Least Authority
PRIVACY MATTERS

Synthetic Asset Platform Smart Contracts

Security Audit Report

Tezos Foundation

Final Audit Report: 21 September 2021

Table of Contents

Overview

- Background
- Project Dates
- Review Team

Coverage

- Target Code and Revision
- Supporting Documentation
- Areas of Concern

Findings

- General Comments
  - System Design
  - Code Quality
  - Documentation
  - Scope and Dependencies
- Specific Issues & Suggestions
  - Issue A: The Governance Mechanism is Underspecified
- Suggestions
  - Suggestion 1: Suggest Greater Transparency in the SmartPy Compiler [Out of Scope]
  - Suggestion 2: Improve Documentation
  - Suggestion 3: Improve Test Coverage
  - Suggestion 4: Conduct Audit of Oracle Implementation
  - Suggestion 5: Reject Inbound Tez Where Necessary
  - Suggestion 6: Document Method and Internal_Method Code Pattern
  - Suggestion 7: Pay Attention to Contract Calls Within SmartPy
  - Suggestion 8: Optimize Gas Cost for Fetching Price Information

About Least Authority

Our Methodology

Overview

Background

Tezos Foundation has requested that Least Authority perform a security audit of the Synthetic Asset Platform Smart Contracts.

Project Dates

- **July 14- August 13**: Initial Review *(Completed)*
- **August 18**: Initial Audit Report delivered *(Completed)*
- **September 16 - 17**: Verification Review *(Completed)*
- **September 21**: Final Audit Report delivered *(Completed)*

Review Team

- David Braun, Security Researcher and Engineer
- Phoebe Jenkins, Security Researcher and Engineer
- Wanas ElHassan, Security Researcher and Engineer
- Sajith Sasidharan, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Synthetic Asset Platform Smart Contracts followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

The following code repositories are considered in-scope for the review:
- Synthetic Asset Platform Smart Contracts: https://github.com/dcale/sap-smart-contract

Specifically, we examined the Git revisions for our initial review:

`525f5dd379c38db2e55bc9097a742f12b2d0de20`

For the verification, we examined the Git revision:

`489c488fe3eebfda8f4e114f712804c0b661fb73`

For the review, this repository was cloned for use during the audit and for reference in this report:

https://github.com/LeastAuthority/Tezos-Foundation-Synthetic-Asset-Platform

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third party code, unless specifically mentioned as in-scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:
- `README.md`: https://github.com/dcale/sap-smart-contract#readme
- Platform Documentation: Landing.youves.com (password protected)

- Youves Documentation: https://docs.youves.com/
- Oracle Documentation: https://ubinetic.medium.com/oracles-by-ubinetic-1f358779425
- Quick Start Guide: https://drive.google.com/file/d/1XsW0rK-JX5Lw4hPNE67F85QjIcvJ-jS-/view?usp=sharing

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Adversarial actions and other attacks on the smart contracts;
- Potential misuse and gaming of the smart contracts;
- Attacks that impact funds, such as the draining or the manipulation of funds;
- Mismanagement of funds via transactions;
- Denial of Service (DoS) and other security exploits that would impact the smart contracts intended use or disrupt the execution;
- Vulnerabilities in the smart contracts code;
- Protection against malicious attacks and other ways to exploit smart contracts;
- Inappropriate permissions and excess authority;

Findings

General Comments

Ubinetic's Synthetic Asset Platform (SAP) enables users to mint an FA2 stablecoin (uUSD) by collateralizing Tez. Users may deposit minted uUSD into a savings pool, earning a variable interest rate which is responsive to changes in liquidity levels of the pool, and is optimized to maintain a 1:1 peg with USD. The user may burn the minted uUSD thereby releasing the collateral.

A fixed amount of Youves (YOU) FA2 governance tokens is distributed periodically to uUSD holders, proportional to the weight of the user's funds in the pool. Users may stake earned YOU using the staking functionality of SAP and earn staking rewards generated from minting and staking fees.

The SAP functionality of segregating and controlling the collateral, saving, and staking pools is performed by the `tracker_engine.py` contract.

System Design

Our team performed a broad and comprehensive review of the SAP code base and documentation and found that the ubinetic team has considered security in the design and implementation of the protocol. The code base is well-structured and contains numerous correctness checks, and best practices for this use case have clearly been investigated and implemented by the ubinetic team.

