API3

API3 Airnode

Smart Contract Security Review

Version: 1.0

March, 2023

Contents

Introduction

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

In particular as some components are intended as building blocks for other developers working with API3, the testing team have included security recommendations that do not currently cause issues but are highlighted as potential pitfalls to avoid.

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 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 smart contracts.

Overview

API3 Airnodes are first-party serverless oracles designed to provide price feeds for other Dapps that are easy to set up and maintain. Each API is served by an oracle that is operated by the owner of the API in an effort to reduce middleman fees and reduce attack vectors on the data feeds. These API feeds can then be aggregated to produce groups of Airnodes known as Beacon sets which allow for the median price of several oracles to be read.

Reducing operating overheads is managed by allowing Airnode Beacon operators to produce off-chain signatures for their price feed, these can then be trustlessly pushed on-chain updating the oracle feed. The oracles are split into two styles: the *Request-reponse-protocol* and the *Publish-subscribe-protocol*. These two different approaches allow for different uses of the oracles in push/pull set-ups.

Security Assessment Summary

This review was conducted on the files hosted on the API3 repository and were assessed at commit 2686828.

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

The manual code review section of the report is focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts. This includes 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 12 issues during this assessment. Categorised by their severity:

- Critical: 1 issue.
- Medium: 2 issues.
- Informational: 9 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the API3 smart contracts. 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, 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	Signature Reuse Allows Airnodes to Skew Reported Price	Critical	Resolved
API3-02	Aggregated Beacons Can Report Inaccurate Price & Timestamp Pairs	Medium	Resolved
API3-03	Beacon Sets Reuse Stale Data	Medium	Resolved
API3-04	Lack Of Access Control Can Lead To Event Spam	Informational	Closed
API3-05	Lack Of Zero Address Checks Could Lead To Loss Of Funds	Informational	Resolved
API3-06	Comment Suggestion To Third Party Developers Can Be Dangerous	Informational	Resolved
API3-07	Reverted Calls Are Indistinguishable From Non-Contract Account Calls	Informational	Resolved
API3-08	Internal Function Calls Are Not Possible With `SelfMulticall.sol`	Informational	Resolved
API3-09	`ExternalMulticall.sol` Design Is Permissive	Informational	Resolved
API3-10	Event `SetIndefiniteAuthorizationStatus` Emitted On All Calls	Informational	Closed
API3-11	No Sponsor Validation in AirnodeProtocol Request Fulfillment	Informational	Closed
API3-12	Miscellaneous General Comments	Informational	Resolved

API3-01	Signature Reuse Allows Airnodes to Skew Reported Price		
Asset	`DapiServer.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

`Beacon sets` aggregate price updates from several `Airnodes` to provide a price. Due to incorrect validation, it is possible for previously used `Airnodes` signatures to be reused, allowing a single malicious `Airnodes` to greatly influence the reported price of a `Beacon set`.

When a price update is supplied to a `Beacon set`, all `Airnodes` providing a new price update must sign the provided data to ensure the correct data is published, as they are not always the on-chain caller of `updateDataFeedWithSignedData()`, `updateDataFeedWithDomainSignedData()` or `updateOevProxyDataFeedWithSignedData()`. This signature is intended to be used once, as evidenced by its use of a timestamp.

However, because the aggregated update timestamp is calculated as a mean of the provided timestamps, this means that only one of the updates actually needs to be new. Therefore, a malicious `Beacon` can provide an update which includes stale signatures where the timestamps are older than the current update timestamp.

The aggregated timestamp passes the following check `require(updatedTimestamp > dataFeeds[beaconSetId].timestamp)`, because the malicious `Beacon` can specify a timestamp greater than the current update timestamp.

The result is that a single `Beacon` can control the `Beacon Set's` published price by taking the signatures of recent but stale updates and reusing them with a new malicious update.

While executing this exploit requires control of a `Beacon` the testing team have rated this as having a high likelihood of occurring as the financial incentive from misrepresenting a price feed would attract bad actors and increase the likelihood of legitimate beacon operators being targeted for key theft.

Recommendations

There are multiple methods that could be applied to solve this issue:

- Include a nonce with `Beacon` price updates, this would then be added to the signature and checked to prevent signature reuse. However, it is possible that this would break multi-chain functionality as different chains may have different nonces on the same feed.
- Record and check `Beacon` timestamps individually rather than as a mean.

Resolution

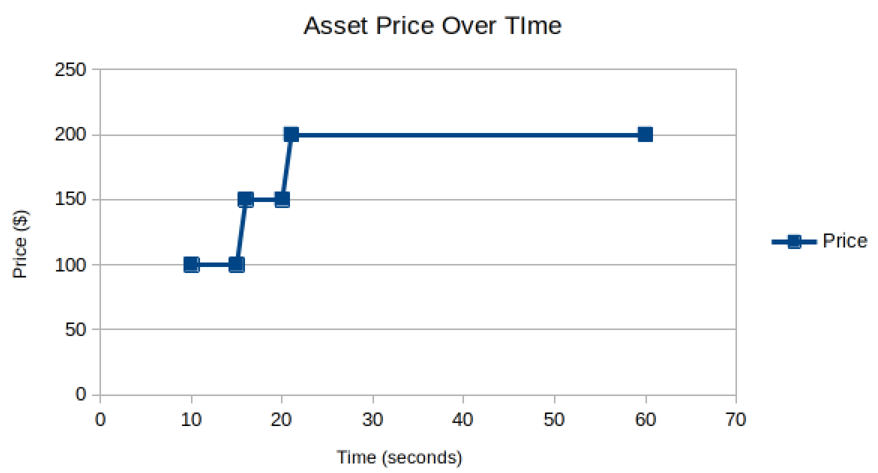
`Beacon sets` now must have `Beacons` updated individually and so must check their own timestamps for each update. The domain signed data update methods have been removed and OEV update methods have been reworked to function off-chain. See pull request 96 and 107.

API3-02	Aggregated Beacons Can Report Inaccurate Price & Timestamp Pairs		
Asset	`DapiServer.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

When aggregating `Beacons` two different mathematical methods are used to return the `value` and `timestamp`. This means they can be out-of-sync and a price can be returned that was not actually achieved at the returned timestamp.

The graph below illustrates an instance of this behaviour.



In this scenario three beacons provided updates at times 15s, 18s and 60s with prices $100, $150 and $200 respectively. If these reported values are aggregated via a call to `aggregateBeacons()`, the aggregate data returned will be a price of $150 at time 31s, however looking at the graph this is wrong as the price at time 31s is $200. Therefore even though each beacon submitted valid data the aggregated data has ended up being inaccurate.

This issue is due to the different calculation methods used for the aggregate price and timestamp. Since the price is calculated via a median while the timestamp is a mean, this issue can occur with timestamp differences as small as 2s between beacon readings if volatile prices are present. Depending on how the oracle is then used this could result in incorrect decisions being made for third party developers who are interested in the price & timestamp pair such as `Time Weighted Average Price` calculations or `Binary Options`.

However, as the issue can only occur as a side effect of genuine updates the testing team have only rated it as having a medium likelihood as any instance is likely to be accidental. The impact of this issue is hard to judge without existing third party uses of the oracle, so it is provisionally rated as having a medium impact, if services like those outlined earlier made use of API3 then this could be a high impact issue.

Recommendations

To resolve this issue the same method should be employed for calculating the aggregate price as the aggregate timestamp:

- Use the median timestamp to report back on update calls for `Beacon sets`.
- Use the mean price to report back on update calls for `Beacon sets`, however this then exposes the price calculation to the extremes of the reported data set. If this method was employed a weighting system could be used to dampen the influence of beacons far from the median value on the reported price and timestamp.

This discrepancy can occur via different paths such as `fulfillPspBeaconSetUpdate()` via `aggregateBeacons()` mentioned beforehand but also with `updateDataFeedWithSignedData()`, `updateDataFeedWithDomainSignedData()` and `updateOevProxyDataFeedWithSignedData()`. All methods of calculating the `Beacon set` data should be fixed.

If users are not intended to trust the timestamp provided by calls for `Beacon sets` then this should be explicitly stated, or the functions should be altered to only provide the price to the end user.

Resolution

The calculation of the `Beacon set` timestamps is now performed via median rather than mean for all aggregation methods except OEV, see pull request 97. For OEV aggregation updates the recorded timestamp is calculated off-chain first then verified on-chain against the `Airnode Beacon` signatures.

API3-03	Beacon Sets Reuse Stale Data		
Asset	`DapiServer.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

If a user attempts to aggregate individual `Beacons` into a set then it is possible to end up with stale data that would have been rejected if it was added at the time of the aggregation.

This is due to the lack of timestamp checks on existing `Beacon` data. New price data is validated via a call to `AirnodeRequester.timestampIsValid()` for each supplied `Beacon's` timestamp, this does not happen for existing data.

This means that stale data can be used to create a `Beacon Set` output. While API3 have stated *"any Beacon set update is welcome"* the testing team feels this is not a good design choice and if intentional should be communicated to the users of the `Beacon Sets`, so they can check timestamps accordingly as using stale data may lead to a `Beacon set` reporting an inaccurate price.

Recommendations

Some possible mitigation strategies are:

- Reject `Beacons` from aggregation if their last update is too old, reverting if not enough `Beacons` of a set are up-to-date.
- Outlining the limitations and intended uses of each different `Beacon` aggregation method in NatSpec, ensuring third party developers are aware of which ones to use.

Resolution

The Airnode team have taken an alternative resolution approach to those listed, by not considering old signatures invalid. This solution is now possible because only `Beacon` updates with more recent timestamps can be accepted. This means previous timestamp restrictions are no longer needed and so can be changed to ensure consistency. See commit 992c587.

API3-04	Lack Of Access Control Can Lead To Event Spam
Asset	`AirnodeProtocol.sol, StorageUtils.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

Due to lack of access control in certain functions in `AirnodeProtocol` and `StorageUtils`, it is possible for a third party to emit events which can be used to spam the event logs.

There are two instances where this can happen:

- In `AirnodeProtocol.sol`, functions `makeRequest()` and `makeRequestRelayed()` do not check `msg.sender` is a valid requester. This was a change from `v0` of Airnode to allow the sponsorship status to be overridden off-chain. Since the sponsorship status is not validated on-chain, anyone can call these functions to emit the `MadeRequest` and `MadeRequestRelayed` events.
- Anyone can call `announceTemplate()` and `announceSubscription()` in `StorageUtils.sol` to emit the `AnnouncedTemplate` and `AnnouncedSubscription` events.

Recommendations

Ensure Airnode is resilient to this potential on-chain DOS attack vector.

Resolution

Events `AnnounceTemplate` and `AnnounceSubscription` were removed, see commit c1ad9bc. Functions `makeRequest()` and `makeRequestRelayed()` were not altered to maintain flexibility of the protocol.

API3-05	Lack Of Zero Address Checks Could Lead To Loss Of Funds
Asset	`DataFeedProxyWithOev.sol, DapiProxyWithOev.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The contracts `DataFeedProxyWithOev` and `DapiProxyWithOev` lack a check for the zero address in the constructor, which can allow the deployer to set `address(0)` as the `oevBeneficiary`. This results in the `withdraw()` function in `DapiServer` reverting which prevents funds relating this OEV proxy from being withdrawn.

The intended method of deploying these OEV proxies is via `ProxyFactory` and if this method is followed then this vulnerability will not occur, but it is possible that third parties will not adhere to this pattern.

Recommendations

It is recommended to move the zero address checks to the constructors of `DataFeedProxyWithOev` and `DapiProxyWithOev` to prevent errors if a user deploys the contracts directly.

Alternatively the developers could include a NatSpec comment to warn third party developers about this risk should they deploy `DataFeedProxyWithOev.sol` or `DapiProxyWithOev.sol` directly.

Resolution

Comments were added to warn third party developers, see commit 960ebb4.

API3-06	Comment Suggestion To Third Party Developers Can Be Dangerous
Asset	`AirnodeRequester.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

On line [**38**] there is a comment to third party developers that they can *"Feel free to use a different condition or even omit it if you are aware of the implications."* regarding the validation checks performed on a `Beacon` timestamp.

The auditing team feel this comment is potentially dangerous, and as this check is currently the only guard against signature reuse (as noted in API3-01) these implications should be made clearer to other developers.

Recommendations

Make the risks of removing or altering timestamp validation clearer in the comments.

Resolution

Comments have been updated to highlight the risks of incorrect timestamp validation. Also, the contract has been made abstract and `timestampIsValid()` is now virtual, forcing third party developers to specify their own design if they use it. See commit 992c587.

Timestamp comments were also updated elsewhere due to changes relating to API3-02, see commit b1ab44e.

API3-07	Reverted Calls Are Indistinguishable From Non-Contract Account Calls
Asset	`ExternalMulticall.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

When a call fails in `tryExternalMulticall()`, the error message given by the failed call is recorded and returned to the user. If no error message is returned the call is logged as `false` for `Success[ind]` and `Returndata[ind]` will be the default value of `0x0`. This behaviour is identical to what will be returned if `target[ind]` address resolves to an externally owned account due to line [**73**].

This is because no entry is made for the related `Success[ind]` and `Returndata[ind]` fields meaning they will remain in the default states of `false` and `0x0`.

This means that it is not possible for the caller to distinguish calls that revert and return no error data and calls to externally owned accounts.

Recommendations

Include a special return data code used for calls sent to an externally owned account.

Resolution

Revert strings have been simulated for non-contract account calls, so they can be distinguished from calls which revert with no error message. See commit f5fe00a.

API3-08	Internal Function Calls Are Not Possible With `SelfMulticall.sol`
Asset	`SelfMulticall.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Due to calls made by `multicall()` or `tryMulticall()` being processed by an external `delegatecall` back into the contract that inherits `SelfMulticall.sol`, it is not possible for `multicall()` or `tryMulticall()` to call `internal` functions. This logic may be confusing for third party developers as usually contracts are capable of calling their own functions marked as `internal`.

Recommendations

Include code comments making this restriction clear to the reader, perhaps document a standard method of replicating `internal` functions that can be called by such as `public` functions using an access modifier.

Resolution

Additional guidance has been added to `SelfMulticall.sol` comments, see commit d0d90f8.

API3-09	`ExternalMulticall.sol` Design Is Permissive
Asset	`ExternalMulticall.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

`ExternalMultiCall.sol` is provided as a utility intended to be integrated by other developers building on top of the Airnode protocol. By design it is intended to batch multiple calls to other contracts together and call them from the contract that inherits it. This design lacks any access controls and would enable any caller to remove any tokens stored in the underlying contract.

Third party developers should be aware of this when inheriting the contract and plan accordingly.

Recommendations

The NatSpec comments of `ExternalMulticall.sol` should highlight the risks of allowing arbitrary external calls and third party developers should either restrict its functionality in their projects or take precautions such as ensuring the contracts that inherit it do not handle any tokens such as `ERC20s` or `ERC721s`.

Resolution

Additional guidance has been added to `ExternalMulticall.sol` comments. The related functions have been made `virtual` so that they can be overridden with access control or other less permissive designs if inherited. See commit ca0757b.

API3-10	Event `SetIndefiniteAuthorizationStatus` Emitted On All Calls
Asset	`RequestAuthorizer.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

In the function `_setIndefiniteAuthorizationStatus()`, the event `SetIndefiniteAuthorizationStatus` is emitted once the function logic is run, regardless of the outcome. While the function's NatSpec mentions that this occurs even if the `airnodeToRequesterToSetterToIndefiniteAuthorizationStatus` boolean has not changed it does not mention that it is also emitted when called by users with no authorization.

In either circumstance this will lead to a misleading event being emitted as no state change has occurred.

Recommendations

Make the event emission dependent on the `airnodeToRequesterToSetterToIndefiniteAuthorizationStatus` boolean changing state so that off-chain tracking can interpret the event more reliably.

Resolution

The issue has been acknowledged by the Airnode team. It is considered a design style and so will not be addressed.

API3-11	No Sponsor Validation in AirnodeProtocol Request Fulfillment
Asset	`AirnodeProtocol.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

`AirnodeProtocol`'s `makeRequest()` function allows requesters to specify a `sponsor` address. However, there is no validation that `msg.sender` is equal `sponsor` in `fulfillRequest()`, relying solely on the `AirnodeProtocol` to provide the correct signature with no on-chain safeguards.

This can lead to the following:

1. The `requester` requests an update from a specific `sponsor`.
2. The `airnode` signs a message for a different `sponsor`.
3. The other `sponsor` is granted permission to fulfill the request.

If the `airnode`'s private key was to be compromised, an attacker could sign a message allowing an arbitrary address to call `fulfillRequest()` and submit false data on-chain.

This behavior is a change from `v0` of the protocol where `fulfillRequest()` is only callable by the specified `sponsor` address.

Recommendations

Review if this change in behavior is intended in line with allowing `airnode` flexibility in overriding `requester/sponsor` relationships off-chain.

The testing team considers it to be prudent to include the `sponsor` address as part of the fulfillment parameters' Keccak256 hash, as implemented in `v0` of the protocol. This would require both the `airnode` and the `sponsor's` private keys to be compromised in order for an attacker to fulfill a request with false data.

Resolution

No action was taken for this issue. The Airnode team outlined that an `airnode` wallet is intended to be derived from the same private key as the `sponsor` and so little is gained by enforcing a specific `sponsor` to fulfil the request. The Airnode team have opted not to fix this issue to maintain a range of sponsorship uses and help improve developer UX.

API3-12	Miscellaneous General Comments
Asset	`contracts/*`
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Misleading comment**

   In `SelfMulticall.sol` on line [12] a comment states `"reverts if at least one of the batched calls reverts"`. However, the call will revert if a batched call reverts, therefore only one call can revert prior to the parent call reverting. Suggest removing "at least" as this suggests more than one call can revert in each parent call. Likewise for the comment on line [12] of `ExternalMulticall.sol`.

2. **Differing Solidity version requirements**

   As a set of contracts integrating with other projects it is expected that the Solidity version required might float. However, some files specify `^0.8.0` and others have `=0.8.17`. Ideally the specified version should be locked at the version where testing occurred, `0.8.17`, or justification as to why some files should only be used with `0.8.17` should be included.

3. **Funds can get stuck in `WithdrawalUtils.claimBalance()`**

   If the sponsor is a contract without a default 'payable' function, `claimBalance()` will revert which can cause funds to be stuck. While this is not an issue with Airnode protocol, this behavior should be documented clearly by way of official documentation and NatSpec comments to avoid potential issues with integrations.

4. **Out of date documentation**

   Recommend updating to reflect the latest changes.

5. **Gas optimisations**

   - Code Reuse in `updateDataFeedWithSignedData()`, `updateDataFeedWithDomainSignedData()`, and `updateOevProxyDataFeedWithSignedData()` it is recommended to refactor these to reduce code duplication and deployment costs.
   - Calling length in the loop on line [89] of `QuickSelect.sol`: To save gas this length should be called once and stored in a local variable rather than called every iteration of the loop.
   - Usage of uints/ints smaller than 32 bytes (256 bits) incurs gas overhead. Instances:
     - `Allocator.sol::17 => uint32 expirationTimestamp;`
     - `RequesterAuthorizer.sol::14 => uint32 expirationTimestamp;`
     - `DapiServer.sol::50 => uint32 timestamp;`
   - Use custom errors in `require()` statements to save gas.
   - Use bytes32 instead of string. String is a dynamic data structure and therefore uses more gas than equivalent bytes32 fields. Instances:
     - `Allocator.sol::21 => string public constant override SLOT_SETTER_ROLE_DESCRIPTION`
     - `RequesterAuthorizer.sol::31 => string public constant override INDEFINITE_AUTHORIZER_ROLE_DESCRIPTION`
     - `string public constant override DAPI_NAME_SETTER_ROLE_DESCRIPTION`

- Splitting `require()` statements that use && saves gas. Instances:
  - `AirnodeRequester.sol::51 =>`
    `timestamp + 1 hours > block.timestamp && timestamp < block.timestamp + 15 minutes`
  - `WithdrawalUtils.sol::86 =>`
    `timestamp + 1 hours > block.timestamp && timestamp < block.timestamp + 15 minutes`
- Not using named return variables when a function returns wastes deployment gas as in `QuickSelect.quickselectKPlusOne()`.

Recommendations

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

Resolution

The development team have acknowledged these findings, addressing them where appropriate as follows:

1. **Misleading comment**: Fixed in commit 03418c8.
2. **Differing Solidity version requirements**: A comment was added to explain different compiler choices 71f7ff3.
3. **Funds can get stuck in `WithdrawalUtils.claimBalance()`**: A comment was added to highlight this possibility, see commit 0a25f6c.
4. **Out of date documentation**: Fixed in commit b43ea44.
5. **Gas optimisations**: Some gas optimisations were made, see commits 8844a01, 5f7e95c and 5177c7c. Custom errors were avoided as some blockchains that Airnode may be deployed to might not support them.

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_externalMulticall               PASSED  [6%]
test_average                         PASSED  [13%]
test_average_edgecases               PASSED  [20%]
test_Sort_long                       PASSED  [26%]
test_Sort_short                      PASSED  [33%]
test_role                            PASSED  [40%]
test_withdrawalUtils                 PASSED  [46%]
test_fulfillRequest                  PASSED  [53%]
test_fulfillRequestDifferentSponsor  PASSED  [60%]
test_setup                           PASSED  [66%]
test_oevWithdrawAddressZero          XFAIL   (Wit...)[
test_BeaconSetsReportMedianData      PASSED  [80%]
test_RejectStaleBeaconUpdates        PASSED  [86%]
test_multicall                       PASSED  [93%]
test_tryMulticall                    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].

σ'