API3

API3 Airnode Protocol

Security Assessment Report

Version: 2.0

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the API3 airnode protocol smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contracts, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the API3 airnode protocol smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the API3 airnode protocol smart contracts.

Overview

API3's airnode protocol enables first–party API providers to run oracle nodes with little to no maintenance. By allowing providers to supply data on–chain, without an intermediary, the API3 airnode protocol can eliminate middlemen costs and further reduce the protocol's attack surface. First–party oracle solutions also help to align the interests of the protocol's users and API providers.

The API3 airnode protocol can be logically separated into the following components:

- `Access Control`:
  - Contracts related to the hierarchy of user roles.
  - Roles relate to each other in a tree-like structure.
- `Whitelist`:
  - Contracts implementing generic whitelisting functionality.
  - Split into separate whitelist contracts intended to manage different roles.
- `Request-Response Protocol (RRP)`:
  - Contracts related to the Request-Response Protocol. Implements withdrawals by sponsor wallets, request template creation, authorization checks and the general sending and receiving of requests.

  - `RrpRequester.sol` is tasked with making requests to `AirnodeRrp.sol`.
  - `RrpBeaconServer.sol` serves live data points for whitelisted readers.
- `Authorizers`:
  - These contracts implement arbitrary business logic for the airnode protocol.
  - Contracts inherit whitelist contracts to enforce which accounts are permitted to interact with the authorizer contracts.

Security Assessment Summary

This review was conducted on the files hosted on the airnode repository and were assessed at commit c6263e9.

Retesting activities were conducted against commit 1929ce5.

Note: the OpenZeppelin libraries and dependencies were excluded from the scope of this assessment.

The design of the API3 airnode protocol incorporates a privileged manager account which has the ability to whitelist and revoke certain roles. As this is by design, this review does not explicitly list attacks where the manager account is malicious, rather we assume the protocol's manager keys are secure and honest. Should the manager's private key be stolen by a malicious actor, the protocol can be severely compromised.

The manual code review section of the report, focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts. Specifically, their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focused on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [1, 2].

To support this review, the testing team used the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 8 issues during this assessment. Categorized by their severity:

- Low: 2 issues.
- Informational: 6 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the scope of this review. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations and comments not directly related to the security posture of the relevant smart contracts, are also described in this section and are labelled as "informational".

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
API3-01	Proxy Manager Can Renounce `onlyOwner` Role	Low	Closed
API3-02	Lack of Transfer Ownership Pattern in OpenZeppelin's `Ownable` Library	Low	Closed
API3-03	Whitelisted Beacon Readers Can Create Secondary Markets For APIs	Informational	Resolved
API3-04	EVM Bytecode is Not Optimised	Informational	Resolved
API3-05	`expirationTimestamp` is Not Greater Than `block.timestamp`	Informational	Closed
API3-06	`setUpdatePermissionStatus()` and `setSponsorshipStatus()` Erroneously Emit Events	Informational	Closed
API3-07	Miscellaneous Gas Optimisations	Informational	Resolved
API3-08	Miscellaneous General Statements	Informational	Resolved

API3-01	Proxy Manager Can Renounce `onlyOwner` Role		
Asset	`AccessControlManagerProxy.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `onlyOwner` role is granted to the deployer account of `AccessControlManagerProxy` by default. `AccessControlRegistry` implements a `renounceRole()` function which allows accounts to renounce granted roles belonging to them. The root role is strictly not renounceable by the `manager` account, however, `AccessControlManagerProxy's` use of OpenZeppelin's `Ownable` library permits the `onlyOwner` role to effectively renounce the root role by calling `AccessControlManagerProxy.renounceOwnership()`.

Recommendations

Consider inheriting a modified `Ownable` contract which does not contain the `renounceOwnership()` function.

Resolution

The ability for the owner account to renounce their `onlyOwner` role is part of the `AccessControlManagerProxy` contract's capabilities. There are potential use-cases where this capability might be needed. As a result, the development team has marked this issue a *wontfix*.

API3-02	Lack of Transfer Ownership Pattern in OpenZeppelin's `Ownable` Library		
Asset	`AccessControlManagerProxy.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `onlyOwner` role is granted to the deployer account of `AccessControlManagerProxy` by default. However, it is unlikely that the DAO has this role initially. Hence, the deployer account must call `transferOwnership()` to transfer the ownership of the contract to the DAO. As OpenZeppelin's `Ownable` library does not implement a two-step transfer pattern, it is possible for such a transfer to lead to an unexpected loss in contract ownership. Any subsequent transfer poses the same risk to the contract.

Recommendations

Consider inheriting a modified `Ownable` contract which implements a two-step transfer ownership process. This should first enforce the nomination of a new account by the existing account with the `onlyOwner` role. The nominated account must then accept the nomination for the transfer to be complete. This ensures the role is not mistakenly transferred to the wrong account, thereby ensuring the `AccessControlManagerProxy` contract maintains proper ownership.

If the DAO is not equipped to handle arbitrary calls to contracts through a typical proposal mechanism, the testing team also recommends implementing the accept and nominate functions on the DAO side.

Resolution

The development team finds the current ownership transfer pattern to be more in line with how the rest of the protocol operates. Most accounts are transferred in a one-way fashion, hence implementing protection against one specific case does not justify the added code complexity. As a result, this issue is considered a *wontfix*.

API3-03	Whitelisted Beacon Readers Can Create Secondary Markets For APIs
Asset	`RrpBeaconServer.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `RrpBeaconServer` contract intends to limit the use of fulfilled requests to a subset of whitelisted readers. These whitelisted contracts are allowed to read the state of any given `templateId` under a subscription based model which is managed offchain.

However, it is possible that certain whitelisted contracts act as proxies for a secondary market whereby subscription costs are distributed between several users. As a result, the incentive for API providers to fulfill onchain requests may diminish over time.

Recommendations

Ensure this is intended behaviour or alternatively validate the functionality of whitelisted contracts offchain.

Resolution

The development team has acknowledged that the highlighted issue will be handled offchain and is part of the service provided by API3.

API3-04	EVM Bytecode is Not Optimised
Asset	`Smart Contract Suite`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Solidity's optimiser simplifies complicated expressions according to a set of simplification rules, reducing both bytecode size and execution cost. Users are able to benefit from the reduced gas costs and improved user experience as a result. There are two modules, namely, the *old*, battle-tested optimizer which operates at an opcode level and the *new* optimizer that operates on Yul IR code. Therefore, one of these optimisers should be enabled in the protocol's `hardhat.config.js`.

Recommendations

Consider updating the project's `hardhat.config.js` file such that the following optimised configuration holds true:

```
solidity: {
        version: '0.8.9',
        settings: {
          optimizer: {
            enabled: true,
            runs: 200,
          },
        },
      },
```

This should help generate significant gas savings throughout the entire `Airnode protocol` smart contract suite. Refer to Solidity's documentation for more information about how the optimiser works.

Resolution

The recommendation will be implemented by the development team in future deployments.

API3-05	`expirationTimestamp` is Not Greater Than `block.timestamp`
Asset	`Whitelist.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The whitelist expiration setter role is able to temporarily whitelist a given `user`. However, there are no explicit checks to ensure `expirationTimestamp` is set to sometime in the future. As a result, it is possible for a setter role to incorrectly configure `expirationTimestamp` and have the expectation that the target `user` has been whitelisted.

Recommendations

Consider checking that `expirationTimestamp` is set in the future. The following snippet of code details what this could look like.

```
require(expirationTimestamp > block.timestamp, "Is not set in the future");
```

This should be checked before making any state changes to a user's temporary whitelist. This check only needs to be enforced in `_setWhitelistExpiration()` and `_extendWhitelistExpiration()` as all deriving contracts eventually execute one of these two functions.

Resolution

The term *set* is used to refer to setting the value, strictly ignoring its previous value. While *extend* refers to updating a timestamp value with a larger one. The proposed recommendation adds unnecessary code complexity at the cost of dependent use-cases. Therefore, the development team has marked this issue a *wontfix*.

API3-06	`setUpdatePermissionStatus()` and `setSponsorshipStatus()` Erroneously Emit Events
Asset	`RrpBeaconServer.sol` and `AirnodeRrp.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `setUpdatePermissionStatus()` and `setSponsorshipStatus()` functions are both called by `sponsor` accounts to enable the respective `requester` accounts to make beacon requests. However, if a `requester` account has already had their status updated, calling the function again with the same `status` will emit a duplicate event.

Recommendations

Consider only emitting the respective events in `setUpdatePermissionStatus()` and `setSponsorshipStatus()` if the `requester's` status has changed. There are potentially other instances of this issue which should also be updated to ensure events are only emitted on valid state changes.

Resolution

The term *set* is used to refer to setting the value, strictly ignoring its previous value. As a result, the proposed recommendation does not accurately reflect the intended behaviour of the mentioned functions. The development team has made this issue a *wontfix*.

API3-07	Miscellaneous Gas Optimisations
Asset	`Smart Contract Suite`
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section describes general observations made by the testing team in which contract callers are able to generate gas savings:

1. **Gas Savings By Caching Values:**
   `TemplateUtils.sol` uses `templateIds.length` multiple times. This can be cached in memory once at the start of `getTemplates()`. Each subsequent use can then refer to the cached value instead.
2. **Function Computation Can Be Done Offchain:**
   - `TemplateUtils.getTemplates()` iterates over a list of `templateIds` to pull the template state from the `AirnodeRrp` contract.
   - Similarly, an `Airnode` will potentially call `checkAuthorizationStatus()` multiple times using the `checkAuthorizationStatuses()` function.

   These calls can be simulated offchain to generate additional gas savings.
3. **Storage Accesses Can Be Replaced By Memory Operations:**
   `TemplateUtils.getTemplates()` iterates over a list of templates. The template is loaded into storage before being subsequently written into the returned arrays. Consider loading struct data into `memory` instead of `storage` for some gas savings.
4. **Variables Can Be Made Immutable For Gas Savings:**
   Both the `string adminRoleDescription` and `bytes32 adminRoleDescriptionHash` variables can utilise the `immutable` keyword to generate small gas savings.
5. **Functions Can Be Made External For Gas Savings:**
   The `WhitelistRolesWithAirnode.deriveAdminRole()` function is used to derive the admin role from a specified `airnode` address. This function is never called internally, hence, it can benefit from small gas savings by updating its visibility from `public` to `external`. This should generate gas savings on contract deployment and on each function call.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The first three issues have been acknowledged by the development team. `getTemplates()` and `checkAuthorizationStatus()` functions are implemented such that an API3 node can call them statically. Hence, gas costs for these functions are not a concern.

The last two issues plan on being integrated into future deployments of API3's smart contract suite.

API3-08	Miscellaneous General Statements
Asset	`Smart Contract Suite`
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section describes general observations made by the testing team during this assessment that do not have direct security implications:

1. **Certain Contracts Can Be Made Abstract:**

   The `RoleDeriver` and `AccessControlClient` contracts are only ever inherited by another contract and should therefore be made `abstract` to reflect this.

2. **Unclear Naming Of `AccessControlRegistry.initializeRole()` Function Arguments:**

   The `initializeRole()` function is intended to be used in conjunction with `initializeAndGrantRoles()` such that the root, admin and any other roles are all initialised in a single transaction. However, the naming of the `adminRole` argument could generate confusion over the intended use of the `initializeRole()` function. Consider renaming this argument or document the intended use of this function to avoid any confusion.

3. **Lack Of Length Checking On Input `bytes`:**

   The `RrpBeaconServer.fulfill()` function is called by the `AirnodeRrp` contract when fulfilling a made request. `fulfill()` takes a `data` argument which is not length checked to ensure it fits within the `int256 decodedData` variable. Consider checking that the `data` argument is *32 bytes* in length, otherwise the `fulfill()` function may read from the EVM stack.

4. **Lack Of Input Validation:**

   There are a number of functions which don't validate the input. For example, `WithdrawalUtils.requestWithdrawal()` does not check if `airnode` is the zero address. Consider performing necessary input validation on the affected functions.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The development team has provided the following resolutions to the above issues:

1. To avoid silent compiler errors when inheriting `abstract` contracts that have unimplemented parts, the development team has decided not to implement the recommendation.

2. Renaming of the `initializeRole()` and `initializeAndGrantRoles()` function arguments in future iterations.
3. Length checking for the `RrpBeaconServer.fulfill()` has been implemented in commit 3c7ce03.
4. Input validation is intended to be added into API3's next iteraton.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are provided alongside this document. The `brownie` framework was used to perform these tests and the output is given below.

```
test_initialize_manager                       PASSED   [6%]
test_initialize_role                          PASSED   [12%]
test_renounce_and_revoke_roles                PASSED   [18%]
test_set_sponsorship_status                   PASSED   [25%]
test_set_sponsorship_status_erroneous_event   PASSED   [31%]
test_make_template_request                    PASSED   [37%]
test_make_full_request                        PASSED   [43%]
test_fulfill_request_successful               PASSED   [50%]
test_fulfill_request_failure                  PASSED   [56%]
test_cancel_request                           PASSED   [62%]
test_withdrawal_request                       PASSED   [68%]
test_template                                 PASSED   [75%]
test_fulfill_request                          PASSED   [81%]
test_deploy_access_control                    PASSED   [87%]
test_deploy_authorizer                        PASSED   [93%]
test_deploy_rrp                               PASSED   [100%]
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
			Likelihood	

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018,
Available: `https://blog.sigmaprime.io/solidity-security.html`. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: `http://www.dasp.co/`. [Accessed 2018].