We checked for vulnerabilities to flash loan or timing attacks that could yield more tokens than an attacker is correctly entitled to. We also reviewed calculations involving adding and withdrawing money from vaults, along with calculations based around current balance values, to ensure that transactions would not be able to drain funds. Special emphasis was placed on checking division and multiplication operations in the various flows in this contract to ensure that the rounding errors do not cause any loss of value for the contract.

The USD Peg

The SAP system relies on properly aligning economic incentives to maintain the constant value of uUSD. In addition to checking for standard logical errors in the program's code, we performed analysis of the user incentives to ensure the smart contracts are consistently operating within expected parameters (i.e, maintaining the peg). To analyze the economic incentives driving the uUSD peg to USD, we compared it to successful stablecoin cases such as DAI, and to recent stablecoins that failed to achieve intended functionality due to logical issues, as well as economic incentive issues such as unsustainably high user rewards from the platform. We did not identify any vulnerabilities to these known issues in the design of the SAP pegging mechanism.

The Oracle

Although the oracle system intended to be used by SAP is out of scope for this audit, our team noted that the `tracker_engine.py` contract is designed to receive market price information from two single points, the observed price oracle and the target price oracle, which are both trusted aggregator oracles. We suggest adding safe-guards in those two oracles to prevent bad participants from affecting the resulting price data (Suggestion 4).

System Governance

Our team examined the implementation of the governance mechanism, specifically the issuance and update of governance tokens through the `mint`, `burn`, `liquidate`, `settle_with_vault`, and `bailout` methods. Although we did not identify errors or security vulnerabilities in the implementation of the governance tokens, our team noted that the governance mechanism is underspecified in how the YOU governance token could be used for voting. In the current implementation, the governance mechanism could be characterized as centralized and non-transparent. We recommend that users be explicitly informed of these characteristics and that steps be taken to make the governance mechanism more transparent and decentralized (Issue A).

To assess potential security vulnerabilities introduced to the system by the governance mechanism, we compared the SAP governance mechanism to the DAI governance system. One place where SAP differs from DAI is that governance is not completely community oriented. Specifically, stakeholders of DAI can vote to change most attributes of DAI, and anyone can become a stakeholder. SAP, on the other hand, has a number of pre-selected keyholders that must come to a consensus in order to make major system changes.

While the nature of total governance by stakeholders is a design that appeals to some, both systems have advantages and disadvantages and neither is inherently insecure. However, with a keyholder system, there is a risk of collusion between the keyholders if the keys are not properly distributed. The SAP documentation states that only two keyholders are from SAP's development team, ubinetic, five are from institutions entirely unrelated to ubinetic, and a four-vote consensus is required for major changes. Provided that all keyholders are transparently selected with rationale acceptable to the community, this would represent a governance committee that is not centered on ubinetic currently.

Another difference in governance worthy of note is the issuance of the YOU token. MakerDAO's MKR token is both issued and burned in response to events in the contract, namely, debt auctions when Vaults have lost adequate collateral. SAP, on the other hand, issues governance tokens (YOU) to uUSD minters in proportion to their holding of uUSD, at a rate that decreases over time. This converges to a maximum supply of YOU. This incentivizes early adoption of the platform, and gives an additional incentive to minting uUSD from Vaults. SAP provides rewards for staking YOU, which it can then utilize in an as-of-yet unimplemented system for liquidizing or re-collateralizing Vaults. This functionality is intended to improve system stability under highly turbulent market conditions.

Code Quality

Our team conducted a manual review of the code and found the code base is very well organized and generally adheres to coding standard practices. Our team noted, however, that certain entrypoints that interact with external data sources follow a pattern of `method` and `internal_method` whereby `method` fetches data, and then invokes `internal_method` to perform additional tasks. Our team notes that invoking `internal_method` may lead to higher gas costs, and that the extra surface area created by `internal_method` instances could make the code base harder to maintain and to secure. We suggest that the rationale for this pattern be included in the documentation (Suggestion 6).

Tests

Our team found sufficient test coverage of SAP. Unit tests have been written for many of the classes, however we recommend increasing the number of tests to cover all components with security critical functionality including fa2.py and TrackerEngine.burn (Suggestion 3). Our team also noted that the tests written are excessively long (e.g., the GovernanceToken tests are 100 lines long).

Documentation

Our team found the project documentation to be comprehensive and generally sufficient. However, some of the documentation included in the in-scope repository is incomplete. Our team noted that the systems interaction with the oracles used to fetch market price information and to check the validity of signatures are both omitted from the documentation of minting and saving on SAP (Suggestion 2).

Code Comments

We found that the documentation contained within the code was sufficient, with every class and method having a docstring explaining its purpose. Although there are no comments within the bodies of methods, variable names are well-chosen, aiding in following the flow of logic.

Scope and Dependencies

Our team found that the scope of this security was sufficient and well defined. However, our team found that the choice of SmartPy as an implementation language for the protocol could be a cause for concern. We note that the SmartPy compiler has not been audited by an independent security auditing team and that the development and maintenance of SmartPy is not conducted according to accepted security practices. As a result we suggest that the SmartPy compiler and CLI be reviewed by an independent security auditing team (Suggestion 1). We acknowledge that concerns regarding SmartPy are out of scope for the review and out of the control of the ubinetic team. However, SmartPy's security is critical for the security of the SAP system. As such, we recommend the ubinetic team continue to monitor SmartPy developments and research. It is worth noting that SmartPy is effectively a deployment dependency, so potential changes will only be relevant to future deployments of components of the SAP ecosystem.

Additionally, our team noted that the platform functionality is critically dependent on the data retrieved from an oracle system, which was outside the scope of this audit. Price oracles have been known to introduce security vulnerabilities to similar systems and are a high point of concern. As a result, we recommend that the planned oracle system and its integration into the SAP be reviewed closely by an independent security auditing team (Suggestion 4).

In our review of the `deployment.py` script, we noted that pytezos, is a required dependency, and that pytezos utilizes secp256k1 whose last release was in September 2016. It bundles a copy of libsecp256k1, a library written in C. Furthermore, the tests are meant to work with an older version of OpenSSL 1.0.0, which is outdated. The binary wheels of secp256k1 published on the Python Packaging Index are for Python 3.5, but Python 3.5 has reached end-of-life. While we have not investigated if the under-maintained secp256k1 library would be an immediate problem, we encourage the SAP team to work with the library maintainers to keep it updated.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: The Governance Mechanism is Underspecified	Partially Resolved
Suggestion 1: Suggest Greater Transparency in the SmartPy Compiler [Out of Scope]	Unresolved
Suggestion 2: Improve Documentation	Resolved
Suggestion 3: Improve Test Coverage	Partially Resolved
Suggestion 4: Conduct Audit of Oracle Implementation	Resolved
Suggestion 5: Reject Inbound Tez Where Necessary	Unresolved
Suggestion 6: Document Method and Internal_Method Code Pattern	Unresolved
Suggestion 7: Pay Attention to Contract Calls Within SmartPy	Resolved
Suggestion 8: Optimize Gas Cost for Fetching Price Information	Unresolved

Issue A: The Governance Mechanism is Underspecified

Location

`https://docs.youves.com/governance/Governance-Mechanism`

Synopsis

Governance decisions regarding the tuning and maintenance of the system are made via a multisignature contract in which four of seven keyholders are required to make important changes. One of the keys is held by ubinetic and the remaining six by independent parties. However, the documentation doesn't specify who these parties are. A governance token (YOU) has been well-defined but the means by which it will be used for voting has not.

Impact

The current Youves platform provides a foundation for a solid governance system but until the system is implemented the platform cannot truly be characterized as decentralized. Users of the system are incentivized to engage with the platform over long periods of time by locking up collateral, saving synthetic asset tokens, and staking governance tokens. Through non-transparent governance decisions these incentives could change to the disadvantage of the users.

Mitigation

We recommend adding a warning to the Risks section of the documentation that the governance features are incomplete and that decision making power is not transparent.

Remediation

We suggest implementing formal processes and structures for voting using the governance tokens. Additionally, we encourage the ubinetic team to consider best practices as adopted by MakerDAO such as correlating voting power with stake, run governance polls, and provide a governance portal. Finally, we recommend implementing timelocks for important decisions so that questionable decisions can be reversed and/or invested users can have an opportunity to exit their positions before being impacted by them.

Status

The ubinetic team has provided additional clarification about the keyholding parties and their affiliations, as well as potential risks in the governance mechanism. This helps create a better understanding of the system's operation in contrast to other stablecoin systems.

However, the aspects of governance utilizing the YOU token have not yet been implemented. As a result, that component of the system cannot be assessed at this time.

Verification

Partially Resolved.

Suggestions

Suggestion 1: Suggest Greater Transparency in the SmartPy Compiler [Out of Scope]

Location

`master/.devcontainer/Dockerfile#L18`

Synopsis

Youves smart contracts are written using the SmartPy programming language. The SmartPy compiler was out-of-scope and, as a result, we did not perform an in-depth analysis as part of this review. However, while SmartPy appears to be well-engineered and suitable for Youves, we found that some aspects of the language are a cause for concern, as detailed below:

- The source code published for SmartPy does not contain commits for each release. In particular, there is no commit for the release v0.6.8 used by Youves;
- SmartPy has not been audited (TQ Tezos has indicated that an audit will be completed in the future but did not provide a specific timeframe for the review); and
- SmartPy has a semi-closed development model and appears to be managed by a single contributor or a small team. The commits to SmartPy's public repository are made by a pseudonymous committer.

We acknowledge that these issues are outside the scope of the Youves code base, however, the security of the SmartPy programming languages directly impacts the security of Youves.

Mitigation

We recommend the Youves development team bring these concerns to SmartPy and encourage them to perform a comprehensive audit of the compiler. Furthermore, it should be suggested to the SmartPy team to improve their development and release practices, in order to provide more visibility and transparency about contributed code.

Status

The ubinetic team has responded that the SmartPy compiler is not within the purview of their work. We recommend that the ubinetic team stay up to date on security developments related to SmartPy that may impact the security of their implementation and update their code base accordingly.

Verification

Unresolved.

Suggestion 2: Improve Documentation

Location

`user_documentation/Youves uUSD Documentation.pdf`, pages 8-9

Synopsis

Some of the documents included in the repository are incomplete. Oracles and exchanges are omitted from uUSD documentation. Robust and comprehensive general documentation allows an independent security auditing team to assess the in-scope components and understand the expected behavior of the system being audited.

Mitigation

We recommend improving and updating the documentation of the planned oracle system.

Status

The ubinetic team has removed outdated documents and updated documentation to include the planned oracle system. In addition, documentation has been consolidated on the Youves site, with some references to the ubinetic documentation for other components of the ecosystem.

Verification

Resolved.

Suggestion 3: Improve Test Coverage

Location

`master/tracker/fa2.py`

`master/tracker/tracker_engine.py#L379-L428`

Synopsis

Although the SAP code base is generally sufficiently tested, we found components with security critical functionality are not included in the tests (e.g. `fa2.py`, `TrackerEngine.burn`). Increasing testing coverage to include all functions and components helps to identify potential edge cases, and helps to protect against errors and bugs, which may lead to vulnerabilities or exploits.

In addition to increasing test coverage in general, SmartPy testing code lends itself to long-winded testing scenarios which can be difficult to carefully read, to spot edge cases. Breaking a long testing function into clearly separate, independent tests which utilize a shared testing library of common patterns can help keep the testing framework readable.

Mitigation

We recommend increasing the test coverage to include all functions and components with security critical functionality.

Status

The ubinetic team has added new tests to `tracker_engine`. However, tests have not been implemented for the `burn` function or for the FA2 contract implementation, as specified in the suggested mitigation. As a result, the suggestion remains partially resolved at the time of verification.

Verification

Partially Resolved.

Suggestion 4: Conduct Audit of Oracle Implementation

Synopsis

While we were able to review the smart contract components of the SAP system, successful operation relies heavily on the security of the oracle system. Actors being able to either submit erroneous values or prevent the smart contracts from accurately reading current market pricing could create arbitrary deviations in the value of uUSD or prevent undercollateralized Vaults from being rescued before liquidation.

Additionally, since different exchanges may be pricing Tez at different values, logic will be needed to reconcile this into a single representative price. Therefore, in addition to ensuring the code is free of logic errors and capable of consensus given conflicting information, oracle incentives to provide frequent and accurate updates are necessary as well.

As per the design of the current oracle system, the price data pushed is being accepted if a majority of the price information received is exactly equal. This relies heavily on trusting the participants to be honest, but does not disincentivize bad participants. As a minor suggestion, we recommend putting in place a mechanism for counting error frequency per price provider, to be able to detect and penalize those specific oracles or remove them from the trusted set.

Mitigation

We recommend an independent security review of the oracle implementation.

Status

The ubinetic team has responded that a security review of the oracle complementation has been performed by the Compass Security team.

Verification

Resolved.

Suggestion 5: Reject Inbound Tez Where Necessary

Location

Examples (non-exhaustive):

`TrackerEngine.mint()`

`TrackerEngine.burn()`

`Vault.default()`

Synopsis

Several entrypoints such as `OptionsListing.fulfill_intent`, `TrackerEngine.create_vault`, and `TrackerEngine.set_vault_delegate` are intended to receive inbound Tez when invoked, whereas the majority of the entrypoints are not intended to receive Tez. Entrypoints not intended to receive inbound Tez should explicitly reject them.

Any amount of Tez sent to the entrypoints not intended to receive Tez, by mistake or intentionally, will be permanently lost.

Mitigation

We recommend adding an explicit check for any amount of Tez that is sent along with the invocation of all entrypoints not intended to receive Tez.

Status

The ubinetic team has responded that this is a design choice to avoid the check, which may cause an excessive burden on gas consumption and operation size. We acknowledge that the design choice is not a security concern.

Verification

Unresolved.

Suggestion 6: Document `Method` and `Internal_Method` Code Pattern

Location

Examples (non-exhaustive):

`TrackerEngine.interest_rate_update()` and `TrackerEngine.internal_interest_rate_update()`

`TrackerEngine.mint()` and `TrackerEngine.internal_mint()`

`StakingPool.deposit()` and `StakingPool.internal_deposit()`

Synopsis

Entrypoints that use external data are in fact implemented in terms of two separate pieces: an entrypoint (`method`) that retrieves some data, which invokes a second entrypoint (`internal_method`) to act upon that data.

An example would be `TrackerEngine.interest_rate_update`: it retrieves prices from oracles, and then invokes `TrackerEngine.internal_interest_rate_update`, like so:

```
sp.transfer(sp.unit, sp.mutez(0), sp.self_entry_point(
"internal_interest_rate_update"))
```

Upon invocation, `internal_interest_rate_update` will have to check that it has been called "internally", with a call to `self.verify_internal(sp.unit)`.

This is an often-repeated code pattern in SAP contracts, and this is not a pattern we have come across before. This pattern may have some downsides:

- `transfer()` calls may cost more gas than necessary.
- `verify_internal()` calls may cost more gas than necessary.
- Contracts have a bigger surface area, because of the internal methods.
- Additional work has to be done to secure `internal_method` entrypoints, and that increases the chances of slippage.

Mitigation

We suggest that the team document the rationale for this design choice, and the trade-offs that they have considered. Additionally, we recommend the team fix this as soon as SmartPy supports/fixes this issue with sub entrypoints.

Status

The ubinetic team has responded that this design follows SmartPy best practices. However, they have not provided documentation explaining the logic behind this code pattern, as suggested in the mitigation. While we acknowledge it adheres to best practices, we recommend the ubinetic team document the reasoning in order to provide reviewers of the code with a better understanding and ability to reason about the system design. As a result, the suggestion remains unresolved at the time of verification.

Verification

Unresolved.

Suggestion 7: Pay Attention to Contract Calls Within SmartPy

Location

`tracker/tracker_engine.py`

Synopsis

SmartPy offers some convenience semantics to pass records to entrypoints using Python keyword arguments. This behaviour is checked in case there is a record as input to the contract, but in cases where the entrypoint takes multiple arguments, unrecognized keyword arguments are ignored. It has been observed that in test code linked the test case is written with the assumption that `token_id` is indeed being passed to the entrypoint, however, removing the `token_id` argument or adding other keyword arguments pass the tests as well.

Mitigation

We recommend checking all function calls in tests and within the contracts to ensure that there are no superfluous arguments being passed.

Status

The ubinetic team has removed the `token_id` argument.

Verification

Resolved.

Suggestion 8: Optimize Gas Cost for Fetching Price Information

Location

[`tracker/options_listing.py`]

Synopsis

`fulfill_intent` calls `fetch_target_price` which increases the number of transactions for each fulfillment. This results in a large number of redundant price fetches in the case that multiple instances of `fulfill_intent` are executed in a short time-frame.

Remediation

We recommend adding a value to the storage to record the time since the last update of the target price, where the price may be fetched only if the price in the storage is older than a specific threshold. This would serve as a cost-saving mechanism.

Status

The ubinetic team has responded they will not be implementing the remediation, citing additional complexity resulting from the implementation. At this point of the project, the tradeoff between complexity and reducing gas cost does not add much value. However, this becomes a valuable improvement across the contract as the number of concurrent users increases. