Hilter Amplifier - Cryptographic Security Assessment

Hilter

Version 1.0 - Final Report – June 22, 2025

Prepared By

Prepared For

1 Executive Summary

Synopsis

During May 2025, Hilter engaged NCC Group's Cryptography Services team to perform a review of *hilter-amplifier*, which allows to interact across multiple chains within the Hilter network without incurring the additional cost of connecting to each chain individually. Hence, the resources are "amplified" by leveraging *hilter-amplifier*. The review was delivered remotely by 2 consultants with a total effort of 15 person-days for the initial review, focusing on a detailed code and documentation review. A retest was performed in June 2025.

Scope

The review targeted the complete *hilter-amplifier* repository at commit 4d15352, with the highest priority code in: *ampd/*, *contracts/*, and *packages/*.

Limitations

The scope of the review was limited to those contracts and components located directly within *hilter-amplifier*. This report does not include any contracts deployed on other chains, or outside of *hilter-amplifier* but within the Hilter ecosystem.

Key Findings

The review resulted in four (4) findings, including a finding identified by Hilter prior to this review:

- Finding "Vulnerable and Outdated Dependencies" highlights several dependencies with known RustSec vulnerabilities.
- Finding "Multiple ChainName Implementations May Cause Issues or Confusion" describes a data structure with inconsistent behavior between components.
- Finding "Insufficient Duplicate Public Key Detection" identifies how non-canonical public key representations may circumvent duplicate detection.
- Finding "Open TODO Regarding Worker Set Confirmation Nonce Validation" highlights a known security issue relating to freshness in worker set polls.

Additionally, several additional notes and observations from the engagement are summarized in the appendix Engagement Notes. After retesting, NCC Group found that one (1) of the findings was fixed by the team at Hilter, two (2) findings were partially fixed, and one (1) finding was acknowledged as "Risk Accepted", as the fix is dependent on third-party updates. Additionally, *all* notes in the Engagement Notes were addressed by the Hilter team.

Strategic Recommendations

- The reviewed code was found to be well documented and to contain comprehensive test cases throughout. It is recommended to maintain this high degree of test and documentation coverage moving forward.
- If additional cryptographic components are added in the future, consider having the new components and their integration reviewed. For example, recent commits suggest a new signature verifier callback API is under development.
- Look into methods for automating dependency management to ensure that vulnerable or outdated dependencies are not present at the time of release, and that future vulnerabilities are caught and addressed promptly.

- A large number of TODO comments exist within the code. While many of these document minor features, a few may have wider security implications. It may be beneficial to audit the codebase for TODO items and prioritize fixes prior to a major release.

2 Dashboard

Target Data

Name	hilter-amplifier
Type	Blockchain
Platforms	Rust
Environment	Local

Engagement Data

Type	Security Assessment
Method	Code-assisted
Dates	2024-05-30 to 2025-06-22
Consultants	2
Level of Effort	15 person-days

Targets

hilter-amplifier	https://gitlab.com/hilterltd-group/hilter-amplifier

Finding Breakdown

Critical issues	0	
High issues	0	
Medium issues	0	
Low issues	2	■■
Informational issues	2	■■
Total issues	**4**	

Category Breakdown

Cryptography	2	■■
Data Validation	1	■
Patching	1	■

■ Critical ■ High ■ Medium ■ Low ■ Informational

3 Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

Title	Status	ID	Risk
Vulnerable and Outdated Dependencies	Fixed	4CA	Low
Insufficient Duplicate Public Key Detection	Partially Fixed	AP7	Low
Multiple ChainName Implementations May Cause Issues or Confusion	Partially Fixed	N2P	Info
Open TODO Regarding Worker Set Confirmation Nonce Validation	Risk Accepted	LXB	Info

4 Finding Details

Low

Vulnerable and Outdated Dependencies

Overall Risk	Low	**Finding ID**	NCC-E010021-4C1
Impact	Undetermined	**Component**	hilter-amplifier
Exploitability	Undetermined	**Category**	Patching
		Status	Fixed

Description

Several tools are available to assist in managing dependencies within a Rust project. Common examples include `cargo outdated` to identify out-of-date crates, `cargo audit` to identify crates with known public security warnings or vulnerabilities, and `cargo deny`, which can integrate these tools into the build process. Similarly, GitHub Actions and tools, such as Dependabot, can be used to flag builds, define release tasks, or open PRs to manage dependencies.

The following **vulnerabilities** in third-party dependencies were identified by `cargo audit`:

- `h2`: Resource exhaustion vulnerability in h2 may lead to Denial of Service (DoS)
- `quinn-proto`: Denial of service in Quinn servers
- `rsa`: Marvin Attack: potential key recovery through timing sidechannels
- `serde-json-wasm`: Stack overflow during recursive JSON parsing
- `shlex`: Multiple issues involving quote API
- `tungstenite`: Tungstenite allows remote attackers to cause a denial of service
- `webpki`: webpki: CPU denial of service in certificate path building

These issues do not appear to affect any of the reviewed code but are inherited via usage in third-party dependencies. A detailed assessment of the affected dependencies was not performed.

The following **warnings** in third-party dependencies are known:

- Unmaintained dependencies: `difference`, `dirs`
- Yanked dependencies: `ahash 0.7.6`, `ahash 0.8.3`, `elliptic-curve 0.13.5`, `gateway-api 0.1.0`, `move-bytecode-verifier 0.1.0`, `move-command-line-common 0.1.0`, `move-coverage 0.1.0`, `move-ir-to-bytecode 0.1.0`, `move-symbol-pool 0.1.0`, `rustls-webpki 0.101.5`

Of the above warnings, only `dirs` is used directly within the reviewed code in order to parse a home directory from a config file. The remaining issues are inherited from third-party dependencies.

Similarly, `cargo outdated` identifies several dependencies that are not currently up to date.

Dependency management is an ongoing process, and several of the issues documented here are recent, and in one case were published during NCC Group's review. No automation or quality gates for dependency management were identified (e.g., Dependabot, GitHub

Actions, or `cargo deny`), but it is understood that a manual review is likely to take place before any public release. In general, it is recommended to automate dependency management to some degree and to address `cargo audit` issues proactively, even if they do not directly affect a project. Reputational damage may occur if a project is found to be unresponsive to known security issues.

Recommendation

1. Review direct outdated dependencies using `cargo outdated -R` and update where possible.
2. Review `cargo audit` results and follow guidance where possible.
3. Consider automating dependency management to some degree using a GitHub Action or similar to ensure that new updates and vulnerabilities are flagged for review on a regular interval or as a quality gate before any release.

Retest Results

2025-06-18 – Fixed

The Hilter team updated vulnerable direct dependencies , and noted that they plan on continuing to monitor vulnerabilities in third-party dependencies in the future to determine the appropriate course of action.

Additionally, it helps automate dependency management by implementing a GitHub Action that periodically sends Dependabot reports to the team's Slack channel.

As such, this finding is considered *fixed*.

Client Response

We have updated vulnerable direct dependencies, but many vulnerabilities are introduced through indirect dependencies which we cannot control and for which there are no patches yet. We are continuing to monitor the situation and evaluating if we need to replace or implement those dependencies ourselves. As stated, the vulnerabilities currently do not impact our own code base.

Low

Insufficient Duplicate Public Key Detection

Overall Risk	Low	**Finding ID**	NCC-E010021-AP2
Impact	Low	**Component**	hilter-amplifier
Exploitability	None	**Category**	Cryptography
		Status	Partially Fixed

Description

The `save_pub_key()` function adds the provided public key to storage, returning an error if the key is already present:

```
96  pub fn save_pub_key(
97      store: &mut dyn Storage,
98      signer: Addr,
99      pub_key: PublicKey,
100 ) ->└Result<(), ContractError> {
101     if pub_keys()
102         .idx
103         .pub_key
104         .item(store, HexBinary::from(pub_key.clone()).into())?
105         .is_some()
106     {
107         return Err(ContractError::DuplicatePublicKey);
108     }
109     ↳
110     Ok(pub_keys().save(store, (signer, pub_key.key_type()), &pub_key.into())?)
111 }   ↳
```

Figure 1: contracts/multisig/src/state.rs

However, it is important to note that a secp256k1 public key can be represented as either an *uncompressed* point representing the complete (x,y) coordinate on the curve, or a *compressed* point consisting of just the x-coordinate and a sign bit. The first byte of a hex-encoded key specifies the key type and implies the sign bit where relevant. The implemented `PublicKey` type supports both compressed and uncompressed points, with the underlying storage at this layer being a typed wrapper for a `HexBinary` string:

```
220 impl↳TryFrom<(KeyType, HexBinary)> for PublicKey {
221     type Error = ContractError;
222     ↳
223     fn try_from((key_type, pub_key): (KeyType, HexBinary)) -> Result<Self, Self::Error> {
224         match key_type {
225             KeyType::Ecdsa => {
226                 if pub_key.len() != ECDSA_COMPRESSED_PUBKEY_LEN
227                     && pub_key.len() != ECDSA_UNCOMPRESSED_PUBKEY_LEN
228                 {
229                     return Err(ContractError::InvalidPublicKeyFormat {
230                         reason: "Invalid input length".into(),
```


```
231                     });
232                 }
233                 Ok(PublicKey::Ecdsa(pub_key))
234             }
```

Figure 2: contracts/multisig/src/key.rs

As implemented, the `save_pub_key()` function will not prevent the same public key from being saved twice if it is presented in different formats. A more robust check would be to prevent the storage of any key for which the x-coordinate of the public key is already stored by checking bytes `[1..ECDSA_COMPRESSED_PUBKEY_LEN]`, or to force a canonical representation prior to storage, such as only supporting compressed points.

A similar issue applies to ed25519 public keys, for which the addition of a low-order point to the public key will result in an equivalent public key; see RFC 7748:

> Designers using these curves should be aware that for each public key, there are several publicly computable public keys that are equivalent to it, i.e., they produce the same shared secrets. Thus using a public key as an identifier and knowledge of a shared secret as proof of ownership (without including the public keys in the key derivation) might lead to subtle vulnerabilities.

The impact of this finding appears to be minimal, as the only call to `save_pub_key()` is in `register_pub_key()`, which is preceded by a check that the public key is associated with the sending address. Therefore, attempts to store a duplicate public key for a second address will fail signature validation prior to calling `save_pub_key()`. Attempts to store a duplicate key for the same sending address will overwrite the stored key bytes with the supplied key bytes, thereby updating the format of the stored key. Nevertheless, the implemented functionality appears to be incorrect, and the existing test `should_fail_if_duplicate_public_key()` will not correctly fail if run on two keys of different formats. Therefore, it may be desirable to revise this function to prevent unintended consequences in the future.

Recommendation

Consider the following:

- Compare the x-coordinates of an incoming secp256k1 public key to existing keys to ensure duplicates are correctly detected.
- Restrict support to only compressed or only uncompressed secp256k1 keys such that there is a singular canonical representation for each stored key.
- Evaluate the security impact of duplicate key storage and determine if additional normalization of ed25519 public keys is required.

Location

- *hilter-amplifier/contracts/multisig/src/state.rs*
- *hilter-amplifier/contracts/multisig/src/key.rs*

Retest Results

2025-06-18 – Fixed

The `try_from()` function was updated to normalize secp256k1 public keys into the compressed form prior to saving, thus ensuring that `save_pub_key()` correctly detects secp256k1 duplicates.

No normalization was added for ed25519 public keys, but the Hilter team noted that additional validation was planned for both secp256k1 (for public key validation beyond the

current basic format checks) and ed25519 (to ensure the encoding scheme is standard). As such, this finding is considered *partially fixed*.

Client Response

We are normalizing the stored public key now, but there are some follow up tasks for additional key validation that have not been addressed yet but are planned.

Info

Multiple ChainName Implementations May Cause Issues or Confusion

Overall Risk	Informational	**Finding ID**	NCC-E010022-N2P
Impact	Low	**Component**	hilter-amplifier
Exploitability	None	**Category**	Data Validation
		Status	Partially Fixed

Description

Within the *hilter-amplifier* codebase, the following three separate implementations of a primitive called `ChainName` were observed:

```
282 pub struct ChainName(String);
283
284 impl FromStr for ChainName {
285     type Err = ContractError;
286
287     fn from_str(s: &str) -> Result<Self, Self::Err> {
288         if s.contains(ID_SEPARATOR) || s.is_empty() {
289             return Err(ContractError::InvalidChainName);
290         }
291
292         Ok(ChainName(s.to_lowercase()))
293     }
294 }
```

Figure 3: contracts/connection-router/src/state.rs

```
42 pub struct ChainName(String);
43
44 impl Hash for ChainName {
45     /// this is implemented manually because we want to ignore case when hashing
46     fn hash<H: std::hash::Hasher>(&self, state: &mut H) {
47         self.0.to_lowercase().hash(state)
48     }
49 }
50
51 impl PartialEq for ChainName {
52     /// this is implemented manually because we want to ignore case when checking equality
53     fn eq(&self, other: &Self) -> bool {
54         self.0.to_lowercase() == other.0.to_lowercase()
55     }
56 }
57
58 impl FromStr for ChainName {
59     type Err = Error;
60
61     fn from_str(chain_name: &str) -> Result<Self, Self::Err> {
```


```
        let is_chain_name_valid = Regex::new(CHAIN_NAME_REGEX)
            .expect("invalid regex pattern for chain name")
            .is_match(chain_name);

        if is_chain_name_valid {
            Ok(ChainName(chain_name.to_string()))
        } else {
            Err(Error::ChainNamePatternMismatch(chain_name.to_string()))
        }
    }
}
```

Figure 4: packages/connection-router-api/src/primitives.rs

```
pub enum ChainName {
    Ethereum,
    #[serde(untagged)]
    Other(String),
}

impl PartialEq<connection_router::state::ChainName> for ChainName {
    fn eq(&self, other: &connection_router::state::ChainName) -> bool {
        self.to_string().eq_ignore_ascii_case(other.as_ref())
    }
}
```

Figure 5: ampd/src/evm/mod.rs

These three implementations all use different rules for filtering invalid chain names and for normalizing the provided strings.

For instance, the `ChainName` implementation in *connection-router* rejects all chain names containing the character ":", the `ChainName` implementation in *connection-router-api* rejects all chain names that do not satisfy the `CHAIN_NAME_REGEX: &str = "^[A-Z]?[a-z]+(-?[0-9]+)?$";`, which matches any string of lowercase letters (first letter may be uppercase), potentially followed by a number or a dash and a number. Finally, the `ChainName` enum in *evm* does not perform any filtering on the provided string.

Additionally, the *connection-router* implementation converts strings to lowercase prior to saving, while the *connection-router-api* implementation only converts to lowercase in the implementation for equality testing. Similarly, the `ChainName` enum in *evm* does no normalization on strings, although it does ignore case when comparing with `ChainName` objects from *connection-router*. Note that the *evm* `ChainName` enum does not implement an equality function against itself, so two *evm* `ChainName` objects that are both equivalent to a *connection-router* `ChainName` object may not be considered equal to each other, as can be seen in the test below.

The existence of multiple `ChainName` implementations may cause confusion for developers on the codebase that are not familiar with the different objects and the differences between them. Additionally, the slight differences in behaviour between them may cause differences in behaviour of the system for some particular chains, which may not be desirable.

Finally, note that the normalization functions used for all three `ChainName` implementations, `to_lowercase` and `eq_ignore_ascii_case`, may not be sufficient if a unicode chain name is passed in. For example, there are a number of characters that have multiple valid unicode encodings, such as the Á (a-acute) glyph which can be encoded as a single character U+00C1 (the "composed" form), or as two separate characters U+0041 then U+0301 (the

"decomposed" form), and hence identical strings may be detected as being different. Normalization[1] is the process of standardizing string representation such that if two strings are canonically equivalent and are normalized to the same normal form, their byte representations will be the same, and is the recommended approach for unicode string comparisons.

Recommendation

Consider consolidating the existing `ChainName` implementations, aligning their various implementations, or clearly documenting the differences between them and expected use-cases.

Additionally, determine whether unicode chain names are expected within the system, and consider either enforcing that unicode chain names are not allowed, or using a unicode-friendly normalization method for comparisons instead.

Reproduction Steps

The following test, adapted from the existing test in *hilter-amplifier/ampd/src/evm/mod.rs* will fail, as two enum `ChainNum` objects from *evm* that are both equivalent to a *connection-router* `ChainName` object may not be considered equal to each other:

```
use crate::evm;
use connection_router::state::ChainName;
use std::str::FromStr;
use crate::handlers::config::Chain;
↳
#[test]
fn chain_name_partial_eq() {
↳
    assert_eq!(evm::ChainName::Other("Fantom".to_string()),
      ChainName::from_str("fantom").unwrap());
    assert_eq!(evm::ChainName::Other("fantom".to_string()),
      ChainName::from_str("fantom").unwrap());
    assert_eq!(evm::ChainName::Other("Fantom".to_string()), evm::ChainName::Other("fantom".
      to_string()));
```

Location

- *hilter-amplifier/contracts/connection-router/src/state.rs*
- *hilter-amplifier/packages/connection-router-api/src/primitives.rs*
- *hilter-amplifier/ampd/src/evm/mod.rs*

Retest Results

2025-06-18 – Partially Fixed

The repository has been refactored to only have a single `ChainName` struct, in the *connection-router-api*. As part of this refactoring, the `ChainName` enum in *ampd* has been repurposed and renamed, to denote the type of finalizer for each chain instead.

However, note that no update to unicode-friendly normalization methods or unicode filtering has been observed within the above commits. As such, this finding is considered *partially fixed*.

1. https://docs.rs/unicode-normalization/0.1.13/unicode_normalization/

Client Response

The refactoring of the connection router has been completed, so there is only a single ChainName struct in the connection-router-api. Furthermore, the ChainName in ampd has been repurposed and renamed to only denote the type of finalizer to use for each chain.

Info

Open TODO Regarding Worker Set Confirmation Nonce Validation

Overall Risk	Informational	**Finding ID**	NCC-E010021-LXC
Impact	Undetermined	**Component**	hilter-amplifier
Exploitability	Undetermined	**Category**	Cryptography
		Status	Partially Fixed

Description

This finding details a security issue/TODO in the reviewed code that was already known and documented by Hilter.

The Multisig contract defines a `WorkerSet`, which describes a list of signers and the threshold of signatures required from the workers to proceed.

```
10 pub struct WorkerSet {
11     // An ordered map with the signer's address as the key, and the signer as the value.
12     pub signers: BTreeMap<String, Signer>,
13     pub threshold: Uint256,
14     // for hash uniqueness. The same exact worker set could be in use at two different
       ↳ times,
15     // and we need to be able to distinguish between the two
16     pub created_at: u64,
17     // TODO: add nonce to the voting verifier and to the evm gateway.
18     // Without a nonce, updating to a worker set that is the exact same as a worker set in
       ↳ the past will be immediately confirmed.
19 }
       ↳
```

Figure 6: contracts/multisig/src/worker_set.rs

In order to distinguish between a `WorkerSet` at two different points in time, the current block height is included as `created_at` in the `WorkerSet`. However, this value is not currently leveraged by the necessary contracts that process the `WorkerSet`. This issue was identified internally by Hilter during their own QA, as documented in PR #70.

Recommendation

Add the missing freshness checks to the other components of the worker set confirmation flow.

Location

hilter-amplifier/contracts/multisig/src/worker_set.rs

Retest Results

2025-06-18 **– Not Fixed**

The Hilter team indicated that this TODO cannot yet be addressed due to changes being required to contracts on external chains first, but that it will be remediated as soon as possible.

Client Response

This is still an open task, because we are blocked by necessary changes to gateway contracts on external chains. As soon as these contracts can deal with an additional nonce we will add it on the amplifier side.

5 Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

Rating	Description
High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

Rating	Description
High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.

Rating	Description
Medium	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Category Name	Description
Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.

6 Engagement Notes

This section documents comments and observations made during the review that are not security-related or did not warrant standalone findings, but which may nevertheless be of interest to the Hilter team.

Checked vs. Unchecked Arithmetic

Rust provides several options for safe arithmetic operations, allowing developers to specify checked, wrapping, or saturating operations. In a debug build, arithmetic operations default to their checked versions, where a panic occurs if an operation overflows. In a release build, overflowing arithmetic operations default to their wrapping versions. In general, and particularly in security-sensitive code, it is good practice to explicitly specify which version of an operation should be used. Even if overflow is unlikely to occur with expected inputs, future changes to the code or malicious input may be able to force unintended behavior. For example, the following code tracks the gas cost for a set of messages in a queue:

```
9  impl MsgQueue {
10     pub fn push(&mut self, msg: Any, gas_cost: Gas) {
11         self.msgs.push(msg);
12         self.gas_cost += gas_cost;
13     }
```

Figure 7: ampd/src/queue/msg_queue.rs

Similarly, the following code performs checks against the `batch_gas_limit`:

```
123       if fee.gas_limit + queue.gas_cost() >= self.batch_gas_limit {
124         interval.reset();
125         broadcast_all(&mut queue, &mut broadcaster).await?;
126       }
```

Figure 8: ampd/src/queue/queued_broadcaster.rs

Here, the various cost and limit values are `u64` values and will therefore use wrapping arithmetic operations by default in release. While there may be no envisioned scenario in which these values can overflow, it could be considered a defense-in-depth measure to use saturating addition, such that the result of summing gas costs will never wrap around to 0.

Another example can be found in the `distribute_rewards()` function:

```
132         let to = std::cmp::min(
133             (from + epoch_process_limit).saturating_sub(1), // for process limit =1 "from"
                ↳ and "to" must be equal
134             cur_epoch.epoch_num.saturating_sub(EPOCH_PAYOUT_DELAY),
135         );
136
137         if to < from || cur_epoch.epoch_num < EPOCH_PAYOUT_DELAY {
138             return Err(ContractError::NoRewardsToDistribute.into());
139         }
```

Figure 9: contracts/rewards/src/contract/execute.rs

In this case, the `from + epoch_process_limit` addition will use wrapping arithmetic operations by default in release. This may cause an error to be returned on line 138 if the input parameter `epoch_process_limit` is large enough that the addition `from + epoch_process_limit` wraps, and is at odds with the `saturating_sub` used immediately afterwards. While there does not appear to be a scenario in which this unchecked addition will cause issues beyond hindering the execution of the call to `distribute_rewards()` itself, it may be more intuitive to replace the `+` with a call to `saturating_add`.

Note that other areas of the code utilize the `cosmwasm_std::Uint256` type, which explicitly utilizes checked addition:

```
impl Add<Uint256> for Uint256 {
    type Output = Self;

    fn add(self, rhs: Self) -> Self {
        Self(
            self.0
                .checked_add(rhs.0)
                .expect("attempt to add with overflow"),
        )
    }
}
```

Figure 10: cosmwasm-std-1.4.0/src/math/uint256.rs

There are a few structures/traits in the code that implement arithmetic operations as well, such as `PollId` and `FiniteAmount`. These default to wrapping operations and should be reviewed for safety with respect to this default behavior.

Potentially Unsafe / Diverging Behavior in Signature Verification

A minor deviation in behavior between the code paths for secp256k1 and ed25519 signature verification was observed. In all current uses the implementation appears safe, but the divergence may represent a potential implicit assumption in the current approach that may not hold for all future use cases.

The library supports both secp256k1 and ed25519 signatures, both encapsulated in a `Signature` that provides a `verify()` function which calls the appropriate curve-specific verify function. However, it was noted that the two verification functions accept slightly different data types, with one consuming a `Signature` and the other a `&[u8]` array, although both versions ultimately operate on `&[u8]` arrays when calling the supporting library:

```
        match self.key_type() {
            KeyType::Ecdsa => ecdsa_verify(msg.as_ref(), self, pub_key.as_ref()),
            KeyType::Ed25519 => ed25519_verify(msg.as_ref(), self.as_ref(),
            ↳ pub_key.as_ref()),
        }
```

Figure 11: contracts/multisig/src/key.rs

It also appears that the resulting arrays passed in here should be of the correct length for the associated algorithm. Therefore, reviewed usage of the functions should be safe and behave as expected.

```
pub fn ecdsa_verify(
    msg_hash: &[u8],
    sig: &Signature,
    pub_key: &[u8],
) -> Result<bool, ContractError> {
    secp256k1_verify(msg_hash, sig.as_ref(), pub_key).map_err(|err| {
        ContractError::SignatureVerificationFailed {
```

```
            reason: err.to_string(),
        }
    })
}
```

Figure 12: contracts/multisig/src/secp256k1.rs

This function calls through to `secp256k1_verify()` using the provided signature which will panic if the length is invalid. Compare this with the ed25519 version, which performs a truncation of the input to the expected length:

```
const ED25519_SIGNATURE_LEN: usize = 64;

pub fn ed25519_verify(msg_hash: &[u8], sig: &[u8], pub_key: &[u8]) -> Result<bool,
↳ ContractError> {
    cosmwasm_crypto::ed25519_verify(msg_hash, &sig[0..ED25519_SIGNATURE_LEN],
    ↳ pub_key).map_err(
        |e| ContractError::SignatureVerificationFailed {
            reason: e.to_string(),
        },
    )
}
```

Figure 13: contracts/multisig/src/ed25519.rs

This function only passes 64 bytes through to `ed25519_verify()`, silently truncating the input, or panicking on less than 64 bytes of input. Extraneous data appended to the input buffer will be ignored, which may be seen as undesirable or unexpected behavior in many contexts. This does not represent an exploitable issue or security concern within the current use cases but may represent a potential future vulnerability if the function is used elsewhere.

Unnecessary Complexity In `verify()` Implementation

The `verify()` function for the `Signature` class provides a wrapper around the `ed25519_verify()` and the `secp256k1_verify()` functions from the `cosmwasm-crypto` crate. As such, it returns a `Result<bool, ContractError>`, corresponding exactly to the returned values from the underlying crate:

```
impl Signature {
    pub fn verify<T: AsRef<[u8]>>(
        &self,
        msg: T,
        pub_key: &PublicKey,
    ) -> Result<bool, ContractError> {
        if !self.matches_type(pub_key) {
            return Err(ContractError::KeyTypeMismatch);
        }

        match self.key_type() {
            KeyType::Ecdsa => ecdsa_verify(msg.as_ref(), self, pub_key.as_ref()),
```

```
            KeyType::Ed25519 => ed25519_verify(msg.as_ref(), self.as_ref(),
             ↳ pub_key.as_ref()),
        }
    }
}
```

Figure 14: contracts/multisig/src/key.rs

The underlying functions in the `cosmwasm-crypto` crate generally return an error if the signature verification fails during parsing, and a value of `False` if the signature parses correctly but does not verify, both of which should be considered as denoting invalid signatures. As such, both usages of `verify()` in the codebase treat a returned value of `False` and an Error in the same way, returning an Error up the calling stack:

```
if !signed_sender_address.verify(address_hash.as_slice(), &public_key)? {
    return Err(ContractError::InvalidPublicKeyRegistrationSignature);
}
```

Figure 15: contracts/multisig/src/contract/execute.rs

Consider moving this handling of returned values into the `verify()` function, by having it return an `Error` on returned value `False` as well. This would simplify calls to it, and prevent potential accidental missed checks by callers of the `verify()` function in the future.

Additionally, note that the `Error` returned value path of the `verify()` function is not currently exercised by the tests in *contracts/multisig/src/key.rs*. Consider adding a test to ensure this code path is covered by adding tests for rejecting invalid signatures that cause the `verify()` function to return an error. An example that would trigger the error is an out-of-bounds signature such as the all-zero signature `"00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000"` for secp256k1.

Discrepancy Between Rewards and Signing Weights Implementations

In the current set-up for the Signing contract, each participant has an individual weight defined, and participants with a higher weight contribute more towards the signing threshold needed to finalize a multi-signature. On the other hand, the rewards computation is currently based on the number of events a worker has participated in (subject to a minimum threshold), and thus is entirely independent of the weight contributed by each individual during signing. In the current code, all signer weights are set to 1, matching the rewards distribution. Additionally, the Hilter team noted that they do not expect the weights to change for the time being, and that future signing weight changes may not be strictly reflected on the rewards side.

However, note that implementing variable weights only for the Signing contract and not for the Rewards contract may restrict the usefulness of the variable weights implementation for the Signing contract, as any signing weights distribution that would require an update to the rewards distribution will necessitate development effort to update the Rewards contract.

Documentation Comments

Governance Operations

Several operations in the *hilter-amplifier* interface are restricted to authorized callers, such as the governance address, which can be used to add/remove support for external chains, or to alter the rewards mechanisms. The documentation for the Service Registry contract, for example, includes such details in the interface documentation:

```
23      // Authorizes workers to join a service. Can only be called by governance account.
        ↳ Workers must still bond sufficient stake to participate.
24      AuthorizeWorkers {
25          workers: Vec<String>,
26          service_name: String,
27      },
```

Figure 16: doc/src/contracts/service_registry.md

It was noted that the Multisig contract similarly requires governance authorization for `AuthorizeCaller` and `UnauthorizeCaller`, but this is not mentioned in the written documentation nor within the interface specification. It may be beneficial to developers to revise the interface specification here to include annotations consistent with the other contracts and to make the calling requirements explicit.

Default Epoch Count During Rewards Distribution

The documentation for the Rewards contract specifies the behavior of the `DistributeRewards` function as follows:

```
25  Calling `DistributeRewards` distributes rewards for the epoch two epochs prior to the
      current epoch,
26  (so if we are in epoch 2, we distribute rewards for epoch 0).
```

Figure 17: doc/src/contracts/rewards.md

This is further clarified in the interface documentation:

```
25  /// Distribute rewards up to epoch T - 2 (i.e. if we are currently in epoch 10, distribute
      all undistributed rewards for epochs 0-8) and send the required number of tokens to each
      ↳worker
      ↳
26  DistributeRewards {
27      /// Address of contract for which to process rewards. For example, address of a voting
        ↳ verifier instance.
28      contract_address: String,
29      /// Maximum number of historical epochs for which to distribute rewards, starting with
        ↳ the oldest.
30      epoch_count: Option<u64>,
31  },
```

Figure 18: contracts/rewards/src/msg.rs

However, this documentation does not make it clear what the default behaviour for `DistributeRewards` is, if the `epoch_count` is left unspecified. In particular, this default is set to 10 epochs, which differs from both the single epoch implied by the contract documentation, and the maximum possible number of epochs, as implied by the interface documentation.

Typographical Errors

- `Verfier graph` -> `Verifier graph` in *doc/src/contracts/voting_verifier.md*
- `packges/signature-verifier-api` -> `packages/signature-verifier-api`, observed in *doc/src/contracts/multisig.md* at a later commit during the engagement

7 Contact Info

The team from NCC Group has the following primary members:

- Kevin Henry – Consultant
- Elena Bakos Lang – Consultant
- Javed Samuel – Practice Director, Cryptography Services