BLOCKSEC

Security Audit Report for Bridge Contracts

Date: Jul 3, 2026 **Version:** 2.0
Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	PopDEX
Target	Bridge Contracts

Version History

Version	Date	Description
1.0	May 14, 2026	First release
2.0	Jul 3, 2026	Second release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-07-09 06:25:43+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	Rust, 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 Bridge Contracts of PopDEX.

The DexChain Bridge implements a Hub-and-Spoke cross-chain bridge architecture consisting of native Rust precompiled contracts on the Hub chain (Core) and Solidity upgradeable contracts on each Spoke chain (EVM). On the Hub side, three Rust modules are deployed as native precompiles: MessageGateway at 0x1001, which handles inbound message verification, hot/cold verifier signature validation, replay protection, and verifier set management; TokenBridge at 0x100f, which manages user withdrawals, deposit execution, and governance configuration synchronization to Spoke chains; and common, which provides shared utilities including EIP-712 digest construction.

On the Spoke side, the bridge is implemented through Solidity UUPS-upgradeable contracts. MessageGateway.sol is responsible for receiving and validating cross-chain messages from the Hub chain, enforcing message execution rules, replay protection, and authorized gateway interactions. TokenBridge.sol manages token deposit and withdrawal flows on the Spoke chain, including locking or releasing bridged assets, initiating outbound bridge messages to the Hub, executing inbound bridge instructions, and applying bridge-related governance or configuration updates propagated from the Hub.

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

- token_bridge.rs
- message_gateway.rs
- common.rs
- TokenBridge.sol
- MessageGateway.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/secmgtcoding/dex-bridge.git

Project	Version	Commit Hash
Bridge Contracts	Version 1	e8d97c1e208aee11806bd624b551bf61e339dc11
	Version 2	541bee9faa6f35936a2bd140bc2cf113815375c4
	Version 3	af679f46b017559d4d03ef9d062016288a62f1d9

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 **ten** potential security issues. Besides, we have **eleven** recommendations and **seven** notes.

- High Risk: 2
- Medium Risk: 4
- Low Risk: 4
- Recommendation: 11
- Note: 7

ID	Severity	Description	Category	Status
1	High	Lack of `challenge_period` initialization in genesis blocks subsequent verifier updates	Security Issue	Fixed
2	High	Spoke custody configuration incorrectly overwrites the Hub `challenge_period`	Security Issue	Fixed
3	Medium	Improper configuration synchronization process	Security Issue	Confirmed
4	Medium	Incorrect address encoding in the function `_calculateMessageHash()`	Security Issue	Fixed
5	Medium	Potential DoS in the function `depositWithPermit()`	Security Issue	Fixed
6	Medium	Inconsistent use of the bridge precompile address	Security Issue	Fixed
7	Low	Potential DoS due to the improper deposit accounting in the contract `TokenBridge`	Security Issue	Confirmed
8	Low	Potential delay of verifier update due to missing duplicate check	Security Issue	Fixed
9	Low	Unbounded signature input may lead to excessive gas consumption	Security Issue	Fixed
10	Low	Inconsistent failure handling in the function `executeMessage()`	Security Issue	Fixed
11	-	Add checks in the function `validate_verifier_set_addresses()`	Recommendation	Fixed
12	-	Ensure consistent governance payload checks between Hub and Spoke	Recommendation	Fixed
13	-	Add destination checks in the function `handle_receive_message()`	Recommendation	Fixed
14	-	Add overflow protection when casting `tx_index`	Recommendation	Fixed
15	-	Add chain ID checks for `source_chain_id` in the function `handle_receive_message()`	Recommendation	Fixed

16	-	Add non-zero checks for the variable `tokenId` in the route configuration	Recommendation	Fixed
17	-	Add minimum value checks in the function `setChallengerPeriod()`	Recommendation	Fixed
18	-	Implement emergency withdrawal in `TokenBridge` to prevent accidental transfers	Recommendation	Confirmed
19	-	Revise the custom error `InvalidSignature`	Recommendation	Fixed
20	-	Remove redundant sorting in function `handle_list_verifiers()`	Recommendation	Fixed
21	-	Remove redundant code	Recommendation	Fixed
22	-	Ensure the same verifier set is shared across all Spoke chains	Note	-
23	-	Proper handling for `USDT` on `Tron`	Note	-
24	-	Proper aggregation logic for relayer signatures	Note	-
25	-	Proper Cross-Spoke liquidity management	Note	-
26	-	Potential centralization risks	Note	-
27	-	Out of scope dependencies and external logic	Note	-
28	-	Ensure proper setup in production	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Lack of `challenge_period` initialization in genesis blocks subsequent verifier updates

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `handle_update_verifiers()`, the verifier set can be initialized during genesis when variable `total_power` is zero and variable `block_number` is zero. In this genesis path, the initial verifier set is applied and variable `Meta` is written to storage, but variable `challenge_period` is not initialized.

As a result, variable `challenge_period` remains at its default value of zero after genesis. However, in the non-genesis execution path, the function explicitly requires variable `challenge_period` to be non-zero and will revert otherwise. This creates an inconsistency between initialization and subsequent updates.

Consequently, any attempt to update the verifier set after genesis via function `handle_update_verifiers()` will revert due to missing initialization of variable `challenge_period`, effectively blocking all regular verifier set updates.

```
230   pub(super) fn handle_update_verifiers<
231       Store: CoreStateStore + CoreStateReader + CoreAccountReader + CoreAccountWriter,
232   >(
233       input: PrecompileInput<'_>,
234       context: &PrecompileContext<'_, Store>,
235       output: &mut PrecompileOutput,
236   ) -> Result<(), PrecompileError> {
237       crate::check_role!(input, context, crate::permission_verifier::Role::Relayer);
238       let meta = read_verifier_meta(context.storage)?;
239       if meta.paused {
240           return Err(PrecompileError::ExecutionError(
241               "Bridge is paused; use emergencyUpdateVerifiers".to_string(),
242           ));
243       }
244
245       let call = IMessageGateway::updateVerifiersCall::abi_decode(input.data()).map_err(|e| {
246           tracing::info!(error = %e, "Failed to decode updateVerifiers call");
247           PrecompileError::InvalidInput(format!("Failed to decode: {}", e))
248       })?;
249       let (epoch, hot_verifiers, cold_verifiers, powers, signatures) =
250           (call.epoch, call.hotVerifiers, call.coldVerifiers, call.powers, call.signatures);
251
252       validate_verifier_set_addresses(&hot_verifiers, &cold_verifiers, &powers)?;
253
254       if meta.total_power.is_zero() {
255           if context.block_number != 0 {
256               return Err(PrecompileError::ExecutionError(
257                   "Bridge verifiers must be initialized in genesis when totalPower == 0".to_string
                        (),
258               ));
259           }
260           if epoch != 0 {
261               return Err(PrecompileError::ExecutionError(format!(
262                   "Genesis verifier initialization requires epoch == 0, got {}",
263                   epoch
264               )));
265           }
266           if !signatures.is_empty() {
267               return Err(PrecompileError::ExecutionError(
268                   "Genesis verifier initialization requires empty signatures".to_string(),
269               ));
270           }
271           let mut meta = meta;
272           apply_verifier_set(context.storage, &mut meta, &hot_verifiers, &cold_verifiers, &powers
                )?;
273           meta.epoch = 0;
274           write_verifier_meta(context.storage, meta)?;
275           output.push_log(create_log(
276               MESSAGE_GATEWAY_PRECOMPILE_ADDRESS,
```

```
277             &IMessageGateway::VerifierSetUpdated {
278                 epoch: 0u64,
279                 hotVerifiers: hot_verifiers.clone(),
280                 coldVerifiers: cold_verifiers.clone(),
281                 powers,
282             },
283         ));
284         ensure_verifier_accounts(context, &hot_verifiers, &cold_verifiers)?;
285         output.bytes = Bytes::new();
286         output.gas_used = GAS_BRIDGE_UPDATE_VERIFIERS;
287         return Ok(());
288     }
289 
290     if epoch <= meta.epoch {
291         return Err(PrecompileError::ExecutionError(format!(
292             "epoch must be > currentVerifierEpoch ({}), got {}",
293             meta.epoch, epoch
294         )));
295     }
296 
297     let digest = update_verifiers_signed_digest(epoch, &hot_verifiers, &cold_verifiers, &powers
        )?;
298     let accumulated_power = verify_signatures_until_required_power_for_set(
299         context.storage,
300         digest,
301         &signatures,
302         VerifierSetKind::Hot,
303         meta.required_power,
304     )?;
305     if accumulated_power < meta.required_power {
306         return Err(PrecompileError::ExecutionError(
307             "Insufficient verifier power for update".to_string(),
308         ));
309     }
310 
311     if meta.challenge_period == 0 {
312         return Err(PrecompileError::ExecutionError(
313             "challenge_period is not configured".to_string(),
314         ));
315     }
316     let now_secs = bridge_block_timestamp_secs(context.block_timestamp);
317     let execute_after = now_secs
318         .checked_add(meta.challenge_period)
319         .ok_or_else(|| PrecompileError::ExecutionError("execute_after overflow".to_string()))?;
320     let params_hash = compute_verifier_params_hash(epoch, &hot_verifiers, &cold_verifiers, &
        powers);
321     write_pending_verifier_update(
322         context.storage,
323         PendingVerifierUpdate { epoch, execute_after, params_hash },
324     )?;
325     output.push_log(create_log(
326         MESSAGE_GATEWAY_PRECOMPILE_ADDRESS,
327         &IMessageGateway::VerifierSetQueued {
```

```
328                 epoch,
329                 paramsHash: params_hash,
330                 executeAfter: U256::from(execute_after),
331             },
332         ));
333         output.bytes = Bytes::new();
334         output.gas_used = GAS_BRIDGE_UPDATE_VERIFIERS;
335         Ok(())
336     }
```

Listing 2.1: message_gateway.rs

Impact After genesis, verifier set updates via function `handle_update_verifiers()` are permanently blocked, as variable `challenge_period` remains zero while the function requires it to be non-zero.

Suggestion Ensure that variable `challenge_period` is properly initialized during the genesis path, so that subsequent verifier updates can proceed as expected.

2.1.2 Spoke custody configuration incorrectly overwrites the Hub `challenge_period`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `handle_set_custody_chain_config()`, `BridgeAdmin` configures per-Spoke runtime parameters such as variable `dispute_period`, variable `verifier_update_period`, and variable `locker_threshold` for a given custody chain. However, after storing the configuration for the specified `Spoke` chain, the function also writes variable `verifier_update_period` into the `Hub`'s global variable `challenge_period`.

