BLOCKSEC

Security Audit

Report for multisig-timelock-contracts

Date: August 11, 2026 **Version:** 1.0

Contact: contact@blocksec.com

目录

Report Manifest

Item	Description
Client	DeAgentAI
Target	multisig-timelock-contracts

Version History

Version	Date	Description
1.0	August 11, 2026	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-08-13 13:26:30+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 multisig-timelock-contracts of DeAgentAI.

The multisig-timelock-contracts is a token vault deployed on BSC and Sui. It implements a weighted multisig scheme based on off-chain signatures, combined with a conditional timelock. Each owner holds a weight, and an operation is authorized once the combined weight of the signers reaches a fixed threshold. The timelock delay applies only when at least one participating signer is flagged as timelocked. Both chains share identical high-level logic, differing solely in the signature primitive: EIP-712 with ECDSA on BSC and Ed25519 on Sui. The configured owners are A (weight 2, timelocked), B (weight 1), and C (weight 1), with a threshold of 2. Accordingly, A alone requires the delay, B and C together may execute immediately, and B or C alone is rejected. The only governed operation is transferring whitelisted tokens out of the vault, and all configuration is fixed at deployment and immutable.

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

- bsc/contracts/MultisigTimelockVault.sol
- sui/sources/vault.move

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
multisig-timelock-contracts	`Version 1`	`a1c4a63b95cda99e243235642494f1c8ec823f3d`
	`Version 2`	`0b8c18f7e8f27cd3e835a800fdaabdc1532cb20a`

[1] https://github.com/test666aamm/multisig-timelock-contracts

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.

表 1.1: Vulnerability Severity Classification

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

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

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

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.

Chapter 2 Findings

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

- Medium Risk: 2
- Recommendation: 6
- Note: 5

ID	Severity	Description	Category	Status
1	Medium	Lack of binding on intended signer set	Security Issue	Confirmed
2	Medium	Lack of balance validation against pending operations	Security Issue	Confirmed
3	-	Revise misleading documentation	Recommendation	Fixed
4	-	Add a view function for allowed tokens	Recommendation	Confirmed
5	-	Align the documentation with the implementation	Recommendation	Fixed
6	-	Add input validation in the Sui module	Recommendation	Confirmed
7	-	Add the vault identifier field and creation events	Recommendation	Confirmed
8	-	Validate owner public keys in function `new_internal()`	Recommendation	Confirmed
9	-	Verification before funding	Note	-
10	-	Potential centralization risks	Note	-
11	-	Weird ERC20 tokens	Note	-
12	-	The upgrade authority of Sui packages	Note	-
13	-	Design of the multi-signature scheme	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Lack of binding on intended signer set

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description In the contract `MultisigTimelockVault`, the function `schedule()` is used to submit signatures authorizing operations. However, the function accepts any composition of signers

and does not bind the intended signer set to the signed message. Consequently, valid signatures for the same operation can be omitted or recombined with signatures from other owners.

For example, a delayed operation that requires A+B signatures can be front-run. C could observe the pending transaction in the mempool and submit B+C signatures instead, scheduling and executing the action immediately. Conversely, A's signature can be included to impose a timelock on an operation that was intended to execute immediately.

The issues also exist for the Sui module `vault`.

```
215  /// @notice Authorize `op` with signatures whose combined weight reaches
216  ///         `threshold`, and queue it. `eta = now + delay` if any signer is
217  ///         timelocked, else `eta = now`. Run it later with `execute`.
218  /// @param signatures 65-byte ECDSA signatures sorted by signer ascending.
219  function schedule(Operation calldata op, bytes[] calldata signatures)
220      external
221      nonReentrant
222  {
223      if (op.nonce != nonce) revert BadNonce();
224      _validate(op);
225      (address[] memory signers, uint256 weight, bool anyTimelocked) =
226          _checkSignatures(hashOperation(op), signatures);
227      if (weight < threshold) revert NotEnoughWeight();
228
229      nonce++;
230
231      // Unified flow: always queue. A non-timelocked signer set (e.g. B+C)
232      // yields an eta of now, i.e. immediately executable via execute().
233      // Nothing runs inline — every op goes schedule -> execute/cancel.
234      bytes32 id = opId(op);
235      uint256 d = anyTimelocked ? delay : 0;
236      uint256 e = block.timestamp + d;
237      uint256 exp = e + GRACE_PERIOD;
238      eta[id] = e;
239      expiresAt[id] = exp;
240      emit Scheduled(
241          id, op.nonce, op.token, op.to, op.amount, e, exp, d == 0, signers
242      );
243  }
```

