently, if function cancelOperation() is assigned a longer timelock delay than a target operation, that operation may become executable before the cancellation takes effect.

```
64   function setSelectorLevels(bytes4[] calldata selectors, RiskLevel[] calldata levels)
65       external
66       timelocked
67   {
68       uint256 length = selectors.length;
69       if (length == 0 || length != levels.length) revert ArrayLengthMismatch();
70       StorageLib.TimelockStorage storage timelockStorage = StorageLib.getTimelockStorage();
71       for (uint256 i = 0; i < length;) {
72           bytes4 selector = selectors[i];
73           require(!timelockStorage.lockedHigh[selector], CanNotUpdateSelectorLevelWhenInLocked(
                 selector));
74           timelockStorage.selectorLevel[selector] = levels[i];
75           emit SelectorLevelSet(selector, levels[i]);
76           unchecked {
77               ++i;
78           }
79       }
80   }
```

Listing 2.4: contracts/DeFiCardTimelock.sol

```
98   function cancelOperation(bytes32 opId) external timelocked {
99       StorageLib.TimelockStorage storage timelockStorage = StorageLib.getTimelockStorage();
100      if (timelockStorage.executableAt[opId] == 0) revert NoSuchOperation(opId);
101      _consumePending(timelockStorage, opId);
102      emit OperationCancelled(opId);
103  }
```

Listing 2.5: contracts/DeFiCardTimelock.sol

Impact The cancellation mechanism may fail to prevent the execution of an unexpected pending operation.

Suggestion Revise the code logic accordingly.

Feedback from the project The project states that this behavior is by design. Function cancelOperation() is classified as Low risk and is executed immediately, while off-chain monitoring will assist in disclosing and validating pending proposals before execution.

2.1.4 Inconsistent timelock enforcement for selector risk level changes

Severity Low

Status Confirmed

Introduced by Version 1

Description In the contract DeFiCardTimelock, the function setSelectorLevels() can change the risk level of a function selector after an operation has already been queued. However, queued operations do not snapshot the delay that was in effect at scheduling time. Instead, function _timelocked() derives the effective delay from the selector's current risk level. As a result, if a selector is later downgraded to LOW, an already queued operation with the same calldata can be executed immediately (line 24). Conversely, updating the selector level with a non-zero delay does not update the stored executableAt and the operation still uses the original scheduled timestamp. This leads to inconsistent timelock enforcement and allows pending operations to execute under a weaker delay than the one originally applied.

```
64  function setSelectorLevels(bytes4[] calldata selectors, RiskLevel[] calldata levels)
65      external
66      timelocked
67  {
68      uint256 length = selectors.length;
69      if (length == 0 || length != levels.length) revert ArrayLengthMismatch();
70      StorageLib.TimelockStorage storage timelockStorage = StorageLib.getTimelockStorage();
71      for (uint256 i = 0; i < length;) {
72          bytes4 selector = selectors[i];
73          require(!timelockStorage.lockedHigh[selector], CanNotUpdateSelectorLevelWhenInLocked(
                selector));
74          timelockStorage.selectorLevel[selector] = levels[i];
75          emit SelectorLevelSet(selector, levels[i]);
76          unchecked {
77              ++i;
78          }
79      }
80  }
```

Listing 2.6: contracts/DeFiCardTimelock.sol

```
19  function _timelocked() internal returns (bool locked) {
20      StorageLib.TimelockStorage storage timelockStorage = StorageLib.getTimelockStorage();
21      uint256 activeDelay = _getDelay(msg.sig);
22      bytes32 opId = keccak256(msg.data);
23      uint256 execAt = timelockStorage.executableAt[opId];
24      if (activeDelay == 0) {
25          _checkPlanProposalAuthorized();
26          if (execAt != 0) _consumePending(timelockStorage, opId);
27          if (timelockStorage.operationData[opId].length == 0) timelockStorage.operationData[opId
                ] = msg.data;
28          _recordTimelockExecution(timelockStorage, opId, msg.sig);
29          return false;
30      }
31      if (execAt == 0) {
32          _checkPlanProposalAuthorized();
33          uint256 executableAt = block.timestamp + activeDelay;
34          timelockStorage.executableAt[opId] = executableAt;
35          timelockStorage.operationData[opId] = msg.data;
36          timelockStorage.pendingIds.add(opId);
```

```
37            emit OperationScheduled(opId, msg.sig, executableAt);
38            return true;
39        }
40        _checkExecuteProposalAuthorized();
41        if (block.timestamp < execAt) revert DelayNotElapsed(opId, execAt);
42        _consumePending(timelockStorage, opId);
43        _recordTimelockExecution(timelockStorage, opId, msg.sig);
44        return false;
45    }
```

Listing 2.7: contracts/DeFiCardTimelock.sol

Impact The processing of pending operations is not consistent when the selector's risk level is updated.

Suggestion Revise the code logic accordingly.

Feedback from the project The project states that the timelock behavior is consistent with the intended design. Function `setSelectorLevels()` is classified as `High` risk, so its proposal execution time will occur after the corresponding function-level change has already taken effect. Besides, when a function is downgraded from `High` risk to `Low` risk, immediate execution is the expected behavior. The project will also use timelock monitoring to assist with tracking and reviewing all proposals.

2.2 Recommendation

2.2.1 Align `maxBorrow` calculation with Morpho's health check implementation

Status Confirmed

Introduced by `Version 1`

Description The local `maxBorrow` amount calculation is intended to determine the borrowable value of a user's collateral under the market LLTV. However, its arithmetic form (line 138) does not exactly mirror Morpho's own health check calculation. Even if the result is usually equivalent, using a different calculation style may introduce subtle rounding differences in edge cases.

```
123   function _checkPositionHealth(MorphoMarketParams memory mp, address user, uint256 safetyBps)
          internal view {
124       bytes32 id = keccak256(abi.encode(mp));
125
126       (, uint128 borrowShares, uint128 collateral) = IMorpho(MORPHO).position(id, user);
127       if (borrowShares == 0) return;
128
129       (,, uint128 totalBorrowAssets, uint128 totalBorrowShares,,) = IMorpho(MORPHO).market(id);
130
131       uint256 borrowed = _mulDivUp(
132           uint256(borrowShares),
133           uint256(totalBorrowAssets) + VIRTUAL_ASSETS,
134           uint256(totalBorrowShares) + VIRTUAL_SHARES
135       );
```

```
136
137         uint256 collateralPrice = IMorphoOracle(mp.oracle).price();
138         uint256 maxBorrow = uint256(collateral) * collateralPrice / ORACLE_PRICE_SCALE * mp.lltv /
                WAD;
139         uint256 allowedBorrow = maxBorrow * safetyBps / BPS;
140
141         if (borrowed > allowedBorrow) {
142             revert PositionUnhealthy(borrowed, allowedBorrow);
143         }
144     }
```

Listing 2.8: contracts/handlers/morphoblue/market/DeFiCardMorphoBlueMarketHandler.sol

```
514     function _isHealthy(MarketParams memory marketParams, Id id, address borrower) internal view
            returns (bool) {
515         if (position[id][borrower].borrowShares == 0) return true;
516
517         uint256 collateralPrice = IOracle(marketParams.oracle).price();
518
519         return _isHealthy(marketParams, id, borrower, collateralPrice);
520     }
521
522     /// @dev Returns whether the position of `borrower` in the given market `marketParams` with
            the given
523     /// `collateralPrice` is healthy.
524     /// @dev Assumes that the inputs `marketParams` and `id` match.
525     /// @dev Rounds in favor of the protocol, so one might not be able to borrow exactly `
            maxBorrow` but one unit less.
526     function _isHealthy(MarketParams memory marketParams, Id id, address borrower, uint256
            collateralPrice)
527         internal
528         view
529         returns (bool)
530     {
531         uint256 borrowed = uint256(position[id][borrower].borrowShares)
532             .toAssetsUp(market[id].totalBorrowAssets, market[id].totalBorrowShares);
533         uint256 maxBorrow = uint256(position[id][borrower].collateral).mulDivDown(collateralPrice,
                ORACLE_PRICE_SCALE)
534             .wMulDown(marketParams.lltv);
535
536         return maxBorrow >= borrowed;
537     }
```

Listing 2.9: Morpho.sol

Suggestion Align `maxBorrow` calculation with Morpho's health check implementation.

2.2.2 Add validation for settlement configuration

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Functions `registerCard()` and `setCardConfig()` configure the `settlement` contract for a card. However, they do not validate that the settlement contract correctly matches the card router, card ID, and payment token. If an incorrect settlement contract is configured, authorization may be routed through the incorrect contract.

```
112   function registerCard(CardConfigParams calldata params) external timelocked {
113       _checkBytes32(params.config.cardId);
114       _checkString(params.config.symbol);
115       _checkAddress(params.config.paymentToken);
116       StorageLib.CardConfigStorage storage cs = _checkCardExistAndReturnStorage(params.config.
              cardId, false);
117       Card storage card = cs.cards[params.config.cardId];
118       card.cardId = params.config.cardId;
119       card.symbol = params.config.symbol;
120       card.paymentToken = params.config.paymentToken;
121       _applyCardConfig(cs, params.config.cardId, params);
122       card.cardEnabled = true;
123       emit CardRegistered(
124           params.config.cardId,
125           params.config.symbol,
126           params.config.paymentToken,
127           params.config.settlement,
128           params.config.settleReceiver
129       );
130   }
```

Listing 2.10: contracts/DeFiCardRouter.sol

```
138   function setCardConfig(CardConfigParams calldata params) external timelocked {
139       StorageLib.CardConfigStorage storage cs = _checkCardExistAndReturnStorage(params.config.
              cardId, true);
140       _applyCardConfig(cs, params.config.cardId, params);
141       emit CardConfigSet(params.config.cardId);
142   }
```

Listing 2.11: contracts/DeFiCardRouter.sol

Suggestion In function `_applyCardConfig()`, verify the `CARD_ROUTER`, `CARD_ID` and `CARD_PAYMENT_TOKEN` in the new settlement contract.

2.2.3 Adopt accurate variable naming

Status Confirmed

Introduced by `Version 1`

Description In contract `DeFiCardCoreV1`, variable `isMultiAsset` is set to true when `params.actions.length > 0` (line 38). However, this name is misleading as it implies that the corresponding authorization involves multiple assets, yet the variable is also set to false when there is only the `paymentToken` asset involved.

```
27   function auth(address coreAddress, AuthParams calldata params)
28       external
```

```
29        onlyAuthCaller(params.cardId)
30        whenNotPaused
31        nonReentrant
32    {
33        _checkAmount(params.authAmount);
34        _checkAddress(params.payer);
35        uint256 totalAuthAmount = params.authAmount + params.feeAmount;
36        require(params.directTransferAmount <= totalAuthAmount, InvalidDirectTransferAmount());
37
38        bool isMultiAsset = (params.actions.length > 0);
```

Listing 2.12: contracts/core/DeFiCardCoreV1.sol

Suggestion Revise the naming to reflect the actual semantics of the variable.

2.2.4 Add validation during action execution

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `DeFiCardMorphoBlueMarketHandler`, the function `_supply()` does not use the variable `action.tokenOut` and implicitly assumes it to be `address(0)`. However, the function does not enforce this check.

```
82    function _supply(Action calldata action, address payer) internal virtual returns (uint256) {
83        require(TokenHelper.balanceOf(action.tokenIn, address(this)) >= action.amountIn,
              HandlerTokenInNotEnough());
84        MorphoMarketParams memory mp = abi.decode(action.extraData, (MorphoMarketParams));
85        bytes32 id = keccak256(abi.encode(mp));
86        require(action.tokenIn == mp.collateralToken, InvalidMorphoMarketHandlerTokenIn());
87        (,, uint256 supplyCollateralBefore) = IMorpho(MORPHO).position(id, payer);
88        TokenHelper.safeApproveMaxIfNeeded(action.tokenIn, MORPHO, action.amountIn);
89        IMorpho(MORPHO).supplyCollateral(mp, action.amountIn, payer, "");
90        (,, uint256 supplyCollateralAfter) = IMorpho(MORPHO).position(id, payer);
91        require(supplyCollateralAfter - supplyCollateralBefore >= action.minAmountOut,
              InsufficientOutput());
92        return 0;
```

Listing 2.13: contracts/handlers/morphoblue/market/DeFiCardMorphoBlueMarketHandler.sol

Suggestion Revise the code logic accordingly.

2.2.5 Support share-based withdrawal in the Morpho handler

Status Confirmed

Introduced by `Version 1`

Description In the contract `DeFiCardMorphoBlueMarketHandler`, the function `_withdraw()` invokes Morpho's `withdraw()` function with `action.minAmountOut` as the `assets` parameter and hard-codes `shares` to `0`. This leaves no path to fully withdraw by specifying shares. For a

full exit, an asset-denominated invocation is unreliable since interest accrual makes the exact redeemable amount unpredictable in advance. The Morpho documentation[1] recommends passing shares for full withdrawals to eliminate rounding errors and dust. It is recommended to add a share-based withdrawal path in the function `_withdraw()`, mirroring the implementation in the function `_repay()`.

```
57  function _withdraw(Action calldata action, address payer) internal virtual returns (uint256
        withdrawAmountOut) {
58      MorphoMarketParams memory mp = abi.decode(action.extraData, (MorphoMarketParams));
59      uint256 balanceBefore = TokenHelper.balanceOf(action.tokenOut, payer);
60      if (action.tokenOut == mp.loanToken) {
61          IMorpho(MORPHO).withdraw(mp, action.minAmountOut, 0, payer, payer);
62      } else if (action.tokenOut == mp.collateralToken) {
63          IMorpho(MORPHO).withdrawCollateral(mp, action.minAmountOut, payer, payer);
64      } else {
65          revert InvalidMorphoMarketHandlerTokenOut();
66      }
67      _checkPositionHealth(mp, payer, MAX_BORROW_BPS);
68      withdrawAmountOut = TokenHelper.balanceOf(action.tokenOut, payer) - balanceBefore;
69      require(withdrawAmountOut >= action.minAmountOut, InsufficientOutput());
70  }
```

Listing 2.14: contracts/handlers/morphoblue/market/DeFiCardMorphoBlueMarketHandler.sol

```
95  function _repay(Action calldata action, address payer) internal virtual returns (uint256) {
96      MorphoMarketParams memory mp = abi.decode(action.extraData, (MorphoMarketParams));
97      require(action.tokenIn == mp.loanToken, InvalidActionTokenIn());
98      require(TokenHelper.balanceOf(action.tokenIn, address(this)) >= action.amountIn,
            HandlerTokenInNotEnough());
99      bytes32 id = keccak256(abi.encode(mp));
100     (, uint128 borrowedShares,) = IMorpho(MORPHO).position(id, payer);
101     require(borrowedShares != 0, MorphoBlueNoDebtToRepay());
102     IMorpho(MORPHO).accrueInterest(mp);
103     (,, uint128 totalBorrowAssets, uint128 totalBorrowShares,,) = IMorpho(MORPHO).market(id);
104     uint256 owedAssets = _mulDivUp(
105         uint256(borrowedShares),
106         uint256(totalBorrowAssets) + VIRTUAL_ASSETS,
107         uint256(totalBorrowShares) + VIRTUAL_SHARES
108     );
109     TokenHelper.safeApproveMaxIfNeeded(action.tokenIn, MORPHO, action.amountIn);
110     uint256 actual;
111     if (action.amountIn >= owedAssets) {
112         (actual,) = IMorpho(MORPHO).repay(mp, 0, borrowedShares, payer, "");
113     } else {
114         (actual,) = IMorpho(MORPHO).repay(mp, action.amountIn, 0, payer, "");
115     }
116     if (action.amountIn > actual) {
117         TokenHelper.safeTransfer(action.tokenIn, payer, action.amountIn - actual);
118     }
119     require(actual >= action.minAmountOut, InsufficientOutput());
120     return 0;
```

[1] https://docs.morpho.org/build/borrow/concepts/market-mechanics#practical-implementation-snippets

```
121    }
```

Listing 2.15: contracts/handlers/morphoblue/market/DeFiCardMorphoBlueMarketHandler.sol

Suggestion Add a share-based withdrawal path in the function `_withdraw()`.

2.2.6 Add allowance revocation logic

Status Confirmed

Introduced by `Version 1`

Description Functions `_repay()` and `_supply()` grant token allowances to Morpho to facilitate supply and repayment operations. However, they do not revoke these allowances after the operation completes, leaving residual allowances that could potentially be exploited by malicious actors. It is recommended to implement allowance revocation logic immediately after each operation to ensure no residual allowances remain.

```
87    function safeApproveMaxIfNeeded(address token, address spender, uint256 amount) internal {
```

Listing 2.16: contracts/libs/TokenHelper.sol

Suggestion Add explicit allowance revocation by invoking `approve(MORPHO, 0)`.

Feedback from the project The project states that the current allowance handling logic will remain unchanged, as the currently integrated protocols have undergone security reviews. For higher-risk protocols, explicit allowance revocation logic will be added where necessary.

2.2.7 Align interface and event with implementation

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `DeFiCardRouter`, two declarations diverge from their corresponding interface and event definitions.

1. The function `getCardAccessCallers()` is defined in the router but is not declared in the interface `IDeFiCardRouter`.

2. The event `CardAccountDailyLimitSet` declares `dailyLimit` as `uint256`, while the function `setAccountDailyLimitByCard()` actually passes a `uint96` value.

It is recommended to align the interface and the event with the actual implementation.

```
218    function getCardAccessCallers(bytes32 cardId, CardAccessCallerType callerType, uint256 offset,
           uint256 limit)
```

Listing 2.17: contracts/DeFiCardRouter.sol

```
218    function getCardAccessCallers(bytes32 cardId, CardAccessCallerType callerType, uint256 offset,
           uint256 limit)
219        public
220        view
221        returns (address[] memory callers, uint256 total)
222    {
```

Listing 2.18: contracts/DeFiCardRouter.sol

```
259   /// @notice Emitted when an account changes its daily limit on a card.
260   event CardAccountDailyLimitSet(bytes32 indexed cardId, address indexed account, uint256
          dailyLimit);
```

Listing 2.19: contracts/interfaces/IDeFiCardRouter.sol

```
152
153   function setAccountDailyLimitByCard(bytes32 cardId, uint96 dailyLimit) external {
154       Card storage card = _checkCardExistAndReturnCardStorage(cardId, true);
155       StorageLib.CardSpendLimitStorage storage ss = StorageLib.getCardSpendLimitStorage();
156       require(uint256(dailyLimit) >= card.minDailyLimit, InvalidMinDailyLimit());
157       ss.settings[msg.sender][cardId].dailyLimit = dailyLimit;
158       emit CardAccountDailyLimitSet(cardId, msg.sender, dailyLimit);
159   }
```

Listing 2.20: contracts/DeFiCardRouter.sol

Suggestion Revise the code logic accordingly.

2.2.8 Add validations on action execution results

Status Confirmed

Introduced by `Version 1`

Description In the contract `DeFiCardCoreV1`, the function `_executeActions()` invokes the handler and gets the collected payment token amounts from its return value. For `SUPPLY` and `REPAY` actions, the returned `actionAmountOut` should remain zero, since these actions do not transfer the payment token to the payer. However, the function does not enforce this.

```
230   function _executeActions(
231       Action[] calldata actions,
232       address payer,
233       address paymentToken,
234       uint256 borrowActionLimit,
235       bool borrowEnabled
236   ) internal returns (uint256 actionsTotalAmountOut) {
237       uint256 borrowCount;
238       for (uint256 i = 0; i < actions.length;) {
239           Action calldata action = actions[i];
240
241           require(_isSystemAccess(action.handler, SystemAccessType.HANDLER), InvalidHandler(
                  action.handler));
242
243           if (action.actionType == ActionType.TRANSFER) {
244               require(action.amountIn > 0, InvalidActionAmountIn());
245               require(action.tokenIn != paymentToken && action.tokenIn != address(0),
                      InvalidActionTokenIn());
246               require(action.tokenOut == paymentToken, InvalidActionTokenOut());
247           } else if (action.actionType == ActionType.SUPPLY) {
248               require(action.amountIn > 0, InvalidActionAmountIn());
249               require(action.tokenIn != paymentToken && action.tokenIn != address(0),
                      InvalidActionTokenIn());
```

```
250             require(action.tokenOut != paymentToken, InvalidActionTokenOut());
251         } else if (action.actionType == ActionType.BORROW) {
252             require(action.amountIn == 0, InvalidActionAmountIn());
253             require(borrowEnabled, BorrowNotEnable());
254             borrowCount++;
255             require(borrowActionLimit == 0 || borrowCount <= borrowActionLimit,
                    BorrowLimitExceeded());
256             require(action.tokenIn == address(0), InvalidActionTokenIn());
257             require(action.tokenOut == paymentToken, InvalidActionTokenOut());
258         } else if (action.actionType == ActionType.WITHDRAW) {
259             require(action.tokenIn != paymentToken, InvalidActionTokenIn());
260             require(action.tokenOut == paymentToken, InvalidActionTokenOut());
261             if (action.tokenIn == address(0)) {
262                 require(action.amountIn == 0, InvalidActionAmountIn());
263             } else {
264                 require(action.amountIn > 0, InvalidActionAmountIn());
265             }
266         } else if (action.actionType == ActionType.REPAY) {
267             require(action.amountIn > 0, InvalidActionAmountIn());
268             require(action.tokenIn != paymentToken && action.tokenIn != address(0),
                    InvalidActionTokenIn());
269             require(action.tokenOut == address(0), InvalidActionTokenOut());
270         }
271
272         if (action.amountIn > 0 && action.tokenIn != address(0) && action.tokenIn !=
                paymentToken) {
273             TokenHelper.safeTransferFrom(action.tokenIn, payer, action.handler, action.amountIn
                    );
274         }
275         uint256 actionAmountOut = IDeFiCardBaseHandler(action.handler).execute(action, payer);
276         actionsTotalAmountOut += actionAmountOut;
```

Listing 2.21: contracts/core/DeFiCardCoreV1.sol

Suggestion Revise the code logic accordingly.

Feedback from the project The project states that this behavior is by design. The `DeFiCardCoreV1` contract does not perform additional validation on action execution results.

2.3 Note

2.3.1 Potential centralization risks

Introduced by `Version 1`

Description In this project, several privileged roles (e.g., `owner`, `operator`, `authCaller`, `settleCaller`, `refundCaller`, `offchainSigner`) can conduct sensitive operations, which introduces potential centralization risks. For example, the router contract's `owner` can schedule contract upgrades, modify card configurations, including `settlement`, `settleReceiver`, and all feature toggles via `setCardConfig()`. The `operator` can perform contract upgrades, execute timelocked proposals, pause or unpause the protocol, and rescue assets. The settlement contract's `owner`

can recover payment tokens via function `rescuePaymentToken()`, and the `operator` could block order settlement via pause functionality. Additionally, the `offchainSigner` is trusted to produce valid authorization signatures, and `authCaller`/`settleCaller`/`refundCaller` are trusted to invoke financial operations correctly. If the private keys of any privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

Feedback from the project The project states that the protocol is designed with a clear separation of privileges between the `owner` and `operator` roles. The `owner` role holds the highest level of authority, is controlled collectively through a multi-signature mechanism, and is responsible for business configuration, permission management, as well as timelock proposal creation and execution. In contrast, the `operator` role is controlled by an EOA address and is limited to handling emergency actions for mitigating or blocking attacks, as well as executing approved timelock proposals. In addition, `authCaller`, `settleCaller`, and `refundCaller` are restricted to initiating their respective business operations only. The corresponding fund flows are predefined within the protocol design and cannot be arbitrarily modified by these caller roles. The `offchainSigner` role is controlled by the signing machine.

2.3.2 Proxy deployment and implementation binding should be atomic

Introduced by `Version 1`

Description To prevent potential front-running attacks, it is recommended that proxy deployment and implementation binding be executed atomically within a single transaction. If these operations are performed separately, malicious actors could exploit the time window between deployment and binding to front-run the implementation binding transaction. An attacker could potentially bind their own malicious implementation to the newly deployed proxy contract, gaining unauthorized control over the protocol's upgrade mechanism and user funds. This race condition poses significant security risks including complete protocol compromise, fund theft, and unauthorized access to privileged functions.

Feedback from the project The project confirmed that proxy deployment and implementation binding are executed atomically.

2.3.3 Assumptions on external protocols

Introduced by `Version 1`

Description The correctness and availability of handler operations depend on the reliability, stability, and security assumptions of the underlying external protocols, and fall outside of the audit scope. This includes Oracle freshness and accuracy, Morpho market accounting and liquidity, interest accrual behavior, token transfer semantics, token blacklist or pause behavior, and the continued availability of external protocol contracts. If additional protocols are integrated in the future, the project should ensure their safety and integration compatibility.

For example, contract `DeFiCardMorphoBlueMarketHandler` relies on Morpho's Oracle pricing mechanism when performing health checks. Morpho protocol does not perform an explicit staleness check, instead assuming that the underlying Chainlink feed maintains its expected

update guarantees, as stated: "Staleness is not checked because it's assumed that the Chainlink feed keeps its promises on this." [2] Therefore, Oracle freshness is inherited as an external assumption rather than enforced by this protocol.

```
137         uint256 collateralPrice = IMorphoOracle(mp.oracle).price();
```

Listing 2.22: contracts/handlers/morphoblue/market/DeFiCardMorphoBlueMarketHandler.sol

Feedback from the project The project confirmed that all integrated protocols undergo audit and assessment by the internal security team. If an integrated protocol has a vulnerability, the project will pause the contract.

2.3.4 Authorize the handler before specific Morpho operations

Introduced by Version 1

Description In the contract `DeFiCardMorphoBlueMarketHandler`, the functions `_withdraw()`, `_borrow()`, and `_repay()` invoke contract `MORPHO` with the `payer` address passed as the `onBehalfOf` parameter. In Morpho, position-changing actions require `onBehalfOf` to authorize `msg.sender` (i.e., handler) via the function `_isSenderAuthorized()`. Therefore, these functions require the `payer` to authorize the handler before these operations. Integrators should account for this requirement during transaction preparation.

```
36    function execute(Action calldata action, address payer)
37        external
38        override
39        whenNotPaused
40        nonReentrant
41        onlyRouter
42        returns (uint256 returnAmount)
43    {
44        if (action.actionType == ActionType.SUPPLY) {
45            return _supply(action, payer);
46        } else if (action.actionType == ActionType.WITHDRAW) {
47            return _withdraw(action, payer);
48        } else if (action.actionType == ActionType.BORROW) {
49            return _borrow(action, payer);
50        } else if (action.actionType == ActionType.REPAY) {
51            return _repay(action, payer);
52        } else {
53            revert InvalidMorphoMarketHandlerActionType();
54        }
55    }
```

Listing 2.23: contracts/handlers/morphoblue/market/DeFiCardMorphoBlueMarketHandler.sol

Feedback from the project The project confirmed that this is an intended design, and the authorization process is ensured off-chain.

[2] https://github.com/morpho-org/morpho-blue-oracles/blob/e32d8902f9518365caa53e9eaed3cbd6cb017a63/src/morpho-chainlink/libraries/ChainlinkDataFeedLib.sol#L17

2.3.5 Trusted off-chain logic

Introduced by Version 1

Description The project depends on the correctness and consistency of the inputs constructed off-chain. In particular, business choices embedded in signed requests, such as the selected `paymentToken`, `settlement`, and Morpho market in actions, must accurately reflect the intended execution. Any inconsistency in these parameters may lead to execution results that deviate from the intended business logic.

Feedback from the project The project confirmed that this is an intended design.

2.3.6 Assumptions on whitelisted handlers

Introduced by Version 1

Description In the contract `DeFiCardCoreV1`, the function `_executeActions()` dispatches each action to a whitelisted handler and accumulates the returned `actionAmountOut` into the total payout. The correctness of this process relies on the following assumptions about every whitelisted handler:

1. The handler enforces slippage checks when executing protocol-specific actions.
2. The value returned by the function `handler.execute()` accurately reflects the amount of `paymentToken` made available to the payer.
3. The handler correctly settles any input tokens it receives, either consuming them in the action or returning the unused portion to the payer.
4. Access controls in the function `execute()` must be properly configured.
5. Each protocol has exactly one whitelisted handler instance.

Feedback from the project The project stated that each whitelisted handler undergoes audits. Enabling a handler requires a multi-signature process, and each protocol is integrated through exactly one handler instance. If adjustments are needed (e.g., to risk parameters), the existing handler is replaced with a new one.

2.3.7 Ensure order resolution during settlement replacement

Introduced by Version 1

Description In the contract `DeFiCardRouter`, the function `setCardConfig()` allows replacing the `settlement` address. If this occurs while unsettled orders remain, escrowed funds may stay in the old settlement contract. However, subsequent `settle()` invocations pull tokens from the new settlement, which may introduce unexpected logic or lack sufficient funds. The project should ensure all outstanding orders are resolved before any settlement address update.

Feedback from the project The project states that the `settlement` address is not expected to be replaced during normal operations. It would only be updated in emergency cases, such as an exploit involving the `settlement` contract. Before replacing the `settlement` address, the backend will complete all pending settlements through the existing `settlement` contract.

2.3.8 Storage layout consistency across core versions

Introduced by Version 1

Description In the contract DeFiCardRouter, the function fallback() delegates calls to the target coreAddress specified in the calldata. The router only checks that this address belongs to the enabled core set, so different invocations on the same card can target different core versions. The project should ensure that every core version maintains a consistent storage layout. Otherwise, state mismatches or data corruption may occur when the same card interacts with multiple core versions over time.

```
63   fallback() external {
64       require(msg.data.length >= 36, InvalidFallBackCalldata());
65       address coreAddr;
66       assembly {
67           coreAddr := calldataload(4)
68       }
69       (bool isEnabled,) = _getSystemCoreVersion(coreAddr);
70       require(isEnabled, CoreVersionNotEnabled());
71       assembly {
72           calldatacopy(0, 0, calldatasize())
73           let result := delegatecall(gas(), coreAddr, 0, calldatasize(), 0, 0)
74           returndatacopy(0, 0, returndatasize())
75           if iszero(result) { revert(0, returndatasize()) }
76           return(0, returndatasize())
77       }
78   }
```

Listing 2.24: contracts/DeFiCardRouter.sol

Feedback from the project The project states that the implementation is intentional and that they will ensure storage layout consistency across core versions.

2.3.9 Refund mechanism relies on off-chain logic

Introduced by Version 1

Description In contract DeFiCardCoreV1, function refund() transfers the refund amount directly from msg.sender (i.e., refundCaller) to the payer. The function requires a non-zero orderId, and does not require the corresponding order to pre-exist. Meanwhile, the refunded amount is independent of the payer's previous payments and only determined by the off-chain signature. The project should ensure that all refund data is correctly constructed off-chain.

Feedback from the project The project states that function refund() supports orders that have not yet been recorded on-chain. The refund amount is decided off-chain due to exchange rate fluctuations.

2.3.10 Weird ERC20 tokens

Introduced by Version 1

Description In this project, it is important to note that some weird ERC20 tokens (e.g., Fee-on-Transfer tokens, rebasing tokens) are not supported and may result in unexpected behaviors. Specifically, if the `paymentToken` is a Fee-on-Transfer token or rebasing token, the actual balance change after action execution may not equal `actionsTotalAmountOut`, causing the strict equality check to revert and permanently blocking multi-asset operations for such tokens.

Feedback from the project The `paymentToken` is expected to be a stablecoin, and USDC is the provisional choice.

2.3.11 Cancel stale operations

Introduced by `Version 1`

Description In the contract `DeFiCardTimelock`, the function `_timelocked()` only checks whether `block.timestamp` has reached `execAt`. Once the delay has elapsed, a queued operation remains executable indefinitely, and removal requires explicit cancellation via the function `cancelOperation()`. The project should track queued operations and cancel any whose original context or assumptions no longer hold.

Feedback from the project The project will implement off-chain monitoring to track and clean up stale proposals.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS