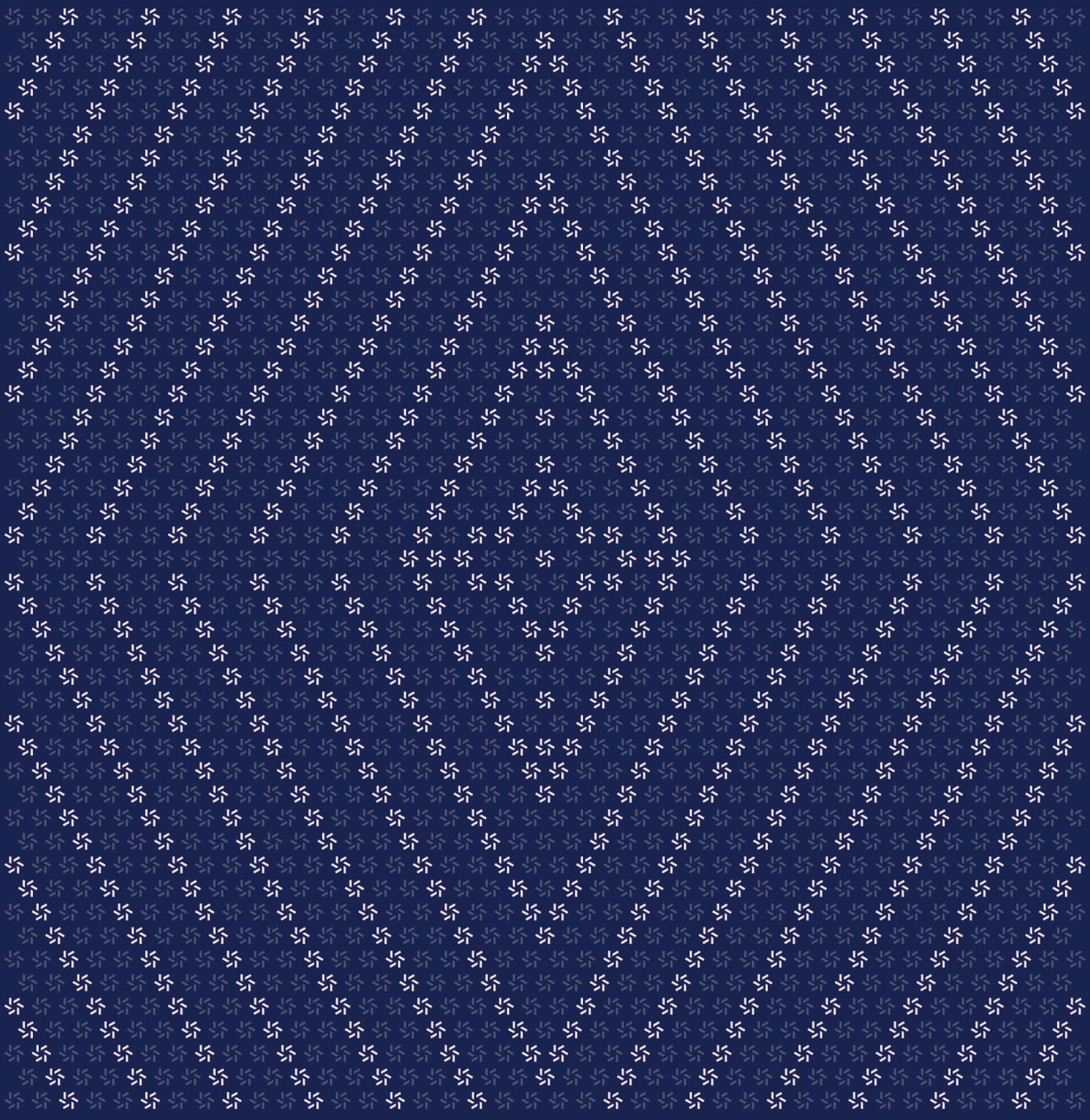
Zellic

Prepared for
Peter Kris
Gasp

Prepared by
Filippo Cremonese
Seunghyeon Kim
Zellic

October 15, 2024

Gasp Node and Monorepo
Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Gasp from September 3rd to October 3rd, 2024. During this engagement, Zellic reviewed Gasp Node and Monorepo's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Are the deposit and withdraw flows designed securely?
- Are the in-scope components implemented securely?
- Are assets held in escrow by the L1 rolldown contract safe?
- Are there specific and practical DOS issues?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Operational security practices including key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Gasp Node and Monorepo contracts, we discovered nine findings. Two critical issues were found. One was of high impact, one was of medium impact, two were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Gasp in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	2
High	1
Medium	1
Low	2
Informational	3

Gasp has remediated all impactful findings and resolved two out of three informational findings.

Finding	Status
3.1	Remediated
3.2	Remediated
3.3	Remediated
3.4	Remediated
3.5	Remediated
3.6	Remediated
3.7	Acknowledged
3.8	Acknowledged
3.9	Remediated

2. Introduction

2.1. About Gasp Node and Monorepo

Gasp contributed the following description of Gasp Node and Monorepo:

> GASP is L2 cross-rollup protocol at first, with the vision to make Ethereum the settlement layer for all cross-chain transactions. Gasp offers native cross-chain swaps without resorting to traditional bridges through the power of escape hatches which guarantee the withdrawal of user funds at all times.
>
> The Gasp network also relies on Ethereum for transaction finality through EigenLayer, making it one of the first AVSs (Actively Validated Service) ↗ in the ecosystem.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Gasp Node and Monorepo Contracts

Types	Solidity, Rust, Go
Platforms	EVM-compatible, Substrate
Target	eigen-layer-monorepo
Repository	https://github.com/mangata-finance/eigen-layer-monorepo ↗
Version	`7c3a652e99f48f45377ef4785e443c00f8b92bb1`

Programs	`contracts/src/FinalizerTaskManager.sol`
	`contracts/src/OperatorStateRetrieverExtended.sol`
	`contracts/src/GaspToken.sol`
	`contracts/src/Faucet.sol`
	`contracts/src/RolldownStorage.sol`
	`contracts/src/GaspMultiRollupService.sol`
	`contracts/src/GaspMultiRollupServiceStorage.sol`
	`contracts/src/FinalizerServiceManager.sol`
	`contracts/src/Rolldown.sol`
	`avs-aggregator/*.go`
	`avs-finalizer/*.rs`
	`rollup-sequencer/src/*.ts`
	`rollup-updater/src/*.ts`

Target	mangata-node
Repository	https://github.com/mangata-finance/mangata-node ↗
Version	`b6d9db6a70efb1db6e86014ca54ca7b25da8fdb6`
Programs	`pallets/rolldown/src/*.rs`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 13 person-weeks. The assessment was conducted by two consultants over the course of 6.5 calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Filippo Cremonese
Engineer
fcremo@zellic.io ↗

Seunghyeon Kim
Engineer
seunghyeon@zellic.io ↗

2.5. Project Timeline

September 3, 2024	Start of primary review period
September 4, 2024	Kick-off call
October 3, 2024	End of primary review period

3. Detailed Findings

3.1. Type confusion between `Withdrawal` and `Cancel` actions

Target	Rolldown		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

After the updater invokes `update_l1_from_l2` to update the Merkle tree root for a given range, anyone can call `close_withdrawal` or `close_cancel` to finalize processing of a pending request, perfecting the withdrawal or cancellation according to the corresponding entry in the Merkle tree.

Merkle trees store both `Withdrawal` and `Cancel` structs, and the leaf hash function is implemented as `keccak256(abi.encode(<struct>))`; it does not distinguish between `Cancel` and `Withdrawal` nodes with a distinct prefix.

The two data types have identical sizes, and their fields overlap as follows:

Cancel struct	Type	Withdrawal struct	Type
`requestId`	`RequestId`	`requestId`	`RequestId`
`Range.start`	`uint256`	`recipient`	`address`
`Range.end`	`uint256`	`tokenAddress`	`address`
`hash`	`bytes32`	`amount`	`uint256`

It is possible to submit a call to `close_cancel` using the `Withdrawal` struct intended to be provided to `close_withdrawal`. The Merkle inclusion check would pass because the ABI encoding and therefore the leaf hash are the same. The request would be marked as canceled, preventing the withdrawal with the given ID from being finalized.[1]

[1] Note that even though an address is 20 bytes, the ABI encoding is padded to 32 bytes (ref. https://docs.soliditylang.org/en/latest/abi-spec.html#formal-specification-of-the-encoding ↗). Therefore, all fields are 32 bytes wide and overlap cleanly.

Impact

An attacker could race the call to `close_withdrawal` and call `close_cancel` using the encoding of the `Withdrawal` struct and the same `merkle_root` and proof that would be supplied to `close_withdrawal`. This would lead to a denial of service and loss of funds.

Recommendations

Use a distinct prefix for all different types of Merkle tree leaves.

We strongly recommend to also differentiate leaves from intermediate nodes. While the current implementation does not seem to be exploitable due to the Merkle tree proof verification functions receiving the depth of the tree as input, this is a common source for vulnerabilities in Merkle tree implementations.

Remediation

This issue has been acknowledged by Gasp, and a fix was implemented in commit `47d82117` ↗.

The commit introduces separate hashing functions for each leaf type which prepend a unique prefix. This prevents collisions between hashes for different leaf types.

3.2. Reentrancy issue allowing repeat withdrawals

Target	Rolldown		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The function `close_withdrawal` has a reentrancy issue. The `processedL2Requests` mapping is only updated at the end of the function and after the withdrawal is processed, which includes sending ETH to the recipient.

```
function close_withdrawal(Withdrawal calldata withdrawal, bytes32 merkle_root,
    bytes32[] calldata proof) public {
    Range memory r = merkleRootRange[merkle_root];
    require(r.start != 0 && r.end != 0, "Unknown merkle root");

    bytes32 withdrawal_hash = keccak256(abi.encode(withdrawal));
    require(processedL2Requests[withdrawal.requestId.id] == false, "Already
    processed");

    uint32 leaves_count = uint32(r.end - r.start + 1);
    uint32 pos = uint32(withdrawal.requestId.id - r.start); // AUDIT: unsafe
    narrowing cast
    require(
      calculate_root(withdrawal_hash, pos, proof, leaves_count) == merkle_root,
      "Invalid proof"
    );
    process_l2_update_withdrawal(withdrawal);
    processedL2Requests[withdrawal.requestId.id] = true;
}
```

Impact

The attacker could repeat the withdrawal and drain the assets by abusing the reentrancy vulnerability.

Recommendations

Implement a reentrancy guard on all public and external functions. We recommend implementing reentrancy guards by default, unless a contract explicitly needs reentrancy.

We also recommend to adopt the checks-effects-interactions pattern as a best practice.

Remediation

This issue has been acknowledged by Gasp, and a fix was implemented in commit `05b0ca86` ↗.

3.3. Improper status tracking

Target	FinalizerTaskManager		
Category	Coding Mistakes	**Severity**	High
Likelihood	N/A	**Impact**	High

Description

The functions `respondToOpTask`, `respondToRdTask`, and `forceRespondToOpTask` contain multiple occurrences of a basic coding mistake. The code uses the comparison operator == instead of the assignment operator = when updating the `idToTaskStatus` mapping. The task statuses are therefore not being updated correctly, leading to inconsistencies in the contract's state management.

1. In the `respondToOpTask` function (lines 257 and 279)

```
idToTaskStatus[TaskType.OP_TASK][taskResponse.referenceTaskIndex] ==
    TaskStatus.RESPONDED;
...
idToTaskStatus[TaskType.OP_TASK][taskResponse.referenceTaskIndex] ==
    TaskStatus.COMPLETED;
```

2. In the `respondToRdTask` function (line 492 and line 511)

```
idToTaskStatus[TaskType.RD_TASK][taskResponse.referenceTaskIndex] ==
    TaskStatus.RESPONDED;
...
idToTaskStatus[TaskType.RD_TASK][taskResponse.referenceTaskIndex] ==
    TaskStatus.COMPLETED;
```

3. In the `forceRespondToOpTask` function (line 383)

```
idToTaskStatus[TaskType.OP_TASK][taskResponse.referenceTaskIndex] ==
    TaskStatus.COMPLETED;
```

Impact

Since the task statuses are not updated due to the incorrect operator, the contract will behave like the tasks are still in their initial state. This can lead to reprocessing of tasks, duplication, or failure to recognize completed tasks.

Recommendations

We recommend correcting these lines to use the assignment operator = as intended.

Remediation

This issue has been acknowledged by Gasp, and a fix was implemented in commit `10e4ba5c` ↗.

3.4. Raw ERC-20 interface usage

Target	Rolldown		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The `deposit_erc20` function directly uses the ERC-20 `transferFrom` interface to transfer tokens. The call is wrapped in a require statement. While this is technically correct, some tokens do not implement the ERC-20 standard correctly, including not returning a boolean value from `transferFrom`. Notably, this includes USDT and BNB.[2]

Additionally, the `process_erc20_withdrawal` function does not check the return value of the `transfer`; some tokens do not revert but return `false` if a transfer fails.

Impact

The code may not be compatible with several assets, including USDT and BNB.

Recommendations

We recommend using SafeERC20 for handling the ERC-20 tokens, as it guarantees compatibility with the vast majority of popular assets without compromising on security checks.

Remediation

This issue has been acknowledged by Gasp, and fixes were implemented in the following commits:

- `05b0ca86` ↗
- `a15df570` ↗

[2] A list of tokens with unusual behaviors can be found here: https://github.com/d-xo/weird-erc20.

3.5. Incorrect tracking of in-use status for alias accounts

Target	Sequencer staking pallet		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The `provide_sequencer_stake` function of the sequencer staking pallet allows to register a new sequencer and optionally provide the address of an alias account to be associated to the sequencer.

The function inserts the provided alias account into the `AliasAccountInUse` storage map to mark it as used. However, it does not remove any alias account that could be already be associated with the sequencer.

The same issue appears in the `set_updater_account_for_sequencer` function, which can replace the alias account associated with a sequencer without removing the entry from `AliasAccountInUse`.

Impact

Alias accounts could incorrectly remain registered in the `AliasAccountInUse` map, preventing them from being reregistered.

This issue could be abused to mark any address as an in-use alias account account. This may lead to a denial-of-service attack, since alias addresses are prevented from registering as sequencers.

Recommendations

Ensure stale entries in `AliasAccountsInUse` are removed by `provide_sequencer_stake` and `set_updater_account_for_sequencer`.

Remediation

This issue has been acknowledged by Gasp, and a fix was implemented in commit `35f7d3e1` ↗.

3.6. Missing maximum-number-of-sequencers check

Target	Sequencer staking pallet		
Category	Coding Mistakes	**Severity**	Low
Likelihood	N/A	**Impact**	Low

Description

The `provide_sequencer_stake` function part of the sequencer staking pallet does not correctly enforce the maximum number of registered sequencers.

Impact

More than the maximum intended number of sequencers may be able to register by using `provide_sequencer_stake`.

Recommendations

Check the number of currently registered sequencers as a precondition to accepting new sequencer registrations.

Remediation

This issue has been acknowledged by Gasp, and a fix was implemented in commit `8d96e174` ↗.

3.7. Missing error handling

Target	Sequencer staking pallet		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `slash_sequencer` function of the sequencer staking pallet does not handle possible errors from a call to `Currency::repatriate_reserve`:

```
let _ = T::Currency::repatriate_reserved(
  to_be_slashed,
  to_reward,
  repatriate_amount,
  frame_support::traits::BalanceStatus::Free,
);
```

Impact

The function is technically not guaranteed to succeed; however, it is not clear whether a failure result can be returned in practice.

Recommendations

We recommend to explicitly handle all error cases, at least with an assertion, to ensure no unexpected condition can be undetected.

Remediation

This issue has been acknowledged by Gasp.

This informational finding constitutes a violation of error handling best practices but is not an exploitable issue. Gasp believes this function will never return an error, and therefore opted not to make changes to the codebase in response to this finding.

3.8. Unhandled error

Target	rollup/runtime/src/runtime_config.rs		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `FeeHelpers::handle_sell_asset` and `FeeHelpers::handle_buy_asset` functions are not handling a potential error that may be returned by invocations of `OFLA::unlock_fee`.

Impact

The `unlock_fee` function could potentially fail and return an error; however, this does not seem possible to occur in practice, and as such, this finding is reported as Informational.

Recommendations

We recommend handling errors explicitly for all functions that could potentially return an error. If expected to never fail, it can be acceptable to handle errors via assertions.

Remediation

This issue has been acknowledged by Gasp.

This informational finding constitutes a violation of error handling best practices but is not an exploitable issue. Gasp believes this function will never return an error, and therefore opted not to make changes to the codebase in response to this finding.

3.9. Missing handling-of-transaction revert

Target	Aggregator service		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

The `SendNewOpTask` and `SendNewRdTask` functions, part of the aggregator-service implementation (in avs_writer.go) do not check whether a transaction was successfully executed.

The functions assume the first entry of `receipt.Logs` exists, but this is not technically guaranteed, as the transaction could potentially revert and have no logs.

Impact

This issue appears to be unlikely to manifest in practice. However, if the transaction submitted by `SendNewOpTask` or `SendNewRdTask` were to revert, it would cause the aggregator to panic and crash.

Recommendations

Handle potential reverts in `SendNewOpTask` and `SendNewRdTask`.

Remediation

This issue has been acknowledged by Gasp, and a fix was implemented in commit `eca9fa7f` ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Mixed-endian usage could cause unintended behavior

We found that crypto/bn254.rs from avs-finalizer mixes big and little endian encodings.

```
impl From<BlsKeypair> for Option<G1Point> {
    fn from(val: BlsKeypair) -> Self {
        if let Some((x, y)) = val.public.xy() {
            Some(G1Point {
                x: U256::from_big_endian(&x.into_bigint().to_bytes_le()),
                y: U256::from_little_endian(&y.into_bigint().to_bytes_le()),
            })
        } else {
            None
        }
    }
}
```

Such usage is inconsistent with other functions (which use little endian exclusively) and could cause unintended behavior. The function appears to be currently unused.

We suggest to correct this so that little endian is used consistently.

The Gasp team fixed this function in commit c68a569a ↗.

4.2. Project maturity considerations

This section discusses some aspects related to maturity of the project and the codebase.

Code and documentation quality

The reviewed codebases show, at times, suboptimal code quality, to a level not unusual for work-in-progress codebases. Examples of aspects that could be improved include the following:

- Inaccurate or irrelevant comments
- TODOs and commented-out code
- Spelling mistakes
- Overly complicated code

Considering the complexity of the system architecture, we recommend expanding the project documentation to make it more accessible and comprehensive.

Opsec and security hygiene practices

The repositories contain several instances of hardcoded credentials and private keys, used for testing the system. These secrets could more clearly be marked as test-only. Any secret related to a production deployment of the system must not be committed in the repository.

Some command line tools receive secrets as command line arguments; command line arguments are generally easier to obtain from a low privilege user and should be avoided, especially on shared servers or hosts that run several services. We suggest to only support reading secrets from environment variables or configuration files.

5. System Design

The following sections provide a description of the role of the various components of the system architecture. Please refer to Appendix 8. ↗ for a diagram illustrating the deposit and withdrawal flows.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

5.1. Rolldown Solidity contract

The rolldown contract is the main L1 contract.

It provides entry points allowing users to initiate a deposit of either the L1 native currency or an ERC-20 asset. When a deposit is made, user funds are transferred to the rolldown contract, which holds them in custody until a withdrawal is made. Upon successful completion of a deposit, the rolldown contract emits an event containing the deposit information (including amount, asset, and recipient). The event is detected by sequencers that relay it to the L2 network, crediting the recipient.

The contract also receives and stores Merkle tree root hashes from the updater component. The Merkle tree summarizes events happening on Gasp L2 and contains two types of leaves: withdrawal requests to be completed and cancellation requests. Once the Merkle root hash for a set of requests is provided to the rolldown contract, the withdrawal or cancellation requests are finalized individually by providing the request data together with a proof showing its inclusion in the tree. In the case of withdrawal requests, finalization also entails transferring the funds to the withdrawal recipient.

Note: In the code revision under review, a Merkle tree root hash submitted by the updater is not verified in any way; as such, the updater is a centralized and trusted system component. This is not reported as a finding since the Gasp team was aware of this limitation beforehand and discussed the work-in-progress state of the components during early stages of the engagement. Their awareness is also made apparent by comments in the rolldown contract `update_l1_from_l2` function and by other documentation shared with us, which document the intent to verify the submitted Merkle tree root against the consensus determined by EigenLayer.

5.2. Sequencer

The sequencer component is responsible for relaying L1 events to Gasp L2. Gasp architecture assumes that multiple parties will independently operate their own sequencers.

Sequencers are registered on Gasp L2 by staking assets that can be slashed in case of misbehavior. One sequencer at a time is selected to be allowed to propose an update to the L2. This is referred to as giving the selected sequencer a "read right", because the sequencer reads events pending submission from the L1 rolldown contract and relays them to the L2. The events submitted by the sequencer with read rights are put in a queue for a number of blocks. During this time, other sequencers verify the submitted information.

The other sequencers are granted cancel rights that allow them to dispute an event submitted to the L2 by the selected sequencer with read rights. Sequencers with cancel rights monitor the L1 updates sent to the L2 and verify them against their own view of the L1 state; if the sequencer with read rights is found misbehaving, the other sequencers submit a cancellation request within the dispute

period, which prevents the incorrect update from taking effect and slashes the misbehaving operator's stake.

5.3. Gasp collator node and rollup pallet

Gasp L2 is built using the Substrate framework, including several off-the-shelf pallets. Our review focused on the bespoke rolldown pallet, which handles deposits and withdrawals.

The `update_l2_from_l1` extrinsic can be invoked by the sequencer with read rights to provide an L1 update (including information about a deposit made on an L1 chain). The update is kept pending for a dispute period, during which other sequencers can invoke the `cancel_requests_from_l1` extrinsic to cancel the update if they detect that the update does not correctly reflect the L1 state.

After the dispute period has passed, pending updates are processed, eventually leading to the minting of the tokens that represent the deposit to the intended recipient. The tokens can be traded using the functionality exposed by the standard xyk pallet AMM.

The rollup pallet exposes a `withdraw` extrinsic that can be used by token owners to initiate a withdrawal of the L2 tokens back to the L1. Withdrawal requests are queued up and normally processed in batches. A batch of withdrawals is created automatically after a certain configurable amount of L2 blocks have occurred or manually via explicit request of an L2 user using the `create_batch` extrinsic. A cryptographic commitment of the L2 updates contained in each batch is relayed back to the L1 in the form of a Merkle tree root hash.

Administrative extrinsics

The pallet exposes several extrinsics for administrative purposes, which can only be invoked by the root origin. These extrinsics allow to forcefully submit a deposit and cancel an L1 update, forcing creation of a batch of updates to be sent to the L1 as well as triggering a refund of a failed deposit (e.g., due to overflow or other errors in minting the L2 tokens).

The existence of these administrative extrinsics further increases the importance of ensuring access to the root origin cannot be abused, as it effectively allows to sidestep the distributed validation otherwise ensured by the multiple independent sequencers. In the Substrate configuration under review, the root origin is accessible to the members of a council via the sudo pallet. Both the council and the sudo functionality are implemented by default pallets. Since the project is not deployed yet, we did not evaluate how the council is managed and who its members are.

5.4. Aggregator

The aggregator component detects if a batch of L2 updates is pending and invokes the AVS Task Manager contract to create a task asking operators to provide a signed witness attesting the validity of the batch. Note that the aggregator is in charge of determining the number of operators needed to reach the quorum. This design choice entrusts great power upon the aggregator, as the number of operators needed to reach the quorum is fundamental to the security of the system.

EigenLayer operators participating in Gasp consensus will pick up the task, independently derive the L2 state by executing the relevant block, and provide a signed attestation of the state they derived.

The signed attestation is submitted by the operators to an endpoint exposed by the aggregator. The aggregator service accumulates the received signatures (delegating cryptographic operations to the off-the-shelf EigenLayer BLS signatures aggregator). When enough signatures attesting the same state have been collected and a quorum is reached, the signatures (in the form of a single aggregate BLS signature) are submitted to the L1 Task Manager contract, which verifies the aggregate signature and marks the task as finalized.

Kicker thread

The aggregator is also responsible for holding EigenLayer operators responsible for inactivity. A thread periodically checks which operators have not participated in any of the last N (configurable) blocks. Any inactive operator found is ejected from the list of operators associated with Gasp AVS.

EigenLayer AVS implementation contracts

Gasp uses EigenLayer to establish a consensus about the L2 state on the L1. Integrating with EigenLayer requires implementing several contracts. Our review included the following:

- FinalizerServiceManager. The AVS service manager handles operator registration and deregistration; this contract extends the default EigenLayer service manager to add a waiting period before an operator can register after updating their stake. It also implements operator ejection, which allows a privileged address (currently the aggregator service) to forcibly remove an operator.
- FinalizerTaskManager. This contract manages the life cycle of a task. New tasks can be created by the aggregator (or forcefully created by the contract owner). A response can then be submitted by the aggregator and is marked as successfully completed if the weight of the operators that have signed it reaches the required quorum. The contract owner can also force a response.
- OperatorStateRetrieverExtended. This contract inherits from EigenLayer OperatorStateRetriever. It implements some utility methods used by the aggregator to get the state of an operator.

5.5. Finalizer

One instance of the finalizer component is run by each AVS operator. The finalizer retrieves AVS tasks from the task manager contract. It reproduces the L2 state by independently reexecuting the relevant block. The execution results (hashes representing the L2 state) are signed and submitted back to the aggregator service.

Note: The finalizer service is not currently including the Merkle tree root hash as part of the response provided by the finalizer for a given task. As discussed in previous sections, this is not reported as a finding due to the clear awareness of the Gasp team about this issue and the work-in-progress state

of the codebase.

5.6. Updater

The updater component is responsible for submitting Merkle root hashes of the trees that contain withdrawal and cancel requests to the L1 rolldown contract.

The updater subscribes to events emitted by the FinalizerTaskManager contract, signaling a task was completed (meaning a quorum of operators submitted a matching response).

Note: It is currently implemented as a centralized trusted component, as the provided Merkle root is not validated in any way by the rolldown contract. This is not reported as a finding due to the work-in-progress state of the project. The Gasp team was aware of this limitation and mentioned it in early conversations. This is also made apparent by comments in the rolldown contract `update_l1_from_l2` function, which document the intent to verify the submitted Merkle tree root.

Gasp plans to modify the finalizer, task manager, and rolldown contract components to include the Merkle tree root hash in the response provided by operators to AVS tasks. This would allow the rolldown contract to verify the Merkle tree root hash directly against the information submitted to the EigenLayer contracts, removing the reliance on the updater as a trusted component.

6. Threat Model

This section provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

6.1. Module: FinalizerTaskManager.sol

Function: `respondToOpTask(OpTask task, OpTaskResponse taskResponse, IBLSSignatureChecker.NonSignerStakesAndSignature nonSignerStakesAndSignature)`

This function is for responding to the OpTask. The function will try to finalize the OpTask and mark it as completed.

Inputs

- `task`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must match the record in the contract.
  - **Impact**: The task to treat.
- `taskResponse`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must have a valid `referenceTaskIndex`.
  - **Impact**: The response for the task.
- `nonSignerStakesAndSignature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a valid signature.
  - **Impact**: The BLS signature–related information for the `init` stage.

Branches and code coverage

Intended branches

- Complete the given task with the `taskResponse`.
  - ☑ Test coverage

Negative behavior

- Revert when the improper `msg.sender` calls this function during the `init` stage.
  - ☑ Negative test
- Revert when the aggregator already responded to the expected task.

- ☑ Negative test

6.2. Module: Rolldown.sol

Function: `close_cancel(Cancel cancel, byte[32] merkle_root, byte[32][] proof)`

This function is for canceling the close request.

Inputs

- `cancel`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The request must not be processed already.
  - **Impact**: Cancel request of the close request.
- `merkle_root`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a valid root for the Merkle proof.
  - **Impact**: The root for the Merkle proof.
- `proof`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be valid with the Merkle proof.
  - **Impact**: The proof for the Merkle proof.

Branches and code coverage

Intended branches

- Check the given `requestId` is valid for the cancel request.
  - ☑ Test coverage

Negative behavior

- Revert when the `merkle_root` is invalid.
  - ☑ Negative test
- Revert when the `requestId` is already processed.
  - ☑ Negative test
- Revert when the proof validation is failed.
  - ☑ Negative test

Function: `close_withdrawal(Withdrawal withdrawal, byte[32] merkle_root, byte[32][] proof)`

This function is for closing the withdrawal request.

Inputs

- `withdrawal`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The given `requestId` must be not processed yet.
  - **Impact**: Request for the withdrawal.
- `merkle_root`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a valid root for the Merkle proof.
  - **Impact**: The root for the Merkle proof.
- `proof`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be valid with the Merkle proof.
  - **Impact**: The proof for the Merkle proof.

Branches and code coverage

Intended branches

- Check the given `requestId` is valid for the closing request.
  - ☑ Test coverage
- Transfer the token if all the conditions are satisfied.
  - ☑ Test coverage

Negative behavior

- Revert when the `merkle_root` is invalid.
  - ☑ Negative test
- Revert when the `requestId` is already processed.
  - ☑ Negative test
- Revert when the proof validation is failed.
  - ☑ Negative test

Function: `deposit_erc20(address tokenAddress, uint256 amount)`

This function is for depositing ERC-20 tokens to the contract.

Inputs

- `tokenAddress`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The address must be nonzero.
  - **Impact**: Address of the token.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be nonzero.
  - **Impact**: Amount to transfer.

Branches and code coverage

Intended branches

- Transfer the expected amount of the tokens.
  - ☑ Test coverage

Negative behavior

- Revert when the token address is zero.
  - ☑ Negative test
- Revert when the amount is zero.
  - ☑ Negative test
- Revert when the transfer fails.
  - ☐ Negative test

Function: `deposit_native()`

This function is for depositing ETH to the contract.

Branches and code coverage

Intended branches

- Add the request for the deposit.
  - ☑ Test coverage
- Emit the events `DepositAcceptedIntoQueue` by using `depositRequest`'s information.
  - ☑ Test coverage

Negative behavior

- Revert when the `msg.value` is zero.
  - ☑ Negative test

7. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Gasp Node and Monorepo contracts, we discovered nine findings. Two critical issues were found. One was of high impact, one was of medium impact, two were of low impact, and the remaining findings were informational in nature.

7.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.

8. Appendix A: deposit and withdraw flows