Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

vlayer

vlayer

Veridise Inc.
May 12, 2025

► **Prepared For:**

vlayer labs
`https://www.vlayer.xyz/`

► **Prepared By:**

Jon Stephens
Tyler Diamond

► **Contact Us:**

`contact@veridise.com`

► **Version History:**

May 12, 2025	V3
Apr 29, 2025	V2
Mar 26, 2025	V1
Mar 26, 2025	Initial Draft

Contents

1 Executive Summary

From Feb. 10, 2025 to Mar. 21, 2025, vlayer labs engaged Veridise to conduct a security assessment of vlayer. Veridise conducted the assessment over 12 person-weeks, with 2 security analysts reviewing the project over 6 weeks on commit `a763614`.

The review strategy involved a tool-assisted analysis of the program source code performed by Veridise security analysts as well as thorough code review.

Project Summary. vlayer enables users to write smart contracts that can take advantage of additional functionality that is unavailable to a typical contract, including the ability to time travel in the current chain, teleport to a new chain, query the web and query emails (see below). This additional functionality is provided via a custom Risc Zero zkVM application that executes a version of the EVM, with the following additional precompiles.

- *Time travel*: Users can prove arbitrary state over a number of blocks that occurred in the past.
- *Teleport*: Users can prove state that exists on another chain. Specifically, this is used for reading the state of OP-stack chains from Ethereum.
- *Web proof*: Users can verify the transcript of a web connection that occurred which utilized the TLSNotary [*].
- *Email proof*: Users can verify the legitimacy of a DKIM signed email.
- *JSON and Regex*: Users can parse JSON and regular expressions on a given string.

This allows a user to deploy a *prover* contract that can interact with arbitrary state on the blockchain while making use of the new precompiles. This execution then produces a ZK proof that can be checked with an on-chain *verifier* contract to (1) verify the execution of the prover contract and (2) update information on-chain in response to the information provided in the ZK proof.

Code Assessment. The vlayer developers provided the source code of the vlayer contracts for the code review. The source code appears to be mostly original code written by the vlayer developers. The `call` component of the project, which provides the functionality of proving the legitimacy of state read by the EVM, appears to take inspiration from Steel[†]. The project contains some documentation in the form of READMEs and documentation comments on functions and storage variables. Additionally, vlayer provides a book that documents the usage and architecture of their components[‡]. To facilitate the Veridise security analysts understanding of the code, the vlayer developers provided access to their internal documentation book, which provided more extensive design decisions than the public counterpart. Although the analysts attempted to understand the intended behavior of the code from the source code and books, they noted that many of the components provided no source code documentation. The source

[*] This project utilizes a trusted notary server to attest to the legitimacy of a TLS connection's data `https://tlsnotary.org/`

[†] `https://github.com/RiscZero/RiscZero-ethereum/tree/main/crates/steel`

[‡] `https://www.book.vlayer.xyz`

code contained a test suite, which the Veridise security analysts noted provided positive tests and some negative tests for most components.

Summary of Issues Detected. The security assessment uncovered 30 issues, 9 of which are assessed to be of high or critical severity by the Veridise analysts. Specifically, a user can prove arbitrary state when using a teleport (V-VLYR-VUL-001), email proofs may be forged to prove an arbitrary email (V-VLYR-VUL-002) and unvalidated email bodies can be injected into email proofs (V-VLYR-VUL-003). The Veridise analysts also identified 6 medium-severity issues, including V-VLYR-VUL-012, which details how a user can bypass the smart contract safeguards surrounding email proofs. Additionally, 8 low-severity issues and 7 warnings were found. Among the 30 issues, 20 issues have been acknowledged by vlayer labs, 10 issues are still unresolved. It should be noted that vlayer labs resolved all critical, high and medium issues so only issues classified as low and warning remain.

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve vlayer.

Documentation and testing. Some modules, such as the `mpt` module, contain extensive documentation along with diagrams. However, many other parts of the code, especially in the Rust code, provide no documentation. For example, many files in the `engine` crate have no documentation on them. This is especially concerning for the verifier components of `teleport` and `time_travel` given their importance in the project.

Dependency inner-workings. Much of vlayer's core functionality makes use of 3rd party dependencies to perform key operations such as reading emails, validating DKIM signatures, reading HTTP requests and validating web proofs. During this audit, several severe issues were identified related to unexpected interactions between dependencies (V-VLYR-VUL-003, V-VLYR-VUL-006, V-VLYR-VUL-008, V-VLYR-VUL-009). We would strongly recommend that the developers investigate the dependencies to understand what guarantees they provide. Additionally, the developers should ensure that any information returned to the user is proven or signed.

Privacy implications documentation. The vlayer developers have indicated they expect many users will opt into using prover networks which will generate zero-knowledge proofs for them, as opposed to proving locally. This will require potentially sensitive input data to be transmitted to said provers, and there should be extensive documentation on what trade-offs are made regarding a user's privacy when using these networks.

Document best practices. Several important security decisions are left to those that use the vlayer infrastructure as mentioned in V-VLYR-VUL-029. To ensure that users are aware of these security recommendations, we would recommend including a best practice page with information about:

1. *Frontrunning*: Frontunning could allow a proof to be verified by anyone once it enters the mempool. This could allow the frontrunner to perform an action or gain a reward rather than the individual who first submitted the transaction.

2. *Proof replay*: Replay attacks allow individuals to verify the same proof more than once. Oftentimes this is undesired behavior and nullifiers are used to ensure a proof may only be verified once. Also note that the seal (or a hash of the seal) should not be used as a nullifier as Groth16 proofs are malleable.
3. *Signatories/Notaries*: Applications that make use of web proofs or email proofs must designate trusted third parties that act as signatories. These signatories must be selected carefully as a compromised or malicious signatory could verify arbitrary web proofs or email proofs. Additionally, applications should ensure that these signatories are practicing proper operational security regarding the secure storage of keys. They should also frequently rotate their keys and, if possible, use a threshold signature scheme so that a single compromise does not compromise the security of the application.
4. *Domain-Specific security assumptions*: Security assumptions specific to certain features of the protocol such as V-VLYR-VUL-028and V-VLYR-VUL-026.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
vlayer	`a763614`	Rust	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Feb. 10 – Mar. 21, 2025	Manual & Tools	2	12 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	3	3	3
High-Severity Issues	6	6	6
Medium-Severity Issues	6	6	6
Low-Severity Issues	8	4	3
Warning-Severity Issues	7	1	0
Informational-Severity Issues	0	0	0
TOTAL	30	20	18

Table 2.4: Category Breakdown.

Name	Number
Data Validation	14
Logic Error	9
Maintainability	3
Access Control	2
Authentication	1
Frontrunning	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of vlayer's smart contracts and Rust code. During the assessment, the security analysts aimed to answer questions such as:

- Can non-canonical blocks be used during execution?
- Are signature operations correctly validated?
- What privacy protections are in place for users?
- Are the state roots of blocks correctly validated in the zkVM?
- Are the storage slots of accounts validated against the executing enviornment's state root?
- Do the EVM extensions allow invalid execution?
- Can unverified data be returned from the `web_proof` and `email_proof` programs?
- Do all inputs to the zkVM programs influence their execution or outputs?

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology. To address the questions above, the security assessment involved a combination of human experts and automated program analysis & testing tools. In particular, the security assessment was conducted with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, security analysts leveraged Veridise's custom smart contract analysis tool Vanguard, as well as the open-source tool Slither. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.

Scope. The scope of this security assessment is limited to the Solidity contracts in the `contracts/vlayer/src` folder and the Rust files in the `rust/` folder of the source code provided by the vlayer developers with the following exceptions:

- `contracts/vlayer/src`
  - `ImageID.sol`
  - `proof_verifier/FakeProofVerifier.sol`
  - `proof_verifier/ProofVerifierFactory.sol`
  - `proof_verifier/ProofVerifierRouter.sol`
- `rust/`
  - Various test files
  - `cli/`
  - `common/cli.rs`
  - `common/rpc.rs`
  - `trace.rs`
  - `provider/`

- `range/`
- `server_utils/`
- `services/call/host/`
- `services/call/server/`
- `services/call/server_lib/`
- `services/chain/client/`
- `services/chain/db/`
- `services/chain/host/`
- `services/chain/mock_server/`
- `services/chain/server/`
- `services/chain/server_lib/`
- `services/chain/worker/`
- `services/dns/`
- `verifiable_dns/dns_over_https/`
- `verifiable_dns/verifiable_dns/`
- `version`

Note that most of the host code, which communicates the inputs of the actual constrained execution of the zkVM, is out of scope.

Methodology. Veridise security analysts inspected the provided tests, read the vlayer documentation and provided examples. They then began a review of the code assisted by both static analyzers and testing. Additionally, proof of concepts were developed for issues which required validation.

During the security assessment, the Veridise security analysts were in regular asynchronous contact with vlayer developers to ask questions about the code. Additionally, security reviews for the Steel codebase were studied during the review of the `call` and `mpt` modules due to their similar architecture.

3.3 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 3.1.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

The likelihood of a vulnerability is evaluated according to the Table 3.2.

The impact of a vulnerability is evaluated according to the Table 3.3:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Trust Model

4.1 Operational Assumptions.

In addition to assuming that any out-of-scope components behave correctly, Veridise analysts assumed the following properties held when modeling security for vlayer.

- The vlayer labs `verifiable_dns` administrators will only sign valid DNS records.
- The vlayer labs TLSNotary Notary server will act honestly.
- The vlayer labs `Repository` owners will only add legitimate DNS signing keys, ImageIDs and Notary keys.

4.2 Privileged Roles.

Roles. This section describes in detail the specific roles present in the system, and the actions each role is trusted to perform. The roles are grouped based on two characteristics: privilege-level and time-sensititivy. *Highly-privileged* roles may have a critical impact on the protocol if compromised. Time-sensitive *emergency* roles may be required to perform actions quickly based on real-time monitoring, while *non-emergency* roles perform actions like deployments and configurations which can be planned several hours or days in advance.

During the review, Veridise analysts assume that the role operators perform their responsiblities as intended. Protocol exploits relying on the below roles acting outside of their privileged scope are considered outside of scope.

- Highly-privileged, emergency roles:
  - The `Repository` admin can change the owner of the `Repository` at any point, with no time delay.
  - The `Repository` owner can remove/rotate keys if a compromise were to occur.
- Highly-privileged, non-emergency roles:
  - `verifiable_dns` approved public keys can associate a DKIM public key with any domain they choose.
  - The approved TLSNotary Notary servers used in `web_proof` are entrusted to only sign valid transcripts.
  - The admin and owner of the `Repository` can whitelist any Risc Zero Image ID, DNS signing key and TLSNotary Notary key.

Operational Recommendations. Highly-privileged, non-emergency operations should be operated by a multi-sig contract or decentralized governance system. This applies to the admin and owner roles of the `Repository`. These operations should be guarded by a timelock to ensure there is enough time for incident response. Highly-privileged, emergency operations should be tested in example scenarios to ensure the role operators are available and ready to respond when necessary.

Full validation of operational security practices is beyond the scope of this review. Users of the protocol should ensure they are confident that the operators of privileged keys are following best practices such as:

- Never storing a protocol key in plaintext, on a regularly used phone, laptop, or device, or relying on a custom solution for key management.
- Using separate keys for each separate function.
- Storing multi-sig keys in a diverse set of key management software/hardware services and geographic locations.
- Enabling 2FA for key management accounts. SMS should *not* be used for 2FA, nor should any account which uses SMS for 2FA. Authentication apps or hardware are preferred.
- Validating that no party has control over multiple multi-sig keys.
- Performing regularly scheduled key rotations for high-frequency operations.
- Securely storing physical, non-digital backups for critical keys.
- Actively monitoring for unexpected invocation of critical operations and/or deployed attack contracts.
- Regularly drilling responses to situations requiring emergency response such as pausing/unpausing.

5 Vulnerability Report

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 5.1 summarizes the issues discovered:

Table 5.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-VLYR-VUL-001	Missing EVM environment validation . . .	Critical	Fixed
V-VLYR-VUL-002	Missing DNS record validation allows . . .	Critical	Fixed
V-VLYR-VUL-003	Inconsistent dependencies allow injection . . .	Critical	Fixed
V-VLYR-VUL-004	SetBlock and SetChain apply to all . . .	High	Fixed
V-VLYR-VUL-005	Time-travel forward admits arbitrary state	High	Fixed
V-VLYR-VUL-006	Information not included in DKIM . . .	High	Fixed
V-VLYR-VUL-007	Inconsistent metadata access allows access . . .	High	Fixed
V-VLYR-VUL-008	Request transcript redaction can change . . .	High	Fixed
V-VLYR-VUL-009	Response transcript redaction may silently . . .	High	Fixed
V-VLYR-VUL-010	Entries with an empty key incorrectly . . .	Medium	Fixed
V-VLYR-VUL-011	Precompile addresses are not unique	Medium	Fixed
V-VLYR-VUL-012	Email proof validation can be passed . . .	Medium	Fixed
V-VLYR-VUL-013	Incorrect From email address can be extracted	Medium	Fixed
V-VLYR-VUL-014	Email address validation does not match . . .	Medium	Fixed
V-VLYR-VUL-015	Delegate calls use incorrect storage	Medium	Fixed
V-VLYR-VUL-016	REVM block number not set for ForgeBlock	Low	Open
V-VLYR-VUL-017	Sequencer output silently overwritten	Low	Open
V-VLYR-VUL-018	Travel block silently truncated	Low	Fixed
V-VLYR-VUL-019	Potential for man in the middle attack	Low	Fixed
V-VLYR-VUL-020	Values with primary redaction character . . .	Low	Fixed
V-VLYR-VUL-021	Primary redaction characters may exist in . . .	Low	Open
V-VLYR-VUL-022	Two step ownership is preferred	Low	Open
V-VLYR-VUL-023	Lack of SpecID may cause confusion to users	Low	Acknowledged
V-VLYR-VUL-024	AnchorStateRegistry reads from a fixed slot	Warning	Open
V-VLYR-VUL-025	Unexpected JSON path syntax for nested . . .	Warning	Open
V-VLYR-VUL-026	Database seeding can cause inconsistencies	Warning	Open
V-VLYR-VUL-027	General smart contract recommendations	Warning	Partially Fixed
V-VLYR-VUL-028	Domain owners can prove arbitrary emails	Warning	Open
V-VLYR-VUL-029	Proofs may be replayed or frontrun	Warning	Open
V-VLYR-VUL-030	General Rust recommendations	Warning	Open

5.1 Detailed Description of Issues

5.1.1 V-VLYR-VUL-001: Missing EVM environment validation allows proof spoofing

Severity	Critical	**Commit**	a763614
Type	Logic Error	**Status**	Fixed
File(s)	`rust/services/call/guest/src/guest.rs`		
Location(s)	main()		
Confirmed Fix At	https://github.com/vlayer-xyz/vlayer/pull/1891, https://github.com/vlayer-xyz/vlayer/pull/1950		

The vLayer "call" zkVM application can be used to prove the output of a given transaction that interacts with vLayer's new precompiles. To do so, the following information needs to be provided to the guest program:

1. Information about how to instantiate the REVM instances required by the call
2. Information about the initial execution environment, including the chainID and block number
3. Information required to prove the correctness of requested travel calls
4. Information about the call to invoke

After execution, the guest program then makes the following information public for use on-chain:

1. The starting block's number
2. The starting block's hash
3. The call target's address
4. The call's selector
5. the return value of the call

Importantly, the exposed public data must include enough information for a user to validate the correctness of the initial execution environment as some of vLayer's correctness guarantees rely on the *initial* execution environment being correct. Using the above data, a user can validate that the block number and has are consistent with their expected environment, but notably it excludes information about the starting chain itself.

The chain ID is used by vLayer in a few ways. First, it is used to select the an appropriate "ChainSpec" which summarizes information about a blockchain's history. This chain spec and transitively the chain ID are then used by the zkVM application in a few ways:

1. The ChainSpec is used along with the block number or timestamp to select the correct execution configuration for REVM.
2. The ChainSpec is used to alter the behavior of vLayer primitives in some cases. For example, as shown below when checking the validity of a "teleport" from one blockchain to another using vLayer's precompile, important validation is skipped if the chain is configured to be a local testnet.

```
async fn verify(
    &self,
    evm_envs: &CachedEvmEnv<D>,
    start_exec_location: ExecutionLocation,
) -> Result<()> {
    info!("Verifying teleport");
    let source_chain_id = start_exec_location.chain_id;
    let source_chain_spec = chain::ChainSpec::try_from(source_chain_id)?;
    if source_chain_spec.is_local_testnet() {
        info!("Skipping teleport verification for local testnet");
        return Ok(());
    }

    ...
}
```

Snippet 5.1: Snippet from the `verify` function used to perform safety checks for a teleport

Impact By not including information about the ChainID or ChainSpec, an attacker can manipulate the initial execution environment. To do so, the attacker would provide the expected block hash, header and number but would change the input ChainID. The result would be an execution that *appears* correct with respect to the public output but could be manipulated. More specifically a malicious user could:

1. Execute a particular block under the wrong REVM context. Since different forks of the EVM have different instruction semantics and some EVM chains have different precompiles, this could cause the Call result to be incorrect.
2. Avoid validation performed by vLayer to check the consistency of certain operations such as teleport. In the teleport case specifically, by selecting a chain ID that corresponds to a "local testnet" a malicious user can trick vLayer into skipping additional validation that the target chain is configured correctly. The resulting proof would appear to prove information about the target chain but would not do so in reality as the attacker could arbitrarily control the target state.

Recommendation Include information about the chain configuration of the initial execution environment. We would recommend including both the Chain ID and a SpecID selected by the ChainSpec.

Developer Response The developers have added the ChainID to the `CallAssumptions` and check that the executing chain's ID matches in the `ProofVerifierBase`.

5.1.2 V-VLYR-VUL-002: Missing DNS record validation allows email forgery

Severity	Critical	**Commit**	a763614
Type	Data Validation	**Status**	Fixed
File(s)	`rust/email_proof/src/lib.rs`		
Location(s)	parse_and_verify()		
Confirmed Fix At	`https://github.com/vlayer-xyz/vlayer/pull/2079`		

DKIM verification is a process to help determine an email's authenticity by validating and ensuring that the email is from the domain of the sender. To do so, a given domain must post a DKIM DNS record on their domain that provides a cryptographic public key P while keeping the associated private key S secret. When an email is sent, the domain can then attest to the authenticity of the email by signing parts of the email with S and providing the cryptographic signature in a DKIM record in the email's header. Recipients may then check the authenticity of the email by querying the domain for the DKIM record and ensuring that the information was signed by the owner of the listed public key P.

To check the authenticity of an email, vLayer verifies the integrity of an email's DKIM record in the `parse_and_verify` function shown below. To do so, a DNS record that has been signed by a trusted entity is provided, along with the email undergoing verification. The function then:

1. Ensures that the DNS record is indeed signed by the trusted entity
2. Extracts the domain of the email's sender
3. Checks the DKIM header's domain matches the sender's domain and that the public key stored in the DNS record can be recovered from the email's content and DKIM signature stored in the header.

While the above validates that a DKIM signature exists for the provided public key, it does not validate the source of the public key. This allows a user to provide any DKIM DNS record, not just the one provided by the domain of the email.

```
pub fn parse_and_verify(calldata: &[u8]) -> Result<Email, Error> {
    let (raw_email, dns_record, verification_data) = UnverifiedEmail::parse_calldata(
    calldata)?;

    verification_data.verify_signature(&dns_record)?;

    let email = mailparse::parse_mail(&raw_email)?;

    let from_domain = from_header::extract_from_domain(&email)?;

    dkim::verify_email(email, &from_domain, dns::parse_dns_record(&dns_record.data)?)
        .map_err(Error::DkimVerification)?
        .try_into()
        .map_err(Error::EmailParse)
}
```

Snippet 5.2: Definition of the `parse_and_verify` function used to check the validity of an email

Impact Since the domain name of the input DNS record is not validated, an attacker can verify arbitrary emails by adding a DKIM Signature header to the email that uses a public key controlled by the user. The malicious actor only needs said public key to be signed by vlayer for an arbitrary domain. He can then sign an email for *any other* domain.

More specifically, an email can be forged via the following steps:

1. Create a DKIM DNS record on any host that the user controls with some newly created public key
2. Craft the forged email that one to verify, complete with the "From", "To" and "Subject" fields
3. Construct a DKIM signature header using the public key from step 1 while setting the "d" tag of the signature to the forged "from" domain.
4. Using the verifiable_dns module, fetch the malicious DNS record from step 1
5. Provide the signed DNS record and forged email to this module, all of which will be verified.

Recommendation Ensure that the DNS record's domain name is consistent with the information in the email

Developer Response The developers defined a custom format for which DNS record names will conform to, and confirm that the `selector` and `domain` tag values of the DKIM-Signature field match this format.

Updated Veridise Response The issue has been fixed, however the Veridise analysts note that this implementation may be too restrictive as the DKIM RFC notes that only a single valid `DKIM-Signature` header is required to pass validation. If users are expected to trim these invalid or irrelevant headers, then provide documentation surrounding this expectation.

5.1.3 V-VLYR-VUL-003: Inconsistent dependencies allow injection of malicious email bodies

Severity	Critical	**Commit**	a763614
Type	Data Validation	**Status**	Fixed
File(s)	`rust/email_proof/src/email.rs`		
Location(s)	get_body()		
Confirmed Fix At	https://github.com/vlayer-xyz/vlayer/pull/2091		

A discrepancy between the handling of newline characters in the `mailparse` and `dkim` libraries enables malicious users to inject arbitrary email bodies into a DKIM signed email and still pass validation.

When parsing an email, the RFC 5322 specification requires that each header field ends in a `CRLF` (0x0D 0x0A). Additionally, the headers and the body of an email are separated by a blank line, i.e one which only contains a `CRLF`. Therefore, there will be two `CRLF` pairs between the last character of the last header field's value, and the first character of the body.

When parsing email headers with `parse_mail()`, the `mailparse` library does not adhere to the requirement that header fields end with a `CRLF`. Instead, it will consider a lone `LF` character as sufficient enough to consider the end of a header field as long as the following line does not contain a `WSP` character indicating a folded header value. This implementation error also extends to the separation between the headers and the body of the email.

However, the `dkim` library correctly follows the spec and extracts the body during the body hash calculation with the below function:

```rust
fn get_body<'a>(email: &'a mailparse::ParsedMail<'a>) -> Result<Vec<u8>, DKIMError> {
    Ok(bytes::get_all_after(email.raw_bytes, b"\r\n\r\n").to_vec())
}
```

Snippet 5.3: Snippet from the `dkim` library:

This discrepancy manifests in the call to the `parse_and_verify()` function, which, amongst other data, parses the inputted email string via `mailparse::parse_mail()`. The DKIM validation will use the correct body separation to verify the email. However, the `ParsedMail` is then converted to an `Email` via the `TryFrom` trait implementation in `email.rs`. This conversion calls the `get_body()` function, which will use the `ParsedMail::get_body` function on every MIME `part` of the `ParsedMail`. Since the `parse_mail()` function already incorrectly parsed the email, this function call can return data that was not validated in the bodyhash of the email. A proof of concept is provided below.

Impact A malicious user can craft emails in which they insert additional MIME body values that will not be included in the DKIM bodyhash, but they are considered bodies in the `ParsedMail` and returned to the user as if they were verified by the precompile. A malicious user can therefore manipulate an existing email by inserting a new "body" section after the headers separated by a blank line that only uses `\n`. The original body is then appended to the malicious body separated by the expected empty line `\r\n`. By doing so, the email will still pass DKIM verification but will be returned to the user as though all parts of the email were verified.

Recommendation Our investigation indicates this is still an issue in the latest version of the upstream `mailparse` library. While this issue still exists, manual verification should be done regarding the line termination in the passed email such that a lone `LF` character is not observed outside of multiline header values.

Proof of Concept The following test can be placed in the `rust/email_proof/src/lib.rs` file, in addition to adding the `pub(crate)` modifier to the `get_body()` function. The test will add an additional MIME body part to the already-signed email, and still pass DKIM validation.

```rust
#[test]
fn veridise_extra_data() -> anyhow::Result<()> {
    use crate::email::get_body;
    let mut email = signed_email_fixture();

    email = email.replace(
        "boundary=\"00000000000064b16c062913f525\"\r\n",
        "boundary=\"00000000000064b16c062913f525\"\n\n--00000000000064b16c062913f525\
    r\nContent-Type: text/plain; charset=\"UTF-8\"\r\n\nTHIS IS EXTRA DATA\r\n",
    );

    let calldata = calldata(&email, &DNS_FIXTURE, &VERIFICATION_DATA);
    let email = parse_and_verify(&calldata)?;

    println!("Email body from parse_and_verify: {}", email.body);
    Ok(())
}
```

After running the test, one can see that the "THIS IS EXTRA DATA" string is shown before the DKIM-validated body of the message, and validation still passes.

Developer Response The developers have implemented a function that checks every character after a `CRLF` is a valid header field character.

Updated Veridise Response The header bytes are now checked to never contain a lone newline character with the `verify_no_fake_separator()` function. Note that the body should also contain no lone newline characters according to the RFC, but this does not effect this issue.

5.1.4 V-VLYR-VUL-004: SetBlock and SetChain apply to all following transactions

Severity	High	**Commit**	a763614
Type	Logic Error	**Status**	Fixed
File(s)	`rust/services/call/engine/src/travel_call/inspector.rs`		
Location(s)	on_call		
Confirmed Fix At	https://github.com/vlayer-xyz/vlayer/pull/1890/		

The vLayer developers integrate several pre-compiles into the REVM instance running within their zkVM application. Among these, two pre-compiles-known as "travel calls"-allow developers to alter the blockchain's execution environment. Specifically, they can query state from other blocks on the same chain (SetBlock) or from a different EVM-based blockchain (SetChain). When either the SetBlock or SetChain precompile is invoked, the execution environment for the *next function call* is modified, as outlined in vLayer's documentation.

To enable this functionality, vLayer uses a REVM inspector to monitor the virtual machine's execution. This inspector can override the behavior of a call via the `on_call` function (given below) if the call needs to be executed in a different environment (i.e. location). However, when a travel call is made (when `self.location` is `Some`), the `on_call` function triggers the travel call by invoking the `transaction_callback` but afterwards it does not reset `self.location` to `None`. As a result, when `on_call` is invoked again, `self.location` remains set, causing another travel call to occur. Additionally, no other API method is provided to reset the `location` to `None`. Therefore, once a travel call is used, all subsequent calls in the execution will continue to be travel calls even if the user attempts to use the precompiles to restore the original environment.

```rust
fn on_call(&mut self, inputs: &CallInputs) -> Option<CallOutcome> {
    let Some(location) = self.location else {
        return None; // If no setChain/setBlock happened, we don't need to teleport
    to a new VM, but can continue with the current one.
    };
    info!(
        "Intercepting the call. Block number: {:?}, chain id: {:?}",
        location.block_number, location.chain_id
    );
    let (result, metadata) =
        (self.transaction_callback)(&inputs.into(), location).expect("Intercepted
    call failed");
    info!("Intercepted call returned: {result:?}");
    self.metadata.extend(metadata);
    let outcome = execution_result_to_call_outcome(&result, inputs);
    Some(outcome)
}
```

Snippet 5.4: Definition of the `on_call` function which is used to implement the travel call behavior

Impact The behavior of travel calls as implemented by `on_call` does not match the behavior described in vLayer's documentation. Additionally, travel calls are more restrictive and expensive to execute than typical function calls as vLayer enforces that they must return a value.

Since there is no way to stop traveling once a travel call has been requested, it is likely that some applications will encounter errors.

Recommendation Before returning the outcome of the call, set `self.location` to `None`

Developer Response The developers now utilize `Option::take()` when checking the location, which will set it to `None`.

5.1.5 V-VLYR-VUL-005: Time-travel forward admits arbitrary state

Severity	High	**Commit**	a763614
Type	Logic Error	**Status**	Fixed
File(s)	`rust/services/call/engine/src/verifier/time_travel.rs`		
Location(s)	verify()		
Confirmed Fix At	https://github.com/vlayer-xyz/vlayer/pull/1996		

The `TravelCallVerifier` will verify the legitimacy of the `CachedEvmEnv` (all `EvmEnv` that will be used) with respect to the `start_execution_location`. It does so by first verifying the teleports, and then verifies the time travel of each chain and its blocks.

The Time Travel verifier checks the provided chain proofs to assert the coherency of a chain of blocks (essentially the correct formation of the `parentHash` field in block headers) and ensures that the information accessed during a travel call is consistent with the given chain proof. The core validation done in the verifier can be seen below:

```rust
let block_numbers = blocks.iter().map(|(block_num, _)| *block_num).collect();
let chain_proof = client.get_chain_proof(chain_id, block_numbers).await?;
self.chain_proof_verifier.verify(chain_proof.as_ref())?;
for (block_num, block_hash) in blocks {
    let trie_block_hash = chain_proof
        .block_trie
        .get(block_num)
        .ok_or(Error::BlockNotFound { block_num })?;
    if trie_block_hash != block_hash {
        return Err(Error::BlockHash {
            block_num,
            hash_in_input: block_hash,
            proven_hash: trie_block_hash,
        });
    }
}
```

Snippet 5.5: Snippet from `rust/services/call/engine/src/verifier/time_travel.rs:verify()`

In the Teleport verifier, it will use the `AnchorStateRegistry` of every destination L2 in order to verify the legitimacy of the claimed blocks. As a part of this check, it will also ensure the latest block number used in a destination teleport is not greater than the latest anchored block.

The combination of the Time Travel and Teleport verifiers means that one can ensure that the latest claimed L2 block exists, in addition to all of its ancestors being correctly formed. However, the Teleport verifier does *not* check the chain of blocks used in Time Travel functionality on the base chain.

Therefore, one can append additional blocks as long as the coherency of them is valid. Doing this allows one to set the future state of the chain to whatever they desire. Notably, the the call guest program's journal does not include the latest block used in the call.

Impact When creating a chain-proof, the consistency of a given chain is checked. For all blocks at or before the "starting block", this is sufficient as the block number and hash are included in the Journal which *should* be verified on-chain. For future blocks, however, such a consistency check will not ensure that a given block is on the given blockchain as one must only ensure that their specified parent hash is correct, all remaining data can be manipulated arbitrarily (including the state root). As such, if a malicious actor is able to time-travel into the future they can arbitrarily manipulate the produced proof.

Recommendation If the developers do not intend to allow time-travel to the future, add a restriction that all time-travels on the starting chain must be to a previous block (similar to the restriction already in teleport). If the developers do want to allow time-travel to the future, include the most-recent block in the produced journal.

Developer Response The developers have added the `ensure_no_forward_jump()` function to the Travel Call `Executor` in order to ensure that all calls to `internal_call()` do not occur at a block number past the starting location. The starting location is checked on chain to be a valid block, therefore this check is sufficient.

5.1.6 V-VLYR-VUL-006: Information not included in DKIM signature can be returned

Severity	High	**Commit**	a763614
Type	Data Validation	**Status**	Fixed
File(s)	`rust/email_proof/src/lib.rs`		
Location(s)	parse_and_verify()		
Confirmed Fix At	https://github.com/vlayer-xyz/vlayer/pull/2116		

The `parse_and_verify()` function will call the DKIM module's `verify_email()` function to verify the DKIM signature on the extracted email. It will then utilize the `try_into()` method implemented for converting a `ParsedMail` to the crate-defined `Email` and will return said `Email`. As seen below, the returned `Email` from the `TryFrom` trait implementation contains the `from`, `body`, `to` and `subject`.

```
fn try_from(mail: ParsedMail) -> Result<Self, Self::Error> {
    ...
    Ok(Email {
      from: from_email,
      body: get_body(&mail)?,
      to,
      subject,
  })
}
```

Snippet 5.6: Snippet from `rust/email_proof/src/email.rs:try_from()`

The DKIM-Signature header field itself contains a field that lists the header fields of the email to be included in the data that is signed. This field (dubbed as `h=`) only requires that the `from` field is signed. In addition to the bodyhash inherently included in the required bodyhash (`bh=`) field, all other fields of the email being attested to are not required to be signed. Therefore, the `to` and `subject` field may not be signed.

Impact The `parse_and_verify()` function does no verification that the returned `to` and `subject` fields on the `Email` returned from `try_from()` actually were validated by DKIM. Therefore, if those header fields were omitted from `h=`, they can be arbitrarily set. Note that this conflicts with the Solidity `VerifiedEmail` struct, since the mentioned fields may not be verified.

Recommendation Require that validated emails have signed the `to` and `subject` fields, or change the documentation and nomenclature surrounding the `VerifiedEmail` struct.

Developer Response The developers now utilize the `verify_required_headers_signed()` function to ensure the `from`, `to` and `subject` fields are signed.

5.1.7 V-VLYR-VUL-007: Inconsistent metadata access allows access to unsigned information

Severity	High	**Commit**	a763614
Type	Logic Error	**Status**	Fixed
File(s)	`rust/email_proof/src/email.rs`, `rust/email_proof/src/from_header.rs`		
Location(s)	header_getter, extract_from_domain		
Confirmed Fix At	`https://github.com/vlayer-xyz/vlayer/pull/2116`, `https://github.com/vlayer-xyz/vlayer/pull/2207`		

When vLayer processes an email, it always retrieves a "From", "To" and "Subject" header from the email header which are returned after verification along with the email body. If the email contains multiple instances of any of these headers, the first occurrence of the header is chosen and returned as shown in the `header_getter` function below.

According to RFC 6376, when there are multiple instances of a header, the last header instance not already presented to the signing algorithm is added to the signature. Since a valid email can have multiple "From" headers and the `mailparse` dependency does not seem to enforce the uniqueness of either "To" or "Subject", an unvalidated instance of a header can be returned despite some other instances being verified. New headers can therefore be added to the email which would be returned in the verified email.

```rust
fn header_getter(headers: Headers) -> impl Fn(&str) -> Option<String> + '_ {
    move |key: &str| headers.get_first_value(key).map(String::from)
}
```

Snippet 5.7: Definition of `header_getter` which returns a function to fetch an email header

Impact Unverified information can be returned returned by the precompile. This allows malicious users to forge information which would then be trusted by prepending a new header to an existing email. Note that while "From" headers can be forged, the verification process restricts the forged email address to have the same domain as a DKIM header included in the email.

Recommendation Ensure that any returned data is signed by checking that the returned header is included in the DKIM signed header field list and is the last instance of the header (if only one instance is returned).

Developer Response The developers now retrieve the last instance of a header and confirm the header field names are in the list of signed fields in the DKIM-Signature.

5.1.8 V-VLYR-VUL-008: Request transcript redaction can change path and headers

Severity	High	**Commit**	a763614
Type	Data Validation	**Status**	Fixed
File(s)	`rust/web_proof/src/transcript_parser.rs`		
Location(s)	parse_request()		
Confirmed Fix At	`https://github.com/vlayer-xyz/vlayer/pull/2168`		

When utilizing the web_proof precompile, the web proof is verified and returns an object containing the `url` of the request to which this TLSNotary web proof attests. This is done by using the `request_url()` method of a `RequestTranscript`, which simply calls the `parse_request_and_validate_redaction()` function. This will substitute the redacted bytes, represented by NULL bytes (0x00), and then parse the redacted request via `parse_request()`.

`parse_request()` utilizes the `httparse` crate's `Request.parse()` method to read the headers and path of the request. This external functionality roughly works as follows:

1. Use `parse_method()` to determine the method of the request. When this is not the normal `POST` or `GET` request, it utilizes the `parse_token()` function which will read bytes as long as they are valid and until a space character is reached.
2. The path/url is parsed via `parse_uri()`, which will similarly read until a space is encountered.
3. The HTTP is version is then parsed, which reads exactly 8 bytes and must either be `HTTP/1.0` or `HTTP/1.1`.
4. The characters after the version must be newlines.
5. Parse the headers

Since the requests' redacted bytes are replaced with * or + characters via `replace_redacted_bytes()`, which are valid tokens in `parse_token()`, then one can redact an arbitrary number of bytes starting with the very first byte.

Impact Due to no limitations on the location of redacted bytes, one can redact the first line (containing the method, uri and version) and all headers in the HTTP request. One can then put their desired URI and headers in the body of their request, which will be incorrectly parsed as the URI and headers of the request.

Due to the web_proof verifier only using the request to return the URL, this is the impactful part of the issue in the context of where this request parsing is used. Note that because the url returned from `parse_url()` is compared to the `server_name` of the web proof's `Presentation` (which is signed and cannot be manipulated), this issue only enables one to change the path to one that exists on the same domain. However, this can still be dangerous as the path of the real request may be one that the attacker controls, in which case an arbitrary body can be returned for the corresponding response.

For example, one can change a request made to `example.com/usercontrolled` appear to be made to `example.com/servercontrolled`.

Recommendation Ensure that the method of the HTTP is not redacted, and therefore the first line of the HTTP request (which contains the path) must be fully parsed. Additionally, this issue arose due to the interaction between two third party crates: one to parse requests and one to redact requests. We recommend that the developers ensure they understand the interactions between these creates, particularly by exercising them with negative tests.

Proof of Concept The below test will successfully pass, and one can imagine that the beginning `\0` bytes are a redaction of the first line and following headers of an HTTP request.

```rust
 #[test]
fn veridise_redacted_request() {
    let raw_request = "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0 https://urlinbody.com
    HTTP/1.1\r\nHeader-in-body: true\r\n";
    let transcript = RequestTranscript::new(raw_request.as_bytes().to_vec());
    let url = transcript.parse_url().unwrap();
    assert_eq!(url, "https://urlinbody.com");
}
```

Developer Response The developers now ensure that the returned method is either `POST` or `GET`. Additionally, the host is returned from the `parse_request()` function and ensured to be un-redacted in `parse_request_and_validate_redaction()`.

5.1.9 V-VLYR-VUL-009: Response transcript redaction may silently overwrite data

Severity	High	**Commit**	a763614
Type	Data Validation	**Status**	Fixed
File(s)	`rust/web_proof/src/transcript_parser.rs`		
Location(s)	parse_response()		
Confirmed Fix At	`https://github.com/vlayer-xyz/vlayer/pull/2314`		

Note that this issue is similar to V-VLYR-VUL-008 with different field manipulations.

When utilizing the web_proof precompile, the web proof is verified and returns an object containing the `body` of the response that this TLSNotary web proof attests to. This is done by using the `parse_body()` method of a `ResponseTranscript`, which simply calls the `parse_response_and_validate_redaction()` function. This will substitute the redacted bytes, represented by NULL bytes (0x00), and then parse the redacted response via `parse_response()`.

`parse_request()` utilizes the `httparse` crate's `Response.parse()` method to read the headers of the response and return the starting index in the byte array of the `body`. This external functionality roughly works as follows:

1. Parse the version of the response (similarly to step 3 in the steps of V-VLYR-VUL-008)
2. Parse the 3-digit response code
3. Parse the reason via `parse_reason()`, which similarly to `parse_method()` will read an arbitrary amount of data, this time until a newline character is observed.
4. Parse the headers

After a response's version, code and reason fields are parsed, one can redact a header's value to an arbitrary number of bytes into the body. This would nullify any following headers and may truncate the body. One may also nullify all header fields by redacting the reason and headers, note that this may allow insertion of headers from the body.

Impact One may nullify arbitrary headers that follow a header whose value is redacted. Additionally, one can truncate the beginning of a body in order for it to appear that the response's body object contains fewer fields than it actually does. This could make it appear that the body starts in a different location without the redaction being observable.

Recommendation Either require that no headers are redacted, or provide documentation to end users that the parsed body may be incomplete. Note, if a malicious actor can control part of a response's body, they can make it appear as though the embedded JSON content is the entire content of the webpage.

Proof of Concept The following test shows how a response may be redacted, starting with its first header field's value, into manipulating the structure of the body. Note that the real transcript contains two objects in the entire body. However, we can manipulate the beginning of the body and redact the last value of the first object in order to maintain a valid JSON shape.

```
#[test]
fn veridise_redacted_response() {
    let unredacted_response = "HTTP/1.1 200 OK\r\nPlainHeader: Value1\r\n\r\n{\"obj1\" : {\"k1\": 5, \"k2\": 10}, \"obj2\": \"IMPORTANTINFO\"}\r\n";
    let unredacted_transcript =
        ResponseTranscript::new(unredacted_response.as_bytes().to_vec());
    println!("{}", unredacted_transcript.parse_body().unwrap());

    let redacted_response = "HTTP/1.1 200 OK\r\nPlainHeader: \0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\r\n\r\n{\"k1\": 5, \"k2\": \"\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\"}\r\n";
    let redacted_transcript = ResponseTranscript::new(redacted_response.as_bytes().to_vec());
    println!("{}", redacted_transcript.parse_body().unwrap());
}
```

This test's print statements will output the following:

```
{"obj1" : {"k1": 5, "k2": 10}, "obj2": "IMPORTANTINFO"}

{"k1": 5, "k2": "*****************"}
```

Developer Response The developers now check that no redaction occurs inside of the body of responses, and all JSON responses must be properly formatted.

Updated Veridise Response The fix should make it sufficiently difficult to perform such a redaction described in this issue in most non-malicious JSON responses. We have some concerns that it could be possible to still perform such an attack in a specifically crafted malicious response. In this case, however, the server is the one sending the malicious payload, so users should carefully validate the domain they are communicating with. Additionally, this fix is sufficient specifically because the body is required to be in properly formatted JSON, so it should be noted that if this requirement is ever removed or relaxed, the issue could return.

5.1.10 V-VLYR-VUL-010: Entries with an empty key incorrectly made as branch nodes

Severity	Medium	**Commit**	a763614
Type	Logic Error	**Status**	Fixed
File(s)	rust/mpt/src/node/insert/entry.rs		
Location(s)	from()		
Confirmed Fix At	https://github.com/vlayer-xyz/vlayer/pull/1899		

In the `From<Entry>` trait implementation for `Node`, the conversion has the following logic to make a branch node for an Entry with an empty key:

```
impl<D> From<Entry> for Node<D> {
    fn from(Entry { key, value }: Entry) -> Self {
        if key.is_empty() {
            Node::branch_with_value(value)
        } else {
            Node::leaf(&*key, value)
        }
  }
```

Snippet 5.8: Snippet from `rust/mpt/src/node/insert/entry.rs:from()`

However, this is not a proper implementation of the merkle patricia trie specification, and the node should instead still be a leaf node, just with an empty key.

Impact Insertions into the mpt will incorrectly encode values.

Recommendation When the key is empty return a `Node::leaf` with an empty key.

Developer Response The developers now return a `Node::leaf` no matter if the `key` is empty or not.

5.1.11 V-VLYR-VUL-011: Precompile addresses are not unique

Severity	Medium	**Commit**	a763614
Type	Data Validation	**Status**	Fixed
File(s)	`rust/services/call/precompiles/src/lib.rs`		
Location(s)	PRECOMPILES		
Confirmed Fix At	`https://github.com/vlayer-xyz/vlayer/pull/2182`		

VLayer adds several custom precompiles to the instance of REVM executed in their zkVM application. These custom precompiles must be assigned to an *unused* address so that the execution environment can distinguish their invocations. The developers provide a summary of each added precompile in the `PRECOMPILES` constant (shown below) which includes the allocated address. In this definition, we noted that `web_proof` precompile is defined at address `0x100` which collides with Optimism's `P256VERIFY` precompile at the same address.

```
pub const PRECOMPILES: [Precompile; NUM_PRECOMPILES] = generate_precompiles![
    // (address, precompile, base_cost, byte_cost, tag)
    (0x100, web_proof,             1000, 10, Tag::WebProof),
    (0x101, email_proof,           1000, 10, Tag::EmailProof),
    (0x102, json_get_string,       1000, 10, Tag::JsonGetString),
    (0x103, json_get_int,          1000, 10, Tag::JsonGetInt),
    (0x104, json_get_bool,         1000, 10, Tag::JsonGetBool),
    (0x105, json_get_array_length, 1000, 10, Tag::JsonGetArrayLength),
    (0x110, regex_is_match,        1000, 10, Tag::RegexIsMatch),
    (0x111, regex_capture,         1000, 10, Tag::RegexCapture),
    (0x120, url_pattern_test,      1000, 10, Tag::UrlPatternTest),
];
```

Snippet 5.9: Definition of the `PRECOMPILES` constant where the allocated address is the first value in the tuple

Impact This address collision could lead to functional failures when both precompiles are called within the same execution context, as the system may not correctly distinguish between the two precompiles. This would likely result in broken or unintended behavior, undermining the reliability of the zkVM application in environments where both precompiles are used.

Recommendation Ensure that the addresses assigned to their custom precompiles do not conflict with precompiles used by other chains, including Optimism. We recommend that vLayer reviews the list of precompile addresses across supported chains and adjusts the allocation to avoid potential collisions.

Developer Response The developers use a calculation similar to ERC-7201 in order to generate a unique address range for their precompiles.

5.1.12 V-VLYR-VUL-012: Email proof validation can be passed during time travel

Severity	Medium	**Commit**	a763614
Type	Data Validation	**Status**	Fixed
File(s)	`contracts/vlayer/src/EmailProof.sol`		
Location(s)	verify()		
Confirmed Fix At	https://github.com/vlayer-xyz/vlayer/pull/2167		

The `EmailProof` contract contains two checks that verify the legitimacy of the DNS record that contains the public key of the DKIM signed email, as seen below:

```
function verify(UnverifiedEmail memory unverifiedEmail) internal view returns (
    VerifiedEmail memory) {
  require(unverifiedEmail.verificationData.validUntil > block.timestamp, "EmailProof:
     expired DNS verification");
  if (ChainIdLibrary.isMainnet() || ChainIdLibrary.isTestnet()) {
      require(
          TestnetStableDeployment.repository().isDnsKeyValid(unverifiedEmail.
    verificationData.pubKey),
          "Not a valid VDNS public key"
      );
  }
  ...
}
```

Snippet 5.10: Snippet from `contract/vlayer/src/EmailProof.sol`

Essentially, the `validUntil` date is checked to have not passed, and the key used to sign the DNS record is checked against the repository to be registered.

However, given that vlayer introduces Time Travel functionality to execute in previous blocks, one can Time Travel to a block in which the `validUntil` field passes the check.

In regards to the DNS signing key, if a key registered in the repository is ever compromised, then one can validate any email they desire as they could Time Travel to a block in which the key was still registered as valid.

Impact Dependent upon the usage of DNS signing keys and the `validUntil` field, one may be able to validate arbitrary emails.

Recommendation Prevent usage of the email proof precompile during time-travel.

Developer Response The developers now check if a time travel or teleport is occurring via the `is_on_historic_block()` function, and will panic if that is true while executing a time sensitive precompile, checked via `is_time_dependent()`.

Updated Veridise Response With the `ProofVerifierBase`, the executing block must occur within the previous`AVAILABLE_HISTORICAL_BLOCKS` number of blocks. Therefore one could still execute with the previous Repository state until this number of blocks have passed. We recommend storing the block number of the last Repository modification so that verifier contracts can ensure they are executing on the latest version.

5.1.13 V-VLYR-VUL-013: Incorrect From email address can be extracted

Severity	Medium	**Commit**	a763614
Type	Logic Error	**Status**	Fixed
File(s)	`rust/email_proof/src/email.rs`		
Location(s)	extract_address_from_header()		
Confirmed Fix At	`https://github.com/vlayer-xyz/vlayer/pull/2133`, `https://github.com/vlayer-xyz/vlayer/pull/2207`		

When an email is processed by vLayer, information must be extracted from the email header including the email address of the sender. The sender's email address is extracted from the "From" field of the email header which has the following format according to RFC 5322:

1. Display Name <Email Address>
2. Email Address

The `extract_address_from_header()` function, shown below, is used to extract the address from the "From" header. It first identifies if the string contains a "<" character. If no such character is found, the email address itself is returned (i.e. the email address is in the second format shown above). If the character is found, the string between the first instance of "<" and the last instance of ">" is extracted and returned. While this will indeed extract the email address in the first case, it may also extract the incorrect email address (e.g. `safe@safe.org` would be extracted from `"<safe@safe.org>"@malicious.net`)

```rust
pub fn extract_address_from_header(header: &str) -> Result<String, MailParseError> {
    let Some(start) = header.find('<') else {
        return Self::validate_email(header);
    };
    let maybe_end = header.rfind('>');

    match maybe_end {
        None => Err(Self::invalid_from_header()),
        Some(end) if end <= start => Err(Self::invalid_from_header()),
        Some(end) => Self::validate_email(&header[start + 1..end]),
    }
}
```

Snippet 5.11: Definition of `extract_address_from_header` which is used to extract an email address from a header entry like "From" or "To"

Impact While quoted email addresses are rarely used, if a user is able to create an arbitrary address, they can spoof other email addresses. Due to the method used to verify the DKIM signature headers, there are restrictions as the spoofed email address must have the same domain as a DKIM header included in the email.

Recommendation More specifically recognize the above pattern. As an example, the `Display Name <Email Address>` case must end with a ">" while the `Email Address` case cannot.

Proof of Concept The following test will incorrectly pass.

```rust
#[test]
fn malicious_email() {
    let email = r#""<safe@safe.org>"@malicious.net"#;
    let extracted = Email::extract_address_from_header(email);
    // Note the real email is @malicious.org, not @safe.org!
    assert_eq!(extracted.unwrap(), "safe@safe.org");
}
```

Developer Response The developers now use the `addrpase()` function from the `mailparse` crate, check that only one address is returned, and trim the whitespace before returning the value.

Updated Veridise Response According to the documentation, the `addrpase_header()` function should be used instead.

Updated Developer Response The developers now use the `addrparse_header()` function.

5.1.14 V-VLYR-VUL-014: Email address validation does not match specification

Severity	Medium	**Commit**	a763614
Type	Data Validation	**Status**	Fixed
File(s)	`rust/email_proof/src/email_address.rs`		
Location(s)	is_valid()		
Confirmed Fix At	https://github.com/vlayer-xyz/vlayer/pull/2133, https://github.com/vlayer-xyz/vlayer/pull/2207		

Various issues in the `email_address.rs` file lead to inconsistencies of email address validation compared to the specification.

RFC 5322 defines what a valid email address is, and the ABNF syntax for an email address (and some of its dependent rules) is defined as below:

FWS = ([*WSP CRLF] 1*WSP) / obs-FWS ; Folding white space ctext = %d33-39 / ; Printable US-ASCII %d42-91 / ; characters not including %d93-126 / ; "(", ")", or "" obs-ctext ccontent = ctext / quoted-pair / comment comment = "(" *([FWS] ccontent) [FWS] ")" CFWS = (1*([FWS] comment) [FWS]) / FWS

atext = ALPHA / DIGIT / ; Printable US-ASCII "!" / "#" / ; characters not including "$" / "%" / ; specials. Used for atoms. "&" / "’" / "" / *"+" / "-" / "/" / "=" / "?" / "^" / "_" / "‘" / "{" / "|" / "}" / " " atom = [CFWS] 1*atext [CFWS] dot-atom-text = 1*atext *("." 1*atext) dot-atom = [CFWS] dot-atom-text [CFWS]

address = mailbox / group mailbox = name-addr / addr-spec name-addr = [display-name] angle-addr angle-addr = [CFWS] "<" addr-spec ">" [CFWS] / obs-angle-addr addr-spec = local-part "@" domain local-part = dot-atom / quoted-string / obs-local-part domain = dot-atom / domain-literal / obs-domain domain-literal = [CFWS] "[" *([FWS] dtext) [FWS] "]" [CFWS] dtext = %d33-90 / ; Printable US-ASCII %d94-126 / ; characters not including obs-dtext ; "[", "]", or ""

The logic to parse an email address in `email_address.rs` deviates from the above specification, allowing both invalid email addresses to be accepted and valid email addresses to be rejected as outlined below.

Accepted invalid email addresses:

1. The `remove_parts_inside_quotes()` function is used to remove quoted sections of an email address during validation as quoted sections allow almost arbitrary text. The logic to remove quotes, however, allows multiple quoted sections in an email but the specification above only allows the entire local part to be quoted. Any `"` characters found within a quoted section must first be escaped.
2. The `remove_parts_inside_quotes()` function removes quoted parts of the email so that the remaining parts can be validated. There are restrictions, however, on the characters that can occur in the quoted part of an email. For example, an unescaped `"` is not allowed.

Rejected valid email addresses:

1. `is_character_not_allowed_in_email_address()`: This function is utilized in `contains_invalid_characters()`, which is called on both the `unquoted_username` and the

domainname. This function is too restrictive for the username (local-part) of an email address, as it does not provide all the characters defined in a dot-atom.

2. The handle_quotes() function is used to recognize quotes so that quoted sections can be removed during email validation. It, however, recognizes any " character as an opening to a quoted section or closing to a quoted section, even if the quotes are escaped. This can prevent properly quoted emails such as "\""@test.org from being accepted.

Impact This logic is used to validate the extracted "From" email address to ensure a proper email address is indeed extracted. Since some invalid addresses are accepted and some valid email addresses are rejected, it can be the case that this validation prevents verification of some emails and may allow verification of improperly formatted (or extracted) emails.

Recommendation While email address extraction/validation is notoriously difficult, we would recommend supporting a fragment of valid email addresses and documenting this fragment. As an example, quoted locals are uncommon and many email providers do not allow the creation of quoted email addresses.

Developer Response The developers now use the addrpase() function from the mailparse crate.

Updated Veridise Response While the fix does appear to fix the issue, the addrpase_header() function should be used instead. Additionally, Veridise has not reviewed the library code itself for correctness and the library notes that it does not always follow RFC5322, which includes the specification of an email address. The library appears to be well-tested and there are no open issues regarding email parsing.

Updated Developer Response The developers now use the addrparse_header() function.

5.1.15 V-VLYR-VUL-015: Delegate calls use incorrect storage

Severity	Medium	**Commit**	a763614
Type	Logic Error	**Status**	Fixed
File(s)	`rust/services/call/engine/src/io.rs`		
Location(s)	from()		
Confirmed Fix At	https://github.com/vlayer-xyz/vlayer/pull/2164		

The travel call `Inspector` intercepts calls to determine whether a travel call API was invoked or if the current call should teleport or time-travel. If the current call is determined to teleport or time-travel, the `Inspector` will invoke its `on_call` function which, in turn, eventually invokes a transaction on a new instance of REVM configured to the desired state. In the process, the REVM's `CallInputs` is transformed into the `Call` struct via the code shown below

```
impl From<&CallInputs> for Call {
    fn from(inputs: &CallInputs) -> Self {
        Self {
            to: inputs.bytecode_address,
            data: inputs.input.clone().into(),
            gas_limit: inputs.gas_limit,
        }
    }
}
```

Snippet 5.12: Snippet from `rust/services/call/engine/src/io.rs:from()`

Notably, the `Call` structure does not have the ability to differentiate between a normal `CALL` and a `DELEGATECALL`. This differentiation arises from the inclusion of the `bytecode_address` while ignoring the `target_address`. The former specifies the account where the bytecode to run is stored, and the latter specifies the storage and context that will be used during execution. For a normal `CALL`, these addresses will be the same, but a `DELEGATECALL` allows these addresses to differ.

Impact The execution of a `DELEGATECALL`, such as during a proxy call, will use the storage of the `bytecode_address` instead of the `target_address` as during normal EVM operations. This will silently lead to the result of the `DELEGATECALL` returning incorrectly.

Recommendation Due to the immutability of the revm's backing database, changes to an addresses storage during execution will lead to inconsistencies surrounding the set state root of the `DELEGATECALL`'s executing evm. Therefore, it does not seem feasible to support the `DELEGATECALL` opcode and the `Inspector` should check to ensure one does not occur.

Developer Response The developers panic in the `on_call()` function of the `Inspector` if a `DELEGATECALL` is used.

5.1.16 V-VLYR-VUL-016: REVM block number not set for ForgeBlock

Severity	Low	**Commit**	a763614
Type	Logic Error	**Status**	Open
File(s)	`rust/block_header/src/forge.rs`		
Location(s)	fill_block_env()		
Confirmed Fix At	N/A		

The `ForgeBlockHeader` implements the `EvmBlockHeader`, which is the trait used by various components store block headers (such as `EvmInput`). When building the EVM environment, the `fill_block_env()` method of the trait is used to set the environment to be consistent with the description given in the block. However, the `ForgeBlockHeader` has an empty implementation and therefore does not set the environment values such as the block number. This may cause execution to be incorrect.

Separately, the `encode()` function for the `ForgeBlockHeader` does not include the `state_root` in the encoding. Given the hash of a `ForgeBlockHeader` simply returns the default hash, this does not seem to have meaningful impact.

Impact Execution may be incorrect in regards to the data stored in the `ForgeBlockHeader`.

Recommendation Set the block number of the REVM execution

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

5.1.17 V-VLYR-VUL-017: Sequencer output silently overwritten

Severity	Low	**Commit**	a763614
Type	Logic Error	**Status**	Open
File(s)	`rust/services/call/optimism/src/client/recording.rs`		
Location(s)	get_output_at_block()		
Confirmed Fix At	N/A		

When populating the recording `Client` used to record `SequencerOutputs` of L2s via `get_output_at_block()`, it will simply write the value returning from the `inner Client` to its cache. Therefore, if there are multiple `ExecutionLocations` with the same `ChainId`, then the `SequencerOutput` will be silently overwritten. One can see that the factories for cache and recording clients only have one entry per `ChainId`.

Additionally, the chain proofs input to the call guest program are of a hashmap indexed by `ChainId`. Therefore the guest execution will fail given that one of the instances of the teleport to the chain will not have access to the correct `SequencerOutput` and `ChainProof`.

Impact The `SequencerOutput` in the recording client will be silently overwritten and lead to the guest program failing at a later point in the proving pipeline.

Recommendation Fail early during this insertion when the cache is occupied. If multiple time travels to a destination chain plan to be supported, then refactor parts of the code to index upon `ExecutionLocation` instead of `ChainId`.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

5.1.18 V-VLYR-VUL-018: Travel block silently truncated

Severity	Low	**Commit**	a763614
Type	Data Validation	**Status**	Fixed
File(s)	rust/services/call/engine/src/travel_call/args.rs		
Location(s)	u64_from_be_slice()		
Confirmed Fix At	https://github.com/vlayer-xyz/vlayer/pull/2265		

When processing the inputs to a travel call, the `u256` arguments will be converted into `u64` via the `u64_from_be_slice` function shown below. This function extracts the last 8 bytes from the input value while it discarding the remainder of the value, truncating the value.

```rust
fn u64_from_be_slice(slice: &[u8]) -> u64 {
    u64::from_be_bytes(*slice.last_chunk().expect("invalid u64 slice"))
}
```

Snippet 5.13: Definition of `u64_from_be_slice()`

Impact The destination of travel calls may be misinterpreted as large numbers can be provided knowing that they will be truncated. This could be misinterpreted by users who do not know that the truncation will occur.

Recommendation Check that the discarded bytes were zero bytes.

Developer Response The developers now check that any bytes above the 8 bytes for a `u64` are 0 value.

5.1.19 V-VLYR-VUL-019: Potential for man in the middle attack

Severity	Low	**Commit**	a763614
Type	Authentication	**Status**	Fixed
File(s)	`rust/verifiable_dns/src/verifiable_dns/resolver/responses_validation.rs`		
Location(s)	validate_response()		
Confirmed Fix At	N/A		

To help check the authenticity of an email, vLayer provides a verifiable DNS service in which a trusted third party perform a request on the user's behalf and signs the resulting DNS record. Therefore, as long as the 3rd party is trustworthy (and has not been compromised), the DNS record should be authentic. DNS Queries, however, are vulnerable to man-in-the-middle attacks which could allow an external party to compromise the DNS record even if the 3rd party validator is trusted. To prevent such attacks, the DNSSEC extension was created to ensure a response came from a trusted server but the DNS validator service does not require that DNSSEC signatures are checked.

Impact Man-in-the-middle attacks can compromise the authenticity of a DNS record. If this occurs, emails could be improperly verified as authentic.

Recommendation Consider requiring DNSSEC verification in `verifiable_dns` or alternatively allow the user to specify whether DNSSEC should be enforced when verifying an email proof.

Developer Response The DNS infrastructure of vlayer is out of scope. Additionally, the module uses DNS-over-HTTPS which should prevent man-in-the-middle attacks.

Updated Veridise Response We acknowledge that this should prevent man-in-the-middle attacks. We do note this does not prevent DNS cache poisoning attacks, and still suggest that DNSSEC verification is employed.

5.1.20 V-VLYR-VUL-020: Values with primary redaction character may be partially redacted

Severity	Low	**Commit**	a763614
Type	Data Validation	**Status**	Fixed
File(s)	`rust/web_proof/src/redaction.rs`		
Location(s)	validate_name_value_redaction()		
Confirmed Fix At	`https://github.com/vlayer-xyz/vlayer/pull/2169`		

vLayer allows information to be proven about HTTP content via a WebProof. As the information contained in a HTTP request may be sensitive, vLayer allows some information to be redacted as long as the following restrictions are met as shown in the code below (note that information is assumed to be organized as key-value pairs):

1. A key may not be redacted
2. A value must be entirely redacted or not redacted at all (i.e. no partial redactions)

To check these properties, the `validate_name_value_redaction()` function takes as input two strings where the only difference is the character used to indicate a redaction. Therefore, if the two strings match then no redaction occurred while if they are different, some content is redacted. To disallow a partial match, a value is checked to determine if the two string versions are different (which can occur in a full redaction) *and* whether one version of the strings is only the redacted character. A partially redacted string can pass this validation though if the unredacted content is equal to the redaction character (in this case ∗).

```
pub(crate) fn validate_name_value_redaction(
    name_values_with_replacement_primary: &[RedactedTranscriptNameValue],
    name_values_with_replacement_secondary: &[RedactedTranscriptNameValue],
    redaction_element_type: RedactionElementType,
) -> Result<(), ParsingError> {
    let zipped_pairs = zip(
        name_values_with_replacement_primary.iter(),
        name_values_with_replacement_secondary.iter(),
    );

    let redacted_name = zipped_pairs.clone().find(|(l, r)| l.name != r.name);

    if let Some(pair) = redacted_name {
        return Err(ParsingError::RedactedName(redaction_element_type, pair.0.
    to_string()));
    }

    let partially_redacted_value = zipped_pairs.clone().find(|(l, r)| {
        !all_match(&l.value, REDACTION_REPLACEMENT_CHAR_PRIMARY as u8) && l.value !=
    r.value
    });

    if let Some(pair) = partially_redacted_value {
        return Err(ParsingError::PartiallyRedactedValue(
            redaction_element_type,
            pair.0.to_string(),
        ));
    }

    Ok(())
}
```

Snippet 5.14: Definition of the `validate_name_value_redaction` function

Impact A user can violate the invariant that information cannot be partially redacted. Consider the string `a*b`, one can redact `a` and `b` individually while still passing the partial redaction validation since `*` is equal to the redaction character. This could potentially cause redacted values to be misinterpreted as it can collide with realistic content.

Recommendation Assert the content of the secondary value (`r`) by running the same `!all_match` check against the `REDACTION_REPLACEMENT_CHAR_SECONDARY`.

Proof of Concept The following code demonstrates that a partially redacted value that contains the `REDACTION_REPLACEMENT_CHAR_PRIMARY` (`*`) will still pass validation.

```
#[test]
fn veridise_partial_redaction_with_star() {
  use crate::utils::bytes::replace_bytes;
  let redact = |x: &[u8], replacement: char| {
      String::from_utf8(replace_bytes(x, 0, replacement as u8)).unwrap()
  };
  let value_with_star = "\0\0*\0*\0";

  let primary_value = redact(value_with_star.as_bytes(),
    REDACTION_REPLACEMENT_CHAR_PRIMARY);
  let secondary_value = redact(value_with_star.as_bytes(),
    REDACTION_REPLACEMENT_CHAR_SECONDARY);

  let nvp = vec![("key1".to_owned(), primary_value).into()];
  let nvs = vec![("key1".to_owned(), secondary_value).into()];
  assert!(
      validate_name_value_redaction(&nvp, &nvs, RedactionElementType::RequestHeader).
    is_ok()
  );
}
```

Developer Response The developers have added a check on if the `r` value is `!all_matched` against the `REDACTION_REPLACEMENT_CHAR_SECONDARY`.

5.1.21 V-VLYR-VUL-021: Primary redaction characters may exist in values

Severity	Low	**Commit**	a7763614
Type	Data Validation	**Status**	Open
File(s)	`rust/web_proof/transcript_parser.rs`		
Location(s)	parse_request_and_validate_redaction(), parse_response_and_validate_redaction()		
Confirmed Fix At	N/A		

Information, such as the URL and http-body, extracted from a WebProof may be redacted to protect the privacy of the user. To indicate that a piece of information is redacted, the missing character is replaced by a $*$ when it is returned to the contract. This character can legitimately occur in the unredacted text, however, which may make it difficult to distinguish between redacted and unredacted content.

Impact Users may mistakenly interpret redacted information as unredacted or vice versa. For example, if a url contained the pattern `lookup=*`, this could indicate that the lookup value was redacted or possibly that all information was queried.

Recommendation Consider selecting a character that is unlikely (or in the case of a URL impossible) to occur in text.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

5.1.22 V-VLYR-VUL-022: Two step ownership is preferred

Severity	Low	**Commit**	a763614
Type	Access Control	**Status**	Open
File(s)	`contracts/vlayer/src/Repository.sol`		
Location(s)	Repository		
Confirmed Fix At	N/A		

The `transferAdminRole()` and `transferOwnershipRole()` functions utilize Openzeppelin's `AccessControlEnumerable` contract to provide ownership functionality. These functions will take in a new owner and will then simultaneously revoke the current owner/admin and grant the new owner/admin the corresponding role.

Snippet from `contracts/vlayer/src/Repository.sol`

```solidity
function transferAdminRole(address newAdmin) public {
    grantRole(DEFAULT_ADMIN_ROLE, newAdmin);
    renounceRole(DEFAULT_ADMIN_ROLE, msg.sender);
}

function transferOwnership(address newOwner) public {
    address owner = getRoleMember(OWNER_ROLE, 0);
    revokeRole(OWNER_ROLE, owner);
    grantRole(OWNER_ROLE, newOwner);
}
```

This is not a safe pattern, as setting the owner or admin to the incorrect address may leave the contract unable to recover from this mistake.

Impact If an invalid admin is set with `transferAdminRole()`, then the contract will be left without an admin. Additionally, if this has occurred and an incorrect `transferOwnership()` call was made prior, then the ownership role cannot be recovered and the Registry will become immutable.

Recommendation Use an ownership structure that requires the receiving owner to confirm reception of the role, such as Openzeppelin's `Ownable2Step`.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

5.1.23 V-VLYR-VUL-023: Lack of SpecID may cause confusion to users

Severity	Low	**Commit**	a763614
Type	Data Validation	**Status**	Acknowledged
File(s)	`rust/services/call/guest/src/guest.rs`		
Location(s)	main()		
Confirmed Fix At	N/A		

The `CallAssumptions` provided as output of the main travel call circuit provides the ChainID after a previously mentioned issue (V-VLYR-VUL-001) was fixed. However, the `SpecID` utilized is still not provided in the output, and therefore the specific rules utilized by the EVM regarding the configuration of the chain are not transparent to validators of this structure.

More specifically, if a chain is forked after the deployment of an image, users may assume that the image will operate on the new fork even if the image is not updated. We recommend additionally including the `SpecID` (or a hash of the specId) in the `CallAssumptions` to make it clear to users what fork configuration was used in the execution so that potential issues can be identified early, such as the need to upgrade an image.

Impact By not including the `SpecID`, in the `CallAssumptions`, users must have intimate knowledge of the `ImageID` to know whether REVM is configured correctly. It is therefore possible that if a chain is forked after an image is deployed, the image could use an incorrect fork when executing transactions on said chain. There would be no indication to the user if this occurs unless they observe changes in the image itself and know information about the REVM configuration in the image.

Recommendation We would recommend providing the `SpecId` or a hash of the `SpecId` in the `CallAssumptions` struct which provides assumptions about the execution environment. This would allow users to compare the `SpecID` when the image is executed on different blocks to ensure that a different spec is used on different forks and possibly allow additional validation logic to be provided on-chain.

Developer Response The developers acknowledge the issue and will not implement the SpecID in the `CallAssumptions`. This is due to leading to the requirement of providing specification validation on-chain, and they rely on the fact that changes in a specification will require newly generated ImageIDs, which are whitelisted by the team.

5.1.24 V-VLYR-VUL-024: AnchorStateRegistry reads from a fixed slot

Severity	Warning	**Commit**	a763614
Type	Maintainability	**Status**	Open
File(s)	`rust/services/call/optimism/src/anchor_state_registry.rs`		
Location(s)	get_latest_confirmed_l2_commitment()		
Confirmed Fix At	N/A		

In `anchor_state_registry.rs`, the `OUTPUT_HASH_SLOT` and `BLOCK_NUMBER_SLOT` are hardcoded constants used to read from the `AnchorStateRegistry` of a given OP-stack rollup. This registry stores the latest anchored output root in a mapping that maps a `GameType` to its anchored state.

However, the main branch on the Optimism repository shows that this will no longer be a valid way to read the latest anchor root and instead the game that is currently anchored will be stored in an `anchorGame` variable.

Impact Once a chain upgrades the implementation of their `AnchorStateRegistry`, the latest anchor root cannot be verified in the VM.

Recommendation Be aware of when chains are expected to upgrade, and implement the new functionality for retrieving the anchor root.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

5.1.25 V-VLYR-VUL-025: Unexpected JSON path syntax for nested arrays

Severity	Warning	**Commit**	a763614
Type	Maintainability	**Status**	Open
File(s)	rust/services/call/precompiles/src/json.rs		
Location(s)	get_value_by_path()		
Confirmed Fix At	N/A		

The JSON precompile implements parsing of a `path` string in order to traverse a JSON object. As can be seen below, the path is split on `.` and each part of the path is assumed to have at most one array index (as specified by the `[]` characters

Snippet from `rust/services/call/precompiles/src/json.rs`

```
fn get_value_by_path<'a>(value: &'a Value, path: &str) -> Option<&'a Value> {
    path.split('.').try_fold(value, |acc, key| {
        if let Some((key, index)) = key.split_once('[').and_then(|(k, rest)| {
            rest.strip_suffix(']')
                .and_then(|i| i.parse::<usize>().ok().map(|i| (k, i)))
        }) {
            if key.is_empty() {
                acc.get(index)
            } else {
                acc.get(key)?.get(index)
            }
        } else {
            acc.get(key)
        }
    })
}
```

This leads to unconventional behavior in which nested arrays must be indexed with a dot between them.

For example, one would expect to be able to access a nested array with a `path` of `root.veridise[0][1]`. However, this syntax requires the array is indexed with a separating dot: `root.veridise[0].[1]`.

Impact Users of the JSON precompile may not know of this behavior and face troubles accessing JSON objects.

Recommendation Provide documentation surrounding this behavior, and support nested array indexing without dots if this is the desired syntax. Additionally, add testing of nested arrays in the corresponding test module.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

5.1.26 V-VLYR-VUL-026: Database seeding can cause inconsistencies

Severity	Warning	**Commit**	a763614
Type	Data Validation	**Status**	Open
File(s)	`rust/services/call/engine/src/db.rs`		
Location(s)	seed_cache_db_with_trusted_data()		
Confirmed Fix At	N/A		

When building the environments for the EVMs that will be used, the `create_env()` function will call `seed_cache_db_with_trusted_data()`. This will do 2 things with the definitions in `rust/services/call/engine/src/config.rs`:

1. Set the `AccountInfo` of each `Address` in `EMPTY_ACCOUNTS` to empty values.
2. Set the storage of accounts defined in `ACCOUNT_TO_STORAGE`.

Except for the `DEFAULT_CALLER`, the defined addresses in the config file all relate to specialized functionality on OP-Stack chains. However, the database of non OP-Stack chains (such as the base chain, Ethereum), will still get seeded with `seed_cache_db_with_trusted_data()`. Therefore, the state of the defined accounts do not match with their real state on-chain.

For example, on Ethereum the `DEFAULT_CALLER` contains ETH in its balance as it is used as a burn address.

Impact A user can prove execution about the account info of these addresses that is inaccurate.

Recommendation Avoid seeding the OP-stack related accounts for non OP-Stack chains and consider selecting a random address as the `DEFAULT_CALLER` as the current address already has token balances on Ethereum.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

5.1.27 V-VLYR-VUL-027: General smart contract recommendations

Severity	Warning	**Commit**	a763614
Type	Data Validation	**Status**	Partially Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/vlayer-xyz/vlayer/pull/1906`		

This issue documents some general recommendations around providing a more hardened configuration than what is currently implemented in order to avoid potential pitfalls for deployers and callers:

1. `contracts/vlayer/src/Verifier.sol`:
   a) The `_setTestVerifier()` function appears to be code that should not be deployed in production in order to prevent changing the verifier in production networks that are not yet checked for in the `ChainIdLibrary`.
   b) The `verifier` returned from the `ProofVerifierFactory` in the constructor should check that its `PROOF_MODE` is set to `Groth16` for mainnet deployments.
2. `contracts/vlayer/src/Seal.sol`:
   a) The `decode()` function would be more accurately named `encode()`, as it encodes the seal for callers.
3. `contracts/vlayer/src/Prover.sol`:
   a) In the `setBlock()` function, validate that the caller is traveling backwards. Although this does not introduce security surrounding the precompile, it would assist honest provers.
4. `contracts/vlayer/src/CallAssumptions.sol`:
   a) The `validateAssumptions()` function is currently unused. The `ProofVerifierBase` `_verifyExecutionEnv()` function duplicates the check that is implemented by `validateAssumptions()`, alongside additional checks surrounding the `settleBlockNumber`. It may be beneficial to move those additional checks into `validateAssumptions()` and use said function inside of `_verifyExecutionEnv()` to deduplicate code.
5. `contracts/vlayer/src/WebProof.sol`
   a) The `recover()` function can be directly invoked by users of the `WebProofLib`, potentially if they do not have a pattern to test the URL against. This function omits the public key check implemented in the `verify()` function. Therefore, even if this function is unintended to be directly used, it would be beneficial to move the public key check to `recover()`.

Impact Potential misconfigurations may occur leading to unintended behavior.

Recommendation Implement the recommendations

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

5.1.28 V-VLYR-VUL-028: Domain owners can prove arbitrary emails

Severity	Warning	**Commit**	a763614
Type	Access Control	**Status**	Open
File(s)	`rust/email_proof/src/lib.rs`		
Location(s)	parse_and_verify()		
Confirmed Fix At	N/A		

To verify an email proof, one verifies that the public key of the host domain signed the extracted information of the email. This public key, however, can be used to sign emails regardless of whether they were sent. As such if a key is compromised or if the host is malicious, signed emails may not actually have been sent.

Impact Certain applications may want to consider this, especially if they allow senders to submit emails as the proof may not imply what the application intends.

Recommendation Consider including a warning or best-practices page documenting potential risks that applications should consider.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

5.1.29 V-VLYR-VUL-029: Proofs may be replayed or frontrun

Severity	Warning	**Commit**	a763614
Type	Frontrunning	**Status**	Open
File(s)	`contracts/vlayer/src/Verifier.sol`		
Location(s)	onlyVerified()		
Confirmed Fix At	N/A		

Applications that interact with ZK proofs need to be aware of two particular issues: replay attacks and frontrunning. Specifically, a replay attack can occur when a proof may be submitted multiple times when an application intends accepted proofs to be unique. Frontrunning may then be an issue if a particular application enforces proof uniqueness as another individual may frontrun verification to spend the proof first. Oftentimes solutions to the above problems are unique to the ZK application however and so it is up to those building applications on top of vLayer to provide their own solutions.

Impact Frontrunning and replay attacks can be used to break invariants that an application intends to hold. This could potentially have significant impacts on the application itself such as theft.

Recommendation Consider providing best-practices in the documentation that explains how these attacks may occur and provide examples of how one can defend against them. For example, to prevent against a replay attack, a prover may return a nullifier and ensure that the nullifier is not used upon verification. Frontrunning is often prevented by indicating ownership in a proof which can be performed here by returning the address of the individual who can spend the proof.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

5.1.30 V-VLYR-VUL-030: General Rust recommendations

Severity	Warning	**Commit**	a763614
Type	Maintainability	**Status**	Open
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	N/A		

This issue provides general recommendations surrounding the Rust code in order to provide clarity or ease of usage.

1. `rust/mpt/src/node/rlp.rs`:
   a) This module should be directly tested to ensure the RLP encoding functionality is correct.
2. `rust/mpt/src/node/constructors.rs`:
   a) `branch_with_child_node()`: Although this function is only called from an `Extension` node, it would be more clear to rename this function to one that signifies this relationship. Alternatively, the function could check that the child is a branch node in order to ensure that a leaf is not mistakenly inserted with the incorrect key nibbles.
3. `rust/mpt/src/key_nibbles.rs`:
   a) `split_first()`: Out of bounds access can panic, it would be useful to provide a better error to pinpoint the issue.
4. `rust/services/call/guest/src/db/state.rs`:
   a) The `KECCAK_EMPTY` constant can be imported from REVM to better ensure the correct hash is used.
5. `rust/services/chain/common/src/lib.rs`:
   a) `fake_proof_result()`: Unlike other testing functions in this file that are only compiled with the testing feature (`#[cfg(feature = "testing")]`), the function is missing this decorator.
6. `rust/services/call/precompiles/src/regex.rs`:
   a) `validate_regex()`: Documentation should be provided surrounding the requirement that regex patterns must match against a full line.
7. `rust/chain/src/fork.rs`:
   a) `partial_cmp()`: The comparison between two `Forks` does not differentiate between a `Block` and `Timestamp`. In the forks specified in the `chain_specs.toml` file, this is fine as all Ethereum blocks pre-merge (specified by block number) are less than blocks post-merge (specified by timestamp). However, this can be fragile and should be documented for future addition of chains.
8. `rust/email_proof/src/dns`:

   a) `parse_dns_record()`: The DKIM RFC *recommends* checking the version field in the DNS record (in which it must be exactly "DKIM1").

9. `rust/email_proof/src/from_header.rs`:

   a) `extract_from_domain()`: The `email_address.rs` file contains the `split_email()` function, which should be used instead of reimplementing the functionality in this function.

10. `rust/email_proof/src/email.rs`:

    a) The `validate_email` function would be more accurately named `validate_email_address` as the function validates an email address rather than the email itself.

11. ****`rust/mpt/src/node/constructors.rs`: 1. The `branch_with_two_children` function takes as input two nodes MPT nodes that are combined using a MPT branch. If the two nodes have the same id, however, the second node will overwrite the first node. Consider validating that `first_idx != second_idx` in this function rather than the callers to ensure it is used properly in the future.

Impact Potential misuse or confusions surrounding the mentioned functions may occur.

Recommendation Implement the recommendations.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

Glossary

EVM The Ethereum Virtual Machine (EVM) is a virtual environment designed for smart contracts that executes code in a deterministic manner . 1

smart contract A self-executing contract with the terms directly written into code. Hosted on a blockchain, it automatically enforces and executes the terms of an agreement between buyer and seller. Smart contracts are transparent, tamper-proof, and eliminate the need for intermediaries, making transactions more efficient and secure. 1

zero-knowledge circuit A cryptographic construct that allows a prover to demonstrate to a verifier that a certain statement is true, without revealing any specific information about the statement itself. See `https://en.wikipedia.org/wiki/Zero-knowledge_proof` for more. 54

zkVM A general-purpose zero-knowledge circuit that implements proving the execution of a virtual machine. This enables general purpose programs to prove their execution to outside observers, without the manual constraint writing usually associated with zero-knowledge circuit development . 1