This creates an incorrect coupling between Spoke-specific configuration and the `Hub`'s own verifier update timelock. Because different `Spoke` chains may legitimately use different variable `verifier_update_period` values, each invocation to function `handle_set_custody_chain_config()` can overwrite the `Hub`'s `challenge_period` with the most recently configured `Spoke`'s value.

As a result, the `Hub`'s verifier update timelock no longer reflects a Hub-specific security parameter and instead becomes dependent on the latest `Spoke` configuration. This is inconsistent with the protocol design, where `Hub` and `Spoke` chains are expected to maintain independent security parameters.

```
579     pub(super) fn handle_set_custody_chain_config<
580         Store: CoreStateStore + CoreStateReader + CoreAccountWriter,
581     >(
582         input: PrecompileInput<'_>,
583         context: &PrecompileContext<'_, Store>,
584         output: &mut PrecompileOutput,
585     ) -> Result<(), PrecompileError> {
586         crate::check_role!(input, context, crate::permission_verifier::Role::BridgeAdmin);
587
588         let call = ITokenBridge::setCustodyChainConfigCall::abi_decode(input.data())
```

```
589             .map_err(|e| PrecompileError::InvalidInput(format!("Failed to decode: {}", e)))?;
590
591         let custody_chain_id: u64 = call
592             .custodyChainId
593             .try_into()
594             .map_err(|_| PrecompileError::InvalidInput("custodyChainId must fit in u64".to_string()
                    ))?;
595         let dispute_period: u64 = call
596             .disputePeriod
597             .try_into()
598             .map_err(|_| PrecompileError::InvalidInput("disputePeriod must fit in u64".to_string())
                    )?;
599         let verifier_update_period: u64 = call.verifierUpdatePeriod.try_into().map_err(|_| {
600             PrecompileError::InvalidInput("verifierUpdatePeriod must fit in u64".to_string())
601         })?;
602         let block_time_millis: u64 = call.blockTimeMillis.try_into().map_err(|_| {
603             PrecompileError::InvalidInput("blockTimeMillis must fit in u64".to_string())
604         })?;
605         let locker_threshold: u64 = call.lockerThreshold.try_into().map_err(|_| {
606             PrecompileError::InvalidInput("lockerThreshold must fit in u64".to_string())
607         })?;
608
609         let to_address = read_chain_token_bridge_address(context, custody_chain_id)?;
610         if to_address == Address::ZERO {
611             return Err(PrecompileError::ExecutionError(
612                 "getChainTokenBridge(remoteChainId) not set; call setChainTokenBridge first for
                        custody target"
613                     .to_string(),
614             ));
615         }
616
617         let current_version = storage::read_custody_chain_config(context.storage, custody_chain_id)
                ?
618             .map_or(0, |config| config.version);
619         let version = current_version.checked_add(1).ok_or_else(|| {
620             PrecompileError::ExecutionError("Custody chain config version overflow".to_string())
621         })?;
622         let config = CustodyChainConfig {
623             dispute_period,
624             verifier_update_period,
625             block_time_millis,
626             locker_threshold,
627             version,
628         };
629         storage::write_custody_chain_config(context.storage, custody_chain_id, config)?;
630         // Use verifier_update_period for MessageGateway verifier-update timelock (>=
                dispute_period invariant enforced on spoke side).
631         write_challenge_period(context.storage, verifier_update_period)?;
632
633         let selector = apply_custody_chain_config_selector();
634         let payload = (
635             call.disputePeriod,
636             call.verifierUpdatePeriod,
```

```
637             call.blockTimeMillis,
638             call.lockerThreshold,
639             version,
640         )
641             .abi_encode();
642         let mut data = Vec::with_capacity(4 + payload.len());
643         data.extend_from_slice(&selector);
644         data.extend_from_slice(&payload);
645
646         super::message_gateway::emit_message_sent(
647             context,
648             output,
649             TOKEN_BRIDGE_PRECOMPILE_ADDRESS,
650             custody_chain_id,
651             to_address,
652             Bytes::from(data),
653         )?;
654
655         output.bytes = Bytes::new();
656         output.gas_used = GAS_BRIDGE_SET_CUSTODY_CHAIN_CONFIG;
657         Ok(())
658     }
```

Listing 2.2: token_bridge.rs

Impact Updating the custody configuration for one `Spoke` chain can unintentionally modify the `Hub`'s own verifier update timelock, causing Hub-side security parameters to drift based on unrelated Spoke-specific settings.

Suggestion Separate Hub-level and Spoke-level timing parameters, and ensure that configuring a `Spoke` custody chain does not overwrite the `Hub`'s global variable `challenge_period`.

2.1.3 Improper configuration synchronization process

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description In the project, the configuration updates (i.e., route and chain configurations) are asynchronized due to the Hub-Spoke design. Updates take effect instantly on the `Hub` chain, while the `Spoke` chains must wait for relayer confirmations and signatures. However, this design introduces risks (e.g., lock of funds or incorrect accounting). Specifically, if the `Hub` chain sends a route configuration update request that changes a variable `tokenId`'s corresponding token address, deposits of the outdated token on a `Spoke` chain may succeed before the `Spoke` chain applies the update. As a result, this improper design may introduce two potential impacts. If the corresponding deposit messages later fail on the `Hub` chain, users' tokens become locked because the contract `TokenBridge` lacks an emergency withdrawal function. Second, if the `Hub` chain processes the deposit messages despite the mismatch, incorrect accounting may occur.

```
389     function _executeDepositToCore(address token, uint256 amount, address receiver, address
            depositor) internal {
```

```
390         if (token == address(0)) {
391             revert InvalidToken();
392         }
393
394         // Validate amount is not zero
395         if (amount == 0) {
396             revert InvalidAmount();
397         }
398
399         // Validate receiver address is not zero
400         if (receiver == address(0)) {
401             revert InvalidReceiver();
402         }
403
404         address destTokenBridge = coreTokenBridge;
405         if (destTokenBridge == address(0)) revert InvalidCoreTokenBridge();
406
407         uint64 tokenId = tokenIdByCustodyTokenAddress[token];
408         if (tokenId == 0) revert TokenMappingNotFound();
409         _enforceDepositRoute(token, tokenId, amount);
410
411         address destTokenAddress = _coreTokenAddress(tokenId);
412
413         uint8 srcDecimals = _getTokenDecimals(token);
414         uint256 amountCanonical = _srcRawToCanonical18(amount, srcDecimals);
415
416         // Transfer tokens from depositor to this contract (lock tokens; use source-chain raw
                amount)
417         IERC20(token).safeTransferFrom(depositor, address(this), amount);
418
419         // Unified payload: CORE_DEPOSIT_SELECTOR + abi.encode(receiver, destTokenAddress,
                amount_canonical)
420         bytes memory payloadData =
421             abi.encodePacked(CORE_DEPOSIT_SELECTOR, abi.encode(receiver, destTokenAddress,
                    amountCanonical));
422
423         // Call MessageGateway to create crosschain message
424         IMessageGateway(messageGateway).sendMessage(coreChainId, destTokenBridge, payloadData);
425
426         emit Deposit(depositor, coreChainId, token, amount, receiver);
427     }
```

Listing 2.3: TokenBridge.sol

```
479         if (selector == APPLY_CUSTODY_ROUTE_SELECTOR) {
480             (
481                 uint64 tokenId,
482                 address custodyTokenAddress,
483                 uint256 minimumDeposit,
484                 uint256 depositCap,
485                 bool depositEnabled,
486                 bool withdrawEnabled,
487                 uint64 version
```

```
488         ) = abi.decode(data[4:], (uint64, address, uint256, uint256, bool, bool, uint64));
489         if (version <= tokenConfigByTokenId[tokenId].version) {
490             return;
491         }
492         address prevCustody = tokenConfigByTokenId[tokenId].custodyTokenAddress;
493         if (prevCustody != address(0) && prevCustody != custodyTokenAddress) {
494             if (tokenIdByCustodyTokenAddress[prevCustody] == tokenId) {
495                 tokenIdByCustodyTokenAddress[prevCustody] = 0;
496             }
497         }
498         tokenConfigByTokenId[tokenId] = RuntimeTokenConfig({
499             custodyTokenAddress: custodyTokenAddress,
500             minimumDeposit: minimumDeposit,
501             depositCap: depositCap,
502             depositEnabled: depositEnabled,
503             withdrawEnabled: withdrawEnabled,
504             version: version
505         });
506         if (custodyTokenAddress != address(0)) {
507             tokenIdByCustodyTokenAddress[custodyTokenAddress] = tokenId;
508         }
509         emit CustodyRouteConfigUpdated(
510             tokenId, custodyTokenAddress, minimumDeposit, depositCap, depositEnabled,
                    withdrawEnabled, version
511         );
512         return;
513     }
```

Listing 2.4: TokenBridge.sol

Impact Potential lock of funds or incorrect accounting on the `Hub` chain.

Suggestion Revise the configuration synchronization design and code accordingly.

Feedback from the project The project stated that the token configuration will not be changed in the current phase. They will handle the token configuration update with a proper solution in the future.

2.1.4 Incorrect address encoding in the function `_calculateMessageHash()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `MessageGateway`, the function `_calculateMessageHash()` uses inline assembly with `shl(96, from)` and `shl(96, to)` to left-align 20-byte addresses in 32-byte words, producing `0xADDRESS0000...0000`. Standard EIP-712 encoding via `abi.encode` stores addresses right-aligned as `0x0000...0000ADDRESS`, so the resulting struct hash will differ from any hash produced by standard EIP-712 libraries. Therefore, if the `Hub` chain constructs the message hash using standard EIP-712 encoding in the function `calculate_message_hash()`, the forwarded messages and signatures will fail to be verified on the Spoke chain due to the

inconsistent message construction logic in the function `_calculateMessageHash()`, resulting in DoS issues.

```
190   function _calculateMessageHash(
191       uint256 fromChainId,
192       uint256 toChainId,
193       address from,
194       address to,
195       uint256 nonce,
196       bytes calldata data
197   ) internal view returns (bytes32 messageHash) {
198       // Calculate data hash first using inline assembly
199       bytes32 dataHash;
200       assembly ("memory-safe") {
201           let dataLen := data.length
202           let dataPtr := data.offset
203           // Allocate memory for data
204           let memPtr := mload(0x40)
205           // Copy data to memory
206           calldatacopy(memPtr, dataPtr, dataLen)
207           // Hash the data
208           dataHash := keccak256(memPtr, dataLen)
209           // Update free memory pointer (round up to next 32-byte boundary)
210           mstore(0x40, add(memPtr, and(add(dataLen, 0x1f), not(0x1f))))
211       }
212
213       // Calculate struct hash using inline assembly for efficiency
214       // Layout: MESSAGE_TYPE_HASH (32) | fromChainId (32) | toChainId (32) | from (32) | to (32)
              | nonce (32) | dataHash (32) = 224 bytes
215       bytes32 messageTypeHash = MESSAGE_TYPE_HASH;
216       bytes32 structHash;
217       assembly ("memory-safe") {
218           let ptr := mload(0x40)
219           // Store MESSAGE_TYPE_HASH
220           mstore(ptr, messageTypeHash)
221           // Store fromChainId
222           mstore(add(ptr, 0x20), fromChainId)
223           // Store toChainId
224           mstore(add(ptr, 0x40), toChainId)
225           // Store from (address padded to 32 bytes)
226           mstore(add(ptr, 0x60), shl(96, from))
227           // Store to (address padded to 32 bytes)
228           mstore(add(ptr, 0x80), shl(96, to))
229           // Store nonce
230           mstore(add(ptr, 0xa0), nonce)
231           // Store dataHash
232           mstore(add(ptr, 0xc0), dataHash)
233           // Calculate hash (224 bytes = 0xe0)
234           structHash := keccak256(ptr, 0xe0)
235           // Update free memory pointer
236           mstore(0x40, add(ptr, 0xe0))
237       }
238       // Calculate EIP712 typed data hash
```

```
239         messageHash = _hashTypedDataV4(structHash);
240     }
```

Listing 2.5: MessageGateway.sol

Impact The inconsistent message hash construction logic results in DoS issues.

Suggestion Align the message hash construction logic between `Hub` and `Spoke` chains.

2.1.5 Potential DoS in the function `depositWithPermit()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `TokenBridge`, the function `depositWithPermit()` invokes function `IERC20Permit.permit()` and reverts unconditionally in the `catch` block when the permit call fails. Since a permit signature can only be consumed once, an attacker can front-run the victim's transaction by calling the function `permit()` directly, causing the function `depositWithPermit()` to always revert. As a result, a malicious actor can grief legitimate users by front-running their token permits.

```
363     function depositWithPermit(
364         address user,
365         address token,
366         uint256 amount,
367         uint256 deadline,
368         uint8 v,
369         bytes32 r,
370         bytes32 s
371     ) external whenNotPaused {
372         if (block.timestamp > deadline) {
373             revert PermitExpired();
374         }
375         try IERC20Permit(token).permit(user, address(this), amount, deadline, v, r, s) {}
376         catch {
377             revert PermitInvalid();
378         }
```

Listing 2.6: TokenBridge.sol

Impact Users may suffer potential DoS issues when invoking the function `depositWithPermit()` due to the front-running attack.

Suggestion Check the token allowance inside the `catch` block and proceed with the transfer if the allowance is already sufficient, rather than reverting unconditionally.

2.1.6 Inconsistent use of the bridge precompile address

Severity Medium

Status Fixed in `Version 2`

Introduced by Version 1

Description In the contract TokenBridge, the deposit and withdrawal progresses use two different variables to refer to the precompile address on the Hub chain. Specifically, the hardcoded constant CORE_TOKEN_BRIDGE_PRECOMPILE_ADDRESS (i.e., 0x100f) is used in the withdrawal progress (i.e., the functions finalizeWithdrawal() and invalidateWithdrawals()) and the coreTokenBridge is used in the deposit progress. However, the inconsistent use of the precompile address may lead to potential DoS issues. Specifically, the coreTokenBridge can be updated via the function setCoreTokenBridge(), while the hardcoded constant CORE_TOKEN_BRIDGE_PRECOMPILE_ADDRESS remains immutable. As a result, the inconsistent use of the precompile address may lead to DoS issues after the invocation of the function setCoreTokenBridge().

```
73    address internal constant CORE_TOKEN_BRIDGE_PRECOMPILE_ADDRESS =
74        address(0x0000000000000000000000000000000000000100f);
```

Listing 2.7: TokenBridge.sol

```
106   address public coreTokenBridge;
```

Listing 2.8: TokenBridge.sol

```
287       IMessageGateway(messageGateway).sendMessage(coreChainId,
              CORE_TOKEN_BRIDGE_PRECOMPILE_ADDRESS, payload);
```

Listing 2.9: TokenBridge.sol

```
424       IMessageGateway(messageGateway).sendMessage(coreChainId, destTokenBridge, payloadData);
```

Listing 2.10: TokenBridge.sol

```
640   function setCoreTokenBridge(address tokenBridgeAddress) external onlyRole(ADMIN_ROLE) {
641       if (tokenBridgeAddress == address(0)) {
642           revert InvalidCoreTokenBridge();
643       }
644       coreTokenBridge = tokenBridgeAddress;
645       emit CoreTokenBridgeUpdated(tokenBridgeAddress);
646   }
```

Listing 2.11: TokenBridge.sol

Impact The inconsistent use of the precompile address may lead to DoS issues after the invocation of the function setCoreTokenBridge().

Suggestion Unify the use of the precompile address in the deposit and withdrawal progresses.

2.1.7 Potential DoS due to the improper deposit accounting in the contract TokenBridge

Severity Low

Status Confirmed

Introduced by Version 1

Description In the contract TokenBridge, the function _enforceDepositRoute() checks the deposit cap (i.e., IERC20(token).balanceOf(address(this)) + amount > tokenConfig.depositCap) based on the contract TokenBridge's token balance (i.e., balanceOf()). However, this validation is potentially vulnerable to donation attacks, leading to potential DoS issues. Specifically, a malicious user can directly transfer tokens to the contract TokenBridge to consume the remaining deposit amount. As a result, users are temporarily unable to deposit due to the improper deposit accounting.

```
222   function _enforceDepositRoute(address token, uint64 tokenId, uint256 amount) internal view {
223       RuntimeTokenConfig memory tokenConfig = tokenConfigByTokenId[tokenId];
224       if (tokenConfig.version == 0) {
225           revert TokenMappingNotFound();
226       }
227       if (!tokenConfig.depositEnabled) revert DepositDisabled();
228       if (tokenConfig.custodyTokenAddress != address(0) && tokenConfig.custodyTokenAddress !=
              token) {
229           revert InvalidToken();
230       }
231       if (amount < tokenConfig.minimumDeposit) revert DepositBelowMinimum();
232       if (tokenConfig.depositCap != 0) {
233           uint256 currentBalance = IERC20(token).balanceOf(address(this));
234           if (currentBalance + amount > tokenConfig.depositCap) revert DepositCapExceeded();
235       }
```

Listing 2.12: TokenBridge.sol

Impact Users are temporarily unable to deposit due to the improper deposit accounting.

Suggestion Use state variables to record the accumulated deposit amount in the contract TokenBridge.

Feedback from the project The project stated that this is an intentional and acceptable design.

2.1.8 Potential delay of verifier update due to missing duplicate check

Severity Low

Status Fixed in Version 2

Introduced by Version 1

Description In the function handle_update_verifiers(), after collecting sufficient signatures, the contract queues a PendingVerifierUpdate with a variable execute_after timestamp set to now + challenge_period. Once this period elapses, the update can be finalized. However, the function does not check whether a pending update with the same variable epoch and variable params_hash already exists. Since the signed digest depends only on the verifier set parameters and does not include any freshness factor (e.g., timestamp or nonce), the same payload and signatures can be resubmitted multiple times.

Each resubmission overwrites the existing `PendingVerifierUpdate` and resets variable `execute_after` to a new future timestamp. This allows a relayer to repeatedly delay the finalization of an already queued update by continuously refreshing the challenge period.

```
230   pub(super) fn handle_update_verifiers<
231       Store: CoreStateStore + CoreStateReader + CoreAccountReader + CoreAccountWriter,
232   >(
233       input: PrecompileInput<'_>,
234       context: &PrecompileContext<'_, Store>,
235       output: &mut PrecompileOutput,
236   ) -> Result<(), PrecompileError> {
237       crate::check_role!(input, context, crate::permission_verifier::Role::Relayer);
238       let meta = read_verifier_meta(context.storage)?;
239       if meta.paused {
240           return Err(PrecompileError::ExecutionError(
241               "Bridge is paused; use emergencyUpdateVerifiers".to_string(),
242           ));
243       }
244
245       let call = IMessageGateway::updateVerifiersCall::abi_decode(input.data()).map_err(|e| {
246           tracing::info!(error = %e, "Failed to decode updateVerifiers call");
247           PrecompileError::InvalidInput(format!("Failed to decode: {}", e))
248       })?;
249       let (epoch, hot_verifiers, cold_verifiers, powers, signatures) =
250           (call.epoch, call.hotVerifiers, call.coldVerifiers, call.powers, call.signatures);
251
252       validate_verifier_set_addresses(&hot_verifiers, &cold_verifiers, &powers)?;
253
254       if meta.total_power.is_zero() {
255           if context.block_number != 0 {
256               return Err(PrecompileError::ExecutionError(
257                   "Bridge verifiers must be initialized in genesis when totalPower == 0".to_string
                          (),
258               ));
259           }
260           if epoch != 0 {
261               return Err(PrecompileError::ExecutionError(format!(
262                   "Genesis verifier initialization requires epoch == 0, got {}",
263                   epoch
264               )));
265           }
266           if !signatures.is_empty() {
267               return Err(PrecompileError::ExecutionError(
268                   "Genesis verifier initialization requires empty signatures".to_string(),
269               ));
270           }
271           let mut meta = meta;
272           apply_verifier_set(context.storage, &mut meta, &hot_verifiers, &cold_verifiers, &powers
                  )?;
273           meta.epoch = 0;
274           write_verifier_meta(context.storage, meta)?;
275           output.push_log(create_log(
276               MESSAGE_GATEWAY_PRECOMPILE_ADDRESS,
```

```
277             &IMessageGateway::VerifierSetUpdated {
278                 epoch: 0u64,
279                 hotVerifiers: hot_verifiers.clone(),
280                 coldVerifiers: cold_verifiers.clone(),
281                 powers,
282             },
283         ));
284         ensure_verifier_accounts(context, &hot_verifiers, &cold_verifiers)?;
285         output.bytes = Bytes::new();
286         output.gas_used = GAS_BRIDGE_UPDATE_VERIFIERS;
287         return Ok(());
288     }
289 
290     if epoch <= meta.epoch {
291         return Err(PrecompileError::ExecutionError(format!(
292             "epoch must be > currentVerifierEpoch ({}), got {}",
293             meta.epoch, epoch
294         )));
295     }
296 
297     let digest = update_verifiers_signed_digest(epoch, &hot_verifiers, &cold_verifiers, &powers
        )?;
298     let accumulated_power = verify_signatures_until_required_power_for_set(
299         context.storage,
300         digest,
301         &signatures,
302         VerifierSetKind::Hot,
303         meta.required_power,
304     )?;
305     if accumulated_power < meta.required_power {
306         return Err(PrecompileError::ExecutionError(
307             "Insufficient verifier power for update".to_string(),
308         ));
309     }
310 
311     if meta.challenge_period == 0 {
312         return Err(PrecompileError::ExecutionError(
313             "challenge_period is not configured".to_string(),
314         ));
315     }
316     let now_secs = bridge_block_timestamp_secs(context.block_timestamp);
317     let execute_after = now_secs
318         .checked_add(meta.challenge_period)
319         .ok_or_else(|| PrecompileError::ExecutionError("execute_after overflow".to_string()))?;
320     let params_hash = compute_verifier_params_hash(epoch, &hot_verifiers, &cold_verifiers, &
        powers);
321     write_pending_verifier_update(
322         context.storage,
323         PendingVerifierUpdate { epoch, execute_after, params_hash },
324     )?;
325     output.push_log(create_log(
326         MESSAGE_GATEWAY_PRECOMPILE_ADDRESS,
327         &IMessageGateway::VerifierSetQueued {
```

```
328                 epoch,
329                 paramsHash: params_hash,
330                 executeAfter: U256::from(execute_after),
331             },
332         ));
333         output.bytes = Bytes::new();
334         output.gas_used = GAS_BRIDGE_UPDATE_VERIFIERS;
335         Ok(())
336     }
```

Listing 2.13: message_gateway.rs

Impact A queued verifier set update can be indefinitely delayed through repeated resubmissions of the same payload.

Suggestion Add a check to ensure the incoming variable `epoch` and variable `params_hash` do not match the existing `PendingVerifierUpdate` before overwriting.

2.1.9 Unbounded signature input may lead to excessive gas consumption

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `verify_signatures_and_accumulate_power_for_set()`, the function iterates over all provided signatures and skips invalid entries (e.g., incorrect length, invalid recovery parameter, failed recovery, zero address, or duplicate signer) using continue. While skipping invalid signatures is acceptable by design, the function does not enforce any upper bound or consistency check on the signatures array, such as validating that the number of provided signatures reasonably matches the expected number of distinct signers required to reach the threshold.

As a result, a relayer can submit an excessively large signatures array containing many invalid or duplicate entries. Although these entries do not contribute to accumulated power, they are still processed individually, incurring unnecessary computation cost. This may lead to inflated gas usage and, in extreme cases, cause the transaction to exceed gas limits or degrade relayer efficiency.

```
449     fn verify_signatures_and_accumulate_power_for_set<Store: CoreStateReader>(
450         storage: &Store,
451         message_hash: B256,
452         signatures: &[alloy_primitives::Bytes],
453         verifier_set_kind: VerifierSetKind,
454         stop_at_power: Option<U256>,
455     ) -> Result<U256, PrecompileError> {
456         let mut accumulated_power = U256::ZERO;
457         let mut seen_signers = HashSet::new();
458         for signature in signatures {
459             if signature.len() != 65 {
460                 continue;
461             }
```

```
462             let r = B256::from_slice(&signature[0..32]);
463             let s = B256::from_slice(&signature[32..64]);
464             let v = signature[64];
465             let parity = match v {
466                 27 | 0 => false,
467                 28 | 1 => true,
468                 _ => continue,
469             };
470             let sig = Signature::from_scalars_and_parity(r, s, parity);
471             let signer = match sig.recover_address_from_prehash(&message_hash) {
472                 Ok(addr) => addr,
473                 Err(_) => continue,
474             };
475             if signer == Address::ZERO {
476                 tracing::info!(
477                     message_hash = %message_hash,
478                     "Signer is zero address",
479                 );
480                 continue;
481             }
482             if seen_signers.contains(&signer) {
483                 tracing::info!(
484                     message_hash = %message_hash,
485                     signer = %signer,
486                     "Signer already seen",
487                 );
488                 continue;
489             }
490             seen_signers.insert(signer);
491             let power = read_verifier_power_for_set(
492                 storage,
493                 signer.as_slice().try_into().map_err(|_| {
494                     PrecompileError::ExecutionError("Invalid signer address format".to_string())
495                 })?,
496                 verifier_set_kind,
497             )?;
498             if !power.is_zero() {
499                 accumulated_power = accumulated_power.checked_add(power).ok_or_else(|| {
500                     PrecompileError::ExecutionError("Accumulated power overflow".to_string())
501                 })?;
502             }
503             tracing::trace!(
504                 accumulated_power = %accumulated_power,
505                 signer = %signer,
506                 power = %power,
507                 "Accumulated power"
508             );
509             if let Some(stop_at_power) = stop_at_power
510                 && accumulated_power >= stop_at_power
511             {
512                 break;
513             }
514         }
```

```
515         Ok(accumulated_power)
516     }
```

Listing 2.14: common.rs

Impact A malicious signer can supply an excessively large signature array, causing unnecessary processing and increased gas consumption, which may lead to out-of-gas failures and potential denial of service.

Suggestion Introduce validation on the signatures input, such as enforcing a reasonable upper bound or requiring consistency between the number of provided signatures and the minimum required distinct signers.

2.1.10 Inconsistent failure handling in the function `executeMessage()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `TokenBridge`, the function `executeMessage()` performs inconsistent handling for different validation failures in the `APPLY_CUSTODY_CHAIN_CONFIG_SELECTOR` and `APPLY_CUSTODY_ROUTE_SELECTOR` branches. Specifically, if the check `version <= currentVersion` fails, the function `executeMessage()` silently returns and the function `MessageGateway.receiveMessage()` marks the message as processed (i.e., `processedMessages[messageHash] = true`). However, when other checks fail (e.g., `disputePeriod < MIN_CHALLENGE_PERIOD`, `verifierUpdatePeriod < disputePeriod`), the function `executeMessage()` reverts directly. As a result, this inconsistent failure handling may affect potential off-chain operations.

```
446         if (selector == APPLY_CUSTODY_CHAIN_CONFIG_SELECTOR) {
447             (
448                 uint256 disputePeriod,
449                 uint256 verifierUpdatePeriod,
450                 uint256 blockTimeMillis,
451                 uint256 lockerThreshold,
452                 uint64 version
453             ) = abi.decode(data[4:], (uint256, uint256, uint256, uint256, uint64));
454             if (version <= runtimeChainConfig.version) {
455                 return;
456             }
457             if (disputePeriod < MIN_CHALLENGE_PERIOD) {
458                 revert ChallengePeriodTooShort();
459             }
460             if (verifierUpdatePeriod < MIN_CHALLENGE_PERIOD) {
461                 revert VerifierUpdatePeriodTooShort();
462             }
463             if (verifierUpdatePeriod < disputePeriod) {
464                 revert VerifierUpdatePeriodInvariantViolated();
465             }
466             runtimeChainConfig = RuntimeChainConfig({
467                 disputePeriod: disputePeriod,
468                 blockTimeMillis: blockTimeMillis,
```

```
469             lockerThreshold: lockerThreshold,
470             version: version
471         });
472         _refreshLocalVotePauseState(lockerThreshold);
473         IMessageGateway(messageGateway).syncVerifierUpdatePeriod(verifierUpdatePeriod);
474         emit CustodyChainConfigUpdated(disputePeriod, blockTimeMillis, lockerThreshold, version
                );
475         return;
476     }
```

Listing 2.15: TokenBridge.sol

```
479     if (selector == APPLY_CUSTODY_ROUTE_SELECTOR) {
480         (
481             uint64 tokenId,
482             address custodyTokenAddress,
483             uint256 minimumDeposit,
484             uint256 depositCap,
485             bool depositEnabled,
486             bool withdrawEnabled,
487             uint64 version
488         ) = abi.decode(data[4:], (uint64, address, uint256, uint256, bool, bool, uint64));
489         if (version <= tokenConfigByTokenId[tokenId].version) {
490             return;
491         }
492         address prevCustody = tokenConfigByTokenId[tokenId].custodyTokenAddress;
493         if (prevCustody != address(0) && prevCustody != custodyTokenAddress) {
494             if (tokenIdByCustodyTokenAddress[prevCustody] == tokenId) {
495                 tokenIdByCustodyTokenAddress[prevCustody] = 0;
496             }
497         }
498         tokenConfigByTokenId[tokenId] = RuntimeTokenConfig({
499             custodyTokenAddress: custodyTokenAddress,
500             minimumDeposit: minimumDeposit,
501             depositCap: depositCap,
502             depositEnabled: depositEnabled,
503             withdrawEnabled: withdrawEnabled,
504             version: version
505         });
506         if (custodyTokenAddress != address(0)) {
507             tokenIdByCustodyTokenAddress[custodyTokenAddress] = tokenId;
508         }
509         emit CustodyRouteConfigUpdated(
510             tokenId, custodyTokenAddress, minimumDeposit, depositCap, depositEnabled,
                    withdrawEnabled, version
511         );
512         return;
513     }
```

Listing 2.16: TokenBridge.sol

```
266 function receiveMessage(
267     uint256 sourceChainId,
```

```
268         uint256 destinationChainId,
269         address source,
270         address destination,
271         uint256 nonce,
272         bytes calldata data,
273         bytes[] calldata signatures
274     ) external nonReentrant {
275         if (paused() && !_isPausedMessageAllowed(destination, data)) revert EnforcedPause();
276         if (destinationChainId != block.chainid) revert WrongTargetChain();
277         if (totalPower == 0) revert InsufficientPower();
278         if (destination != tokenBridge) revert InvalidDestination();
279
280         bytes32 messageHash = _calculateMessageHash(sourceChainId, destinationChainId, source,
                destination, nonce, data);
281         if (processedMessages[messageHash]) revert MessageAlreadyProcessed();
282
283         uint256 accumulatedPower = _accumulatePowerFromSet(messageHash, signatures, _hotVerifiers);
284         if (accumulatedPower * 3 <= totalPower * 2) revert InsufficientPower();
285
286         processedMessages[messageHash] = true;
287         IMessageHandler(destination).executeMessage(sourceChainId, source, data, messageHash);
288         emit MessageReceived(sourceChainId, destinationChainId, source, destination, nonce, data,
                messageHash);
289     }
```

Listing 2.17: MessageGateway.sol

Impact The inconsistent failure handling may affect potential off-chain operations.

Suggestion Revise the code accordingly.

2.2 Recommendation

2.2.1 Add checks in the function `validate_verifier_set_addresses()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description After collecting sufficient signatures, a user with the `Relayer` role can update the verifier set via function `handle_update_verifiers()`. The function `calidate_verifier_set_addresses()` ensures that all verifier addresses are non-zero, that hot and cold addresses differ for each verifier, and that all addresses are globally unique. However, there is no check for empty input arrays. If the caller mistakenly passes empty arrays for variables `hot_verifiers`, variables `cold_verifiers`, and variable `powers`, the function would execute without error rather than reverting, which is inappropriate.

```
106     pub(super) fn validate_verifier_set_addresses(
107         hot_verifiers: &[Address],
108         cold_verifiers: &[Address],
109         powers: &[U256],
110     ) -> Result<(), PrecompileError> {
111         if hot_verifiers.len() != cold_verifiers.len() || hot_verifiers.len() != powers.len() {
```

```
112             return Err(PrecompileError::InvalidInput(
113                 "hotVerifiers, coldVerifiers and powers must have the same length".to_string(),
114             ));
115         }
116
117         let mut seen = HashSet::with_capacity(hot_verifiers.len().saturating_mul(2));
118         for (hot, cold) in hot_verifiers.iter().zip(cold_verifiers.iter()) {
119             if *hot == Address::ZERO || *cold == Address::ZERO {
120                 return Err(PrecompileError::InvalidInput(
121                     "verifier addresses must be non-zero".to_string(),
122                 ));
123             }
124             if hot == cold {
125                 return Err(PrecompileError::InvalidInput(
126                     "hot and cold verifier address must differ for each verifier".to_string(),
127                 ));
128             }
129             if !seen.insert(*hot) || !seen.insert(*cold) {
130                 return Err(PrecompileError::InvalidInput(
131                     "verifier addresses must be globally unique across hot and cold sets".to_string
                        (),
132                 ));
133             }
134         }
135
136         Ok(())
137     }
```

Listing 2.18: common.rs

Suggestion Add a check to ensure the input arrays are non-empty.

2.2.2 Ensure consistent governance payload checks between Hub and Spoke

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Hub-side validation is weaker than the corresponding Spoke-side validation for some governance payloads. In the function `handle_update_verifiers()`, Hub does not enforce the Spoke-side requirement that both variables `hotVerifiers` and variables `coldVerifiers` be strictly ascending, so a verifier set may be accepted on `Hub` but rejected on `Spoke`. Similarly, functions `handle_set_finalizer_batch()` and `handle_set_locker_batch()` do not check that the address array and the enabled array have the same length, while the `Spoke` token bridge explicitly reverts on such inputs. `Hub` should enforce the same acceptance rules as `Spoke` before accepting or emitting these payloads.

```
732     pub(super) fn handle_set_finalizer_batch<
733         Store: CoreStateStore + CoreStateReader + CoreAccountWriter,
734     >(
735         input: PrecompileInput<'_>,
736         context: &PrecompileContext<'_, Store>,
737         output: &mut PrecompileOutput,
```

```
738   ) -> Result<(), PrecompileError> {
739       crate::check_role!(input, context, crate::permission_verifier::Role::BridgeAdmin);
740
741       let call = ITokenBridge::setFinalizerBatchCall::abi_decode(input.data())
742           .map_err(|e| PrecompileError::InvalidInput(format!("Failed to decode: {}", e)))?;
743       let custody_chain_id: u64 = call
744           .custodyChainId
745           .try_into()
746           .map_err(|_| PrecompileError::InvalidInput("custodyChainId must fit in u64".to_string()
                  ))?;
747
748       let selector = apply_finalizer_batch_selector();
749       let payload = (call.finalizerAddresses, call.enabled).abi_encode_params();
750       emit_custody_governance_message(context, output, custody_chain_id, selector, payload)?;
751
752       output.bytes = Bytes::new();
753       output.gas_used = GAS_BRIDGE_SET_FINALIZER_BATCH;
754       Ok(())
755   }
756
757   /// Handle set locker batch call.
758   pub(super) fn handle_set_locker_batch<
759       Store: CoreStateStore + CoreStateReader + CoreAccountWriter,
760   >(
761       input: PrecompileInput<'_>,
762       context: &PrecompileContext<'_, Store>,
763       output: &mut PrecompileOutput,
764   ) -> Result<(), PrecompileError> {
765       crate::check_role!(input, context, crate::permission_verifier::Role::BridgeAdmin);
766
767       let call = ITokenBridge::setLockerBatchCall::abi_decode(input.data())
768           .map_err(|e| PrecompileError::InvalidInput(format!("Failed to decode: {}", e)))?;
769       let custody_chain_id: u64 = call
770           .custodyChainId
771           .try_into()
772           .map_err(|_| PrecompileError::InvalidInput("custodyChainId must fit in u64".to_string()
                  ))?;
773
774       let selector = apply_locker_batch_selector();
775       let payload = (call.lockerAddresses, call.enabled).abi_encode_params();
776       emit_custody_governance_message(context, output, custody_chain_id, selector, payload)?;
777
778       output.bytes = Bytes::new();
779       output.gas_used = GAS_BRIDGE_SET_LOCKER_BATCH;
780       Ok(())
781   }
```

Listing 2.19: token_bridge.rs

```
230   pub(super) fn handle_update_verifiers<
231       Store: CoreStateStore + CoreStateReader + CoreAccountReader + CoreAccountWriter,
232   >(
233       input: PrecompileInput<'_>,
```

```
234         context: &PrecompileContext<'_, Store>,
235         output: &mut PrecompileOutput,
236     ) -> Result<(), PrecompileError> {
237         crate::check_role!(input, context, crate::permission_verifier::Role::Relayer);
238         let meta = read_verifier_meta(context.storage)?;
239         if meta.paused {
240             return Err(PrecompileError::ExecutionError(
241                 "Bridge is paused; use emergencyUpdateVerifiers".to_string(),
242             ));
243         }
244
245         let call = IMessageGateway::updateVerifiersCall::abi_decode(input.data()).map_err(|e| {
246             tracing::info!(error = %e, "Failed to decode updateVerifiers call");
247             PrecompileError::InvalidInput(format!("Failed to decode: {}", e))
248         })?;
249         let (epoch, hot_verifiers, cold_verifiers, powers, signatures) =
250             (call.epoch, call.hotVerifiers, call.coldVerifiers, call.powers, call.signatures);
251
252         validate_verifier_set_addresses(&hot_verifiers, &cold_verifiers, &powers)?;
253
254         if meta.total_power.is_zero() {
255             if context.block_number != 0 {
256                 return Err(PrecompileError::ExecutionError(
257                     "Bridge verifiers must be initialized in genesis when totalPower == 0".to_string
                        (),
258                 ));
259             }
260             if epoch != 0 {
261                 return Err(PrecompileError::ExecutionError(format!(
262                     "Genesis verifier initialization requires epoch == 0, got {}",
263                     epoch
264                 )));
265             }
266             if !signatures.is_empty() {
267                 return Err(PrecompileError::ExecutionError(
268                     "Genesis verifier initialization requires empty signatures".to_string(),
269                 ));
270             }
271             let mut meta = meta;
272             apply_verifier_set(context.storage, &mut meta, &hot_verifiers, &cold_verifiers, &powers
                )?;
273             meta.epoch = 0;
274             write_verifier_meta(context.storage, meta)?;
275             output.push_log(create_log(
276                 MESSAGE_GATEWAY_PRECOMPILE_ADDRESS,
277                 &IMessageGateway::VerifierSetUpdated {
278                     epoch: 0u64,
279                     hotVerifiers: hot_verifiers.clone(),
280                     coldVerifiers: cold_verifiers.clone(),
281                     powers,
282                 },
283             ));
284             ensure_verifier_accounts(context, &hot_verifiers, &cold_verifiers)?;
```

```
285             output.bytes = Bytes::new();
286             output.gas_used = GAS_BRIDGE_UPDATE_VERIFIERS;
287             return Ok(());
288         }
289
290         if epoch <= meta.epoch {
291             return Err(PrecompileError::ExecutionError(format!(
292                 "epoch must be > currentVerifierEpoch ({}), got {}",
293                 meta.epoch, epoch
294             )));
295         }
296
297         let digest = update_verifiers_signed_digest(epoch, &hot_verifiers, &cold_verifiers, &powers
                )?;
298         let accumulated_power = verify_signatures_until_required_power_for_set(
299             context.storage,
300             digest,
301             &signatures,
302             VerifierSetKind::Hot,
303             meta.required_power,
304         )?;
305         if accumulated_power < meta.required_power {
306             return Err(PrecompileError::ExecutionError(
307                 "Insufficient verifier power for update".to_string(),
308             ));
309         }
310
311         if meta.challenge_period == 0 {
312             return Err(PrecompileError::ExecutionError(
313                 "challenge_period is not configured".to_string(),
314             ));
315         }
316         let now_secs = bridge_block_timestamp_secs(context.block_timestamp);
317         let execute_after = now_secs
318             .checked_add(meta.challenge_period)
319             .ok_or_else(|| PrecompileError::ExecutionError("execute_after overflow".to_string()))?;
320         let params_hash = compute_verifier_params_hash(epoch, &hot_verifiers, &cold_verifiers, &
                powers);
321         write_pending_verifier_update(
322             context.storage,
323             PendingVerifierUpdate { epoch, execute_after, params_hash },
324         )?;
325         output.push_log(create_log(
326             MESSAGE_GATEWAY_PRECOMPILE_ADDRESS,
327             &IMessageGateway::VerifierSetQueued {
328                 epoch,
329                 paramsHash: params_hash,
330                 executeAfter: U256::from(execute_after),
331             },
332         ));
333         output.bytes = Bytes::new();
334         output.gas_used = GAS_BRIDGE_UPDATE_VERIFIERS;
335         Ok(())
```

```
336     }
```

Listing 2.20: message_gateway.rs

Suggestion Revise the logic to ensure the checks are aligned on both sides.

2.2.3 Add destination checks in the function `handle_receive_message()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `handle_receive_message()`, the destination address is validated twice. Although execution logic is gated by `destination == TOKEN_BRIDGE_PRECOMPILE_ADDRESS`, the function does not return early when this condition is not met. Instead, it proceeds to compute the message hash, check replay status, and perform signature verification before ultimately rejecting the request.

As a result, invalid messages with incorrect destination still trigger unnecessary computation, including full signature verification, which has no impact on the final execution outcome but still incurs unnecessary computational cost.

```
47    pub(super) fn handle_receive_message<
48        Store: CoreStateStore + CoreStateReader + CoreAccountReader + CoreAccountWriter + 'static,
49    >(
50        input: PrecompileInput<'_>,
51        context: &PrecompileContext<'_, Store>,
52        output: &mut PrecompileOutput,
53    ) -> Result<(), PrecompileError> {
54        crate::check_role!(input, context, crate::permission_verifier::Role::Relayer);
55        let meta = read_verifier_meta(context.storage)?;
56        if meta.paused {
57            return Err(PrecompileError::ExecutionError("Bridge is paused".to_string()));
58        }
59
60        let call = IMessageGateway::receiveMessageCall::abi_decode(input.data()).map_err(|e| {
61            error!(
62                target: "precompile::bridge::message_gateway",
63                error = %e,
64                "Failed to decode receiveMessage call"
65            );
66            PrecompileError::InvalidInput(format!("Failed to decode receiveMessage: {}", e))
67        })?;
68        let (
69            destination_chain_id_raw,
70            source_chain_id_raw,
71            source,
72            destination,
73            nonce,
74            data,
75            signatures,
76        ) = (
77            call.destinationChainId,
78            call.sourceChainId,
```

```
79              call.source,
80              call.destination,
81              call.nonce,
82              call.data,
83              call.signatures,
84          );
85          if signatures.is_empty() {
86              error!(
87                  target: "precompile::bridge::message_gateway",
88                  "signatures cannot be empty in receiveMessage"
89              );
90              return Err(PrecompileError::ExecutionError("signatures cannot be empty".to_string()));
91          }
92
93          let destination_chain_id: u64 = destination_chain_id_raw
94              .try_into()
95              .map_err(|_| PrecompileError::ExecutionError("toChainId must fit in u64".to_string()))
                    ?;
96          if destination_chain_id != context.chain_id {
97              error!(
98                  target: "precompile::bridge::message_gateway",
99                  destination_chain_id = destination_chain_id,
100                 chain_id = context.chain_id,
101                 "toChainId mismatch in receiveMessage"
102             );
103             return Err(PrecompileError::ExecutionError(
104                 "toChainId must equal this chain's chain_id".to_string(),
105             ));
106         }
107
108         let source_chain_id: u64 = source_chain_id_raw
109             .try_into()
110             .map_err(|_| PrecompileError::ExecutionError("fromChainId must fit in u64".to_string())
                    )?;
111         if destination == TOKEN_BRIDGE_PRECOMPILE_ADDRESS {
112             super::token_bridge::validate_source_chain_token_bridge(
113                 context.storage,
114                 source_chain_id,
115                 source,
116             )?;
117         }
118         let message_hash = calculate_message_hash(
119             data.as_ref(),
120             source,
121             destination,
122             destination_chain_id,
123             source_chain_id,
124             // Nonce is part of the signed message hash and replay protection key.
125             // We intentionally do not require continuity here so relayers can submit
126             // independent messages out of order.
127             nonce,
128         );
129
```

```
130         if is_message_processed(context.storage, message_hash)? {
131             warn!(
132                 target: "precompile::bridge::message_gateway",
133                 message_hash = %message_hash,
134                 "Message already processed (replay attack detected)"
135             );
136             return Err(PrecompileError::ExecutionError(
137                 "Message already processed (replay)".to_string(),
138             ));
139         }
140 
141         let total_power = meta.total_power;
142         if total_power.is_zero() {
143             error!(
144                 target: "precompile::bridge::message_gateway",
145                 signature_count = signatures.len(),
146                 "Cannot relay: no verifier set (totalPower == 0). Please initialize bridge
                        verifiers in genesis."
147             );
148             return Err(PrecompileError::ExecutionError(
149                 "Cannot relay: no verifier set (totalPower == 0). Please initialize bridge
                        verifiers in genesis.".to_string(),
150             ));
151         }
152         let required_power = meta.required_power;
153         let accumulated_power = verify_signatures_until_required_power_for_set(
154             context.storage,
155             message_hash,
156             &signatures,
157             VerifierSetKind::Hot,
158             required_power,
159         )?;
160         if accumulated_power < required_power {
161             warn!(
162                 target: "precompile::bridge::message_gateway",
163                 accumulated_power = %accumulated_power,
164                 required_power = %required_power,
165                 total_power = %total_power,
166                 signature_count = signatures.len(),
167                 "Insufficient verifier power for relay message"
168             );
169             return Err(PrecompileError::ExecutionError("Insufficient verifier power".to_string()));
170         }
171 
172         if destination == TOKEN_BRIDGE_PRECOMPILE_ADDRESS {
173             super::token_bridge::ensure_supported_execute_message_selector(data.as_ref())?;
174             super::token_bridge::handle_execute_message(
175                 context,
176                 output,
177                 source_chain_id,
178                 source,
179                 data.as_ref(),
180             )?;
```

```
181             mark_message_processed(context.storage, message_hash)?;
182             output.bytes = Bytes::new();
183             output.gas_used = GAS_BRIDGE_RECEIVE_MESSAGE;
184             return Ok(());
185         }
186
187         error!(
188             target: "precompile::bridge::message_gateway",
189             to = %destination,
190             expected = %TOKEN_BRIDGE_PRECOMPILE_ADDRESS,
191             "to must equal TOKEN_BRIDGE_PRECOMPILE_ADDRESS"
192         );
193         Err(PrecompileError::ExecutionError(
194             "to must equal TOKEN_BRIDGE_PRECOMPILE_ADDRESS".to_string(),
195         ))
196     }
```

Listing 2.21: message_gateway.rs

Suggestion Add an early validation on destination and return immediately if it is not equal to `TOKEN_BRIDGE_PRECOMPILE_ADDRESS`, to avoid unnecessary processing and gas consumption.

2.2.4 Add overflow protection when casting `tx_index`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In functions `handle_withdraw()` and `handle_finalize_withdraw_result_message()`, the variable `context.tx_index` (of type `usize`) is cast to `u16` using `as u16` in multiple places when constructing `CoordinatorMessageHeader` and `TradingCmdHeader`. The `as` cast in `Rust` silently truncates values that exceed the target type's range, which would produce an incorrect variable `block_tx_index` or variable `sequence` value without any error. Using a silent truncation cast in a financial protocol is inconsistent with the defensive coding style applied elsewhere in the codebase. For example, variable `chain ID` conversions consistently use `try_into().map_err(...)` for safe narrowing. Following the same pattern here would ensure that any unexpected truncation is caught explicitly rather than silently corrupting downstream message fields.

```
279     let withdraw_done_message = CoordinatorMessage {
280         header: CoordinatorMessageHeader {
281             block_tx_index: context.tx_index as u16,
282             block_height: context.block_number,
283             message_kind: CoordinatorMessageKind::BalanceUpdateRequest,
284             timestamp,
285             wallet_id: Some(wallet_id),
286             coordinator_internal_index: 0,
287             last_of_tx_index: false,
288         },
289         payload: CoordinatorPayload::TradingCmd(Box::new(TradingCmd {
290             header: TradingCmdHeader {
291                 block_tx_index: context.tx_index as u16,
292                 sequence: context.tx_index as u16,
```

```
293             timestamp,
294             symbol_id: None,
295             wallet_id: Some(wallet_id),
296             trigger_price: None,
297             trigger_entry_sequence: None,
298         },
299         cmd_type: TradingCmdType::WithdrawDone(WithdrawDoneBody {
300             wallet_id,
301             order_id: from_sol_order_id(decoded.withdrawal_id),
302             token_id: pending.token_id,
303             amount: amount_decimal,
304             chain_id: from_chain_id,
305             result: decoded.result_code == WITHDRAW_RESULT_SUCCEEDED,
306         }),
307     })),
308 };
```

Listing 2.22: token_bridge.rs

```
430 let withdraw_message = CoordinatorMessage {
431     header: CoordinatorMessageHeader {
432         block_tx_index: context.tx_index as u16,
433         block_height: context.block_number,
434         message_kind: CoordinatorMessageKind::BalanceUpdateRequest,
435         timestamp,
436         wallet_id: Some(wallet_id),
437         coordinator_internal_index: 0,
438         last_of_tx_index: false,
439     },
440     payload: CoordinatorPayload::TradingCmd(Box::new(TradingCmd {
441         header: TradingCmdHeader {
442             block_tx_index: context.tx_index as u16,
443             sequence: context.tx_index as u16,
444             timestamp,
445             symbol_id: None,
446             wallet_id: Some(wallet_id),
447             trigger_price: None,
448             trigger_entry_sequence: None,
449         },
450         cmd_type: TradingCmdType::Withdraw(DepositWithdrawBody {
451             wallet_id,
452             master_wallet_id: wallet_id,
453             token_id,
454             amount: amount_decimal,
455             chain_id: call.toChainId.try_into().map_err(|_| {
456                 PrecompileError::InvalidInput("toChainId must fit in u64".to_string())
457             })?,
458         }),
459     })),
```

Listing 2.23: token_bridge.rs

Suggestion Use checked conversions instead of as for narrowing casts to prevent silent truncation.

2.2.5 Add chain ID checks for `source_chain_id` in the function `handle_receive_message()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `handle_receive_message()`, the code validates that `destination_chain_id == context.chain_id` to ensure the message is intended for this chain, but does not verify that `source_chain_id != context.chain_id`. Since all legitimate cross-chain messages originate from external `Spoke` chains, a message where the source and destination chain are the same is semantically invalid in the `Hub-and-Spoke` architecture. If the `Hub`'s own `chain_id` were ever registered via the function `setChainTokenBridge()` (a privileged operation restricted to the `BridgeAdmin` role), a self-referential message could pass the existing check in the function `validate_source_chain_token_bridge()`, potentially causing unintended state mutations such as a self-deposit crediting assets without any actual custody lock on an external chain. While such misconfiguration is unlikely under normal operations, adding an explicit guard against self-referential messages provides defense-in-depth consistent with the existing destination chain check.

```
47  pub(super) fn handle_receive_message<
48      Store: CoreStateStore + CoreStateReader + CoreAccountReader + CoreAccountWriter + 'static,
49  >(
50      input: PrecompileInput<'_>,
51      context: &PrecompileContext<'_, Store>,
52      output: &mut PrecompileOutput,
53  ) -> Result<(), PrecompileError> {
54      crate::check_role!(input, context, crate::permission_verifier::Role::Relayer);
55      let meta = read_verifier_meta(context.storage)?;
56      if meta.paused {
57          return Err(PrecompileError::ExecutionError("Bridge is paused".to_string()));
58      }
59
60      let call = IMessageGateway::receiveMessageCall::abi_decode(input.data()).map_err(|e| {
61          error!(
62              target: "precompile::bridge::message_gateway",
63              error = %e,
64              "Failed to decode receiveMessage call"
65          );
66          PrecompileError::InvalidInput(format!("Failed to decode receiveMessage: {}", e))
67      })?;
68      let (
69          destination_chain_id_raw,
70          source_chain_id_raw,
71          source,
72          destination,
73          nonce,
74          data,
75          signatures,
76      ) = (
77          call.destinationChainId,
78          call.sourceChainId,
```

```
79              call.source,
80              call.destination,
81              call.nonce,
82              call.data,
83              call.signatures,
84          );
85          if signatures.is_empty() {
86              error!(
87                  target: "precompile::bridge::message_gateway",
88                  "signatures cannot be empty in receiveMessage"
89              );
90              return Err(PrecompileError::ExecutionError("signatures cannot be empty".to_string()));
91          }
92
93          let destination_chain_id: u64 = destination_chain_id_raw
94              .try_into()
95              .map_err(|_| PrecompileError::ExecutionError("toChainId must fit in u64".to_string()))
                    ?;
96          if destination_chain_id != context.chain_id {
97              error!(
98                  target: "precompile::bridge::message_gateway",
99                  destination_chain_id = destination_chain_id,
100                 chain_id = context.chain_id,
101                 "toChainId mismatch in receiveMessage"
102             );
103             return Err(PrecompileError::ExecutionError(
104                 "toChainId must equal this chain's chain_id".to_string(),
105             ));
106         }
107
108         let source_chain_id: u64 = source_chain_id_raw
109             .try_into()
110             .map_err(|_| PrecompileError::ExecutionError("fromChainId must fit in u64".to_string())
                    )?;
111         if destination == TOKEN_BRIDGE_PRECOMPILE_ADDRESS {
112             super::token_bridge::validate_source_chain_token_bridge(
113                 context.storage,
114                 source_chain_id,
115                 source,
116             )?;
117         }
118         let message_hash = calculate_message_hash(
119             data.as_ref(),
120             source,
121             destination,
122             destination_chain_id,
123             source_chain_id,
124             // Nonce is part of the signed message hash and replay protection key.
125             // We intentionally do not require continuity here so relayers can submit
126             // independent messages out of order.
127             nonce,
128         );
129
```

```
130         if is_message_processed(context.storage, message_hash)? {
131             warn!(
132                 target: "precompile::bridge::message_gateway",
133                 message_hash = %message_hash,
134                 "Message already processed (replay attack detected)"
135             );
136             return Err(PrecompileError::ExecutionError(
137                 "Message already processed (replay)".to_string(),
138             ));
139         }
140 
141         let total_power = meta.total_power;
142         if total_power.is_zero() {
143             error!(
144                 target: "precompile::bridge::message_gateway",
145                 signature_count = signatures.len(),
146                 "Cannot relay: no verifier set (totalPower == 0). Please initialize bridge
                        verifiers in genesis."
147             );
148             return Err(PrecompileError::ExecutionError(
149                 "Cannot relay: no verifier set (totalPower == 0). Please initialize bridge
                        verifiers in genesis.".to_string(),
150             ));
151         }
152         let required_power = meta.required_power;
153         let accumulated_power = verify_signatures_until_required_power_for_set(
154             context.storage,
155             message_hash,
156             &signatures,
157             VerifierSetKind::Hot,
158             required_power,
159         )?;
160         if accumulated_power < required_power {
161             warn!(
162                 target: "precompile::bridge::message_gateway",
163                 accumulated_power = %accumulated_power,
164                 required_power = %required_power,
165                 total_power = %total_power,
166                 signature_count = signatures.len(),
167                 "Insufficient verifier power for relay message"
168             );
169             return Err(PrecompileError::ExecutionError("Insufficient verifier power".to_string()));
170         }
171 
172         if destination == TOKEN_BRIDGE_PRECOMPILE_ADDRESS {
173             super::token_bridge::ensure_supported_execute_message_selector(data.as_ref())?;
174             super::token_bridge::handle_execute_message(
175                 context,
176                 output,
177                 source_chain_id,
178                 source,
179                 data.as_ref(),
180             )?;
```

```
            mark_message_processed(context.storage, message_hash)?;
            output.bytes = Bytes::new();
            output.gas_used = GAS_BRIDGE_RECEIVE_MESSAGE;
            return Ok(());
        }

        error!(
            target: "precompile::bridge::message_gateway",
            to = %destination,
            expected = %TOKEN_BRIDGE_PRECOMPILE_ADDRESS,
            "to must equal TOKEN_BRIDGE_PRECOMPILE_ADDRESS"
        );
        Err(PrecompileError::ExecutionError(
            "to must equal TOKEN_BRIDGE_PRECOMPILE_ADDRESS".to_string(),
        ))
    }
```

Listing 2.24: message_gateway.rs

Suggestion Add an explicit check to reject messages where `source_chain_id == context.chain_id` to prevent self-referential processing and provide defense-in-depth against potential misconfiguration.

2.2.6 Add non-zero checks for the variable `tokenId` in the route configuration

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `executeMessage()`, the `APPLY_CUSTODY_ROUTE_SELECTOR` branch does not validate that variable `tokenId` is non-zero. Neither the `Hub` chain nor the `Spoke` chain enforces this check. As a result, the `Hub` chain could configure a token with zero `tokenId`, which would be accepted on the `Spoke` chain, but any subsequent deposit or withdrawal operations involving that token would fail because those code paths require a non-zero `tokenId`. Therefore, it is recommended to add such non-zero checks for the variable `tokenId` to prevent potential misconfigurations.

```
        if (selector == APPLY_CUSTODY_ROUTE_SELECTOR) {
            (
                uint64 tokenId,
                address custodyTokenAddress,
                uint256 minimumDeposit,
                uint256 depositCap,
                bool depositEnabled,
                bool withdrawEnabled,
                uint64 version
            ) = abi.decode(data[4:], (uint64, address, uint256, uint256, bool, bool, uint64));
            if (version <= tokenConfigByTokenId[tokenId].version) {
                return;
            }
            address prevCustody = tokenConfigByTokenId[tokenId].custodyTokenAddress;
            if (prevCustody != address(0) && prevCustody != custodyTokenAddress) {
```

```
494             if (tokenIdByCustodyTokenAddress[prevCustody] == tokenId) {
495                 tokenIdByCustodyTokenAddress[prevCustody] = 0;
496             }
497         }
498         tokenConfigByTokenId[tokenId] = RuntimeTokenConfig({
499             custodyTokenAddress: custodyTokenAddress,
500             minimumDeposit: minimumDeposit,
501             depositCap: depositCap,
502             depositEnabled: depositEnabled,
503             withdrawEnabled: withdrawEnabled,
504             version: version
505         });
506         if (custodyTokenAddress != address(0)) {
507             tokenIdByCustodyTokenAddress[custodyTokenAddress] = tokenId;
508         }
509         emit CustodyRouteConfigUpdated(
510             tokenId, custodyTokenAddress, minimumDeposit, depositCap, depositEnabled,
                    withdrawEnabled, version
511         );
512         return;
513     }
```

Listing 2.25: TokenBridge.sol

Suggestion Add non-zero checks accordingly.

2.2.7 Add minimum value checks in the function `setChallengerPeriod()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `MessageGateway`, the function `setChallengerPeriod()` allows `ADMIN_ROLE` to set `challengePeriod` to any value including zero without enforcing a minimum value check. It is recommended to add the minimum value (i.e., `MIN_CHALLENGE_PERIOD`) check in the function `setChallengerPeriod()`.

```
543 function setChallengerPeriod(uint256 challengePeriod_) external onlyRole(ADMIN_ROLE) {
544     challengePeriod = challengePeriod_;
545 }
```

Listing 2.26: MessageGateway.sol

Suggestion Add a minimum value check in the function `setChallengerPeriod()`.

2.2.8 Implement emergency withdrawal in `TokenBridge` to prevent accidental transfers

Status Confirmed

Introduced by `Version 1`

Description In the contract `TokenBridge`, there is a lack of an emergency withdrawal function to withdraw accidental transfers. It is recommended to add an emergency withdrawal function in the contract `TokenBridge` to rescue accidental token transfers.

```
222   function _enforceDepositRoute(address token, uint64 tokenId, uint256 amount) internal view {
223       RuntimeTokenConfig memory tokenConfig = tokenConfigByTokenId[tokenId];
224       if (tokenConfig.version == 0) {
225           revert TokenMappingNotFound();
226       }
227       if (!tokenConfig.depositEnabled) revert DepositDisabled();
228       if (tokenConfig.custodyTokenAddress != address(0) && tokenConfig.custodyTokenAddress !=
              token) {
229           revert InvalidToken();
230       }
231       if (amount < tokenConfig.minimumDeposit) revert DepositBelowMinimum();
232       if (tokenConfig.depositCap != 0) {
233           uint256 currentBalance = IERC20(token).balanceOf(address(this));
234           if (currentBalance + amount > tokenConfig.depositCap) revert DepositCapExceeded();
235       }
236   }
```

Listing 2.27: TokenBridge.sol

Suggestion Implement an emergency withdrawal function in the contract `TokenBridge`.

2.2.9 Revise the custom error `InvalidSignature`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `MessageGateway`, the custom error `InvalidSignature` is used for the validity check (i.e., `epoch <= lastAppliedVerifierEpoch`) of the input epoch. It is recommended to revise the misleading error name for better code readability.

```
468       if (epoch <= lastAppliedVerifierEpoch) revert InvalidSignature();
```

Listing 2.28: MessageGateway.sol

```
432       if (epoch <= lastAppliedVerifierEpoch) revert InvalidSignature();
```

Listing 2.29: MessageGateway.sol

```
371       if (epoch <= lastAppliedVerifierEpoch) revert InvalidSignature();
```

Listing 2.30: MessageGateway.sol

Suggestion Revise the error name.

2.2.10 Remove redundant sorting in function `handle_list_verifiers()`

Status Fixed in `Version 3`

Introduced by `Version 2`

Description In the function `handle_list_verifiers()`, both `hotVerifiers` and `coldVerifiers` are sorted before being returned. However, the function `validate_verifier_set_addresses()` already enforces that both verifier sets are strictly stored in ascending order at write time. This

guarantees that verifier data in storage is always ordered. As a result, the additional sorting in function `handle_list_verifiers()` operates on already ordered data and is unnecessary for correctness.

```
805   pub(super) fn handle_list_verifiers<Store: CoreStateStore + CoreStateReader>(
806       _input: PrecompileInput<'_>,
807       context: &PrecompileContext<'_, Store>,
808       output: &mut PrecompileOutput,
809   ) -> Result<(), PrecompileError> {
810       let mut hot_keys = collect_hot_verifier_keys(context.storage)?;
811       hot_keys.sort_unstable();
812       let hot_verifiers: Vec<Address> = hot_keys.into_iter().map(Address::from).collect();
813
814       let mut cold_keys = collect_cold_verifier_keys(context.storage)?;
815       cold_keys.sort_unstable();
816       let cold_verifiers: Vec<Address> = cold_keys.into_iter().map(Address::from).collect();
817
818       let return_data = IMessageGateway::listVerifiersCall::abi_encode_returns(
819           &IMessageGateway::listVerifiersReturn {
820               hotVerifiers: hot_verifiers,
821               coldVerifiers: cold_verifiers,
822           },
823       );
824       output.bytes = Bytes::from(return_data);
825       output.gas_used = GAS_BRIDGE_LIST_VERIFIERS;
826       Ok(())
827   }
```

Listing 2.31: message_gateway.rs

```
138   if idx + 1 < hot_verifiers.len()
139       && (hot.as_slice() >= hot_verifiers[idx + 1].as_slice()
140           || cold.as_slice() >= cold_verifiers[idx + 1].as_slice())
141   {
142       return Err(PrecompileError::InvalidInput(
143           "hotVerifiers and coldVerifiers must each be strictly ascending".to_string(),
144       ));
145   }
```

Listing 2.32: common.rs

Suggestion Remove the redundant sorting in function `handle_list_verifiers()`.

2.2.11 Remove redundant code

Status Fixed in `Version 3`

Introduced by `Version 2`

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

```
271   function _enforceWithdrawRoute(address tokenAddress) internal view {
```

```
272         uint64 tokenId = tokenIdByCustodyTokenAddress[tokenAddress];
273         if (tokenId == 0) {
274             revert TokenMappingNotFound();
275         }
276         RuntimeTokenConfig memory tokenConfig = tokenConfigByTokenId[tokenId];
277         if (tokenConfig.version == 0) {
278             revert TokenMappingNotFound();
279         }
280         if (!tokenConfig.withdrawEnabled) revert WithdrawDisabled();
281         if (tokenConfig.custodyTokenAddress != address(0) && tokenConfig.custodyTokenAddress !=
                tokenAddress) {
282             revert InvalidToken();
283         }
284     }
```

Listing 2.33: TokenBridge.sol

```
310     function _loadWithdrawRoute(address tokenAddress) internal view returns (RuntimeTokenConfig
            memory tokenConfig) {
311         uint64 tokenId = tokenIdByCustodyTokenAddress[tokenAddress];
312         if (tokenId == 0) {
313             revert TokenMappingNotFound();
314         }
315         tokenConfig = tokenConfigByTokenId[tokenId];
316         if (tokenConfig.version == 0) {
317             revert TokenMappingNotFound();
318         }
319         if (tokenConfig.custodyTokenAddress != address(0) && tokenConfig.custodyTokenAddress !=
                tokenAddress) {
320             revert InvalidToken();
321         }
322     }
```

Listing 2.34: TokenBridge.sol

```
148     /// @dev Snapshot tokenId for newly created pending withdraws. Legacy pending records may
            leave this as zero.
149     mapping(uint128 => uint64) internal pendingWithdrawTokenIds;
```

Listing 2.35: TokenBridge.sol

Suggestion Remove the redundant code.

2.3 Note

2.3.1 Ensure the same verifier set is shared across all Spoke chains

Introduced by `Version 1`

Description The function `updateVerifiers()` in the contract `MessageGateway` computes the signed digest via function `_updateVerifiersMessageHash()`. This digest does not include `block.chainid` and `address(this)`, meaning the same set of verifier signatures is valid on other `Spoke` chains

with the same verifier set. The project must ensure the same verifier set is used across all Spoke chains.

2.3.2 Proper handling for USDT on Tron

Introduced by Version 1

Description In the contract TokenBridge, the functions safeTransferFrom() and safeTransfer() are used to transfer users' tokens. However, if the project supports USDT on Tron network, these functions may lead to DoS issues as USDT on Tron network is incompatible with the functions safeTransfer() and safeTransferFrom(). The project stated that only USDT on Arbitrum is supported in the current phase, and that USDT on Tron will be handled properly when added later.

```
415
416       // Transfer tokens from depositor to this contract (lock tokens; use source-chain raw
              amount)
417       IERC20(token).safeTransferFrom(depositor, address(this), amount);
```

Listing 2.36: TokenBridge.sol

```
617
618       // Interactions
619       IERC20(pending.tokenAddress).safeTransfer(pending.receiver, amountDestRaw);
```

Listing 2.37: TokenBridge.sol

2.3.3 Proper aggregation logic for relayer signatures

Introduced by Version 1

Description In the contract MessageGateway, the function _accumulatePowerFromSet() uses a two-pointer merge algorithm that assumes the signatures[] array is ordered by recovered signer address in ascending order. If a relayer submits signatures in an incorrect order, the algorithm will advance the verifier pointer past all verifiers, causing all legitimate signatures to be skipped and the transaction to revert. While the message remains unprocessed and can be resubmitted, this allows any relayer to continuously block message delivery. The project must ensure relayers correctly sequence collected messages and signatures.

```
632   function _accumulatePowerFromSet(
633       bytes32 digestOrMessageHash,
634       bytes[] calldata signatures,
635       address[] memory verifierSet
636   ) internal view returns (uint256 accumulatedPower) {
637       uint256 sigIdx = 0;
638       uint256 verIdx = 0;
639       uint256 nSig = signatures.length;
640       uint256 nVer = verifierSet.length;
641
642       while (verIdx < nVer && sigIdx < nSig) {
643           address verifier = verifierSet[verIdx];
644           bytes calldata signature = signatures[sigIdx];
```

```
645             if (signature.length != 65) {
646                 unchecked {
647                     sigIdx++;
648                 }
649                 continue;
650             }
651
652             address signer = ECDSA.recoverCalldata(digestOrMessageHash, signature);
653             if (signer == address(0)) {
654                 unchecked {
655                     sigIdx++;
656                 }
657                 continue;
658             }
659
660             if (signer == verifier) {
661                 accumulatedPower += signerPower[verifier];
662                 unchecked {
663                     sigIdx++;
664                     verIdx++;
665                 }
666                 continue;
667             }
668
669             if (uint160(signer) < uint160(verifier)) {
670                 unchecked {
671                     sigIdx++;
672                 }
673                 continue;
674             }
675
676             unchecked {
677                 verIdx++;
678             }
679         }
680     }
```

Listing 2.38: MessageGateway.sol

2.3.4 Proper Cross-Spoke liquidity management

Introduced by `Version 1`

Description In the Hub-Spoke model, users can deposit tokens on one `Spoke` chain and withdraw through the `Hub` to a different `Spoke` chain, while actual liquidity is held in `TokenBridge` contracts on individual `Spoke` chains. In scenarios where a large volume of withdrawals targets a specific `Spoke`, the corresponding `TokenBridge` may not have sufficient balance, causing the function `finalizeWithdraw()` to fail with `InsufficientBalance` and leaving withdrawals in a pending state. The project stated that they will handle cross-Spoke liquidity in the future and that there is currently only one Spoke chain.

```
611             emit WithdrawFailed(withdrawalId, pending.sourceMessageHash, "InsufficientBalance");
```

```
612             return;
613         }
```

Listing 2.39: TokenBridge.sol

```
233             uint256 currentBalance = IERC20(token).balanceOf(address(this));
234             if (currentBalance + amount > tokenConfig.depositCap) revert DepositCapExceeded();
235         }
```

Listing 2.40: TokenBridge.sol

2.3.5 Potential centralization risks

Introduced by Version 1

Description In this project, several privileged roles can conduct sensitive operations, which introduces potential centralization risks. On the Spoke chain (Solidity), the ADMIN_ROLE of the contract MessageGateway can directly modify bridge configurations (e.g., function setTokenBridge()). On the Hub chain (Rust precompile), the BridgeAdmin role can unilaterally pause the entire bridge, modify custody configurations, and invalidate pending withdrawals without timelock or multisig requirements, and these governance operations remain executable even when the bridge is paused. Additionally, the Relayer role controls cross-chain message submission and verifier update requests, and its off-chain signature aggregation logic is outside the scope of this audit. If the private keys of these privileged accounts are compromised, it could pose significant risks to both the protocol's security and liveness.

2.3.6 Out of scope dependencies and external logic

Introduced by Version 1

Description The audited code relies on multiple external functions, modules, and dependencies whose implementations were not provided and are therefore considered out of scope. This includes, but is not limited to, internal business logic functions such as transfer_from_token_bridge(), load_balance(), compute_required_verifier_power(), and check_role!(), as well as amount conversion utilities like canonical_18_to_raw().

In addition, all external crates and runtime components referenced via use statements (e.g., precompile_interfaces, reth_storage_api, core_state_provider) were not reviewed. The protocol also depends on a modified execution environment (reth/revm fork) with custom extensions to the precompile execution model.

Accordingly, all external logic and dependencies invoked by the audited code are assumed to be correctly implemented and secure, and were not assessed as part of this audit.

2.3.7 Ensure proper setup in production

Introduced by Version 1

Description The project must ensure the proper setup (i.e., the variable MIN_CHALLENGE_PERIO) in production.

```
138    // FIXME: This is a temporary value for testing. It should be set to 1 hours in production.
139    uint256 public constant MIN_CHALLENGE_PERIOD = 10 seconds;
```

Listing 2.41: TokenBridge.sol


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS