



COMMIT BOOST

Multiplexer & Signer Security Assessment Report

Version: 2.1

January, 2026

Contents

Introduction	2
Disclaimer	2
Document Structure	2
Overview	2
Security Assessment Summary	3
Scope	3
Approach	3
Coverage Limitations	3
Findings Summary	3
Detailed Findings	5
Summary of Findings	6
JWT Replay Attack Vulnerability	7
Cross-Module Signature Request Forgery	9
Shared JWT Secret Across Modules	12
Inability To Revoke Or Add JWTs Via Reload Mechanism	13
Lack of Route-Specific Authorization in JWT Middleware	15
JWT Authentication Lacks Rate Limiting Protection	17
Missing PBS Config Validation	19
Insufficient Participant ID Validation In Distributed Signing	20
Inappropriate Key Uniqueness Enforcement In Muxer	22
Improper Use Of Dealing With Errors	23
Insufficient Path Validation For <code>MuxKeysLoader</code> File Path	24
Insecure Handling of External HTTP Requests	25
Early Return Missing For Empty Key Registry	27
Miscellaneous General Comments	28
A Vulnerability Severity Classification	29

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Commit Boost components. The review focused solely on the security aspects of these components, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the components in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Commit Boost components contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see [Vulnerability Severity Classification](#)), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Commit Boost components in scope.

Overview

Commit Boost is a modular sidecar that allows Ethereum validators to safely interact with the PBS pipeline and commitment protocols. It provides a flexible system for interacting with Ethereum network components, focusing on builder and relay interactions.

This review is a continuation of the [first review](#) completed by Sigma Prime in December 2024, focusing on the Signer and Multiplexer components of the Commit Boost Client.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the [commit-boost-client](#) repository.

The scope of this time-boxed review was strictly limited to files at commit [d3244bb](#).

The fixes of the identified issues were assessed at commit [19121f3](#).

Note: third party libraries and dependencies were excluded from the scope of this assessment.

Approach

The security assessment covered components written in Rust.

The manual review focused on identifying issues associated with the business logic implementation of the components in scope. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Rust language.

Additionally, the manual review process focused on identifying vulnerabilities related to known Rust anti-patterns and attack vectors, such as unsafe code blocks, integer overflow, floating point underflow, deadlocking, error handling, memory and CPU exhaustion attacks, and various panic scenarios including index out of bounds, `panic!()`, `unwrap()`, and `unreachable!()` calls.

To support the Rust components of the review, the testing team also utilised the following automated testing tools:

- Clippy linting: <https://doc.rust-lang.org/stable/clippy/index.html>
- Cargo Audit: <https://github.com/RustSec/rustsec/tree/main/cargo-audit>
- Cargo Outdated: <https://github.com/kbknapp/cargo-outdated>
- Cargo Geiger: <https://github.com/rust-secure-code/cargo-geiger>
- Cargo Tarpaulin: <https://crates.io/crates/cargo-tarpaulin>

Output for these automated tools is available upon request.

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 14 issues during this assessment. Categorized by their severity:

- High: 2 issues.
- Medium: 3 issues.
- Low: 4 issues.
- Informational: 5 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Commit Boost components in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: [Vulnerability Severity Classification](#).

A number of additional properties of the components, including optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- **Open**: the issue has not been addressed by the project team.
- **Resolved**: the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- **Closed**: the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
CBST2-01	JWT Replay Attack Vulnerability	High	Resolved
CBST2-02	Cross-Module Signature Request Forgery	High	Resolved
CBST2-03	Shared JWT Secret Across Modules	Medium	Resolved
CBST2-04	Inability To Revoke Or Add JWTs Via Reload Mechanism	Medium	Resolved
CBST2-05	Lack of Route-Specific Authorization in JWT Middleware	Medium	Closed
CBST2-06	JWT Authentication Lacks Rate Limiting Protection	Low	Resolved
CBST2-07	Missing PBS Config Validation	Low	Resolved
CBST2-08	Insufficient Participant ID Validation In Distributed Signing	Low	Resolved
CBST2-09	Inappropriate Key Uniqueness Enforcement In Muxer	Low	Resolved
CBST2-10	Improper Use Of Dealing With Errors	Informational	Resolved
CBST2-11	Insufficient Path Validation For <code>MuxKeysLoader</code> File Path	Informational	Resolved
CBST2-12	Insecure Handling of External HTTP Requests	Informational	Resolved
CBST2-13	Early Return Missing For Empty Key Registry	Informational	Resolved
CBST2-14	Miscellaneous General Comments	Informational	Resolved

CBST2-01	JWT Replay Attack Vulnerability		
Asset	signer/src/service.rs, common/src/utils.rs		
Status	Resolved: See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The JWT authentication mechanism lacks replay protection and does not bind the token to the specific request. Furthermore, the signing service communicates over unencrypted HTTP, making JWT interception significantly easier. This allows an attacker who intercepts a valid JWT to reuse it for authenticating arbitrary requests to the signing service, potentially leading to the signing of malicious messages and loss of funds.

A vulnerability exists because the signing service communicates over unencrypted HTTP. The `axum` server is started with a standard TCP listener, without TLS encryption enabled:

`signer/src/service.rs::run()`

```
let address = SocketAddr::from([0, 0, 0, 0], config.server_port);
let listener = TcpListener::bind(address).await?;

axum::serve(listener, app).await.wrap_err("signer server exited")
```

This lack of TLS means that any entity capable of monitoring network traffic between the client modules and the signing service can easily intercept the JWTs sent in the `Authorization: Bearer` header.

Furthermore, the JWT validation performed by `validate_jwt()` only checks standard claims and the signature. The JWT payload (`JwtClaims`) does not include any request-specific information, such as a nonce or a hash of the request body.

`common/src/utils.rs::validate_jwt()`

```
pub fn validate_jwt(jwt: Jwt, secret: &str) -> eyre::Result<()> {
    let mut validation = jsonwebtoken::Validation::default();
    validation.leeway = 10;

    jsonwebtoken::decode::(
        jwt.as_str(),
        &jsonwebtoken::DecodingKey::from_secret(secret.as_ref()),
        &validation,
    )
    .map(|_| ())
    .map_err(From::from)
}
```

The combination of easy interception (due to no TLS) and the lack of request-binding in the JWT means an attacker can capture a valid token and replay it to authenticate *different* requests (e.g., `request_signature` with arbitrary data, or `generate_proxy` with attacker-owned keys) sent to the signer service, as long as the token has not expired (within the 10-second leeway). This allows the attacker to effectively impersonate the legitimate module and make arbitrary calls to the signing endpoints.

The impact is classified as high because gaining the ability to make arbitrary `request_signature` calls allows an attacker to sign any message they choose using the validator's or proxy keys. Depending on how these signatures are used downstream, this could directly lead to loss of funds or other severe consequences. The likelihood is medium because, while the lack of TLS encryption significantly lowers the barrier for JWT interception, successfully sniffing

network traffic still requires specific preconditions (e.g., being on the same network segment, compromising network infrastructure).

Recommendations

Consider implementing any/both of the following:

1. Configure the `axum` server to use TLS (HTTPS) to encrypt all communication between clients and the signing service. This will protect JWTs from interception over the network.
2. Incorporate request-specific data into the JWT claims to prevent replay attacks even if a token is somehow compromised. Two possible approaches are:
 - Include a unique, single-use nonce (such as a `jti` claim) in the JWT payload. The server must maintain a list of used nonces for the lifetime of the tokens and reject any JWT with a previously seen nonce.
 - Include a hash of the request body in the JWT payload. The server would then recompute the hash upon receiving the request and compare it with the hash in the JWT, ensuring the token is tied to that specific request. Keep in mind that this does not prevent replay attacks for the same request, but prevents the same JWT from being reused in other requests.

Resolution

The development team has implemented the following changes in the following PRs:

1. [#357](#): Configured the `axum` server to use TLS (HTTPS) to encrypt all communication between clients and the signing service if the user configures the `tls_mode` to `TlsMode::Certificate` with a path to the TLS certificate and private key. Keep in mind that this is intentionally optional and the default is `TlsMode::Insecure`.
2. [#356](#): Added a `payload_hash` field to the `JwtClaims` struct. This field is computed by hashing the request body and is included in the JWT payload. The `payload_hash` is verified in `validate_jwt()`.
3. [#354](#): Added a `nonce` field to the `SignConsensusRequest` struct for replay protection. The requesting commit module (not the CB signer) is responsible for storing, comparing, and validating the nonces.

This issue has been resolved.

CBST2-02 Cross-Module Signature Request Forgery			
Asset	signer/src/service.rs, signer/src/manager/dirk.rs, signer/src/manager/local.rs		
Status	Resolved: See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

Signature requests handled by the signing service do not incorporate the requesting `module_id` into the signing domain. This allows a malicious or compromised module to request signatures that can be replayed and accepted by other modules, potentially leading to unauthorized actions like fund withdrawal.

The `handle_request_signature()` function processes requests for signing messages from modules. While the function authenticates the requesting module using JWT and extracts the `module_id`, this `module_id` is not subsequently used when calling the underlying signing methods:

signer/src/service.rs::handle_request_signature()

```

/// Implements request_signature from the Signer API
async fn handle_request_signature(
    Extension(module_id): Extension, // @audit module_id is extracted here
    // ...
) -> Result {
    // ...
    let res = match &*manager {
        SigningManager::Local(local_manager) => match request {
            // @audit module_id is NOT used in these calls
            SignRequest::Consensus(SignConsensusRequest { object_root, pubkey }) => local_manager
                .sign_consensus(&pubkey, &object_root)
            // ...
            SignRequest::ProxyBls(SignProxyRequest { object_root, proxy: bls_key }) => {
                local_manager
                    .sign_proxy_bls(&bls_key, &object_root)
                // ...
            }
            SignRequest::ProxyEcdsa(SignProxyRequest { object_root, proxy: ecdsa_key }) => {
                local_manager
                    .sign_proxy_ecdsa(&ecdsa_key, &object_root)
                // ...
            }
        },
        SigningManager::Dirk(dirk_manager) => match request {
            // @audit module_id is NOT used in these calls
            SignRequest::Consensus(SignConsensusRequest { object_root, pubkey }) => dirk_manager
                .request_consensus_signature(&pubkey, *object_root)
            // ...
            SignRequest::ProxyBls(SignProxyRequest { object_root, proxy: bls_key }) => dirk_manager
                .request_proxy_signature(&bls_key, *object_root)
            // ...
            SignRequest::ProxyEcdsa(_) => {
                // ... error handling ...
                Err(SignerModuleError::DirkNotSupported)
            }
        },
    };
    // ...
}

```

Consider a scenario where `module_A` is a malicious module and `module_B` is an optimistic preconfirmation service managing staking and slashing. If `module_B` uses consensus signer signatures to authorize withdrawals, `module_A` could

request a signature from the signing service for a withdrawal message specific to `module_B`. Since the `module_id` is not part of the signing domain, this signature would be valid for `module_B`, allowing `module_A` to illegitimately withdraw funds staked via `module_B`.

Additionally, a similar issue exists for signed proxy delegations, such that the `module_id` is not included in the delegation message, allowing a signed delegation created for one module to be replayed and used by a different module.

signer/src/manager/local.rs::create_proxy_ecdsa()

```
pub async fn create_proxy_ecdsa(
    &mut self,
    module_id: ModuleId,
    delegator: BlsPublicKey,
) -> Result {
    let signer = EcdsaSigner::new_random();
    let proxy_pubkey = signer.address();

    // @audit The message should also contain the module_id, otherwise the same signature can be used for multiple modules
    let message = ProxyDelegationEcdsa { delegator, proxy: proxy_pubkey };
    let signature = self.sign_consensus(&delegator, &message.tree_hash_root().0).await?;
    let delegation = SignedProxyDelegationEcdsa { signature, message };
    // ... rest of the function ...
    Ok(delegation)
}
```

The impact is classified as high because this vulnerability could lead to direct loss of user funds (e.g., staked assets). The likelihood is medium because exploiting this vulnerability requires control over a registered module within the system; it cannot be triggered by an arbitrary external actor but could be triggered by a compromised or intentionally malicious registered module.

Recommendations

Consider implementing the following:

1. Modify the signing process to incorporate the requesting `module_id` into the signing domain. This ensures that a signature generated for `module_A` is cryptographically bound to `module_A`'s context and cannot be replayed for `module_B`. Keep in mind that this would require a global registry of module IDs, such that different modules cannot register with the same ID.
2. When handling proxy signing requests, verify that the provided proxy key is actually associated with the requesting `module_id`. This could involve calling `has_proxy_bls_for_module()` or `has_proxy_ecdsa_for_module()` before proceeding with signing.
3. Modify the `ProxyDelegation` struct to include a `module_id` field so that signed proxy delegations are module-specific.

Resolution

The development team has implemented a signing ID that is unique for each module. This signing ID is included in the signed message and ensures domain separation across messages signed between modules.

During the retesting phase, the testing team informed the development team of the potential of a malicious module using the same signing ID as another module. The development team has acknowledged this issue with the following comment:

"This is a known and accepted security risk of the system due to the lack of a global registry, and the fact that there is a trust assumption for installed modules to be honest."

This issue has been resolved in PR [#329](#).

CBST2-03	Shared JWT Secret Across Modules		
Asset	cli/src/docker_init.rs		
Status	Resolved: See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

All modules are configured to use the same JWT secret when the `SIGNER_JWT_SECRET_ENV` environment variable exists, creating a security vulnerability where a compromised module could impersonate any other module.

When building the docker compose file in `docker_init.rs`, if the `SIGNER_JWT_SECRET_ENV` variable is set, the system configures all modules to use the identical JWT secret. This significantly reduces the security isolation between modules, as a malicious module with access to the shared secret could forge JWTs for any other module.

cli/src/docker_init.rs:handle_docker_init()

```
for module in modules_config {
    let module_cid = format!("cb_{}", module.id.to_lowercase());

    let module_service = match module.kind {
        // a commit module needs a JWT and access to the signer network
        ModuleKind::Commit => {
            let mut ports = vec![];

            let jwt_secret = load_optional_env_var(SIGNER_JWT_SECRET_ENV)
                .unwrap_or_else(random_jwt_secret);
            // ...
        }
        // ...
    }
    // ...
}
```

If the shared JWT secret is exposed, then a malicious module can pretend to be another authorized module and perform actions on behalf of that module, such as requesting signatures or creating proxy keys.

Recommendations

Consider removing the option to use a shared JWT secret and enforce unique secrets for each module.

Resolution

The development team has removed the option to use a shared JWT secret and now enforces unique random secrets for each module.

This issue has been resolved in PR [#294](#).

CBST2-04 Inability To Revoke Or Add JWTs Via Reload Mechanism			
Asset	signer/src/service.rs		
Status	Resolved: See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The signing service lacks a mechanism to revoke compromised JWTs or add new module JWTs without a full server restart. The `reload` endpoint only updates the signing manager configuration, not the JWT secrets used for authentication.

The `SigningState` struct holds both the `SigningManager` and a map of module IDs to JWT secrets.

signer/src/service.rs::SigningState

```
struct SigningState {
    // Manager handling different signing methods
    manager: Arc<RwLock<SigningManager>>,
    // Map of modules ids to JWT secrets. This also acts as registry of all
    // modules running
    jwt: Arc<HashMap<ModuleId, String>>,
}
```

The `jwt_auth()` middleware uses the `state.jwt` map to authenticate incoming requests. The `handle_reload()` function is intended to refresh the service configuration from environment variables. However, while it successfully creates and installs a `new_manager` based on the reloaded configuration, it does not update `state.jwt`.

signer/src/service.rs::handle_reload()

```
async fn handle_reload(
    State(mut state): State,
) -> Result {
    let req_id = Uuid::new_v4();

    debug!(event = "reload", ?req_id, "New request");

    let config = match StartSignerConfig::load_from_env() {
        Ok(config) => config,
        Err(err) => {
            error!(event = "reload", ?req_id, error = ?err, "Failed to reload config");
            return Err(SignerModuleError::Internal("failed to reload config".to_string()));
        }
    };

    let new_manager = match start_manager(config).await {
        Ok(manager) => manager,
        Err(err) => {
            error!(event = "reload", ?req_id, error = ?err, "Failed to reload manager");
            return Err(SignerModuleError::Internal("failed to reload config".to_string()));
        }
    };

    // @audit missing update to state.jwt
    state.manager = Arc::new(RwLock::new(new_manager));

    Ok(StatusCode::OK)
}
```

Consequently, if a module's JWT is compromised, an attacker can continue authenticating requests using that JWT even after an administrator attempts to reload the configuration to remove it. The compromised JWT remains valid

until the signing service is completely restarted with a new initial configuration. Similarly, new modules cannot be registered by simply reloading the configuration; a restart is required.

The impact is assessed as medium because it significantly hinders incident response by preventing timely revocation of compromised credentials and impacts operational flexibility by requiring restarts for module updates. The likelihood is also medium because the vulnerability (lack of revocation/update) is always present, but exploiting the security aspect requires a prior JWT compromise. Administrators attempting to add new modules after starting the service will inevitably encounter this limitation.

Recommendations

Consider modifying the `handle_reload()` function to update the `state.jwts` field based on the `jwt`s loaded in the new `StartSignerConfig`.

Furthermore, consider adding an endpoint specifically for JWT revocation. This would allow administrators to immediately invalidate a known compromised JWT, preventing its further use and thereby mitigating the impact of the compromise until the token can be replaced through configuration updates.

Resolution

The development team has added the following changes in the following PRs:

1. [#295](#):

- Added a `jwt_secrets` field to the `ReloadRequest` struct used to update the `state.jwts` field
- Added a `revoke_module` endpoint to quickly remove the permissions of compromised modules.
- Implemented separate JWT authentication for admin endpoints.

2. [#376](#):

- Added rate limit count increment to the admin JWT authentication middleware.
- Implemented atomic changes to `handle_reload()` function, and doesn't update any state if any failures occur.

3. [#387](#):

- Added a call to `mark_jwt_failures()` if payload conversion failed.

This issue has been resolved.

CBST2-05	Lack of Route-Specific Authorization in JWT Middleware		
Asset	signer/src/service.rs, common/src/utils.rs		
Status	Closed: See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The JWT authentication mechanism allows a valid JWT for a specific module to access any API endpoint, regardless of the intended privilege level for that JWT. This could allow unauthorized access to sensitive operations like requesting signatures or generating proxy keys using a JWT intended for less sensitive operations.

The `axum` router in `service.rs` applies the `jwt_auth()` middleware globally to the routes `/request_signature`, `/get_pubkeys`, and `/generate_proxy_key`.

signer/src/service.rs::build_router()

```
let app = axum::Router::new()
    .route(REQUEST_SIGNATURE_PATH, post(handle_request_signature))
    .route(GET_PUBKEYS_PATH, get(handle_get_pubkeys))
    .route(GENERATE_PROXY_KEY_PATH, post(handle_generate_proxy))
    .route_layer(middleware::from_fn_with_state(state.clone(), jwt_auth))
    .route(RELOAD_PATH, post(handle_reload))
    .with_state(state.clone())
    .route_layer(middleware::from_fn(log_request))
    .route(STATUS_PATH, get(handle_status));
```

The `jwt_auth()` middleware function correctly decodes the JWT, extracts the `module_id`, retrieves the corresponding secret, and validates the token using `validate_jwt()`:

signer/src/service.rs::jwt_auth()

```
async fn jwt_auth(
    // ...
) -> Result {
    let jwt: Jwt = auth.token().to_string().into();

    // We first need to decode it to get the module id and then validate it
    // with the secret stored in the state
    let module_id = decode_jwt(jwt.clone()).map_err(|e| {
        error!("Unauthorized request. Invalid JWT: {e}");
        SignerModuleError::Unauthorized
    })?;

    let jwt_secret = state.jwts.get(&module_id).ok_or_else(|| {
        error!("Unauthorized request. Was the module started correctly?");
        SignerModuleError::Unauthorized
    })?;

    validate_jwt(jwt, jwt_secret).map_err(|e| {
        error!("Unauthorized request. Invalid JWT: {e}");
        SignerModuleError::Unauthorized
    })?;

    req.extensions_mut().insert(module_id);

    Ok(next.run(req).await)
}
```

However, upon successful validation, `jwt_auth()` simply passes the request to the next handler without checking if

the authenticated `module_id` and the specific JWT used is authorized to access the requested route. This violates the principle of least privilege, as a compromised JWT intended for a low-privilege operation (like `/get_pubkeys`) could be misused to perform high-privilege actions (like `/request_signature` or `/generate_proxy_key`).

The impact is classified as medium because it allows bypassing intended access controls for sensitive signing operations, although it does not directly lead to fund loss. The likelihood is medium as an attacker needs to obtain a valid JWT, which could occur through module compromise or network sniffing.

Recommendations

Consider implementing route-specific authorization checks. This can be achieved by:

1. Adding custom claims to the JWT payload specifying the allowed routes or operations.
2. Modifying the `jwt_auth()` middleware to extract these claims after validation and verify that the requested route is present in the allowed list. Reject the request if the route is not permitted for the given JWT.

Resolution

The development team has acknowledged the issue above with the following comment:

"We elected, for simplicity, to only have one JWT per module that has access to all of the routes. The onus is on the user to trust the modules that they're installing."

CBST2-06 JWT Authentication Lacks Rate Limiting Protection			
Asset	signer/src/service.rs		
Status	Resolved: See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The JWT authentication mechanism lacks rate limiting, potentially allowing attackers to brute-force credentials or cause a Denial of Service (DoS) by overwhelming the server with invalid requests.

The `jwt_auth()` middleware validates incoming JWTs. However, there is no mechanism to track the number of failed attempts from a specific source (e.g., IP address) or to introduce delays or temporary blocks after multiple failures.

An attacker can continuously send requests with invalid JWTs, consuming resources without penalty. This could lead to a DoS condition, preventing legitimate modules from accessing the signing service, or potentially aid in brute-forcing valid tokens if the secret space or token generation is weak.

This issue has a medium impact as it could potentially lead to compromised credentials that would allow an attacker to request arbitrary signatures however, it would not apply to securely generated JWTs. However, the likelihood of this attack is low due to the limited success of brute-forcing attacks.

Recommendations

Consider implementing rate limiting on the JWT authentication middleware. This could involve:

1. Tracking failed authentication attempts per IP address.
2. Implementing a strategy like exponential backoff, where the delay between allowed attempts increases after each failure.
3. Temporarily blocking IP addresses that exceed a certain threshold of failed attempts within a specific timeframe.

Resolution

The development team has fixed the issue above in the following PRs:

1. [#310](#): Added socket IP-based rate limiting to the JWT authentication middleware.
2. [#376](#):
 - Added support for reverse proxy IP-based rate limiting
 - Added a task to periodically sweep and clean up expired JWT authorization failures.
3. [#388](#): Refactored reverse proxy IP-based rate limiting to allow user to configure the header name to use to determine the real client IP.

4. [#410](#): Refactored rightmost value parsing based on trusted count.
5. [#420](#): Refactored rightmost value parsing to support "comma+space" separated values and account for multiple occurrences of the same header.

During the retesting process, the testing team identified a race condition issue that was introduced in [#310](#). The development team acknowledged the issue with the following comment:

"We had a fix for this implemented, but it ended up queueing every incoming request so they ran sequentially rather than in parallel because of the fact that the body needs to be decoded and hashed in order to validate the JWT. This hampered parallel performance enough to raise concerns, so we're opting to leave it alone for now."

CBST2-07 Missing PBS Config Validation			
Asset	common/src/config/mux.rs		
Status	Resolved: See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The PBS configuration is created by combining user-provided values with defaults, but there is no validation step after the config is created.

```
common/src/config/mux.rs::validate_and_fill()
```

```
let config = PbsConfig {
  timeout_get_header_ms: mux
    .timeout_get_header_ms
    .unwrap_or(default_pbs.timeout_get_header_ms),
  late_in_slot_time_ms: mux
    .late_in_slot_time_ms
    .unwrap_or(default_pbs.late_in_slot_time_ms),
  ..default_pbs.clone()
};
// @audit No validation call to PbsConfig::validate()
```

Without validation, it is possible to create a `PbsConfig` with invalid or nonsensical values. For example, if either `timeout_get_header_ms` or `late_in_slot_time_ms` is set to 0, the system would immediately fall back to local block building, preventing the validator from participating in the PBS supply chain and thus losing out on potential MEV rewards.

This issue has a medium impact as it can result in a loss of MEV rewards. The likelihood is low as it requires a misconfiguration in the mux configuration.

Recommendations

Add an explicit validation step after creating the `PbsConfig`.

Resolution

The development team has added a call to `PbsConfig::validate()` after creating the `PbsConfig`.

This issue has been resolved in [PR #317](#).

CBST2-08	Insufficient Participant ID Validation In Distributed Signing		
Asset	signer/src/manager/dirk.rs		
Status	Resolved: See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The distributed signing mechanism does not properly validate participant IDs obtained from Dirk hosts. It fails to check if participant IDs are non-zero or unique before associating them with a distributed account. This could lead to signature aggregation failures or potentially incorrect signature generation under specific circumstances.

The function `load_distributed_accounts()` processes information about distributed accounts received from Dirk hosts. When associating a host connection (channel) with a participant ID for a given distributed account, two checks are missing:

1. The code extracts the `participant_id` but does not ensure it is non-zero. The Lagrange interpolation used in `aggregate_partial_signatures()` assumes distinct, non-zero identifiers. Using an ID of 0 can lead to mathematical errors during coefficient calculation.

```
signer/src/manager/dirk.rs::load_distributed_accounts()
let Some(endpoint { id: participant_id, .. }) =
    account.participants.iter().find(|participant| participant.name == host_name)
// @audit No check that participant_id != 0
```

2. When adding a participant ID and its corresponding channel to an existing `DistributedAccount`, the code uses `HashMap::insert()` without checking its return value. If a participant ID already exists in the map, the existing entry will be silently overwritten. This could lead to an incorrect set of participants being used for threshold signing.

```
signer/src/manager/dirk.rs::load_distributed_accounts()
match consensus_accounts.get_mut(&public_key) {
    Some(Account::Distributed(DistributedAccount { participants, .. })) => {
        participants.insert(participant_id as u32, channel.clone());
        // @audit No check that insert() returns None
    }
}
```

Recommendations

Consider modifying the `load_distributed_accounts()` function to perform the following checks:

1. After obtaining the `participant_id`, verify that it is not zero. Log a warning and skip the participant if it is zero.
2. When inserting into the participants map for an existing `DistributedAccount`, check the return value of `insert()`. If it returns `Some(_)`, indicating a duplicate ID, log a warning and handle the duplication appropriately (e.g., skip the update or raise an error, depending on the desired behavior for conflicting configurations).

Resolution

The development team has added a check to skip the participant if the ID is zero and log a warning if the ID is already in the `DistributedAccount.participants` map.

This issue has been resolved in PR [#296](#).

CBST2-09	Inappropriate Key Uniqueness Enforcement In Muxer		
Asset	common/src/config/mux.rs		
Status	Resolved: See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The muxer enforces key uniqueness in external registries like Lido's `NodeOperatorRegistry` contract and SSV Network, causing failures when importing valid keys that happen to have duplicates, rather than simply filtering duplicates.

The code checks for duplicates in key registries and returns an error when duplicates are found:

```
common/src/config/mux.rs::fetch_lido_registry_keys()
let unique = keys.iter().collect::<HashSet<_>>();
ensure!(unique.len() == keys.len(), "found duplicate keys in registry");
```

This approach is problematic because Lido's `NodeOperatorRegistry` does not enforce uniqueness when keys are added, and SSV Network's uniqueness checks are based on a combination of `pubkey` and `msg.sender`, not just the `pubkey`.

By enforcing uniqueness and failing on duplicates, the system prevents importing valid keys from these registries when duplicates exist. This creates an unnecessary operational barrier and could prevent valid validators from being used.

Recommendations

Change the approach to filter duplicates rather than rejecting them.

Resolution

The development team has removed the uniqueness checks from both `fetch_lido_registry_keys()` and `fetch_ssv_pubkeys()`. Duplicate keys are now filtered after fetching all keys in `MuxKeysLoader::load()`.

This issue has been resolved in PR [#318](#).

CBST2-10	Improper Use Of Dealing With Errors	
Asset	common/src/signer/loader.rs, common/src/signer/schemes/ecdsa.rs	
Status	Resolved: See Resolution	
Rating	Informational	

Description

The code uses `unwrap()` when decrypting keystore passwords, which will cause the application to panic if decryption fails rather than handling the error gracefully.

In `loader.rs`, the keystore decryption operation uses `unwrap()` on the result.

```
common/src/signer/loader.rs::load_ecdsa_signer()
let key_reader = std::io::BufReader::new(key_file);
let keystore: JsonKeystore = serde_json::from_reader(key_reader)?;
let password = std::fs::read(secrets_path)?;
let decrypted_password = eth2_keystore::decrypt(&password, &keystore.crypto).unwrap();
```

Similarly, in `ecdsa.rs`:

```
common/src/signer/schemes/ecdsa.rs::new_random()
Self::new_from_bytes(secret.as_slice()).unwrap()
```

Using `unwrap()` in these contexts is problematic because it will cause the entire application to crash if decryption fails, which could happen for several legitimate reasons (corrupted keystore, incorrect password, etc.). This prevents proper error handling and could lead to service disruptions.

Recommendations

Replace `unwrap()` with proper error handling.

Resolution

The development team has replaced the `unwrap()` instances above with proper error handling and propagation.

This issue has been resolved in [PR #298](#).

CBST2-11	Insufficient Path Validation For MuxKeysLoader File Path	
Asset	common/src/config/mux.rs	
Status	Resolved: See Resolution	
Rating	Informational	

Description

When processing the file path for loading keys, the code does not perform basic validation that the provided file path exists, is a file, or is a JSON file, which could lead to runtime errors when attempting to load keys.

common/src/config/mux.rs:loader_env()

```
pub fn loader_env(&self) -> Option<(String, String, String)> {
    self.loader.as_ref().and_then(|loader| match loader {
        MuxKeysLoader::File(path_buf) => {
            let path = path_buf.to_str().unwrap_or_else(|| panic!("invalid path: {:?}", path_buf));
            let internal_path = get_mux_path(&self.id);
            Some((get_mux_env(&self.id), path.to_owned(), internal_path))
        }
        // ...
    })
}
```

Recommendations

Add validation to check that the path exists, is a file, and has a JSON extension.

Resolution

The development team has added a validation step in `loader_env()` to ensure that the path exists, is a file, and can be deserialised from a JSON file.

This issue has been resolved in PR [#300](#).

CBST2-12	Insecure Handling of External HTTP Requests	
Asset	common/src/config/mux.rs, common/src/pbs/event.rs	
Status	Resolved: See Resolution	
Rating	Informational	

Description

The functions `load()` and `fetch_ssv_pubkeys()` in `common/src/config/mux.rs`, and the function `publish()` in `common/src/pbs/event.rs`, handle external HTTP requests insecurely when fetching data from provided URLs.

This can cause several problems:

1. Requests lack a timeout. When the server is slow or does not respond, the program may wait indefinitely. This can cause the program to hang and use up valuable resources from pending threads.
2. The URL may not use HTTPS. Without TLS encryption data may be modified by actors in the network. Furthermore, the data would not be encrypted allowing eavesdropping.
3. The file downloaded from the URL could be excessive in size, exhausting memory. This occurs on the line `let pubkeys = response.text().await?;`, which loads the entire response into memory as a `String` without validating the length.

These issues manifest in the following code:

common/src/config/mux.rs:load()

```
let client = request::Client::new();
// @audit no timeout configured on the client, and URL scheme (HTTPS) is not checked.
let response = client.get(url).send().await?;
// @audit this loads the entire response into memory as a String
let pubkeys = response.text().await?;
```

common/src/config/mux.rs:fetch_ssv_pubkeys()

```
let client = request::Client::new();
let mut pubkeys: Vec<BlsPublicKey> = vec![];
let mut page = 1;

loop {
    // @audit no timeout configured on the client, and URL scheme (HTTPS) is not checked.
    let response = client
        .get(format!(
            "...
            chain_name, node_operator_id, MAX_PER_PAGE, page
        ))
        .send()
        .await
        .map_err(|e| eyre::eyre!("Error sending request to SSV network API: {e}"))?
    // @audit this loads the entire response into memory before parsing JSON.
    .json::<SSVResponse>()
    .await?;
```

common/src/pbs/event.rs::publish()

```
tokio::spawn(async move {
    trace!("Sending events to {}", endpoint);
    // @audit no timeout configured on the client, and URL scheme (HTTPS) is not checked.
    if let Err(err) = client
        .post(endpoint)
        .json(event)
        .send()
        .await
        .and_then(|res| res.error_for_status())
    {
        error!("Failed to publish event: {:?}", err)
    }
});
```

This issue is marked as informational as the URL comes from a trusted configuration file.

Recommendations

To improve the safety and reliability of using external URLs:

1. Add a timeout to all HTTP requests. Use `request::ClientBuilder` to create a client with a specific timeout (e.g., `std::time::Duration::from_secs(10)`) and reuse this client instance for multiple requests, especially within loops.
2. Ensure the URL uses HTTPS. Parse the URL and explicitly check if the scheme is HTTPS (`url.scheme() == "https"`) before proceeding with the request.
3. Limit the size of the downloaded response before loading it fully into memory. Reading the entire response with `.text()` or `.json()` can lead to memory exhaustion if the response is unexpectedly large. First, check the `Content-Length` header. If the header is present and indicates a size exceeding a predefined limit, abort the request immediately. Otherwise (if the header is missing or indicates an acceptable size), read the response body using a streaming approach such as `response.bytes_stream().take(1024)`. While streaming, continuously count the bytes received and abort the operation if the count exceeds the predefined limit. This prevents excessive memory usage even if the `Content-Length` header is missing or maliciously forged to be smaller than the actual response size.

Resolution

The development team has implemented the following changes:

1. Log a warning if the URL scheme is not HTTPS.
2. Add a timeout to the HTTP client. The timeout is set to 10 seconds by default and can be configured by the user.
3. Read the response body in chunks, enforcing a maximum size of 10 MiB.

This issue has been resolved in PR [#327](#).

CBST2-13	Early Return Missing For Empty Key Registry	
Asset	common/src/config/mux.rs	
Status	Resolved: See Resolution	
Rating	Informational	

Description

The code does not check if a node operator has any keys in the registry before proceeding with key loading, leading to unnecessary processing when working with empty registries.

When loading keys from a Lido's `NodeOperatorRegistry`, the code retrieves the total key count but does not immediately verify that this count is greater than zero.

```
common/src/config/mux.rs::fetch_lido_registry_keys()
```

```
let total_keys =  
    registry.getTotalSigningKeyCount(node_operator_id).call().await?._0.try_into()?;  
// @audit Missing check for total_keys > 0 with early return
```

If there are no keys, the function will continue execution only to return an empty result set later.

Recommendations

Add an early check and return if the key count is zero.

Resolution

The development team has added an early return if the key count is zero.

This issue has been resolved in PR [#299](#).

CBST2-14 Miscellaneous General Comments		
Asset	All files	
Status	Resolved: See Resolution	
Rating	Informational	

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. Misleading Variable Naming For Ethereum Addresses

Related Asset(s): *signer/src/manager/local.rs*

Throughout the codebase, the variable name `proxy_pubkey` is incorrectly used to represent Ethereum addresses, which are derived from but not equivalent to public keys.

There are instances where Ethereum addresses are mistakenly referred to as "public keys".

`signer/src/manager/local.rs::create_proxy_ecdsa()`

```
pub async fn create_proxy_ecdsa(
    &mut self,
    module_id: ModuleId,
    delegator: BlsPublicKey,
) -> Result {
    let signer = EcdsaSigner::new_random();
    let proxy_pubkey = signer.address(); // @audit This is an Ethereum address, not a public key
    // ...
}
```

While this does not create a functional security issue, it introduces confusion and could lead to misunderstandings during development, code review, or when new contributors work with the codebase.

Rename the variables to accurately reflect that they represent Ethereum addresses.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The development team has fixed the issue above in PRs [#319](#) and [#376](#).

Appendix A Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurrence. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

σ'