BLOCKSEC

Security Audit
Report for bgw7702

Date: October 30, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Chapter 1 Introduction **1**
1.1 About the Audit Target . . . 1
1.2 Disclaimer . . . 2
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 **5**
2.1 Security Issue . . . 5
2.1.1 Lack of proof of possession in function `setAccountKey()` . . . 5
2.1.2 Lack of verification on the renouncement status . . . 6
2.1.3 Stale validator configuration after emergency deactivation . . . 8
2.2 Recommendation . . . 9
2.2.1 Enforce hybrid signature scheme . . . 9
2.3 Note . . . 11
2.3.1 Potential centralization risks . . . 11
2.3.2 SLH-DSA deployment requirement . . . 11
2.3.3 EntryPoint trust assumption and EIP-4337 compatibility . . . 11
2.3.4 Integration assumption . . . 11

Report Manifest

Item	Description
Client	bitgetwallet
Target	bgw7702

Version History

Version	Date	Description
1.0	October 30, 2026	First release

Signature



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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-11-02 08:43:39+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 bgw7702 of bitgetwallet.

BGW7702 is a smart account framework built for EIP-7702 delegation with optional ERC-4337 compatibility. This upgrade introduces support for EIP-1271 and EIP-7851. Specifically, contract `BGW7702Logic` adds function `isValidSignature()` and EIP-7851 delegation renunciation checks, along with new entry functions `aggregateWithValidator()` and `verifySignaturesWithValidator()` (the latter in the contract `BGW7702Admin`). The upgrade also introduces a verification router, `BGWVerifyRouterWithTimeLock`, that maintains a mapping from `verifyId` to validator contracts, including Passkey/WebAuthn, SLH-DSA post-quantum signatures, and Guard-EOA. These external validators enable smart accounts to support additional signature schemes beyond traditional ECDSA.

Note this audit only focuses on the codes located in the following directories/files.

- src/BGW7702Admin.sol
- src/BGW7702Logic.sol
- src/BGWVerifyRouterWithTimeLock.sol
- src/validators/SlhDsaValidator.sol
- src/validators/PasskeyValidator.sol
- src/validators/GuardValidator.sol
- src/utils/BGWEcdsaLib.sol
- src/utils/BGWSigLib.sol
- src/interfaces/IValidator.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.

[1] https://github.com/bitgetwallet/bgw7702

Project	Version	Commit Hash
bgw7702	Version 0	49e729fa20137a37bfa88b0c263c6713d393df15
	Version 1	a9ee9aa9dc7832412360c71ca277f85b416f01e2
	Version 2	7b179912697b0beed9163878087da53d1c057296

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**,

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

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

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.

Chapter 2 Findings

In total, we found **three** potential security issues. Besides, we have **one** recommendation and **four** notes.

- Low Risk: 3
- Recommendation: 1
- Note: 4

ID	Severity	Description	Category	Status
1	Low	Lack of proof of possession in function `setAccountKey()`	Security Issue	Confirmed
2	Low	Lack of verification on the renouncement status	Security Issue	Fixed
3	Low	Stale validator configuration after emergency deactivation	Security Issue	Confirmed
4	-	Enforce hybrid signature scheme	Recommendation	Confirmed
5	-	Potential centralization risks	Note	-
6	-	SLH-DSA deployment requirement	Note	-
7	-	EntryPoint trust assumption and EIP-4337 compatibility	Note	-
8	-	Integration assumption	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Lack of proof of possession in function `setAccountKey()`

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the contract `BGWVerifyRouterWithTimeLock`, the function `setAccountKey()` binds a public key to an account under a `verifyId`. It then tracks the registration status in the mapping `outsidePubkeyIsExist`, preventing multiple accounts from registering the same public key under a `verifyId`. However, the function does not verify that the caller controls the corresponding private key. As a result, an attacker who observes a victim's public key in a pending transaction can front-run it by registering the same key from the attacker's own account, causing the victim's transaction to revert.

```
154     if (verifyId == 1) revert InvalidVerifyId(verifyId);
155     emit CallerVerifyIdSet(caller, verifyId);
156 }
157
158 function setAccountKey(uint16 verifyId, bytes calldata pubkey) external {
159     if (verifyId < 2) revert ReservedVerifyId(verifyId);
160     VerifyInfo storage info = _verifyInfos[verifyId];
161     if (info.validator == address(0) || !info.isActive) revert InactiveValidator();
```

```
162
163     // A 20-byte "Guard" key equal to the account itself is just the account's
164     // own ECDSA key wearing a validator costume: it would survive EIP-7851
165     // renouncement and defeat BGW7702Logic._ecdsaRenounced().
166     // length == 20: first 20 bytes are the address, no truncation.
167     // forge-lint: disable-next-line(unsafe-typecast)
168     if (pubkey.length == 20 && address(bytes20(pubkey)) == msg.sender) {
169         revert InvalidGuardPubkey();
170     }
171
172     IValidator v = IValidator(info.validator);
173     if (pubkey.length != v.pubkeyLen() || !v.isValidPubkey(pubkey)) revert InvalidPubkey();
174
175     // Occupied check first: same-key re-set and another account's key both revert.
176     // Then release the caller's previous key (if any) and claim.
177     bytes32 pubkeyHash = keccak256(pubkey);
```

Listing 2.1: src/BGWVerifyRouterWithTimeLock.sol

Impact An attacker can prevent a victim from registering a public key under a selected `verifyId`.

Suggestion Require a proof of possession that binds the public key to the registering account and `verifyId`.

Feedback from the project The project will submit private transactions to help users register public keys if binding transactions are frequently front-run.

2.1.2 Lack of verification on the renouncement status

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `BGW7702Logic`, functions `aggregate()` and `aggregateV2()` verify platform signer and owner signatures via the function `verifySignatures()` of contract `BGW7702Admin`. However, the function `verifySignatures()` never verifies whether the owner has renounced its ECDSA authority. After an owner renounces its original ECDSA key, a holder of that key can still authorize calls via the function `aggregate()` or `aggregateV2()` with a valid platform signer signature, circumventing the intended restriction on the renounced key.

```
190 function aggregate(
191     Call[] calldata _calls,
192     uint256 _nonce,
193     uint256 _deadline,
194     bytes calldata _signerSig,
195     bytes calldata _ownerSig
196 ) external payable {
197     if (block.timestamp > _deadline) revert SignatureExpired();
198
199     bgw7702Admin.verifySignatures(_calls, _nonce, _deadline, _signerSig, _ownerSig);
200
201     _executeCalls(_calls);
```

```
202 }
```

Listing 2.2: src/BGW7702Logic.sol

```
146 function aggregateV2(
147     Call[] calldata _calls,
148     uint256 _nonce,
149     uint256 _deadline,
150     bytes calldata _signerSig,
151     bytes calldata _ownerSig,
152     uint16 _businessType,
153     uint256 _businessAmount,
154     bytes calldata _extraData
155 ) external payable {
156     if (block.timestamp > _deadline) revert SignatureExpired();
157
158     bgw7702Admin.verifySignatures(_calls, _nonce, _deadline, _signerSig, _ownerSig);
159
160     _executeCalls(_calls);
161
162     emit Aggregate(_businessType, _businessAmount, _extraData);
163 }
```

Listing 2.3: src/BGW7702Logic.sol

```
 97 function verifySignatures(
 98     Call[] calldata _calls,
 99     uint256 _nonce,
100     uint256 _deadline,
101     bytes calldata _signerSig,
102     bytes calldata _eoaSig
103 ) external whenNotPaused returns (bool) {
104     bytes32 _digest = _consumeAndHash(_calls, _nonce, _deadline);
105     _checkSigner(_digest, _signerSig);
106     _verifySignature(msg.sender, _digest, _eoaSig);
107     return true;
108 }
```

Listing 2.4: src/BGW7702Admin.sol

```
354 function _verifySignature(
355     address _expectedSigner,
356     bytes32 _digest,
357     bytes calldata _signature
358 ) private pure {
359     address _recoveredAddr = _digest.recover(_signature);
360     if (_recoveredAddr != _expectedSigner) {
361         revert InvalidSignature();
362     }
363 }
```

Listing 2.5: src/BGW7702Admin.sol

```
23 function verify(
24     bytes calldata pubkey,
25     bytes32 hash,
26     bytes calldata realSig,
27     bytes calldata /* extraData */
28 ) external view whenNotPaused returns (bytes4) {
29     if (pubkey.length != 20) return ERC1271_INVALID;
30     // length checked above.
31     // forge-lint: disable-next-line(unsafe-typecast)
32     address guard = address(bytes20(pubkey));
33     if (guard == address(0)) return ERC1271_INVALID;
34     return BGWEcdsaLib.recover(hash, realSig, guard) ? ERC1271_MAGIC_VALUE : ERC1271_INVALID;
35 }
```

Listing 2.6: src/validators/GuardValidator.sol

Impact A renounced ECDSA key can still authorize calls through functions `aggregate()` and `aggregateV2()`.

Suggestion Verify the account's renouncement status before accepting the ECDSA owner signature.

Feedback from the project The project team resolved this issue by verifying the renouncement status in functions `aggregate()` and `aggregateV2()`. However, ECDSA authority cannot currently be revoked on-chain, as EIP-7851 is still a draft and not yet live.

2.1.3 Stale validator configuration after emergency deactivation

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In contract `BGWVerifyRouterWithTimeLock`, function `deactivateValidator()` allows an operator to immediately disable an active validator as an emergency measure. However, it does not invalidate previously queued `setVerifyInfos()` operations. Such an operation may set the validator's `isActive` flag back to true. Once the timelock expires, an authorized executor can execute such an operation and reactivate the validator despite its emergency deactivation. A validator disabled in response to a security issue is then restored, exposing its accounts to the same risk.

```
131 /// @notice Emergency kill-switch: an operator deactivates one validator
132 ///         (sets isActive=false) for fast incident response. Stored pubkeys
133 ///         and extraData are untouched. Re-activation is owner-only via
134 ///         `setVerifyInfos`.
135 function deactivateValidator(uint16 verifyId) external onlyOperator {
136     if (verifyId < 2) revert ReservedVerifyId(verifyId);
137     VerifyInfo storage info = _verifyInfos[verifyId];
138     address validator = info.validator;
139     if (validator == address(0)) revert InvalidVerifyId(verifyId);
140     if(!info.isActive) revert ValidatorAlreadyDeactivated(verifyId, validator);
141     info.isActive = false;
```

```
142     emit ValidatorDeactivated(verifyId, validator);
143 }
```

Listing 2.7: src/BGWVerifyRouterWithTimeLock.sol

```
104 function setVerifyInfos(
105     uint16[] calldata verifyIds,
106     address[] calldata validators,
107     bool[] calldata isActiveList,
108     bytes[] calldata extraDataList
109 ) external timelocked {// timelocked already check only owner can call
110     uint256 len = verifyIds.length;
111     if (len == 0) revert EmptyBatch();
112     if (validators.length != len || isActiveList.length != len || extraDataList.length != len) {
113         revert LengthMismatch();
114     }
115
116     for (uint256 i = 0; i < len;) {
117         uint16 id = verifyIds[i];
118         if (id < 2) revert ReservedVerifyId(id);
119         _checkZero(validators[i]);
120
121         VerifyInfo storage info = _verifyInfos[id];
122         info.validator = validators[i];
123         info.isActive = isActiveList[i];
124         info.extraData = extraDataList[i];
125         emit VerifyInfoSet(id, validators[i], isActiveList[i], extraDataList[i]);
126
127         unchecked { ++i; }
128     }
129 }
```

Listing 2.8: src/BGWVerifyRouterWithTimeLock.sol

Impact A deactivated validator can be reactivated by a `setVerifyInfos()` operation queued before the deactivation.

Suggestion Revise the logic accordingly.

Feedback from the project The project will add an operational SOP to mitigate this issue. Specifically, after a validator is disabled, the multisig must decode each queued `setVerifyInfos()` operation and cancel any proposal containing that validator ID.

2.2 Recommendation

2.2.1 Enforce hybrid signature scheme

Status Confirmed

Introduced by `Version 1`

Description In contract `BGW7702Admin`, function `verifySignaturesWithValidator()` verifies a platform signer signature and an owner signature. Each signature scheme is selected independently through its `verifyId`, allowing both signatures to use SLH-DSA. Given that SLH-DSA is

still experimental, relying solely on this algorithm may introduce security risks. If cryptographic weaknesses are discovered, both signatures can be forged at the same time. Therefore, it is recommended to combine post-quantum signatures with traditional signatures for authentication, as Arc [1] suggests.

```
114 ///     `_signerSet`; the owner `expectedSigner` must be `msg.sender`.
115 function verifySignaturesWithValidator(
116     Call[] calldata _calls,
117     uint256 _nonce,
118     uint256 _deadline,
119     bytes4 _selector,
120     SigParam calldata _signerSigParam,
121     SigParam calldata _ownerSigParam
122 ) external whenNotPaused returns (bool) {
123     bytes32 _digest = _consumeAndHashWithSelector(_calls, _nonce, _deadline, _selector);
124     // Same as two `InvalidSigner` reverts;
125     // `||` short-circuits and skips verify if not a signer.
126     if (!isValidSigner(_signerSigParam.expectedSigner)
127         || !BGWSigLib.verify(
128             bgwVerifyRouter,
129             _signerSigParam.expectedSigner,
130             _digest,
131             _signerSigParam.verifyId,
132             _signerSigParam.signature
133         )
134     ) {
135         revert InvalidSigner();
136     }
137
138     // Same as two `InvalidSignature` reverts;
139     // `||` short-circuits and skips verify if not the owner.
140     if (_ownerSigParam.expectedSigner != msg.sender
141         || !BGWSigLib.verify(
142             bgwVerifyRouter,
143             _ownerSigParam.expectedSigner,
144             _digest,
145             _ownerSigParam.verifyId,
146             _ownerSigParam.signature
147         )
148     ) {
```

Listing 2.9: src/BGW7702Admin.sol

Suggestion Enforce a hybrid signature policy that requires one signature to use a post-quantum scheme and the other to use ECDSA.

Feedback from the project The project decided not to make any code changes and will constrain the selection of signature schemes at the backend.

[1] https://github.com/circlefin/arc-node/blob/main/contracts/src/pq/IPQ.sol#L23-L24

2.3 Note

2.3.1 Potential centralization risks

Introduced by Version 1

Description In this project, several privileged roles (e.g., the owner, operators, and platform signers) can conduct sensitive operations, which introduces potential centralization risks. For example, the owner governs time-locked changes to the safe address, the operator set, validator configuration, and timelock delays. Operators can pause signature verification and sweep all ERC-20 and ETH balances. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

2.3.2 SLH-DSA deployment requirement

Introduced by Version 1

Description In this project, the contract `BGW7702Logic` forwards external signature verification to contract `BGWVerifyRouterWithTimeLock`, which invokes the configured `SlhDsaValidator`. Contract `SlhDsaValidator` verifies SLH-DSA-SHA2-128s signatures by calling Arc's dedicated precompile at `0x1800000000000000000000000000000000000004`. If the project plans to deploy contracts on more chains, the chain must have an available precompile at this address that implements the expected verification behavior.

```
 8 interface IArcPq {
 9     function verifySlhDsaSha2128s(bytes memory vk, bytes memory message, bytes memory sig)
10         external
11         view
12         returns (bool);
13 }
14
15 address constant ARC_SLH_DSA_PRECOMPILE = 0x1800000000000000000000000000000000000004;
16 uint256 constant SLH_DSA_SIG_LENGTH = 7856;
```

Listing 2.10: src/validators/SlhDsaValidator.sol

2.3.3 EntryPoint trust assumption and EIP-4337 compatibility

Introduced by Version 1

Description The project integrates with an external `EntryPoint` contract for ERC-4337 account abstraction, including `validateUserOp()` invocation, prefund handling, and execution settlement. The `EntryPoint` implementation is not included in this audit scope. This audit assumes that the deployed `EntryPoint` is correct, secure, and behaviorally compatible with this protocol's account logic, including validation semantics, gas accounting, and post-validation execution flow.

2.3.4 Integration assumption

Introduced by Version 1

Description In the project, several functions verify a signature against a given hash, including function `isValidSignature()` in contract `BGW7702Logic`, function `verifyRawSignature()` in contract `BGWVerifyRouterWithTimeLock`, and function `verify()` in the validator contracts. These functions do not verify whether the hash corresponds to a message bound to the account, chain ID, nonce, or other replay-protection parameters. Callers must account for this behavior and ensure that submitted messages include sufficient replay protection. Otherwise, a valid signature may be reused across accounts, chains, or transactions.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS