Gattaca Security Review

Auditors

Mattsse, Lead Security Researcher

Alex Stokes, Lead Security Researcher

Report prepared by: Lucas Goiriz

February 26, 2024

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Gattaca's Helix is a Rust-based MEV-Boost Relay implementation, developed as an entirely new code base from the ground up. It has been designed with key foundational principles at its core, such as modularity and extensibility, low-latency performance, robustness and fault tolerance, geo-distribution, and a focus on reducing operational costs.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of helix according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 14 days in total, Gattaca engaged with Spearbit to review the helix protocol. In this period of time a total of **35** issues were found.

Summary

Project Name	Gattaca
Repository	helix
Commit	4c3a5b...910a8d
Type of Project	Infrastructure, MEV
Audit Timeline	Jan 22 to Feb 2
One day fix period	Feb 7

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	2	2	0
High Risk	2	2	0
Medium Risk	4	4	0
Low Risk	9	8	1
Gas Optimizations	0	0	0
Informational	18	10	8
Total	**35**	**26**	**9**

5 Findings

5.1 Critical Risk

5.1.1 No connection timeout on `reqwest::Client`s

Severity: Critical Risk

Context: crates/beacon-client/src/beacon_client.rs#L38-L38, crates/api/src/service.rs#L80-L80

Description: The default `reqwest::Client` applies no timeout from when the request starts connecting until the response body has finished. This can result in indefinitely hanging requests when invoking the beacon API via the `MultiBeaconClient`, e.g. `get_state_validators`:

```
async fn get_state_validators(
    &self,
    state_id: StateId,
) -> Result<Vec<ValidatorSummary>, BeaconClientError> {
    let clients = self.beacon_clients_by_last_response();
    let mut last_error = None;

    for (i, client) in clients.into_iter().enumerate() {
        match client.get_state_validators(state_id.clone()).await {
```

This can prevent the `housekeeper` from fetching and updating the validators:

```
let validators = match self.beacon_client.get_state_validators(StateId::Head).await
```

which is locked by the `refresh_validators_lock` mutex guard:

```
let _guard = self.refresh_validators_lock.try_lock()?;
```

Recommendation: Configure appropriate timeouts for the `reqwest::Client`. With the `MultiBeaconClient`, timeouts can accumulate as the requests are sent one after the other if a request failed. Consider tighter timeouts or requesting with multiple clients concurrently.

Gattaca: Request timeouts have been added for the Simulator, the BeaconClient and the API, in commit 7a543580.

Spearbit: Fixed.

5.1.2 `decode_payload` and `decode_header_submission` consume arbitrary amounts of memory in `submit_block(_v2)` and `submit_header`

Severity: Critical Risk

Context: crates/api/src/builder/api.rs#L1356-L1356

Description: `decode_payload` called by the public `submit_block` and `submit_block_v2` functions with the raw http `Request<Body>`:

```
pub async fn decode_payload(
    req: Request<Body>,
```

consumes the entire body using

```
// Read the body
let body = req.into_body();
let mut body_bytes = hyper::body::to_bytes(body).await?;
```

This is also the case for `decode_header_submission` in `submit_header`

```
pub async fn decode_header_submission(
    req: Request<Body>,
```

hyper::body::to_bytes does not apply any length checks. This call consumes the entire body into memory. While axum configures a DefaultBodyLimit of 2MB this is only applied to `FromRequest` implementations that explicitly apply it.

Recommendation: Enforce body size limts via RequestExt::with_limited_body or

```
req: Request<Limited<Body>>
```

see also difference between DefaultBodyLimit and RequestBodyLimit.

Gattaca: Addressed in commit f1a37e7a. The router now has additional RequestBodyLimit layers attached for the Builder & Proposer API with specific max length parameters for `MAX_PAYLOAD_LENGTH`, `MAX_BLINDED_BLOCK_LENGTH`, `MAX_VAL_REGISTRATIONS_LENGTH`.

Spearbit: Fixed.

5.2 High Risk

5.2.1 Builders can grief other builders with duplicate block hash check

Severity: High Risk

Context: api.rs#L200, api.rs#L348

Description: The concern would be a "replay"-style attack where a malicious builder:

1. Listens for a (assumed) valid block hash from another relay.
2. Then immediately sends a payload with the same block hash to the `helix` relay.

Note that the crafted payload in (2) does not need to be valid, simply have the correct syntax with the target block hash in the message.

Due to the current implementation of the duplicate block hash checks, the honest builder from the remote relay would be prevented from submitting their block to the `helix` relay, opening a griefing vector that would constitute a denial-of-service against this relay.

Recommendation: Refactor the duplicate block hash check. Another suggestion is to implement finer-grained tracking of the block state. With finer-grained tracking, the relay could essentially process any blocks and only skip the heavier weight checks (e.g. simulation) if this was the first time the relay saw a block. It should not reject the block like it currently does.

Gattaca: Initially the duplicate block hash check was just to reduce load when processing gossiped requests. Dropping duplicates in submit_block and header was just an optimisation. Therefore, to resolve this issue, we now only exit if we see a duplicate in the 2 process gossip functions. In submit_block and submit_header we save the block hash to the duplicate list but don't exit if it's already seen (see commit 15520c62).

Spearbit: Fixed.

5.2.2 `handle_new_payload_attributes` removes payload attributes greater or equal than current head slot

Severity: High Risk

Context: crates/api/src/builder/api.rs#L1306-L1306

Description: The `handle_new_payload_attributes` function is invoked on a new `ChainUpdate::PayloadAttributesUpdate` it cleans up old payload attributes

```
// Clean up old payload attributes
let mut all_payload_attributes = self.payload_attributes.write().await;
all_payload_attributes.retain(|_, value| value.slot < head_slot);
```

This retains all payload attributes with a slot lower than the current `head_slot` and removes payload attributes with a slot equal or higher than the current head.

This clears the current `head_slot` from the map which can result in header gossip failure..

Recommendation: Retain payload attributes that are equal or higher than the current slot:

```
all_payload_attributes.retain(|_, value| value.slot >= head_slot);
```

Gattaca: Addressed in commit 014b64ac. Now it retains payload attributes that are equal or higher than the current slot.

Spearbit: Fixed.

5.3 Medium Risk

5.3.1 Builders can get locked out of `submit_block` due to unrelated error

Severity: Medium Risk

Context: api.rs#L200, api.rs#L408, api.rs#L723

Description: The function will return early if a builder submits the same block twice. However, there are many (completely unrelated) points in the rest of this function where something could fail (e.g. a redis internal error here) and it would prevent the builder from re-submitting their block. It's possible that the key expiry (cf. here) would help an honest builder here, although the cache timing of 45s is multiples of the slot time, so a builder would be prevented from submitting their block, really from no fault of their own.

Recommendation: Remove this Redis entry upon failure in any subsequent failure in this function. This may be a bit untenable but could try something like a custom drop guard with some internal mutable state, or reconsider the duplicate submission logic to avoid this failure mode.

Gattaca: See the fix for issue "Builders can grief other builders with duplicate block hash check".

Spearbit: Fixed.

5.3.2 Race condition when updating Redis state on latest delivered payload

Severity: Medium Risk

Context: redis_cache.rs#L337-L338, redis_cache.rs#L339-L343, redis_cache.rs#L356

Description: The state to record the latest delivered payload is split into two updates to the underlying Redis instance, and importantly across an async/await checkpoint in the rust code. It is possible that (e.g.) a buggy client calls `getPayload` honestly multiple times in a way that interacts with the error checking here such that the payload is not released, even when otherwise the exchange is "*honest*".

Recommendation: To reduce the chance of this race condition causing any issue, I suggest to update the Redis state synchronously. A further, although more heavy-handed, enhancement would be to have some global lock across the entire execution of the `getPayload` implementation so even upon multiple calls to this endpoint we can be sure to not suffer shifting state relative to the relay's execution.

Gattaca:

- Since commit b8f4699e, it uses a redis pipeline to make `check_and_set_last_slot_and_hash_delivered` atomic.
- Setting the floor and top bid need to be done synchronously as we only update the bid value if the copy was successful. See the check here.

Spearbit: Fixed.

5.3.3 No request timeouts introduces resource consumption related DoS

Severity: Medium Risk

Context: crates/api/src/service.rs#L130-L130

Description: The `ApiService Router` does not configure a `TimeoutLayer`. This introduces a DoS where incomplete requests are kept alive with allocated memory read from the request's body. In the current implementation the `decode_payload` allocates arbitrary amounts of memory while reading the request's body (see the issue "`decode_payload` and `decode_header_submission` consume arbitrary amounts of memory in `submit_block(_v2)` and `submit_header`"). Even if reading is restricted to `MAX_PAYLOAD_LENGTH`, malicious requests can send `MAX_PAYLOAD_LENGTH` bytes per request and never terminate the request.

Consider this rust example for submitting payloads to `submit_block`:

```
let mut body : FuturesOrdered<Pin<Box<dyn Future< Output = Result<Vec<u8>, Infallible>> + Send>>> =
↪  FuturesOrdered::new();
body.push_back(Box::pin(async move {
    // Stream max allowed payload length bytes
    Ok::<_, Infallible>(vec![0u8;MAX_PAYLOAD_LENGTH])
}));
body.push_back(Box::pin(async move {
    // never complete the request
    pending::<()>().await;
    Ok::<_, Infallible>(vec![])
}));
let body = Body::wrap_stream(body);
// Send request by streaming the malicious body
let resp = reqwest::Client::new()
    .post(req_url.as_str())
    .header("accept", "*/*")
    .header("Content-Type", "application/json")
    .body(body)
    .send()
    .await
    .unwrap();
```

The `decode_payload` function will now allocate the first `MAX_PAYLOAD_LENGTH` bytes of the received request's body and wait for it to complete, which never happens.

Recommendation: Introduce reasonable timeouts for the server, (see also error handling for middleware). Consider introducing rate limits.

Gattaca: Request timeouts have been added for the Simulator, the BeaconClient and the API in commit 7a543580.

Spearbit: Fixed.

5.3.4 Chain state is sourced in multiple places, possibly leading to "*split brain*" type behavior

Severity: Medium Risk

Context: housekeeper.rs#L99, chain_event_updater.rs#L74

Description: When a relay is configured as a "housekeeper", it gets the head event in two different places and this could potentially lead to issues where part of the codebase thinks the head is `A` and another part of the codebase thinks the head is `B`. I haven't dug more deeply into what could go wrong but given the importance of the housekeeper, it feels like this could cause issues.

Recommendation: Refactor the architecture to have one (1) input channel for head events and share this to all relevant parties in the codebase. There should only be one possible view inside the entire relay at any given (wall-clock) time.

Gattaca: Addressed in commit 76bd5b17. We now use a broadcaster to ensure a single source of truth.

Spearbit: Fixed.

5.4 Low Risk

5.4.1 `ChainEventUpdater` skips all `HeadEventData` if any previously received slot is higher

Severity: Low Risk

Context: crates/housekeeper/src/chain_event_updater.rs#L119-L119

Description: The updater always sets the `last_slot` to the highest received `HeadEventData`, skipping all updates in between. If one `BeaconClient` sends a faulty `HeadEventData` that's far in the future, all `HeadEventData` `PayloadAttributesEvent` updates up until that block are skipped:

```
async fn process_head_event(&mut self, event: HeadEventData) {
    if self.last_slot >= event.slot {
        return;
    }

    self.last_slot = event.slot;
```

Recommendation: Consider adding event validation for events received for the same slot from multiple beacon clients. The `PayloadAttributes` timestamp can be used for validation.

Gattaca: Addressed in commit 3bccbdf4. Now it validates the head update slot. We compare the current timestamp vs the expected timestamp of the slot and only accept slots less than 60 seconds in the future.

Spearbit: Fixed.

5.4.2 `best_beacon_instance` for `beacon_clients_by_last_response` and `beacon_clients_by_least_used` does not map to the corresponding client

Severity: Low Risk

Context: client/src/multi_beacon_client.rs#L40-L47

Description: The `MultiBeaconClient` functions use the `beacon_clients_by_last_response` function to get a list of `of beacon clients, prioritized by the last successful response`.

This function swaps the `best_beacon_index` with position 0:

```
let mut instances = self.beacon_clients.clone();
let index = self.best_beacon_instance.load(Ordering::Relaxed);
if index != 0 {
    instances.swap(0, index);
}
```

Several functions update the `best_beacon_instance` with the client that responded first using the order of `beacon_clients_by_last_response`:

```
let clients = self.beacon_clients_by_last_response();
let mut last_error = None;

for (i, client) in clients.into_iter().enumerate() {
    match client.get_genesis().await {
        Ok(genesis_info) => {
            self.best_beacon_instance.store(i, Ordering::Relaxed);
```

The `publish_block` function updates the first index received from parallel executed requests.

The index obtained via `.enumerate()` no longer represents the order in the immutable `beacon_clients` list.

A successful request in `publish_block` can then shift a, potential failing client to first position, leading to delayed processing in the sequential calls like `get_proposer_duties`.

Recommendation: To maintain correct mappings, the constant indices can be attached to the clients directly:

```
pub beacon_clients: Vec<Arc<(usize, BeaconClient)>>,
```

Gattaca: Addressed in commit a91379b8. Now it uses fixed IDs for beacon client instances, as suggested.

Spearbit: Fixed.

5.4.3 Consider verifying `proposer_public_key` matches our expected proposer when accepting blocks

Severity: Low Risk

Context: api.rs#L1698

Description: The linked function performs a number of sanity checks against incoming bids/blocks from builders. One thing it does not check that should be simple to verify is that the `proposer_public_key` in the bid trace matches the one we expect based on the "*next duty*" we have saved.

Recommendation: Verify `bid_trace.proposer_public_key` matches the public key reachable from `next_duty`. This is low-severity as something else would likely fail verification (esp. against the simulator) but I'd generally recommend validating *all* fields that are incoming.

Gattaca: Added extra proposer public key validation to `sanity_check_block_submission` in commit 37b591e4.

Spearbit: Fixed.

5.4.4 Unsaved `pending_validator_registrations` can be cleared

Severity: Low Risk

Context: crates/database/src/postgres/postgres_db_service.rs#L128-L137

Description: Newer `pending_validator_registrations` are saved to db on an interval basis if the list is not empty after filtering cached validators:

```
match self_clone.pending_validator_registrations.len() {
    0 => continue,
    _ => {
        let mut entries = Vec::new();
        // AUDIT COMMENT: This acquires a lock while inserting over the all current entries
        for key in self_clone.pending_validator_registrations.iter() {
            if let Some(entry) = self_clone.validator_registration_cache.get(&*key)
            {
                entries.push(entry.registration_info.clone());
            }
        }
        match self_clone._save_validator_registrations(entries).await {
            Ok(_) => {
                // AUDIT COMMENT: This clears __all__ currently pending registrations
                self_clone.pending_validator_registrations.clear();
                info!("Saved validator registrations");
            }
            Err(e) => {
                error!("Error saving validator registrations: {}", e);
            }
        };
    }
};
```

`pending_validator_registrations` can be populated with new entries while pending entries are being inserted:

```
for entry in entries.iter() {
    self.pending_validator_registrations
        .insert(entry.registration.message.public_key.clone());
    self.validator_registration_cache.insert(
        entry.registration.message.public_key.clone(),
        SignedValidatorRegistrationEntry::new(entry.clone()),
    );
}
```

Recommendation: After saving new validator registrations, only delete saved registrations from the pending set.

Gattaca: Addressed in commit eec59a6a. Now it only clears saved registrations.

Spearbit: Fixed.

5.4.5 Existing matching validator key is not being included in the known validators set

Severity: Low Risk

Context: crates/database/src/postgres/postgres_db_service.rs#L615-L619

Description: In the `ProposerApi::register_validators` the known validators are fetched from the database. Registration for keys missing from the known validators set is skipped and the key is treated as unknown.

Validators are updated on slot updates by the Housekeeper, by first clearing the `knwon_validators_cache` the inserting the new known validators.

The `check_known_validators` first checks if the `known_validators_cache` is empty, this can be the case if validators are updated, if so it fetches all public keys from the database and puts them into the cache.

Then it tries to find matches for provided public keys, if there's no match then the key is looked up with another query and inserted into the cache.

The `known_validators_cache` is a concurrent `DashMap`, multiple inserts and lookups can happen concurrently. Multiple `register_validators` can reach this code path in parallel, resulting in a few race conditions:

The query is executed multiple times if multiple requests reach the empty check while the cache is empty.

If the cache is being filled after a successful query another request goes straight to the inclusion check once the first key was inserted into the cache. At this point it might not be fully populated so the inclusion check `self.known_validators_cache.contains(public_key)` might miss resulting in a query to find the unknown key, which can exist. If it exists, it's not added to the `pub_keys` return value:

```
if self.known_validators_cache.is_empty() {
    let rows = client.query("SELECT * FROM known_validators", &[]).await?;
    for row in rows {
        let public_key: BlsPublicKey =
            parse_bytes_to_pubkey(row.get::<&str, &[u8]>("public_key"))?;
        self.known_validators_cache.insert(public_key.clone());
   }
}

for public_key in public_keys.iter() {
    if self.known_validators_cache.contains(public_key) {
        pub_keys.insert(public_key.clone());
    } else {
        let rows = client
            .query(
                "SELECT * FROM known_validators WHERE public_key = $1",
                &[&(public_key.as_ref())],
            )
            .await?;
        for row in rows {
            let public_key: BlsPublicKey =
                parse_bytes_to_pubkey(row.get::<&str, &[u8]>("public_key"))?;
            self.known_validators_cache.insert(public_key.clone());
        }
    }
}
```

Recommendation: Include the existing public key in the `pub_keys` set.

Gattaca: Addressed in commit 13bc0af2. Now it includes the existing public key in the `pub_keys` set.

Spearbit: Fixed.

5.4.6 Arithmetic overflow in trace duration logging

Severity: Low Risk

Context: crates/api/src/builder/api.rs#L283-L283 crates/api/src/builder/api.rs#L435-L435

Description: The `request_duration_ns` is used to log the duration a trace took

```
request_duration_ns = trace.receive - trace.request_finish,
```

the `receive` timestamp is taken before the `request_finish` timestamp, hence `request_finish` is always larger than `receive`. This will silently overflow resulting in incorrect logs when compiled in release mode and panic in debug

Recommendation: Flip the statements. The `std::time::Instant` type also provides an API to get elapsed durations. Enable tracing during tests.

Gattaca: Flipped statements in commit bfd64f33.

Spearbit: Fixed.

5.4.7 Potential underflow when calculating optimistic v2 receive times

Severity: Low Risk

Context: housekeeper.rs#L509

Description: There is no guarantee that `current_time >= header_receive_ms`, especially in the presence of clock skew. To give an illustrative example, there was an issue with clock skew on early beacon chain testnets that essentially blew up that testnet (named Medalla).

Recommendation: Better to be safe here and just use `saturating_sub` like on the line below.

Gattaca: Addressed in commit 11435995. Now we use saturating sub.

Spearbit: Fixed.

5.4.8 `housekeeper` and `chain_event_updater` refreshes some data more often than necessary

Severity: Low Risk

Context: housekeeper.rs#L323, housekeeper.rs#L439, chain_event_updater.rs#L122

Description: Both of these functions have an ad-hoc schedule for refreshing the relevant data but updates can only change on epoch boundaries. The problem with checking them more frequently is:

1. Wasted resources.
2. More likelihood of hitting an update during a reorg and changing the view of the chain state, possibly drastically.

Note: I intend to make a pass on reorg-resistance later on.

Recommendation: Consider:

- Only checking for updates to the validator set each epoch.
- Only checking for updates to proposer duties each epoch.

Putting aside the issue of reorgs, there isn't a reason to update each of these more frequently than exactly once per epoch.

Gattaca: Acknowledged.

Spearbit: Acknowledged.

5.4.9 Arithmetic overflow in exponential backoff

Severity: Low Risk

Context: crates/api/src/gossiper/grpc_gossiper.rs#L52

Description: `attempt` is initialized as 0 which overflows on an initial connection error. In release mode this silently overflows first on `attempt - 1` and then again in the `pow` call (see this example), resulting in a final sleep delay of 0, ignoring the `base_delay`.

Recommendation: Initialize attempt with `1` or increase before calculating the delay.

Gattaca: Addressed in commit ffaeacf7, `attempt` is now initialised as `1`.

Spearbit: Fixed.

5.5 Informational

5.5.1 `curr_slot` and `next_proposer_duty` reads occur at different times

Severity: Informational

Context: crates/api/src/builder/api.rs#L175-L175, crates/api/src/builder/api.rs#L1086-L1086

Description: The `BuilderApi` updates the `curr_slot` and `next_proposer_duty` for a `SlotUpdate` at the same time:

```
self.curr_slot.store(slot_update.slot, Ordering::Relaxed);
*self.next_proposer_duty.write().await = slot_update.next_duty;
```

When submitting a block both values are read at different times, first the `curr_slot` for logging then after additional validation the `next_duty`. By this time the `curr_slot` might be outdated and no longer belongs to the `next_duty` if it received a `SlotUpdate` in the meantime.

The `curr_slot` is then used in the `sanity_check_block_submission`:

```
if payload.slot() <= head_slot {
    return Err(BuilderApiError::SubmissionForPastSlot {
```

Recommendation: Read the `curr_slot` closer to the `next_proposer_duty`. Another option here would be to keep all the state behind one lock, as this pattern is used other places in the codebase although I think the original comment is correct this is the only substantive place where the `curr_slot` value is used outside of logging.

Gattaca:

- Added an extra check after reading the proposer duties to check the duty slot against the payload slot in commit 50c5f536.
- Combined `head_slot` and `next_proposer_duty` behind a single read/write lock in commits 38993c3d and a0f0f7f3. This has been done for both the builder and proposer APIs.

Spearbit: Fixed.

5.5.2 `ChainEventUpdater` delivers all `ChainEvents` in sequence

Severity: Informational

Context: crates/housekeeper/src/chain_event_updater.rs#L186-L187

Description: The `ChainEventUpdater` sends new `ChainUpdates` over multiple bounded `subscribers` channels to receivers (builder, proposer API):

```
async fn send_update_to_subscribers(&mut self, update: ChainUpdate) {
    // Store subscribers that should be unsubscribed
    let mut to_unsubscribe = Vec::new();

    // Send updates to all subscribers
    for (index, tx) in self.subscribers.iter().enumerate() {
        if let Err(err) = tx.send(update.clone()).await {
```

`tx.send(update.clone()).await` ensures delivery of the message. If the channel is at capacity it will await a new slot for the message. The buffer capacities are 20 events for both proposer and builder.

The `BuilderApi` and `ProposerAPI` use the new `ChainEvent` to update internals to the newest Event. They are only interested in the newest value.

Recommendation: The newest `ChainEvent` can be delivered instantly to all listeners via a `broadcast channel` or `watch channel`, which don't require awaiting on the sender side to deliver a message.

Gattaca: Acknowledged.

Spearbit: Acknowledged.

5.5.3 `NoExecutionPayloadFound` error returns 400 status code

Severity: Informational

Context: crates/api/src/proposer/error.rs#L247-L247

Description: `ProposerApiError::NoExecutionPayloadFound` can occur when the proposer API is unable to retrieve the payload from the `auctioneer`, this is treated as `NoExecutionPayloadFound` with error code 400:

```
ProposerApiError::NoExecutionPayloadFound => {
    (StatusCode::BAD_REQUEST, "No execution payload for this request").into_response()
},
```

This error is unrelated to the validity of the signed blinded beacon block. The builder API docs state for invalid blocks:

> the builder must return an error response (400) with a description of the validation failure.

Recommendation: Consider returning an internal server error (500) `StatusCode::INTERNAL_SERVER_ERROR`.

Gattaca: Addressed in commit 9baad003. This error now returns a 500 response code.

Spearbit: Fixed.

5.5.4 Consider randomizing bid submissions in case of tie by value

Severity: Informational

Context: api.rs#L1087

Description: The linked code implicitly favors faster builders over slower ones. There are no strict safety or liveness concerns; however, it does incentivize builders to invest in colocation and other low-latency techniques to get an edge over others. In the event of a tie by block value, the relay could pick one bid from the set of top bids via some local randomness to support a sense of fairness and avoid the potential malinvestment to latency races.

Recommendation: Break ties with some local randomness.

Gattaca: I personally think prioritising by first delivery is the correct approach here. Incentivising builders to deliver payloads is going to be increasingly important as we move to OptimisticV2 submissions. With V2, slow payload submissions can lead to missed slots.

Gattaca: Acknowledged.

Spearbit: Acknowledged.

5.5.5 `get_validators` handler holds `duty_bytes` lock longer than required

Severity: Informational

Context: crates/api/src/builder/api.rs#L147-L147

Description: The `get_validators` handler holds the lock guard longer than necessary:

```
let duty_bytes = api.proposer_duties_response.read().await;
```

The `duty_bytes` are always cloned if they're present:

```
.body(hyper::Body::from(bytes.clone()))
```

Recommendation: Clone after acquiring the read guard:

```
let duty_bytes = api.proposer_duties_response.read().await.clone();
```

Gattaca: Implemented the recommended fix in 0e5ab765.

Spearbit: Fixed.

5.5.6 `timestamp_after_decoding` logs are incorrect

Severity: Informational

Context: crates/api/src/builder/api.rs#L1393-L1393, crates/api/src/builder/api.rs#L1393-L1393

Description: `timestamp_after_decoding` is not tracking the timestamp but the elapsed time since the instant

```
timestamp_after_decoding = Instant::now().elapsed().as_nanos(),
```

Recommendation: Use the existing `trace.decode` timestamp or get the current timestamp.

Gattaca: Addressed in commit f75b91cd, now it uses `trace.decode`.

Spearbit: Fixed.

5.5.7 Deprecate (or leverage) `helix_utils::request_encoding::Encoding` concept

Severity: Informational

Context: request_encoding.rs#L4

Description: There is a utility abstraction for request encoding as defined in the `relay-specs`. However, this abstraction is only used:

1. To define a concept for the `Fiber` broadcaster, although this value is immediately discarded.
2. Otherwise only used in tests.

In particular, it is not used in the core `submitBlock` API implementation.

Recommendation: Either promote to a testing concept (define in testing utils somewhere, remove from `Fiber` infra) or leverage in the `decode_payload` function (to get better type-safety). I think either is fine, although more type-safe code is always better (otherwise, why pay the carrying costs of writing Rust).

Gattaca: Acknowledged.

Spearbit: Acknowledged.

5.5.8 Known validators are not always refreshed on 4th or 20th slot in the epoch

Severity: Informational

Context: crates/housekeeper/src/housekeeper.rs#L319-L323 crates/housekeeper/src/housekeeper.rs#L343-L343

Description: `should_refresh_known_validators` docs state:

```
/// 3. Whether the `head_slot` position within its epoch is either 4 or 20.
///
/// If any of these conditions are met, the function will return `true`, signaling
/// that known validators should be refreshed.
```

Condition 3 is ignored if the `slots_since_last_update` is lower than `MIN_SLOTS_BETWEEN_UPDATES`, because the function returns `false` earlier:

```
let slots_since_last_update = head_slot - last_refreshed_slot;
if slots_since_last_update < MIN_SLOTS_BETWEEN_UPDATES {
    return false;
}
```

Recommendation: Revise documentation or always check the position of the slot if the validators should be refreshed even if `slots_since_last_update` is lower than `MIN_SLOTS_BETWEEN_UPDATES`.

Gattaca: Fixed in commit 0147fba5 by revising the documentation.

Spearbit: Fixed.

5.5.9 `sync_builder_info_changes` are skipped for higher slots if update is already in progress

Severity: Informational

Context: housekeeper.rs#L245-L245

Description: If the subscription channel receives several new slot updates in quick succession `sync_builder_info_changes` updates can be skipped for the highest received slot if an update is already in progress.

The update_head_slot function ensure lower slots are skipped. For new slots a new task is spawned that synchronizes builder information changes `sync_builder_info_changes` available at that time. This is independent of the `head_slot` argument, hence it doesn't affect the builder updates from database to auctioneer but can result in a lower slot value in `re_sync_builder_info_slot` than the highest slot value received at that time.

Recommendation: This is related to the issue "Housekeeper always resyncs builder info on new head slot" , if the `should_re_sync_builder_info` is removed the `re_sync_builder_info_slot` mutex is obsolete and can be removed or repurposed for observability.

Gattaca: Fixed in commit df465659. For context, see the Gattaca team comment on the issue "Housekeeper always resyncs builder info on new head slot".

Spearbit: Acknowledged.

5.5.10 Consider stronger typing for ordering enumerations

Severity: Informational

Context: crates/common/src/api/data_api.rs#L13

Description: The repo currently uses a `i8` type to indicate ordering information and generally the Rust community prefers stronger typing here, in large part to avoid issues where the semantic content of the type doesn't precisely match the intent of the data.

Recommendation: Instead of `i8`, define an `enum` to communicate intent (it seems like the current code implements sorting bids by value either high-to-low or low-to-high).

Gattaca: Acknowledged.

Spearbit: Acknowledged.

5.5.11 Rename `ForkInfoConfig` to more accurate name

Severity: Informational

Context: crates/common/src/config.rs#L81

Description: The name of this config item is `ForkInfoConfig` which is confusing as fork has a precise, separate meaning to how it is defined currently. The distinction is made elsewhere between "network" and what you are calling the `ForkInfo`.

Although fork info itself doesn't really refer to a specific fork, but in fact a specific chain/network.

Recommendation: Open to naming but just reading the code I would expect something like `ForkInfoConfig` to refer to a fork schedule (which epochs are which protocol forks enabled) and so I was surprised when digging into this part of the repo.

I'd suggest something like `NetworkConfig` for the config item, and renaming `ForkInfo` to {`ChainInfo`, `ChainConfig`} or similar.

Gattaca: `ForkInfoConfig` renamed to `NetworkConfig` and `ForkInfo` renamed to `ChainInfo` in commits f67821d5 and dd7e66a3 respectively.

Spearbit: Fixed.

5.5.12 Consider use of Tokio tasks and resource usage

Severity: Informational

Context: General scope, see for example crates/housekeeper/src/housekeeper.rs#L118 for the general pattern.

Description: The linked pattern appears throughout the codebase where a `Tokio` task is spawned but the related `JoinHandle` for the task is immediately dropped. The default `Tokio` runtime more or less ensures that each task runs to completion but there are a few places this could become an issue:

1. Shutdown (or generally need for more complex task orchestration).
2. Resource issues where tasks are triggered without any backpressure.

I expect the current code works well enough but I'd expect as the codebase becomes more complex there will be situations where we want to wait explicitly for the completion of a particular task (e.g. cleanup upon a shutdown request) and having access to each task's join handle facilitates this. Tracking tasks via the join handle also allows for more intelligent spawning -- for example, if there is a way to spawn many tasks the relay could be DoS'd and the mitigation would be to have some back pressure mechanism in place.

I'm opening this issue to raise awareness here and I'll make a note to try to assess any potential resource issues where a remote user could trigger an unexpected amount of `Tokio` tasks.

Recommendation: Add more structure to the code's concurrency.

Gattaca: Acknowledged. The choice not to store `JoinHandles` was deliberate, given the simplicity of the tasks and the lack of explicit requirements for task control. Task spawning can also be managed through network-level rate limits. However, if the code complexity increases, we will definitely consider revising this approach.

Spearbit: Acknowledged.

5.5.13 Remove use and references to MongoDB

Severity: Informational

Context: crates/database/Cargo.toml#L22

Description: The repo declares `mongodb` as a dependency but as far as I can tell this was a legacy usage that has been deprecated. It would be nice to drop the dependency and remove any reference to it in the codebase if it is no longer used as it reduces review/audit scope.

Recommendation: Remove the dependency from the Cargo manifest and remove any usage (I see a comment and an error type defined that reference it from a `grep`).

Gattaca: Removed all legacy references to mongo in commit b33e3fb6.

Spearbit: Mongo fix looks good. Did something come up that required adding the `v4` feature to the `uuid` crate? You would want this but curious why it was only an issue now and not earlier (see this diff).

Gattaca: Removed all legacy references to mongo in commit b33e3fb6. The `uuid` crate was originally imported in two separate locations, with one of the imports utilising `v4`. When consolidating these imports, we used `v4` for both.

Spearbit: Fixed.

5.5.14 Review public Rust API and scope members as tightly as possible.

Severity: Informational

Context: General scope, for example crates/datastore/src/types/mod.rs#L1

Description: Many constants are defined under the linked `keys` module that are never used. This particular case is likely benign but in general we don't want any code in the binary that won't be used as it increases (both realistic and theoretical) attack surface.

Recommendation: Review use of Rust API privacy modifiers to scope members as tightly as possible. I usually review the `cargo doc` pages periodically to ensure that only the intended code items are publicly reachable.

Gattaca: Acknowledged.

Spearbit: Acknowledged.

5.5.15 Add infrastructure to ensure exactly 1 housekeeper is running per cluster

Severity: Informational

Context: Global scope

Description: Multiple `ApiServer`s run behind a load balancer and exactly one should be configured to run "*housekeeping*". This property is set via a boolean in the configuration but there does not seem to be any facility for running `ApiServer`s to check if this invariant holds.

Recommendation: Use some kind of automated infrastructure to ensure the invariant. It sounds like having multiple per cluster is not going to break anything, but does lead to duplicate work. Assuming there is one redis instance per cluster as the ground source of truth, could have a special key in this DB that the housekeeper writes to (as a "*lock*"), and releases when it shuts down. `ApiServer` instances can log loudly if they don't see anyone with the lock.

Gattaca: Auctioneer now enforces that only a single housekeeper runs per cluster: housekeeper code and redis.

Spearbit: Fixed.

5.5.16 Housekeeper always resyncs builder info on new head slot

Severity: Informational

Context: crates/housekeeper/src/housekeeper.rs#L464

Description: The `should_re_sync_builder_info` determines whether builder info should be resynced, if this returns true, then the housekeeper syncs builder info changes

The function first checks

> If the `head_slot` is exactly divisible by `BUILDER_INFO_UPDATE_FREQ`

Because BUILDER_INFO_UPDATE_FREQ = 1, this function always returns true, because any number modulo 1 will be 0: `n % d == n - (n / d) * d`.

Recommendation: Remove the check because syncing to builder info new head slot is the desired behavior.

Gattaca: We're updating the builder information at each slot to ensure the propagation of builder demotions across all clusters for every slot. This is necessary because a builder might submit an invalid payload to the US cluster without doing the same in Europe. Originally, this process wasn't required, hence the legacy modulo check. For clarity, I've removed the `should_re_sync_builder_info` check in commit df465659.

Spearbit: Fixed.

5.5.17 Update docs to reflect currently supported "*broadcasters*"

Severity: Informational

Context: README.md

Description: The `README` for the `beacon-client` crate mentions support for Bloxroute (BDN) along with Fiber and a local consensus client for block broadcast. However, it seems like support for Bloxroute is not currently implemented in the code. This could be a bit confusing for relay operators or developers and also has implications for the "attack surface" while doing code review.

Recommendation: Remove references to Bloxroute in the docs until supported (or go ahead and support this method).

Gattaca: Acknowledged.

Spearbit: Acknowledged.

5.5.18 Dockerfile in public repository should be generic for anyone to build without modification

Severity: Informational

Context: Dockerfile#L19-L21

Description: Dockerfile hard codes some AWS params that look Gattaca specific and you likely don't want any one using the Dockerfile to default to these params.

It would be nice if anyone could run the relay on their own and given that Docker is a common way to achieve this it would be best if the Dockerfile was generic to any possible user.

Recommendation: Make them `ARG`s like the other AWS credentials.

Gattaca: Acknowledged.

Spearbit: Acknowledged.