BLOCKSEC

Security Audit
Report for Commerce Payments

Date: April 2, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	DCS Protocol
Target	Commerce Payments

Version History

Version	Date	Description
1.0	April 2, 2026	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-04-02 09:22:54+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 Commerce Payments of DCS Protocol.

The Commerce Payments Protocol is a permissionless protocol for on-chain payments that mimics traditional authorize-and-capture payment flows. It facilitates secure escrow-based payments through modular token collectors and operator-driven execution, while separating authorization, settlement, capture, reclaim, and refund flows. The core payment state machine is implemented in `AuthCaptureEscrow`, while custody and settlement are handled through a dedicated `TokenStore` for each operator to avoid commingling funds across operators. The protocol supports multiple collection mechanisms, including direct transfers, signature-based transfers, and swap-assisted settlement, while providing relatively clear fund and state boundaries among the payer, receiver, and fee recipient.

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

- src/collectors
- src/AuthCaptureEscrow.sol
- src/TokenStore.sol

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
commerce-payments	`Version 1`	`7608eb092f09eb1fba59803a3e420e72102f48b3`
	`Version 2`	`9343625fdc6eb0d4a722540cdfd36111cf9fd8f0`

[1] https://github.com/dcs-protocol-x/commerce-payments

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.

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.

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

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

- **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.

Chapter 2 Findings

In total, we found **two** potential security issues. Besides, we have **six** recommendations and **nine** notes.

- Low Risk: 2
- Recommendation: 6
- Note: 9

ID	Severity	Description	Category	Status
1	Low	Incorrect refund amount in contract `BitgetSwapPaymentCollector`	Security Issue	Fixed
2	Low	Lack of binding between `paymentInfo` and Permit2 authorization	Security Issue	Fixed
3	-	Add checks in function `_collectTokens()`	Recommendation	Fixed
4	-	Atomize proxy deployment and initialization	Recommendation	Confirmed
5	-	Fix fee rounding direction	Recommendation	Confirmed
6	-	Align comments with implementation	Recommendation	Fixed
7	-	Add a token rescue mechanism	Recommendation	Confirmed
8	-	Add `swapRouter` whitelist restriction	Recommendation	Fixed
9	-	Weird ERC20 tokens	Note	-
10	-	Integration assumptions	Note	-
11	-	Operator fee estimation assumption	Note	-
12	-	Consistency of `tokenStoreImplementation` across upgrades	Note	-
13	-	Security assumption on the variable `PaymentInfo.salt`	Note	-
14	-	Potential centralization risks	Note	-
15	-	EIP-1153 compatibility on supported chains	Note	-
16	-	The fee and refund designIntroduced by Version 1	Note	-
17	-	The operator trust assumption	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Incorrect refund amount in contract `BitgetSwapPaymentCollector`

Severity Low

Status Fixed in `Version 2`

Introduced by Version 1

Description In the contract BitgetSwapPaymentCollector, the function _collectTokens() swaps sourceToken for paymentToken and then refunds any remaining sourceToken to the payer. However, the refund amount is derived from IERC20(params.sourceToken).balanceOf(address(this)). This design uses the collector's full token balance rather than the residual amount produced by the current swap. As a result, this design may refund previously stranded tokens to a later payer with the same sourceToken, leading to unintended transfer of assets held by the collector.

```
102       // Step 2: Approve swap router and execute the swap
103       IERC20(params.sourceToken).forceApprove(params.swapRouter, params.sourceAmount);
104
105       address paymentToken = paymentInfo.token;
106
107       uint256 balanceBefore = IERC20(paymentToken).balanceOf(address(this));
108
109       (bool success,) = params.swapRouter.call(params.swapCalldata);
110       if (!success) revert SwapFailed();
111
112       // Step 3: Revoke router allowance for safety
113       IERC20(params.sourceToken).forceApprove(params.swapRouter, 0);
114
115       // Step 3.5: Refund any remaining source tokens to the payer
116       uint256 remainingSource = IERC20(params.sourceToken).balanceOf(address(this));
117       if (remainingSource > 0) {
118           IERC20(params.sourceToken).safeTransfer(paymentInfo.payer, remainingSource);
119       }
```

Listing 2.1: src/collectors/BitgetSwapPaymentCollector.sol

Impact Unrelated residual funds held by the collector may be transferred to a later payer.

Suggestion Track the source token balance difference for the current swap, and refund only the residual amount attributable to the current execution rather than the collector's full token balance.

2.1.2 Lack of binding between paymentInfo and Permit2 authorization

Severity Low

Status Fixed in Version 2

Introduced by Version 1

Description In contract BitgetSwapPaymentCollector, function _collectTokens() pulls payer funds via Permit2 using a signature supplied in the collectorData, then settles them according to the paymentInfo provided by the operator. However, the Permit2 authorization is not sufficiently bound to paymentInfo. Specifically, the Permit2 nonce is taken solely from data.permit2Nonce, and the Permit2 deadline is taken from data.permit2Deadline. This differs from contract Permit2PaymentCollector, which derives the Permit2 nonce using the function _getHashPayerAgnostic(paymentInfo) and uses paymentInfo.preApprovalExpiry as the authorization deadline. Function _getHashPayerAgnostic() commits critical fields in paymentInfo,

e.g., `receiver`, `token`, and `salt`, to the signed intent. As a result, a valid Permit2 signature can be used with arbitrary `paymentInfo` in contract `BitgetSwapPaymentCollector`, leading to unauthorized payment parameters and unexpected settlement behavior.

```
73  function _collectTokens(
74      AuthCaptureEscrow.PaymentInfo calldata paymentInfo,
75      address tokenStore,
76      uint256 amount,
77      bytes calldata collectorData
78  ) internal override {
79      // Decode collector data
80      CollectorData memory data = abi.decode(collectorData, (CollectorData));
81
82      SwapParams memory params = data.swapParams;
83
84      if (params.swapRouter == address(0)) revert InvalidSwapRouter();
85
86      // Step 1: Pull source tokens from the payer into this contract via Permit2
87      permit2.permitTransferFrom({
88          permit: ISignatureTransfer.PermitTransferFrom({
89              permitted: ISignatureTransfer.TokenPermissions({
90                  token: params.sourceToken, amount: params.sourceAmount
91              }),
92              nonce: data.permit2Nonce,
93              deadline: data.permit2Deadline
94          }),
95          transferDetails: ISignatureTransfer.SignatureTransferDetails({
96              to: address(this), requestedAmount: params.sourceAmount
97          }),
98          owner: paymentInfo.payer,
99          signature: data.permit2Signature
100     });
```

Listing 2.2: src/collectors/BitgetSwapPaymentCollector.sol

```
32  function _collectTokens(
33      AuthCaptureEscrow.PaymentInfo calldata paymentInfo,
34      address tokenStore,
35      uint256 amount,
36      bytes calldata collectorData
37  ) internal override {
38      permit2.permitTransferFrom({
39          permit: ISignatureTransfer.PermitTransferFrom({
40              permitted: ISignatureTransfer.TokenPermissions({
41                  token: paymentInfo.token, amount: paymentInfo.maxAmount
42              }),
43              nonce: uint256(_getHashPayerAgnostic(paymentInfo)),
44              deadline: paymentInfo.preApprovalExpiry
45          }),
46          transferDetails: ISignatureTransfer.SignatureTransferDetails({to: tokenStore,
                requestedAmount: amount}),
47          owner: paymentInfo.payer,
48          signature: collectorData
```

```
49        });
```

Listing 2.3: src/collectors/Permit2PaymentCollector.sol

Impact A valid Permit2 signature can be used with an arbitrary `paymentInfo`, resulting in an unintended settlement.

Suggestion Bind the Permit2 nonce to `paymentInfo` using function `_getHashPayerAgnostic(paymentInfo)` and use `paymentInfo.preApprovalExpiry` as the Permit2 authorization deadline.

2.2 Recommendation

2.2.1 Add checks in function `_collectTokens()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `BitgetSwapPaymentCollector`, the function `_collectTokens()` swaps `sourceToken` into `paymentToken` and then forwards the required payment amount. However, the current implementation lacks two explicit validations to improve both execution safety and failure clarity.

1. The function gets excess swap output as `received - amount` (line 129). However, the function does not check that the `received >= amount` beforehand. Consequently, the function may revert without an explicit reason if `received` is less than `amount`. Meanwhile, the error `InsufficientSwapOutput` (line 61) is defined but unused. It is recommended to either add an explicit check that reverts with this error or remove the unused error definition.

2. The function refunds excess funds to the payer after swapping based on the formula `received = balanceAfter - balanceBefore`. When `sourceToken` equals `paymentToken`, the intermediate refund of the remaining `sourceToken` balance transfers out the same token whose post-swap balance is later used for output accounting. Consequently, `balanceAfter` may be lower than `balanceBefore`, resulting in a revert without an explicit reason.

```
128       // Step 5: Refund any excess swap output to the payer
129       uint256 excess = received - amount;
130       if (excess > 0) {
131           IERC20(paymentToken).safeTransfer(paymentInfo.payer, excess);
132       }
```

Listing 2.4: src/collectors/BitgetSwapPaymentCollector.sol

```
73    function _collectTokens(
74        AuthCaptureEscrow.PaymentInfo calldata paymentInfo,
75        address tokenStore,
76        uint256 amount,
77        bytes calldata collectorData
78    ) internal override {
```

Listing 2.5: src/collectors/BitgetSwapPaymentCollector.sol

```
58  /// @notice Thrown when the swap output is less than the required payment amount.
59  /// @param received The amount of payment tokens received from the swap.
60  /// @param required The amount of payment tokens needed.
61  error InsufficientSwapOutput(uint256 received, uint256 required);
```

Listing 2.6: src/collectors/BitgetSwapPaymentCollector.sol

Suggestion Add a check that rejects `sourceToken == paymentToken` and a check that reverts with `InsufficientSwapOutput()` when `received < amount`.

2.2.2 Atomize proxy deployment and initialization

Status Confirmed

Introduced by `Version 1`

Description The script `DeployAll.run()` deploys the proxy and then invokes the function `initialize()` in a later step (lines 43-44). This leaves a bootstrap window before ownership is set on the proxy. Add atomic initialization during deployment or otherwise constrain the first successful initialization.

```
209  function initialize(address owner_) external initializer {
210      _initializeOwner(owner_);
211  }
```

Listing 2.7: src/AuthCaptureEscrow.sol

Suggestion Pass initialization calldata during proxy deployment or use a deployment flow that removes the public initialization window.

Feedback from the project The front-running window between proxy deployment and initialization is acknowledged but accepted. Since function `LibClone.deployDeterministicERC1967()` does not support initialization calldata, switching proxy mechanisms would break deterministic cross-chain addresses. To minimize the exposure window, both deployment and initialization are executed within the same `vm.startBroadcast()` batch.

2.2.3 Fix fee rounding direction

Status Confirmed

Introduced by `Version 1`

Description In contract `AuthCaptureEscrow`, the function `_distributeTokens()` computes the fee as `amount * feeBps / 10000`, which rounds down due to integer division. This can cause the amount transferred to `feeReceiver` to be lower than the nominal fee implied by `feeBps`. It is recommended to revise the fee calculation so that fees are rounded up.

```
424  /// @notice Sends tokens to receiver and/or feeReceiver
425  ///
426  /// @param token Token to transfer
427  /// @param receiver Address to receive payment
428  /// @param amount Total amount to split between payment and fees
429  /// @param feeBps Fee percentage in basis points
```

```
430   /// @param feeReceiver Address to receive fees
431   function _distributeTokens(address token, address receiver, uint256 amount, uint16 feeBps,
          address feeReceiver) internal {
432       uint256 feeAmount = amount * feeBps / _MAX_FEE_BPS;
433       // send fee to feeReceiver
434       if (feeAmount > 0) _sendTokens(msg.sender, token, feeReceiver, feeAmount);
435       // send payment to receiver
436       if (amount > feeAmount) _sendTokens(msg.sender, token, receiver, amount - feeAmount);
437   }
```

Listing 2.8: src/AuthCaptureEscrow.sol

Suggestion Revise the fee calculation to round fees up.

Feedback from the project The rounding down favors the `receiver` (merchant), which the project considers the preferred default.

2.2.4 Align comments with implementation

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Some comments in the current implementation do not accurately reflect the implemented behavior. It is recommended to revise them for better code accuracy and readability. Specifically, the following comments should be aligned with the implementation.

1. In the function `_validatePayment()`, the inline comment states that the logic checks whether authorization expiry is after pre-approval (line 561). However, the adjacent check actually compares the current time against `preApprovalExpiry`.

```
561       // check authorization expiry is after pre-approval
562       if (currentTime >= preApprovalExp) revert AfterPreApprovalExpiry(currentTime,
              preApprovalExp);
563
564       // check expiry timestamps properly ordered
565       if (preApprovalExp > authorizationExp || authorizationExp > refundExp) {
566           revert InvalidExpiries(preApprovalExp, authorizationExp, refundExp);
567       }
```

Listing 2.9: src/AuthCaptureEscrow.sol

2. In the function `authorize()`, the comment states that the function transfers funds from `payer` to `escrow` (line 278). However, collected tokens are sent to the operator token store rather than being held by the `AuthCaptureEscrow` contract.

```
278   /// @notice Transfers funds from payer to escrow
279   ///
280   /// @param paymentInfo PaymentInfo struct
281   /// @param amount Amount to authorize
282   /// @param tokenCollector Address of the token collector
283   /// @param collectorData Data to pass to the token collector
```

Listing 2.10: src/AuthCaptureEscrow.sol

3. In the function `_collectTokens()`, the comment states that the function transfers tokens into this contract (line 468). However, the transfer is expected to increase the token store's balance rather than the balance of the current contract.

```
468   /// @notice Transfer tokens into this contract
469   ///
470   /// @param paymentInfo PaymentInfo struct
471   /// @param amount Amount of tokens to collect
472   /// @param tokenCollector Address of the token collector
473   /// @param collectorData Data to pass to the token collector
474   /// @param collectorType Type of collector to enforce (payment or refund)
```

Listing 2.11: src/AuthCaptureEscrow.sol

Suggestion Align the comments with the implementation.

2.2.5 Add a token rescue mechanism

Status Confirmed

Introduced by `Version 1`

Description Contracts `AuthCaptureEscrow` and `TokenStore` lack a recovery function for tokens that are transferred in by mistake or left outside the intended settlement flow. This weakens operational recovery for accidental transfers and unsupported assets. It is advised to add a narrowly scoped rescue mechanism and document when it may be used.

```
22   /// @notice Send tokens to a recipient, called by escrow during capture/refund
23   ///
24   /// @param token The token being received
25   /// @param recipient Address to receive the tokens
26   /// @param amount Amount of tokens to receive
27   ///
28   /// @return success True if the transfer was successful
29   function sendTokens(address token, address recipient,uint256 amount) external returns (bool){
30       if(msg.sender != authCaptureEscrow) revert OnlyAuthCaptureEscrow();
31       SafeERC20.safeTransfer(IERC20(token), recipient, amount);
32       return true;
33   }
```

Listing 2.12: src/TokenStore.sol

Suggestion Add an owner-controlled rescue function with clear scope, events, and documentation.

Feedback from the project A `rescueTokens()` function will be added to the contract `TokenStore`, allowing operators to recover stranded tokens from their own token store. While accidental transfers are unlikely given that token store addresses are deterministically derived and publicly queryable via the function `getTokenStore()`, a recovery mechanism is prudent. No equivalent is planned for contract `BitgetSwapPaymentCollector`, given its transient, non-custodial design.

2.2.6 Add `swapRouter` whitelist restriction

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `BitgetSwapPaymentCollector`, the function `_collectTokens()` accepts a parameter `swapRouter` as the call target for swap execution. However, the function only validates `swapRouter` against the zero address. While the operator is intended to be a trusted centralized entity, the absence of an on-chain whitelist leaves the swap execution unconstrained at the contract level. As a result, a compromised or misconfigured operator could invoke an unauthorized router to execute unintended swap logic.

```
102       // Step 2: Approve swap router and execute the swap
103       IERC20(params.sourceToken).forceApprove(params.swapRouter, params.sourceAmount);
104
105       address paymentToken = paymentInfo.token;
106
107       uint256 balanceBefore = IERC20(paymentToken).balanceOf(address(this));
108
109       (bool success,) = params.swapRouter.call(params.swapCalldata);
110       if (!success) revert SwapFailed();
```

Listing 2.13: src/collectors/BitgetSwapPaymentCollector.sol

Suggestion Add an on-chain router whitelist to reduce the impact of a single point of failure.

2.3 Note

2.3.1 Weird ERC20 tokens

Introduced by `Version 1`

Description In this protocol, there are no whitelisting restrictions on ERC20 tokens. It is important to note that some weird ERC20 tokens, such as fee-on-transfer tokens, rebasing tokens, blocklisted tokens, and paused tokens, are not supported and may result in unexpected behaviors. In particular, the token collection logic used by the contract `AuthCaptureEscrow` and the token transfer logic used by the contract `TokenStore` are expected to follow standard ERC20 semantics with exact nominal accounting. If an integrated token does not preserve the expected transfer or balance semantics, the related collection, capture, refund, reclaim, and settlement flows may deviate from intended results.

```
475   function _collectTokens(
476       PaymentInfo calldata paymentInfo,
477       uint256 amount,
478       address tokenCollector,
479       bytes calldata collectorData,
480       TokenCollector.CollectorType collectorType
481   ) internal {
482       // check collector type
483       if (TokenCollector(tokenCollector).collectorType() != collectorType) revert
              InvalidCollectorForOperation();
```

```
484
485         // Measure token store balance before and after collection
486         address token = paymentInfo.token;
487         address tokenStore = getTokenStore(paymentInfo.operator);
488         uint256 tokenStoreBalanceBefore = IERC20(token).balanceOf(tokenStore);
489         TokenCollector(tokenCollector).collectTokens(paymentInfo, tokenStore, amount, collectorData
                );
490         uint256 tokenStoreBalanceAfter = IERC20(token).balanceOf(tokenStore);
491         if (tokenStoreBalanceAfter != tokenStoreBalanceBefore + amount) revert
                TokenCollectionFailed();
492     }
```

Listing 2.14: src/AuthCaptureEscrow.sol

Feedback from the project The protocol targets standard ERC20 tokens (e.g., USDT, USDC). Fee-on-transfer, rebasing, blocklisted, and paused tokens are explicitly not supported and will be documented as such.

2.3.2 Integration assumptions

Introduced by Version 1

Description This protocol relies on standard-compliant implementations of the external token authorization systems that it integrates. The integration model should be a strict token-compatibility requirement.

For ERC-3009, the underlying token is expected to implement the standard semantics of the function `receiveWithAuthorization()`, including the additional check that the payee address matches `msg.sender`. Meanwhile, the ERC-3009 integration also assumes that nonces remain independent between different authorizers. In addition, EIP-3009 is generally designed for EOAs (externally owned accounts) rather than smart contract accounts. ERC-3009 relies on ECDSA signature verification, which smart contract accounts cannot natively produce.

For Permit2-based flows, the current design also assumes that nonces remain independent between different authorizers.

Feedback from the project We target standard ERC-3009 implementations (primarily USDC) and the canonical Permit2 deployment. Nonce independence across different authorizers is guaranteed. Smart contract wallet support is not a current requirement.

2.3.3 Operator fee estimation assumption

Introduced by Version 1

Description In the protocol, the `operator` bears the on-chain transaction cost of driving the payment flow. As a result, the operator is expected to estimate execution cost before submitting a transaction, so that the expected fee revenue can cover the actual cost. Meanwhile, transaction fee models may differ across supported chains. For example, on Morph, the total transaction cost includes both an L2 execution fee and an L1 data fee rather than a single gas component.

Feedback from the project The operator backend estimates gas costs, including L1 data fees on L2s (e.g., Morph), prior to transaction submission. This is handled at the application layer.

2.3.4 Consistency of `tokenStoreImplementation` across upgrades

Introduced by `Version 1`

Description The variable `tokenStoreImplementation` is immutable in each escrow implementation and is used by function `getTokenStore()` to derive the per-operator `TokenStore` address. A new escrow implementation deployed with a different constructor value would derive different `TokenStore` addresses for the same operators. Upgrade procedures should preserve this constructor value consistently.

```
78   /// @notice Implementation contract for operator token stores
79   address public immutable tokenStoreImplementation;
```

Listing 2.15: src/AuthCaptureEscrow.sol

```
499  function getTokenStore(address operator) public view returns (address) {
500      return LibClone.predictDeterministicAddress({
501          implementation: tokenStoreImplementation,
502          salt: bytes32(bytes20(operator)),
503          deployer: address(this)
504      });
505  }
```

Listing 2.16: src/AuthCaptureEscrow.sol

Feedback from the project The upgrade procedure enforces that the new implementation must reference the same `tokenStoreImplementation` address, verified in both the upgrade script and CI (continuous integration) checks.

2.3.5 Security assumption on the variable `PaymentInfo.salt`

Introduced by `Version 1`

Description In contract `AuthCaptureEscrow`, the function `getHash()` incorporates the variable `paymentInfo.salt` as entropy to derive a unique `paymentInfoHash` for each instance of the struct `PaymentInfo`. This design assumes that `paymentInfo.salt` is both unique and sufficiently unpredictable, because the payer-agnostic hash produced from `PaymentInfo` is used directly as the on-chain nonce across EIP-3009 and Permit2 authorisation flows. However, the contract does not impose constraints or validation on how `paymentInfo.salt` is generated, which places operational responsibility on the entity populating `PaymentInfo` to prevent accidental reuse or predictable values. Furthermore, the project should ensure that the operator consistently sources `paymentInfo.salt` from a high-quality randomness mechanism (or an equivalently collision-resistant construction) and maintains process-level guarantees that each payment is assigned a distinct value.

Feedback from the project The operator backend generates salts using a cryptographically secure RNG. Uniqueness is additionally enforced at the application layer via database constraints.

2.3.6 Potential centralization risks

Introduced by Version 1

Description In this project, several privileged roles (e.g., `owner` and `operator`) can conduct sensitive operations, which introduces potential centralization risks. For example, the `owner` can unilaterally add or remove operators via function `setOperatorWhitelist()` or permanently revoke the ownership via function `renounceOwnership()`. A whitelisted `operator` has extensive unilateral control over the full payment lifecycle. Specifically, it selects the `tokenCollector` and constructs arbitrary `collectorData`, including swap router addresses and calldata in contract `BitgetSwapPaymentCollector`, determines whether to charge, authorize, capture, void, or refund a payment, and controls fee parameters within the bounds set by `PaymentInfo`. In the contract `BitgetSwapPaymentCollector`, slippage protection is entirely delegated to the operator through the `swapCalldata` it constructs, with no on-chain enforcement. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

Feedback from the project The owner is a Safe multisig with multiple signers, and operator transaction signing is handled through Fireblocks MPC-TSS with policy-based approval controls. The centralization trade-off is accepted for operational efficiency within this permissioned payment protocol.

2.3.7 EIP-1153 compatibility on supported chains

Introduced by Version 1

Description The protocol uses `ReentrancyGuardTransient`, which depends on EIP-1153 transient storage. All currently supported chains (Ethereum, BSC, Base, and Morph) are compatible. If additional chains are added in the future, EIP-1153 support should be verified before deployment.

```
13contract AuthCaptureEscrow is ReentrancyGuardTransient, UUPSUpgradeable, Ownable, Initializable {
```

Listing 2.17: src/AuthCaptureEscrow.sol

Feedback from the project The project confirmed that the current target chains support EIP-1153.

2.3.8 The fee and refund designIntroduced by Version 1

Introduced by

Description In this project, protocol fees apply only on successful settlement via function `charge()` or `capture()`. The operator must account for this when initiating operations.

Refunds are funded by the operator rather than clawed back from the payment receiver on-chain. Therefore, the operator must maintain sufficient funds for refunds and is responsible for coordinating the corresponding recovery from the receiver off-chain.

```
264        // set payment state with refundable amount
265        _getEscrowStorage().paymentState[paymentInfoHash] =
```

```
266                     PaymentState({hasCollectedPayment: true, capturableAmount: 0,
                            refundableAmount: uint120(amount)});
267         // emit event
268         emit PaymentCharged(paymentInfoHash, paymentInfo, amount, tokenCollector, feeBps,
                feeReceiver);
269
270         // transfer tokens into tokenStore
271         _collectTokens(paymentInfo, amount, tokenCollector, collectorData, TokenCollector.
                CollectorType.Payment);
272
273         // transfer tokens to receiver and fee receiver
274         _distributeTokens(paymentInfo.token, paymentInfo.receiver, amount, feeBps, feeReceiver);
```

Listing 2.18: src/AuthCaptureEscrow.sol

```
415         // Update refundable amount
416         _getEscrowStorage().paymentState[paymentInfoHash].refundableAmount = captured - uint120(
                amount);
417         emit PaymentRefunded(paymentInfoHash, amount, tokenCollector);
418
419         // Transfer tokens into escrow and forward to payer
420         _collectTokens(paymentInfo, amount, tokenCollector, collectorData, TokenCollector.
                CollectorType.Refund);
421         _sendTokens(paymentInfo.operator, paymentInfo.token, paymentInfo.payer, amount);
```

Listing 2.19: src/AuthCaptureEscrow.sol

Feedback from the project 1. Fee mechanism: Protocol fees apply only on successful settlement (functions `charge()` and `capture()`), where value is delivered to the receiver. Operations that return funds to the payer (`authorize()`, `void()`, `reclaim()`, `refund()`) are fee-free, as no settlement occurs. 2. Refund funding: Refunds are operator-funded by design. The operator maintains sufficient liquidity to cover refunds. When a refund is triggered, contract `OperatorRefundCollector` pulls tokens from the operator's address to the token store, which forwards them to the payer. The operator then settles the refunded amount with the receiver off-chain. This avoids the complexity of on-chain clawbacks from the receiver and ensures prompt refunds without requiring receiver cooperation.

2.3.9 The operator trust assumption

Introduced by `Version 1`

Description In the project, whitelisted operators drive the payment lifecycle and control crucial arguments (e.g, collector selection, settlement timing, and the amount that the `receiver` ultimately receives). Therefore, the project is safe only if operators are a trusted party and the whitelist is managed under strict criteria. If this trust assumption does not hold, the operator-controlled actions may lead to unfavorable outcomes for the payer.

For example, in contract `BitgetSwapPaymentCollector`, the function `_collectTokens()` uses the `amount` as the minimum output threshold, a value specified by the operator. As a result, a malicious operator can set the `amount` artificially low, allowing the swap to return the minimum

output needed for settlement, with little or no excess refunded to the payer. This weakens the payer's practical protection against unfavorable swaps.

Feedback from the project The project states that the operator is their backend payment service operated by the DCS Protocol team. Only DCS-controlled addresses can be whitelisted via function `setOperatorWhitelist()`, which is managed by the owner, i.e., a Safe multisig. In addition, operator transactions are signed through Fireblocks MPC-TSS and governed by policy-based approval controls.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS