BLOCKSEC

Security Audit
Report for Merchant Factory Contracts

Date: July 28, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	DCS Protocol
Target	Merchant Factory Contracts

Version History

Version	Date	Description
1.0	July 28, 2026	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-07-31 03:14:21+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 Merchant Factory Contracts of DCS Protocol.

Merchant Factory Contracts of DCS Protocol is a stablecoin acquiring infrastructure that enables recurring, operator-driven pull payments, deterministic per-merchant payout routing, and allowlisted fee distribution. The contract MerchantFactory serves as the single entrypoint for charge authorization, fee validation, and on-demand proxy deployment. The contract MerchantContract is a singleton implementation, and the factory deploys a minimal-proxy clone of it for each merchant and token, which pulls funds directly from the payer to the merchant destination and the fee receiver within the same transaction.

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

- src

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
merchant-factory	`Version 1`	`f79bd1c470a91fcb6502750299b3aac06fc9379d`
	`Version 2`	`673a8e4cfc49550fd4f8955b1db52658d8e2e16c`

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. 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.

[1] https://github.com/dcs-protocol-x/merchant-factory.git

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 have **six** recommendations and **two** notes.

- Recommendation: 6
- Note: 2

ID	Severity	Description	Category	Status
1	-	Remove redundant code	Recommendation	Fixed
2	-	Add non-zero checks for parameters of the contract `MerchantFactory`	Recommendation	Partially Fixed
3	-	Revise the visibility of the function `_getCompoundSalt()`	Recommendation	Fixed
4	-	Revise the events `MerchantCreated` and `TransferExecutedWithFee`	Recommendation	Confirmed
5	-	Revise the file `README` to align with the implementation	Recommendation	Fixed
6	-	Revise the event emission logic for the event `TransferExecutedWithFee`	Recommendation	Confirmed
7	-	Potential centralization risks	Note	-
8	-	Ensure `uuid` is unique across invocations and all supported chains	Note	-

The details are provided in the following sections.

2.1 Recommendation

2.1.1 Remove redundant code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description There are several redundant value assignments and events. It is recommended to remove them for better code readability. Specifically, the following code should be removed or revised.

1. Redundant value assignments.

```
146         permitApplied = permitApplied;
```

Listing 2.1: src/MerchantContract.sol

```
190         permitApplied = permitApplied;
```

Listing 2.2: src/MerchantContract.sol

2. Redundant events.

```
91    event TransferExecuted(
92        bytes32 indexed uuid, address from, address destination, address token, uint256 totalAmount
93    );
```

Listing 2.3: src/MerchantFactoryErrorsAndEvents.sol

Suggestion Remove the redundant code.

2.1.2 Add non-zero checks for parameters of the contract `MerchantFactory`

Status Partially Fixed in `Version 2`

Introduced by `Version 1`

Description 1. In the contract `MerchantFactory`, the function `createMerchant()` lacks performing a non-zero check for the input `token`, allowing a creation for an unusable merchant proxy. It is recommended to add non-zero checks for the input `token` in the function `createMerchant()`.

2. The four charging functions (e.g., the function `transferToDestination()`) do not perform the check `amount > 0`, which may emit zero-amount transfer events. It is recommended to add a non-zero check for the input `amount`.

```
80   function createMerchant(
81       bytes32 merchantId,
82       address token,
83       address initialManager,
84       address initialDebitor
85   ) external onlyOwner whenNotPaused returns (address merchant) {
86       require(initialManager != address(0), InvalidManager());
87       address destination = address(uint160(uint256(merchantId)));
88       require(destination != address(0), InvalidDestination());
89
90       MerchantFactoryStorage storage $ = MerchantFactoryStorageLib.getStorage();
91       // Refuse to silently overwrite an already-configured merchant. Once a manager is set,
92       // changes must go through setAuthorizedManager / updateAuthorizedDebitor. A lazily
93       // deployed merchant has no manager yet, so it can still be configured here once.
94       require($.managers[merchantId] == address(0), MerchantAlreadyConfigured());
95
96       bytes32 compoundSalt = _getCompoundSalt(merchantId, token);
97       merchant = LibClone.predictDeterministicAddress(
98           MERCHANT_CONTRACT_IMPLEMENTATION, compoundSalt, address(this)
99       );
100      // Deploy only if not already materialized (e.g. by a prior lazy-deploy charge),
101      // so explicit provisioning and lazy deployment can coexist without reverting.
102      if (merchant.code.length == 0) {
103          LibClone.cloneDeterministic({
104              implementation: MERCHANT_CONTRACT_IMPLEMENTATION,
105              salt: compoundSalt
106          });
107          emit MerchantCreated(merchantId, merchant, initialManager, destination);
108      }
109      $.managers[merchantId] = initialManager;
110      // Report the manager assignment via ManagerUpdated, mirroring setAuthorizedManager. The
111      // MerchantAlreadyConfigured guard above proves the prior manager was address(0), and the
112      // MerchantCreated NatSpec directs indexers to reconcile manager state from ManagerUpdated
         -
113      // necessary because MerchantCreated is skipped when the proxy was already lazily deployed.
114      emit ManagerUpdated(merchantId, address(0), initialManager);
115      // Skip the zero address: it can never be a msg.sender, so authorizing it is dead state.
```

```
116         // Emit DebitorUpdated so the initial debitor grant is observable off-chain like every
                other.
117         if (initialDebitor != address(0)) {
118             $.allowedDebitors[merchantId][initialDebitor] = true;
119             emit DebitorUpdated(merchantId, initialDebitor, true);
120         }
121     }
```

Listing 2.4: src/MerchantFactory.sol

3. The function `isAuthorizedManager()` returns `true` for unconfigured merchants. It is recommended to add a non-zero check for the `manager`.

```
528     function isAuthorizedManager(bytes32 merchantId, address manager) external view returns (bool)
            {
529         return MerchantFactoryStorageLib.getStorage().managers[merchantId] == manager;
530     }
```

Listing 2.5: src/MerchantFactory.sol

Suggestion Add non-zero checks for the mentioned parameters in the contract `MerchantFactory`.

Clarification from BlockSec The project decided not to fix point 2 and stated that zero-amount charges are used for subscription setup.

2.1.3 Revise the visibility of the function `_getCompoundSalt()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `MerchantFactory`, the function `_getCompoundSalt()` is declared as `public`, whereas all other helper functions are declared as `internal`. It is recommended to revise the visibility of the function `_getCompoundSalt()` for consistency.

```
577     function _getCompoundSalt(bytes32 merchantId, address token)
578         public
579         pure
580         returns (bytes32 result)
581     {
582         /// @solidity memory-safe-assembly
583         assembly {
584             mstore(0, merchantId)
585             mstore(0x20, token)
586             result := keccak256(0, 0x40)
587         }
588     }
```

Listing 2.6: src/MerchantFactory.sol

Suggestion Change the visibility of the function `_getCompoundSalt()` from public to internal.

2.1.4 Revise the events `MerchantCreated` and `TransferExecutedWithFee`

Status Confirmed

Introduced by `Version 1`

Description It is recommended to add the variable `token` in the event `MerchantCreated` and the variable `merchantId` in the event `TransferExecutedWithFee` to facilitate the off-chain tracking.

```
52   event MerchantCreated(
53       bytes32 indexed merchantId,
54       address indexed merchant,
55       address indexed manager,
56       address destination
57   );
```

Listing 2.7: src/MerchantFactoryErrorsAndEvents.sol

```
107  event TransferExecutedWithFee(
108      bytes32 indexed uuid,
109      address from,
110      address destination,
111      address token,
112      uint256 totalAmount,
113      uint256 feeAmount,
114      address feeReceiver
115  );
```

Listing 2.8: src/MerchantFactoryErrorsAndEvents.sol

Suggestion Revise the events `MerchantCreated` and `TransferExecutedWithFee`.

Feedback from the project The project stated that the backend already processes `merchantId` and `token` for each charge and tracks every `uuid`.

2.1.5 Revise the file `README` to align with the implementation

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `MerchantFactory`, the function `setAuthorizedManager()` is invocable when the contract is paused and the function `updateAuthorizedDebitor()` is invocable when the contract is paused and the variable `authorized` is `false`. However, the file `README` indicates that both operations are unavailable when the contract is paused. It is recommended to revise the file `README` to align with the implementation.

```
381  function setAuthorizedManager(bytes32 merchantId, address manager)
382      external
383      onlyOwner
384  {
385      require(manager != address(0), InvalidManager());
386      MerchantFactoryStorage storage $ = MerchantFactoryStorageLib.getStorage();
387      address oldManager = $.managers[merchantId];
388      $.managers[merchantId] = manager;
```

```
389         emit ManagerUpdated(merchantId, oldManager, manager);
390     }
```

Listing 2.9: src/MerchantFactory.sol

```
351     function updateAuthorizedDebitor(bytes32 merchantId, address debitor, bool authorized)
352         external
353         onlyAuthorizedManager(merchantId)
354     {
355         if (authorized) {
356             _requireNotPaused();
357         }
358         // Reject the zero address, mirroring addOperators: it can never be a msg.sender, so
359         // toggling its authorization is dead state and a misleading DebitorUpdated event.
360         require(debitor != address(0), InvalidDebitor());
361         MerchantFactoryStorageLib.getStorage().allowedDebitors[merchantId][debitor] = authorized;
362         emit DebitorUpdated(merchantId, debitor, authorized);
363     }
```

Listing 2.10: src/MerchantFactory.sol

```
103## Incident Response
104
105Charges can only be initiated by protocol-controlled keys (global operators or per-merchant
       debitors), so most incident containment is on-chain and targeted:
106
107| Compromise | Lever | Scope | Callable while paused |
108|------------|-------|-------|-----------------------|
109| Global operator key | `removeOperators([key])` | that key, across all merchants | yes |
110| Per-merchant debitor key | `updateAuthorizedDebitor(merchantId, debitor, false)` | one merchant
       | no (blocked by `whenNotPaused`) |
111| Per-merchant manager key | `setAuthorizedManager(merchantId, newManager)`, then revoke its
       debitors (no cascade-revoke) | one merchant | no |
112| A single merchant (any/unknown key) | `setMerchantFrozen(merchantId, true)` | one merchant, all
       tokens | yes |
113| Systemic / unknown | `pause()` | everything | - |
```

Listing 2.11: README.md

Suggestion Revise the file `README` to align with the implementation.

2.1.6 Revise the event emission logic for the event `TransferExecutedWithFee`

Status Confirmed

Introduced by `Version 1`

Description In the contract `MerchantFactory`, when the variable `feeBps` equals to zero, the function `_validateFeeParams()` skips the allowlist check for the variable `feeReceiver`. However , the unvalidated variable `feeReceiver` is then included in the event `TransferExecutedWithFee`, potentially affecting the off-chain tracking. It is recommended to revise the logic to correctly reflect the variable `feeReceiver` in the event `TransferExecutedWithFee`.

```
619   function _validateFeeParams(uint16 feeBps, address feeReceiver) internal view {
620       if (feeBps == 0) {
621           return;
622       }
623       MerchantFactoryStorage storage $ = MerchantFactoryStorageLib.getStorage();
624       require(feeBps <= $.maxFeeBps, FeeBpsExceedsMax());
625       require($.allowedFeeReceivers[feeReceiver], NotAllowedFeeReceiver());
626   }
```

Listing 2.12: src/MerchantFactory.sol

Suggestion Revise the logic to correctly reflect the variable `feeReceiver` in the event `TransferExecutedWithFee`.

Feedback from the project The project stated that `feeReceiver` has no economic meaning and should be ignored when `feeAmount` is zero.

2.2 Note

2.2.1 Potential centralization risks

Introduced by `Version 1`

Description In this project, several privileged roles (e.g., `owner`) can conduct sensitive operations, which introduce potential centralization risks. For example, the role `owner` can arbitrarily freeze or unfreeze any merchant through the function `setMerchantFrozen()`. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

2.2.2 Ensure `uuid` is unique across invocations and all supported chains

Introduced by `Version 1`

Description The contract `MerchantFactory` does not store consumption status of the variable `uuid`. The project must ensure that duplicate `uuid` cannot be consumed across different invocations and supported chains.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS