Rhinestone

Module Registry

by Ackee Blockchain

3.7.2024

Contents

1. Document Revisions

0.1	Draft report	19.4.2024
1.0	Final report	5.6.2024
1.1	Fix review	19.6.2024
1.2	Fix review	3.7.2024

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain

Ackee Blockchain is an auditing company based in Prague, Czech Republic, specializing in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run free certification courses School of Solana, Summer School of Solidity and teach at the Czech Technical University in Prague. Ackee Blockchain is backed by the largest VC fund focused on blockchain and DeFi in Europe, RockawayX.

2.2. Audit Methodology

1. **Technical specification/documentation** - a brief overview of the system is requested from the client and the scope of the audit is defined.
2. **Tool-based analysis** - deep check with automated Solidity analysis tools and Wake is performed.
3. **Manual code review** - the code is checked line by line for common vulnerabilities, code duplication, best practices and the code architecture is reviewed.
4. **Local deployment + hacking** - the contracts are deployed locally and we try to attack the system and break it.
5. **Unit and fuzz testing** - run unit tests to ensure that the system works as expected, potentially write missing unit or fuzz tests.

2.3. Finding classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**-**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.), but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or configuration (see above) was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review team

Member's Name	Position
Štěpán Šonský	Lead Auditor
Michal Převrátil	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We've put our best effort to find all vulnerabilities in the system, however our findings shouldn't be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

Revision 1.0

Rhinestone engaged Ackee Blockchain to perform a security review of the Rhinestone protocol with a total time donation of 7 engineering days in a period between April 10 and April 19, 2024, with Štěpán Šonský as the lead auditor.

The audit was performed on the commit 6f5e84a [1] and the scope was the following:

- ./core/Attestation.sol
- ./core/AttestationManager.sol
- ./core/ModuleManager.sol
- ./core/ResolverManager.sol
- ./core/SchemaManager.sol
- ./core/SignedAttestation.sol
- ./core/TrustManager.sol
- ./core/TrustManagerExternalAttesterList.sol
- ./lib/AttestationLib.sol
- ./lib/Helpers.sol
- ./lib/ModuleDeploymentLib.sol
- ./lib/ModuleTypeLib.sol
- ./lib/StubLib.sol
- ./lib/TrustLib.sol
- ./Common.sol

- ./DataTypes.sol
- ./Registry.sol

We began our review using static analysis tools, including Wake in companion with Tools for Solidity VS Code extension. We then took a deep dive into the logic of the contracts. For testing and fuzzing, we have involved Wake testing framework. Implemented fuzz tests are available on GitHub [2]. During the review, we paid special attention to:

- checking module deployment can not be misused,
- detecting possible reentrancies in the code,
- checking possible front-running,
- checking for denial of service attacks,
- ensuring access controls are not too relaxed or too strict,
- looking for common issues such as data validation.

Our review resulted in 20 findings, ranging from Info to High severity. The most severe issue is the invalid behavior of the registry in the case when the threshold is equal to 1, which leads to denial of service in most situations (see H1). The finding was discovered by fuzzing with the Wake testing framework. The overall code quality is good. The project is well-structured and contains detailed documentation and comments.

Ackee Blockchain recommends Rhinestone:

- remove the optimization for `threshold = 1`,
- separate the factory logic to the neutral contract,
- resolve the risk of front-running,
- implement two-step ownership transfer,

- address all other reported issues.

See Revision 1.0 for the system overview of the codebase.

Revision 1.1

The review was done on the given commit: `0b4b232`. 14 issues were fixed, 3 issues acknowledged, 1 issue invalidated and 2 issues (M2 and W3) were not fixed. We recommend to fix the remaining issues. The fix review was focused only on issues remediations, other code changes (if any) were not audited.

See Revision 1.1 for the review of the updated codebase and additional information we consider essential for the current scope.

Revision 1.2

Rhinestone engaged Ackee Blockchain to review the fixes of the two issues that were not fixed in the previous revision. The review was performed on the commit `10c3719` [3]. The scope only included the fixes for issues M2 and W3, and no other changes in the codebase were reviewed.

[1] full commit hash: 6f5e84a0b38ab40de377177f51d59e71be783cee

[2] fuzz tests: https://github.com/Ackee-Blockchain/tests-rhinestone-registry

[3] full commit hash: 10c3719589252529f40f170f49cb2768508c8572

4. Summary of Findings

The following table summarizes the findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- a *Description*,
- an *Exploit scenario*,
- a *Recommendation* and if applicable
- a *Fix*.

There might often be multiple ways to solve or alleviate the issue, with varying requirements regarding the necessary changes to the codebase. In that case, we will try to enumerate them all, clarifying which solves the underlying issue better (albeit possibly only with architectural changes) than others.

	Severity	Reported	Status
H1: `threshold = 1` optimization DoS	High	1.0	Fixed
M1: Arbitrary call on factory	Medium	1.0	Fixed
M2: Attesters are not de-duplicated	Medium	1.0	Fixed
M3: `registerModule` front-running	Medium	1.0	Acknowledged
M4: `trustAttesters` downcast	Medium	1.0	Fixed
L1: Resolver one-step ownership transfer	Low	1.0	Acknowledged

	Severity	Reported	Status
W1: Deployment and attestation denial of service	Warning	1.0	Invalidated
W2: Inconsistent revert errors	Warning	1.0	Fixed
W3: EIP-712 compliance	Warning	1.0	Fixed
W4: `findTrustedAttesters` revert on no attesters	Warning	1.0	Fixed
W5: `trustAttesters` zero address validation	Warning	1.0	Fixed
W6: Inconsistent data validation	Warning	1.0	Fixed
W7: `TrustLib` high-order bits not cleared	Warning	1.0	Fixed
I1: Multiple interfaces	Info	1.0	Fixed
I2: Inconsistent parameter naming	Info	1.0	Fixed
I3: Duplicated code	Info	1.0	Acknowledged
I4: Modifier placement	Info	1.0	Fixed
I5: Missing NatSpec documentation	Info	1.0	Fixed
I6: `_storeAttestation` false comment	Info	1.0	Fixed
I7: `NewTrustedAttesters` event	Info	1.0	Fixed

Table 2. Table of Findings

5. Report revision 1.0

5.1. System Overview

This section contains an outline of the audited contracts. Note that this is meant for understandability purposes and does not replace project documentation.

Contracts

Contracts we find important for better understanding are described in the following section.

Attestation.sol

Provides basic operations for attestations. Functions `attest` and `revoke` call internal `_attest` and `_revoke` functions from parent contract `AttestationManager`. View functions `findAttestation` and `findAttestations` return attestations from `AttestationManager` parent.

AttestationManager.sol

Contains internal logic for attestation management, namely functions `_attest`, `_revoke`, `_storeAttestation`, and `_storeRevocation`. Store functions convert requests to storage records. Function `_storeAttestation` utilizes `AttestationLib` library for storing attestation data using `sstore2` pattern.

ModuleManager.sol

Provides functions for module deployment (direct or via factory), module registration and storing logic for `ModuleRecord` struct. After the deployment and registration, it calls the `resolveModuleRegistrationexternal` function on the resolver (if present) and reverts if the call returns `false`.

ResolverManager.sol

Manager for registering and setting external resolvers. Also, it provides ownership transfer for resolvers.

SchemaManager.sol

Allows registering attestation schemas with the optional validator (contract implements `IExternalSchemaValidator` interface).

SignedAttestation.sol

Inherits from the `Attestation` contract and adds `attest` and `revoke` functions with `signature` parameter to allow perform these operations on behalf of the attester using off-chain signing. Signature validation is provided by Solady `SignatureCheckerLib` library and performed using the `ECDSA.recover` function for EOA and ERC-1271 for smart contracts.

TrustManager.sol

Allows `msg.sender` to trust attesters (`trustAttesters` function) by defining the addresses of attesters and threshold. Threshold sets, how many attestations are required to consider the module trustworthy. The `TrustManager` contract also provides `check` functions for checking module attestations for `msg.sender` or other accounts. And finally, the function `findTrustedAttesters` which returns trusted attesters for the specified `smartAccount` address.

TrustManagerExternalAttesterList.sol

Inherits from `TrustManager` contract and adds another two `check` functions to query attestations with trusted `attesters` array in calldata parameters.

AttestationLib.sol

Library for `SSTORE2` operations from Solady (functions `sload2` and `sstore2`), calculating salt (`sstore2Salt` function), and hashing `AttestationRequest` and

`RevocationRequest` structs with the `nonce` parameter.

Helpers.sol

Library with `getUID` functions which calculate UIDs for `SchemaRecord` and `ResolverRecord` structs using the `wrap` function.

ModuleDeploymentLib.sol

Library with helper functions for CREATE2 deployment (`deploy` function) and calculating the target address (`calcAddress` function). Also, it contains `containsCaller` modifier for the `deploy` function which validates `msg.sender` against the first 20 bytes of the `salt` parameter.

ModuleTypeLib.sol

Library for packing module types into `PackedModuleTypes` data type (`uint32`), and function `isType` for checking if the specified type is included in the `PackedModuleTypes` instance.

StubLib.sol

Helper library for interacting with resolvers (`IExternalResolver` interface) and schema validators (`IExternalSchemaValidator` interface). Included functions call schema validator contract to validate the schema (`requireExternalSchemaValidation`) and perform additional logic on `resolver` contracts during the module attestation using the `requireExternalResolverOnAttestation` function, on module revocation using the `tryExternalResolverOnRevocation` function, and on module registration using the `requireExternalResolverOnModuleRegistration` function.

TrustLib.sol

Implements validation checks for the storage reference of the `AttestationRecord` struct. The `enforceValid` function enforces the validity and reverts in case of invalid data. The `checkValid` function returns `false` in case of invalid data.

Common.sol

Contains all basic constant definitions and helper functions `_time` which returns `block.timestamp` and `_isContract` which checks the code length of the address and determines whether it is a contract.

DataTypes.sol

Contains all structs used by the project, namely `AttestationRecord`, `ModuleRecord`, `SchemaRecord`, `ResolverRecord`, `TrustedAttesterRecord`, `AttestationRequest`, and `RevocationRequest`. Also, it contains custom type definitions with operator overrides (`SchemaUID`, `ResolverUID`, `AttestationDataRef`, `PackedModuleTypes`, and `ModuleType`).

Registry.sol

Inherits from the `SignedAttestation` contract and does not contain any additional logic. See the inheritance graph, generated by the [Tools for Solidty (Wake)] tool.

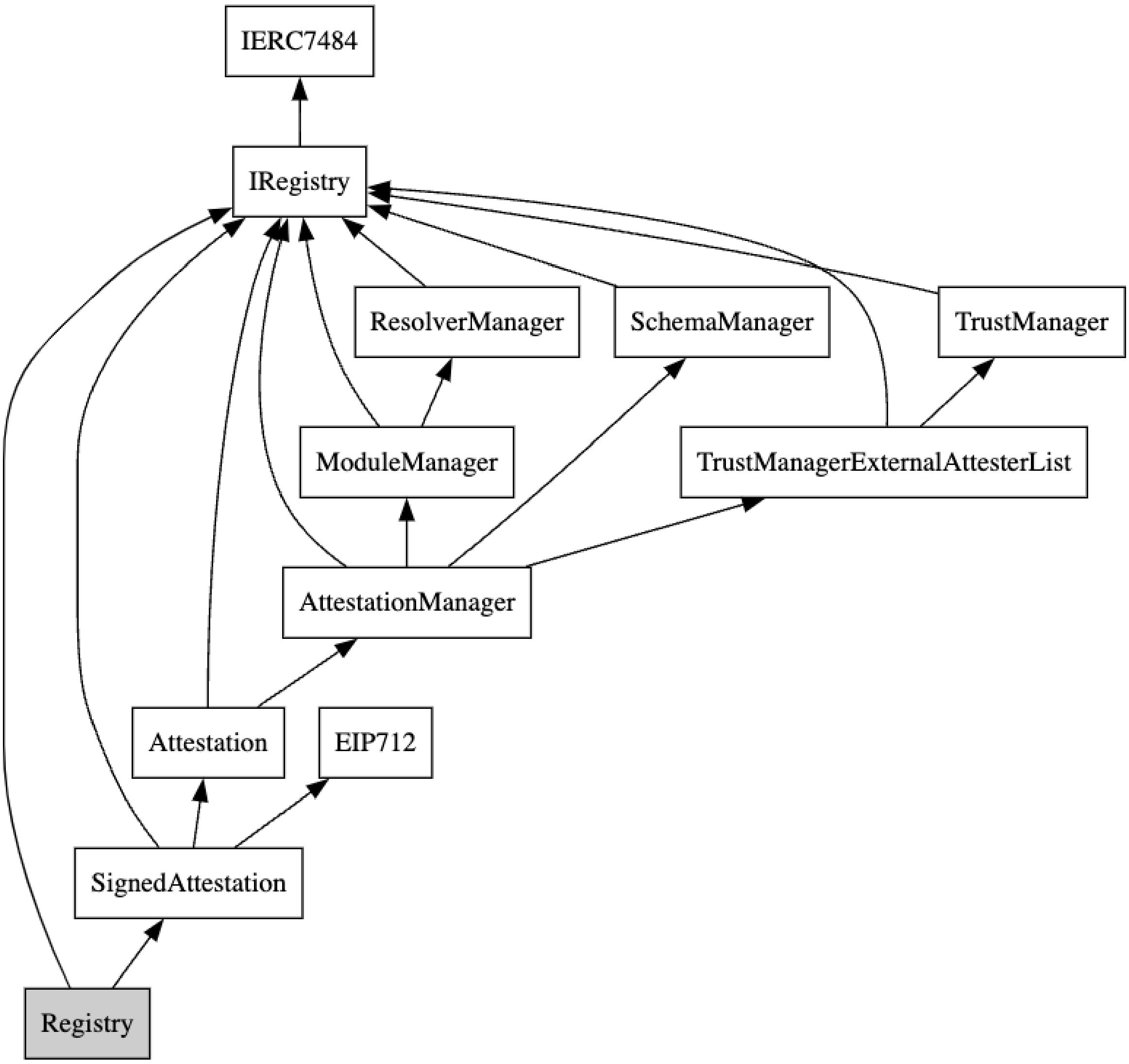

Figure 1. Registry inheritance graph

Actors

Any EOA

Can deploy and register their own resolvers, schema validators and modules.

Smart account

Smart accounts can manage and query trusted attesters and threshold through the `TrustManager`.

Registry

The core entity of the system with complete set of features described above. According to the client, it is not an entity intended to hold any funds and interact with 3rd party contracts.

Resolver

Resolver is used to perform additional checks and logic after the module deployment and registration. Anyone can deploy and register their own resolver.

Resolver owner

The resolver owner can call the `ResolverManager.setResolver` function and transfer the ownership to another address. Also, it is possible to renounce the ownership by transferring it to zero-address.

Schema validator

Schema validators are used for schema validations after attestations using the `validateSchema` function. Anyone can deploy and register their own schema validator.

Owner

The deployer of the `Registry` contract does not have any special privileges in the system.

5.2. Trust Model

From actor roles arise the following trust assumptions. The `Registry` contract does not include any privileged roles and the contract is not upgradeable and, therefore is completely autonomous after the deployment. However, users need to trust external resolvers and schema validators which can be arbitrary and potentially malicious.

H1: `threshold = 1` optimization DoS

High severity issue

Impact:	High	Likelihood:	Medium
Target:	TrustManager.sol	Type:	Denial of service

Description

The function `_check` in the `TrustManager` contract is used to verify if there are enough attestations of a module from smart account trusted attesters. The code contains an optimization for `threshold = 1`, checking only the first attester and enforcing their attestation to be present and valid.

Listing 1. Excerpt from [TrustManager]()

```
        // smart account only has ONE trusted attester
        // use this condition to save gas
        else if (threshold == 1) {
            AttestationRecord storage $attestation = $getAttestation({ module: module, attester: attester });
            $attestation.enforceValid(moduleType);
        }
```

The finding was discovered by a fuzz test written in the [Wake]() testing framework.

Exploit scenario

A smart account owner sets 3 trusted attesters with `threshold = 1`, i.e., a single valid attestation is required. The last-stored attester submits an attestation for a new module.

The smart account owner wants to integrate the module and calls the `check` function. Due to the broken optimization, the function will return `false` as the attestation of the first attester is not present.

Recommendation

Remove the optimization for `threshold = 1` and check all attesters.

Fix 1.1

Fixed. The `_check` function logic was updated. The optimization for `threshold = 1` is still present, but all attesters are checked in case the first attestation is not valid.

Go back to Findings Summary

M1: Arbitrary call on factory

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	ModuleManager.sol	Type:	External call

Description

The function `ModuleManager.deployViaFactory` allows executing arbitrary calls on arbitrary `factory` contract on behalf of the `Registry` contract using the `callOnFactory` parameter. The validation in the `requireExternalResolverOnModuleRegistration` function can be bypassed using the custom `resolver`.

Listing 2. Excerpt from [ModuleManager]

```
        (bool ok, bytes memory returnData) = factory.call{ value: msg.value
}(callOnFactory);
```

According to the client the `Registry` contract is not meant to interact with any 3rd party protocols. Therefore the risk of misusing this attack vector is not actual. Also, the `_storeModuleRecord` function contains the condition `_isContract` for the external call's return value which limits the attack possibilities. However, the solution design should be future-proof and all potential back doors should be closed.

Exploit scenario

The following example exploit scenarios are not applicable in the current state rather explain the general risk of this approach.

First example:

1. Let's assume that the `Registry` contract holds some tokens.
2. The attacker passes an address of the ERC-20 token as the `factory` parameter into the `deployViaFactory` function and encodes the `ERC20.approve` function with the attacker's address as a `spender` into the `callOnFactory` parameter.
3. Then the `Registry` contract calls the `ERC20.approve` function on the token contract and approves the attacker to drain all tokens from the `Registry` contract.

Second example:

1. Let's assume the `Registry` contract is staking tokens in some staking protocol.
2. The attacker passes the address of the staking contract as the `factory` into the `deployViaFactory` function and encodes the `withdraw` function into the `callOnFactory` parameter.
3. The `Registry` contract executes the token withdrawal from the staking contract.
4. The attacker uses the first example scenario to drain the funds.

Allowing to pass the target address and raw call data into the function and executing this external call opens a limitless amount of possible attack vectors.

Recommendation

Move the external call to the separated neutral contract divided from the `Registry` contract to remove the `Registry` from `msg.sender`.

Fix 1.1

Fixed using the new `FactoryTrampoline` contract.

> Added factory call trampoline, so calls made to factory don’t come from msg.sender == registry.
>
> — Rhinestone

Go back to Findings Summary

M2: Attesters are not de-duplicated

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	TrustManagerExternalAttesterList.sol	Type:	Data validation

Description

The overloaded function `check` in `TrustManagerExternalAttesterList` does not verify if the supplied list of attesters does not contain duplicates.

Listing 3. Excerpt from [TrustManagerExternalAttesterList]()

```
        for (uint256 i; i < attestersLength; ++i) {
            if ($getAttestation(module, attesters[i]).checkValid(ZERO_MODULE_TYPE)) {
                --threshold;
            }
            if (threshold == 0) return;
        }
```

Listing 4. Excerpt from [TrustManagerExternalAttesterList]()

```
        for (uint256 i; i < attestersLength; ++i) {
            if ($getAttestation(module, attesters[i]).checkValid(moduleType)) {
                --threshold;
            }
            if (threshold == 0) return;
        }
```

Exploit scenario

Due to an off-chain implementation issue, the attesters array contains duplicated addresses. The `check` function returns `true` even for `threshold = 2` and the attesters array `[A, A, B]` with `A` being the only attester that attested a given module.

Recommendation

Sort and de-duplicate the attester arrays. Optionally, assume the arrays are already sorted, verify that and check for duplicates. Make the behavior consistent with the `TrustManager` contract (see W6).

Update 1.1

The following condition was added to the `check` functions:

```
if (attester < _attesterCache) revert InvalidTrustedAttesterInput();
else _attesterCache = attester;
```

The check prevents supplying the zero address but does not prevent duplicates due to the incorrect inequality sign (`<`) used in the condition.

Fix 1.2

The inequality sign was fixed to `<=` to prevent duplicates.

Go back to Findings Summary

M3: `registerModule` front-running

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	ModuleManager.sol	Type:	Front running

Description

The function `ModuleManager.registerModule` is vulnerable to the front-running attack. The `metadata` parameter validation relies on the external `resolver` contract. The front-runner can inject arbitrary data into the `metadata` parameter if the `metadata` parameter validation is missing in the `resolver` contract. Also, the validation can be bypassed by deploying and registering a custom `resolver` contract.

Listing 5. Excerpt from ModuleManager

```
    function registerModule(ResolverUID resolverUID, address moduleAddress, bytes calldata metadata) external {
        ResolverRecord storage $resolver = $resolvers[resolverUID];

        // ensure that non-zero resolverUID was provided
        if ($resolver.resolverOwner == ZERO_ADDRESS) revert InvalidResolver($resolver.resolver);

        ModuleRecord memory record = _storeModuleRecord({
            moduleAddress: moduleAddress,
            sender: ZERO_ADDRESS, // setting sender to address(0) since anyone can invoke this function
            resolverUID: resolverUID,
            metadata: metadata
        });

        // resolve module registration
        record.requireExternalResolverOnModuleRegistration({ moduleAddress: moduleAddress, $resolver: $resolver });
    }
```

Exploit scenario

A malicious front-running bot is waiting for `registerModule` transaction in the pool and creates a front-running transaction with injected arbitrary `resolverUID` and/or `metadata` for the registered module. After the successful module registration, the arbitrary `resolverUID` and/or `metadata` get stored in the `ModuleRecord`. Using this technique the bot can systematically sabotage registrations of externally deployed modules.

Recommendation

One of the possible solutions would be moving the `metadata` into the deployed module contract. Also, the module contract can contain `resolverUID` whitelist to avoid arbitrary `resolver` assignments.

Fix 1.1

Acknowledged.

> We are aware of the frontrunning issue as described in the code. The problem specifically impacts modules not deployed through the registry as a factory. The majority of modules are unaffected. The registration function susceptible to frontrunning is intentionally positioned to accommodate external factories.
>
> — Rhinestone

[Go back to Findings Summary]

M4: `trustAttesters` downcast

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	TrustManager.sol	Type:	Data validation

Description

The function `trustAttesters` in `TrustManager` contract is used to set a new set of trusted attester addresses for a given account.

Listing 6. Excerpt from [TrustManager](#)

```
        if (threshold > attestersLength) {
            threshold = uint8(attestersLength);
        }

        $trustedAttester.attesterCount = uint8(attestersLength);
```

The length of the attesters array is downcasted to `uint8`, making the function dysfunctional for 256 attesters or more.

Recommendation

Consider using a bigger data type to store the attesters length. In all cases, use `SafeCast` or an alternative to revert the transaction if the length of the attesters array is greater than the maximum for the used data type.

Fix 1.1

Fixed, the downcasting was removed.

[Go back to Findings Summary](#)

L1: Resolver one-step ownership transfer

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	ResolverManager.sol	Type:	Access controls

Description

The `ResolverManager` contract uses a one-step ownership transfer. Transferring ownership to the wrong address can lead to permanent loss of access to the `ResolverManager.setResolver` and `transferResolverOwnership` functions protected by `onlyResolverOwner` modifier.

Also, it lacks a zero-address validation therefore allowing permanent ownership renouncement.

Listing 7. Excerpt from [ResolverManager]()

```
    function transferResolverOwnership(ResolverUID uid, address newOwner)
external onlyResolverOwner(uid) {
        $resolvers[uid].resolverOwner = newOwner;
        emit NewResolverOwner(uid, newOwner);
    }
```

Recommendation

Implement a two-step ownership transfer and add a zero-address check if renouncing ownership is not an intended feature.

Fix 1.1

Acknowledged.

> Not checking zero address is a feature that allows a resolver to

> become non-changeable.
>
> — Rhinestone

[Go back to Findings Summary]

W1: Deployment and attestation denial of service

Impact:	Warning	Likelihood:	N/A
Target:	ModuleManager.sol	Type:	Denial of service

Description

Resolver and schema validators can block module deployment and attestation using the `resolveAttestation` function on the resolver and the `validateSchema` function on the schema validator. The more important revocation process cannot be blocked using this scenario. According to the client, this is known and intended behavior.

> This is expected behavior.
>
> — Rhinestone

Recommendation

Including this warning in the report for external readers' information to be aware of this non-severe risk.

Fix 1.1

Acknowledged.

> This is expected behavior.
>
> — Rhinestone

[Go back to Findings Summary]

W2: Inconsistent revert errors

Impact:	Warning	Likelihood:	N/A
Target:	ModuleManager.sol	Type:	Errors

Description

The function `ModuleManager.deploy` reverts with `InvalidResolver` error and `ModuleManager.deployViaFactory` reverts with `InvalidResolverUID` error.

Also, due to the resolver ownership transfer it is able to set the owner to zero-address and in this edge-case, the `InvalidResolverUID` error would not be relevant.

Listing 8. Excerpt from [ModuleManager]

```
        if ($resolver.resolverOwner == ZERO_ADDRESS) revert InvalidResolver(
    $resolver.resolver);
```

Listing 9. Excerpt from [ModuleManager]

```
        if ($resolver.resolverOwner == ZERO_ADDRESS) revert
    InvalidResolverUID(resolverUID);
```

Recommendation

Consider unifying these errors.

Fix 1.1

Fixed. The error in the `deployModule` function was changed to `InvalidResolverUID`.

[Go back to Findings Summary]

W3: EIP-712 compliance

Impact:	Warning	Likelihood:	N/A
Target:	AttestationLib.sol	Type:	EIP compliance

Description

The signatures verified in the `TrustManagerExternalAttesterList` contract are not EIP-712 compliant.

The standard requires a top-level struct defined for signed data with type hash of the struct signed together with the data.

Listing 10. Excerpt from AttestationLib

```
    function hash(AttestationRequest calldata data, uint256 nonce) internal pure returns (bytes32 _hash) {
        _hash = keccak256(abi.encode(ATTEST_TYPEHASH, keccak256(abi.encode(data)), nonce));
    }

    function hash(AttestationRequest[] calldata data, uint256 nonce) internal pure returns (bytes32 _hash) {
        _hash = keccak256(abi.encode(ATTEST_TYPEHASH, keccak256(abi.encode(data)), nonce));
    }

    function hash(RevocationRequest calldata data, uint256 nonce) internal pure returns (bytes32 _hash) {
        _hash = keccak256(abi.encode(REVOKE_TYPEHASH, keccak256(abi.encode(data)), nonce));
    }

    function hash(RevocationRequest[] calldata data, uint256 nonce) internal pure returns (bytes32 _hash) {
        _hash = keccak256(abi.encode(REVOKE_TYPEHASH, keccak256(abi.encode(data)), nonce));
    }
```

However, in the code snippet, type hashes of a single attestation and revocation requests are used. The nonce is not a part of any struct, and the type hashes do not describe that arrays of requests are signed.

Recommendation

Define four more wrapper structs for attestation and revocation requests, wrapping the nonce and the array of requests or a single request. Follow [EIP-712] when preparing type hashes and the final data payload.

Update 1.1

Wrapper structs were introduced in the `TrustManagerExternalAttesterList` contract. However, the hash generation still does not comply with EIP-712 because the data encoding is not applied correctly on nested structs and arrays.

Fix 1.2

Fixed. The EIP-712 compliance was achieved by applying the correct data encoding on nested structs and arrays and unifying the naming in structs and type hashes.

Go back to Findings Summary

W4: `findTrustedAttesters` revert on no attesters

Impact:	Warning	Likelihood:	N/A
Target:	TrustManager.sol	Type:	User experience

Description

The function `findTrustedAttesters` returns the array of the attesters currently set for a given smart account.

Listing 11. Excerpt from [TrustManager]

```
    function findTrustedAttesters(address smartAccount) public view returns
(address[] memory attesters) {
        TrustedAttesterRecord storage $trustedAttesters =
$accountToAttester[smartAccount];

        uint256 count = $trustedAttesters.attesterCount;
        address attester0 = $trustedAttesters.attester;

        attesters = new address[](count);
        attesters[0] = attester0;

        for (uint256 i = 1; i < count; i++) {
            // get next attester from linked List
            attesters[i] = $trustedAttesters.linkedAttesters[attesters[i -
1]];
        }
    }
```

The function reverts with the panic code 50 (out-of-bounds index access) when a smart contract has no attesters set.

Recommendation

Consider returning an empty array or reverting with a more user-friendly error message/data.

Fix 1.1

Fixed. The `findTrustedAttesters` function returns an empty array when `attesterCount == 0`.

[Go back to Findings Summary](#)

W5: `trustAttesters` zero address validation

Impact:	Warning	Likelihood:	N/A
Target:	TrustManager.sol	Type:	Data validation

Description

The function `trustAttesters` sets trusted attesters for an account. It performs the zero address validation for all attester addresses except the last one.

Listing 12. Excerpt from [TrustManager]()

```
        $trustedAttester.attesterCount = uint8(attestersLength);
        $trustedAttester.threshold = threshold;
        $trustedAttester.attester = attesters[0];

        attestersLength--;

        // setup the linked list of trusted attesters
        for (uint256 i; i < attestersLength; i++) {
            address _attester = attesters[i];
            // user could have set attester to address(0)
            if (_attester == ZERO_ADDRESS) revert InvalidTrustedAttesterInput();
            $trustedAttester.linkedAttesters[_attester] = attesters[i + 1];
        }
```

Consequently, it is possible to call the `trustAttesters` function with the last attester being the zero address.

Recommendation

Perform the zero address validation even for the last attester.

Fix 1.1

Fixed. Added `attesters[0] != ZERO_ADDRESS` validation in combination with `attesters.isSortedAndUniquified()` validation.

[Go back to Findings Summary](#)

W6: Inconsistent data validation

Impact:	Warning	Likelihood:	N/A
Target:	TrustManager.sol, TrustManagerExternalAttesterList.sol	Type:	Data validation

Description

The contracts `TrustManager` and `TrustManagerExternalAttesterList` contain multiple behavioral inconsistencies even though the functionality is almost identical.

The inconsistencies are:

- `threshold` is truncated to `attestersLength` in the `TrustManager` contract,
- the case `threshold = 0` is handled differently in both contracts,
- different revert data in the case of no trusted attesters set,
- attester de-duplication is not performed in the `TrustManagerExternalAttesterList` contract.

Recommendation

It is recommended to revert on the first occurrence of invalid data and not to adjust parameters to reasonable values.

Specifically, it is recommended to:

- revert in the `TrustManagerExternalAttesterList` contract if `threshold = 0` with the same error as in `TrustManager` contract,
- revert in the `trustAttesters` function if `threshold > attestersLength`, make the behavior consistent with the `TrustManagerExternalAttesterList`

contract,

- use the same revert data in the case of no trusted attesters (empty array) in both contracts,
- perform the attester de-duplication in the `TrustManagerExternalAttesterList` contract or check that attesters are sorted and unique, keep the behavior consistent with `TrustManager`.

Fixed. The recommendations were applied.

[Go back to Findings Summary]

W7: `TrustLib` high-order bits not cleared

Impact:	Warning	Likelihood:	N/A
Target:	TrustLib.sol	Type:	Data validation

Description

In the following code snippets, assembly is used to extract data from a single storage slot.

Listing 13. Excerpt from [TrustLib]()

```
        assembly {
            let mask := 0xffffffffffff
            let slot := sload($attestation.slot)
            attestedAt := and(mask, slot)
            slot := shr(48, slot)
            expirationTime := and(mask, slot)
            slot := shr(48, slot)
            revocationTime := and(mask, slot)
            slot := shr(48, slot)
            packedModuleType := and(mask, slot)
        }
```

Listing 14. Excerpt from [TrustLib]()

```
        assembly {
            let mask := 0xffffffffffff
            let slot := sload($attestation.slot)
            attestedAt := and(mask, slot)
            slot := shr(48, slot)
            expirationTime := and(mask, slot)
            slot := shr(48, slot)
            revocationTime := and(mask, slot)
            slot := shr(48, slot)
            packedModuleType := and(mask, slot)
        }
```

`packedModuleType` is of the type `uint32`, but the used mask extracts 48 bits, opening a possibility of preserving dirty high-order bits.

Recommendation

Use a 32-bit mask when extracting `packedModuleType`.

Fix 1.1

Fixed. The 32-bit mask is now used.

Go back to Findings Summary

I1: Multiple interfaces

Impact:	Info	Likelihood:	N/A
Target:	IRegistry.sol	Type:	Best practices

Description

The `IRegistry.sol` file contains `IERC7484` and `IRegistry` interfaces.

Recommendation

Move the `IERC7484` interface to a separate file.

Fix 1.1

Fixed. The `IERC7484` interface was separated and moved to the interfaces directory.

Go back to Findings Summary

I2: Inconsistent parameter naming

Impact:	Info	Likelihood:	N/A
Target:	DataTypes.sol	Type:	Naming

Description

Parameters in `DataTypes.schemaEq` and `DataTypes.schemaNeq` overloaded operators are named `uid1` and `uid`, all other functions use `uid1` and `uid2`.

Recommendation

Unify the naming.

Fix 1.1

Fixed. The parameters were renamed.

Go back to Findings Summary

I3: Duplicated code

Impact:	Info	Likelihood:	N/A
Target:	TrustLib.sol	Type:	Code quality

Description

The `TrustLib` library contains a duplicated assembly code.

Listing 15. Excerpt from [TrustLib]()

```
        assembly {
            let mask := 0xffffffffffff
            let slot := sload($attestation.slot)
            attestedAt := and(mask, slot)
            slot := shr(48, slot)
            expirationTime := and(mask, slot)
            slot := shr(48, slot)
            revocationTime := and(mask, slot)
            slot := shr(48, slot)
            packedModuleType := and(mask, slot)
        }
```

Listing 16. Excerpt from [TrustLib]()

```
        assembly {
            let mask := 0xffffffffffff
            let slot := sload($attestation.slot)
            attestedAt := and(mask, slot)
            slot := shr(48, slot)
            expirationTime := and(mask, slot)
            slot := shr(48, slot)
            revocationTime := and(mask, slot)
            slot := shr(48, slot)
            packedModuleType := and(mask, slot)
        }
```

Recommendation

Move the block into a separated private function.

Fix 1.1

Acknowledged.

> Using the duplicated code, we can save gas by removing a jump instruction.
>
> — Rhinestone

[Go back to Findings Summary]

I4: Modifier placement

Impact:	Info	Likelihood:	N/A
Target:	SchemaManager.sol	Type:	Best practices

Description

In the `SchemaManager` contract, the modifier `onlySchemaValidator` is placed between functions.

Recommendation

Move the `onlySchemaValidator` modifier above all functions according to best practices and improve readability.

Fix 1.1

Fixed. The modifier was moved.

Go back to Findings Summary

I5: Missing NatSpec documentation

Impact:	Info	Likelihood:	N/A
Target:	AttestationLib.sol	Type:	Documentation

Description

The `AttestationLib` library is missing NatSpec documentation.

Recommendation

Cover the `AttestationLib` library by NatSpec documentation.

Fix 1.1

Fixed. The missing documentation in the `AttestationLib` was added.

Go back to Findings Summary

I6: `_storeAttestation` false comment

Impact:	Info	Likelihood:	N/A
Target:	AttestationManager.sol	Type:	Documentation

Description

The function `_storeAttestation` is internally used to store attestations of modules. It is documented with the following comment:

Listing 17. Excerpt from [AttestationManager]()

```
     * @dev This function will revert if the same module is attested twice by the same attester.
      *      If you want to re-attest, you have to revoke your attestation first, and then attest again.
```

The comment is not true, because it is possible to overwrite (even valid) attestations.

Recommendation

Correct the comment or adjust the behavior of the `_storeAttestation` function to reflect the comment.

Fix 1.1

The finding was fixed by keeping the comment but updating the logic to prevent overwriting existing attestations without a revocation.

[Go back to Findings Summary]()

I7: `NewTrustedAttesters` event

Impact:	Info	Likelihood:	N/A
Target:	TrustManager.sol	Type:	User experience

Description

The event `NewTrustedAttesters` is emitted when new trusted attesters are set for a given account.

Listing 18. Excerpt from [TrustManager]()

```
        emit NewTrustedAttesters();
```

However, the information is only relevant to the account that updated the attesters set.

Recommendation

Add the account (`msg.sender`) as a parameter (optionally `indexed`) to the event. Also, consider including `threshold` and hash from `attesters`.

Fix 1.1

Fixed. The `msg.sender` parameter was added to the `NewTrustedAttesters` event.

[Go back to Findings Summary]()

ackee | blockchain security

6. Report revision 1.1

6.1. System Overview

Updates and changes we find important for fix-review.

Contracts

ModuleManager.sol

The parameter `bytes resolverContext` was added to functions `deployModule`, `registerModule`, and `deployViaFactory`. The parameter is passed to the `ModuleRecord.requireExternalResolverOnModuleRegistration` function (from `StubLib` library)

StubLib.sol

The parameter `bytes resolverContext` was added to the function `requireExternalResolverOnModuleRegistration`. The parameter is forwarded to the `IExternalResolver.resolveModuleRegistration` function.

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, Rhinestone: Module Registry, 3.7.2024.

Appendix B: Glossary of terms

The following terms might be used throughout the document:

Superclass/Ancestor of C

A contract that C inherits/derives from.

Subclass/Child of C

A contract that inherits/derives from C.

Syntactic contract

A Solidity contract. May have an inheritance chain, and may be deployed.

Deployed contract

An EVM account with non-zero code. If its source was written in Solidity, it was created through at least one syntactic contract. If that contract had superclasses (parents), it would be composed of multiple syntactic contracts.

Init/initialization function

A non-constructor function that serves as an initializer. Often used in upgradeable contracts.

External entrypoint

A `public` or `external` function.

Public/Publicly-accessible function/entrypoint

An `external` or `public` function that can be successfully executed by any network account.

Mutating function

A non-`view` and non-`pure` function.

Thank You

Ackee Blockchain a.s.

Prague, Czech Republic

hello@ackeeblockchain.com

https://twitter.com/AckeeBlockchain