Listing 2.1: bsc/contracts/MultisigTimelockVault.sol

```
487  fun build_message(
488      action: u8,
489      vault_id: vector<u8>,
490      token: vector<u8>,
491      recipient: address,
492      amount: u64,
493      nonce: u64,
494  ): vector<u8> {
495      let mut m = PREFIX;
```

```
496         vector::push_back(&mut m, action);
497         vector::append(&mut m, vault_id);
498         vector::append(&mut m, u16_be(vector::length(&token)));
499         vector::append(&mut m, token);
500         vector::append(&mut m, address::to_bytes(recipient));
501         vector::append(&mut m, u64_be(amount));
502         vector::append(&mut m, u64_be(nonce));
503         m
504     }
```

Listing 2.2: sui/sources/vault.move

Impact An owner can change the effective timelock of an operation by recombining signatures from other owners.

Suggestion Bind the intended signer set into the signed message.

Feedback from the project The project states this implementation is by design, and these signatures are bound only to the operation. Additionally, they will exchange only signatures via private channels and submit the scheduled transaction to the private node.

2.1.2 Lack of balance validation against pending operations

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description In contract `MultisigTimelockVault`, function `schedule()` authorizes transfer operations without reserving the corresponding token balance. It also does not account for amounts committed to other pending operations. Since scheduled operations become executable once their delay has passed, a later operation with no delay can execute first, depleting the available balance. An earlier operation may fail at execution unless the contract is replenished before its execution window expires.

The issues also exist for the Sui module `vault`.

```
219     function schedule(Operation calldata op, bytes[] calldata signatures)
220         external
221         nonReentrant
222     {
223         if (op.nonce != nonce) revert BadNonce();
224         _validate(op);
225         (address[] memory signers, uint256 weight, bool anyTimelocked) =
226             _checkSignatures(hashOperation(op), signatures);
227         if (weight < threshold) revert NotEnoughWeight();
228
229         nonce++;
230
231         // Unified flow: always queue. A non-timelocked signer set (e.g. B+C)
```

```
232         // yields an eta of now, i.e. immediately executable via execute().
233         // Nothing runs inline — every op goes schedule -> execute/cancel.
234         bytes32 id = opId(op);
235         uint256 d = anyTimelocked ? delay : 0;
236         uint256 e = block.timestamp + d;
237         uint256 exp = e + GRACE_PERIOD;
238         eta[id] = e;
239         expiresAt[id] = exp;
240         emit Scheduled(
241             id, op.nonce, op.token, op.to, op.amount, e, exp, d == 0, signers
242         );
243     }
```

Listing 2.3: bsc/contracts/MultisigTimelockVault.sol

```
288 public fun schedule_transfer<T>(
289     vault: &mut Vault,
290     recipient: address,
291     amount: u64,
292     signer_indices: vector<u64>,
293     signatures: vector<vector<u8>>,
294     clock: &Clock,
295 ) {
296     let tn = type_name::with_defining_ids<T>();
297     assert!(vec_set::contains(&vault.allowed_tokens, &tn), ETokenNotAllowed);
298     assert!(amount > 0, EZeroAmount);
299
300     let token_bytes = ascii::into_bytes(type_name::into_string(tn));
301     let msg = build_message(
302         ACTION_SCHEDULE, object::uid_to_bytes(&vault.id), copy token_bytes, recipient, amount,
                vault.nonce,
303     );
304     let (weight, any_tl) = verify_sigs(vault, &msg, &signer_indices, &signatures);
305     assert!(weight >= vault.threshold, ENotEnoughWeight);
306     vault.nonce = vault.nonce + 1;
307
308     // Unified flow: always queue. `delay == 0` (e.g. B+C) just yields an eta
309     // of now, i.e. immediately executable via execute_transfer.
310     let id = hash::blake2b256(&msg);
311     let now = clock::timestamp_ms(clock);
312     let n = vault.nonce - 1;
313     let d = if (any_tl) { vault.delay_ms } else { 0 };
314     let e = now + d;
315     let exp = e + GRACE_PERIOD_MS;
316     table::add(&mut vault.queue, id, QueueEntry { eta_ms: e, expires_at_ms: exp });
317     event::emit(Scheduled {
318         id, nonce: n, token: token_bytes, recipient,
319         amount, eta_ms: e, expires_at_ms: exp, immediate: d == 0, signers: signer_indices,
320     });
```

```
321    }
```

Listing 2.4: sui/sources/vault.move

Impact Delayed transfer operations may fail if the required balance was consumed.

Suggestion Reserve balances for scheduled operations, and release them upon execution, cancellation, or expiration.

Feedback from the project The project states that off-chain operators will enforce balance validation against pending operations before authorizing new transfers and will correctly process the amounts corresponding to cancelled operations.

2.2 Recommendation

2.2.1 Revise misleading documentation

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The implementation allows a scheduled operation to be cancelled immediately, regardless of the signer's `timelocked` flag. However, files docs/生产环境密钥管理方案.md (e.g., at line 211) and docs/测试网全流程演练指南.md (e.g., at line 67) still describe a delayed cancel and refer to the removed function `executeCancel()`. It is recommended to update these documents to the current semantics.

```
205**场景 2: A 延迟取消**
206```
2071. 发现待执行操作有问题
2082. Owner A 签名 Cancel 摘要
2093. Relayer 调用 vault.cancel(op, [sigA])
210   → 取消进入队列，eta = now + 4个月
2114. 4 个月后，Relayer 调用 vault.executeCancel(op)
212```
```

Listing 2.5: docs/生产环境密钥管理方案.md

```
676. **Scenario 3**: B+C schedule + A cancel + 等待 delay 后 executeCancel
```

Listing 2.6: docs/测试网全流程演练指南.md

Suggestion Update the cancel description in these documents.

2.2.2 Add a view function for allowed tokens

Status Confirmed

Introduced by `Version 1`

Description The Sui module `vault` stores a fixed `allowed_tokens` whitelist and uses it to validate deposits and scheduled transfers. However, it does not expose a view function to read the configured allowed token list, while the BSC contract implements function `getAllowedTokens()`. This makes the two scoped implementations less consistent and forces integrators or operators to infer the whitelist from creation inputs or events instead of querying the vault state directly.

```
105      /// Token whitelist, fixed at creation.
106      allowed_tokens: VecSet<TypeName>,
```

Listing 2.7: sui/sources/vault.move

Suggestion Add a public view function in module `vault` that returns the configured allowed token type names.

2.2.3 Align the documentation with the implementation

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The documentation `bsc/README.md` states that operations with `delay = 0` are queued with `eta = now` and can be executed in the next block. However, function `execute()` in contract `MultisigTimelockVault` only requires the current `block.timestamp` to be greater than or equal to `eta`. Therefore, the operation can be scheduled and executed in the same block and even the same transaction.

```
57`delay = 0` ops (e.g. B+C) are queued with `eta = now`, so `execute` can be
58called in the next block — same path as timelocked ops, just no wait.
```

Listing 2.8: bsc/README.md

```
249   function execute(Operation calldata op) external nonReentrant {
250       bytes32 id = opId(op);
251       uint256 e = eta[id];
252       if (e == 0) revert NotScheduled();
253       uint256 exp = expiresAt[id];
254       if (block.timestamp < e) revert NotYetExecutable();
255       if (block.timestamp > exp) revert GracePeriodExpired();
```

Listing 2.9: bsc/contracts/MultisigTimelockVault.sol

Suggestion Either enforce execution in a subsequent block, or update the documentation.

2.2.4 Add input validation in the Sui module

Status Confirmed

Introduced by `Version 1`

Description Contract `MultisigTimelockVault` on BSC enforces several input validations that are absent from the corresponding Sui module `vault`:

1. The BSC constructor limits the owner set through `MAX_OWNERS`. However, the Sui vault-creation functions impose no explicit maximum.

2. The BSC function `schedule()` invokes `_validate()` to reject `address(0)` as a recipient. However, the Sui function `schedule_transfer()` accepts the zero address.

3. The BSC function `deposit()` rejects a zero amount, whereas the Sui equivalent accepts a zero-value coin.

```
138   constructor(
139       address[] memory owners_,
140       uint256[] memory weights_,
141       bool[] memory timelocked_,
142       uint256 threshold_,
143       uint256 delay_,
144       address[] memory allowedTokens_
145   ) EIP712("MultisigTimelockVault", "1") {
146       uint256 n = owners_.length;
147       if (n == 0 || weights_.length != n || timelocked_.length != n) {
148           revert LengthMismatch();
149       }
150       if (n > MAX_OWNERS) revert TooManyOwners();
```

Listing 2.10: bsc/contracts/MultisigTimelockVault.sol

```
170       // Validate owners: 32-byte Ed25519 pubkeys, distinct (no duplicate keys
171       // — otherwise one key could be double-counted via two indices), weight > 0.
172       let mut total = 0;
173       let mut seen = vec_set::empty<vector<u8>>();
174       let mut i = 0;
175       while (i < n) {
176           let pk = vector::borrow(&owners, i);
177           assert!(vector::length(pk) == 32, EBadOwnerKey);
178           assert!(!vec_set::contains(&seen, pk), EDuplicateOwner);
179           vec_set::insert(&mut seen, *pk);
180           let w = *vector::borrow(&weights, i);
181           assert!(w > 0, EZeroWeight);
182           total = total + w;
183           i = i + 1;
184       };
```

Listing 2.11: sui/sources/vault.move

```
219   function schedule(Operation calldata op, bytes[] calldata signatures)
220       external
221       nonReentrant
222   {
223       if (op.nonce != nonce) revert BadNonce();
224       _validate(op);
```

Listing 2.12: bsc/contracts/MultisigTimelockVault.sol

```
function _validate(Operation calldata op) internal view {
    if (!isAllowedToken[op.token]) revert TokenNotAllowed();
    if (op.to == address(0)) revert ZeroAddress();
    if (op.amount == 0) revert ZeroAmount();
}
```

Listing 2.13: bsc/contracts/MultisigTimelockVault.sol

```
public fun schedule_transfer<T>(
    vault: &mut Vault,
    recipient: address,
    amount: u64,
    signer_indices: vector<u64>,
    signatures: vector<vector<u8>>,
    clock: &Clock,
) {
    let tn = type_name::with_defining_ids<T>();
    assert!(vec_set::contains(&vault.allowed_tokens, &tn), ETokenNotAllowed);
    assert!(amount > 0, EZeroAmount);

    let token_bytes = ascii::into_bytes(type_name::into_string(tn));
    let msg = build_message(
        ACTION_SCHEDULE, object::uid_to_bytes(&vault.id), copy token_bytes, recipient, amount,
            vault.nonce,
    );
    let (weight, any_tl) = verify_sigs(vault, &msg, &signer_indices, &signatures);
    assert!(weight >= vault.threshold, ENotEnoughWeight);
    vault.nonce = vault.nonce + 1;
```

Listing 2.14: sui/sources/vault.move

```
function deposit(IERC20 token, uint256 amount) external nonReentrant {
    if (!isAllowedToken[address(token)]) revert TokenNotAllowed();
    if (amount == 0) revert ZeroAmount();
```

Listing 2.15: bsc/contracts/MultisigTimelockVault.sol

```
public fun deposit<T>(vault: &mut Vault, c: coin::Coin<T>) {
    let tn = type_name::with_defining_ids<T>();
    assert!(vec_set::contains(&vault.allowed_tokens, &tn), ETokenNotAllowed);
    let amount = coin::value(&c);
```

Listing 2.16: sui/sources/vault.move

Suggestion Validate the inputs in the Sui module.

Feedback from the project The project states that Sui vault creation is controlled by deploy scripts, transfers to the zero address would require explicit owner signatures, and zero-value deposits are considered harmless no-ops.

2.2.5 Add the vault identifier field and creation events

Status Confirmed

Introduced by Version 1

Description In contract `vault`, the events emitted by functions `deposit()`, `schedule_transfer()`, `execute_transfer()`, and `cancel_inner()` do not contain an explicit `vault_id` field. As a result, off-chain indexers must parse transaction inputs to attribute events to the correct vault, rather than relying on the event data alone.

Vault creation also does not emit a dedicated event. This makes it harder for indexers to detect new vault instances and associate subsequent events with them.

```
277     event::emit(Deposited {
278         token: ascii::into_bytes(type_name::into_string(type_name::with_defining_ids<T>())),
279         amount,
280     });
```

Listing 2.17: sui/sources/vault.move

```
317     event::emit(Scheduled {
318         id, nonce: n, token: token_bytes, recipient,
319         amount, eta_ms: e, expires_at_ms: exp, immediate: d == 0, signers: signer_indices,
320     });
```

Listing 2.18: sui/sources/vault.move

```
341     event::emit(Executed { id, token: token_bytes, recipient, amount, nonce });
```

Listing 2.19: sui/sources/vault.move

```
419     event::emit(Cancelled {
420         id, token: token_bytes, recipient, amount, nonce, signers: *signer_indices,
421     });
```

Listing 2.20: sui/sources/vault.move

Suggestion Add `vault_id` to all vault events, and emit a dedicated event when a vault is created.

2.2.6 Validate owner public keys in function `new_internal()`

Status Confirmed

Introduced by Version 1

Description In the Sui module `vault`, function `new_internal()` accepts arbitrary 32-byte values as owner public keys and validates their length and uniqueness. However, it does not reject small-order Ed25519 points. If a small-order public key is mistakenly registered as an owner, any account can submit a forged signature and obtain that owner's configured weight.

```
170     // Validate owners: 32-byte Ed25519 pubkeys, distinct (no duplicate keys
171     // — otherwise one key could be double-counted via two indices), weight > 0.
172     let mut total = 0;
173     let mut seen = vec_set::empty<vector<u8>>();
174     let mut i = 0;
175     while (i < n) {
176         let pk = vector::borrow(&owners, i);
177         assert!(vector::length(pk) == 32, EBadOwnerKey);
178         assert!(!vec_set::contains(&seen, pk), EDuplicateOwner);
179         vec_set::insert(&mut seen, *pk);
180         let w = *vector::borrow(&weights, i);
181         assert!(w > 0, EZeroWeight);
182         total = total + w;
183         i = i + 1;
184     };
```

Listing 2.21: sui/sources/vault.move

Suggestion Reject all small-order Ed25519 public keys in function `new_internal()` before storing owner keys.

Feedback from the project The project states that owner keys are generated through standard Ed25519 and will be validated by deploy scripts and pre-funding verification.

2.3 Note

2.3.1 Verification before funding

Introduced by `Version 1`

Description The owner configuration is fixed at deployment and cannot be changed afterward. Each configured owner counts toward the `threshold` regardless of whether its key can actually produce a valid signature. These parameters therefore need to be verified before any funds are deposited, since misconfiguration could otherwise leave funds permanently unwithdrawable. For example, in the contract `MultisigTimelockVault`, the function `_checkSignatures()` recovers signers through `ECDSA.recover()` only, meaning any owner that is a contract cannot produce a valid signature. Meanwhile, the function `new_internal()` of the Sui module `vault` only requires each owner's public key to be 32 bytes.

Feedback from the project The project states that they will perform a comprehensive verification before any funds are deposited into a newly deployed vault. This covers independent owner confirmations, end-to-end dust-signing tests, and on-chain read-backs of weights, threshold, delay, whitelist, and Sui immutability against the intended config.

2.3.2 Potential centralization risks

Introduced by Version 1

Description In this project, several privileged roles (i.e., the owners) can conduct sensitive operations, which introduces potential centralization risks. Specifically, under the intended configuration of `ownerA` with weight 2 and timelocked, `ownerB` and `ownerC` with weight 1 each, and `threshold` 2, the pair `ownerB` plus `ownerC` reaches `threshold` without any timelocked signer, and can therefore transfer the whitelisted tokens out of the vault immediately. Meanwhile, any pair of owners can cancel a queued operation at once through the function `cancel()`. 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 project states that they will deploy hardware wallets for all keys, enforce physically separated custody, and require secure signing environments with no shared infrastructure.

2.3.3 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. For example, the function `deposit()` in contract `MultisigTimelockVault` invokes `safeTransferFrom()` for the requested `amount` and emits that same amount in the `Deposited` event without measuring the vault's actual balance increase. If an allowed token charges a transfer fee, the vault receives fewer tokens than the event reports.

Feedback from the project Non-standard tokens are not supported.

2.3.4 The upgrade authority of Sui packages

Introduced by Version 1

Description Publishing a Sui package creates an `UpgradeCap` object whose holder can publish compatible upgrades. By default, the deployment script consumes this object through `package::make_immutable()` in the publishing transaction, permanently preventing future upgrades.

If `SUI_BURN_UPGRADE_CAP` is set to false, the `UpgradeCap` is transferred to the deployer and remains valid until it is consumed through the script `seal_vault.mjs`. For an immutable production deployment, the project should ensure that no corresponding `UpgradeCap` remains.

```
73 const makeImmutable = (process.env.SUI_BURN_UPGRADE_CAP ?? "true") !== "false";
```

Listing 2.22: sui/scripts/deploy.mjs

```
130 // ---- 2. publish ----
131 const pub = new Transaction();
132 const [cap] = pub.publish({ modules, dependencies });
133 if (makeImmutable) {
134   pub.moveCall({ target: "0x2::package::make_immutable", arguments: [cap] });
135 } else {
136   pub.transferObjects([cap], sender);
137 }
```

Listing 2.23: sui/scripts/deploy.mjs

Feedback from the project The project confirms that for production deployments, `SUI_BURN_UPGRADE_CAP` is set to true and confirm immutability on-chain before funding.

2.3.5 Design of the multi-signature scheme

Introduced by `Version 1`

Description Currently, each production vault uses a fixed configuration with three owners. Owner A has a weight of 2 and is subject to a timelock. Owners B and C each have a weight of 1, and the execution threshold is 2.

Although Owner A's weight equals the combined weight of B and C, the two paths are not equivalent. The B + C path requires two independent signatures, which represents a higher requirement than A acting alone. Therefore, B + C operations execute immediately, while any A-only action is subject to the timelock.

The A+B and A+C paths have the same on-chain authorization semantics as the A-only path. The additional signatures only satisfy off-chain operational requirements and do not change the on-chain execution conditions.

Feedback from the project The project states that this behavior is intentional. When the A+B or A+C authorization path is required, it is enforced through off-chain controls.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS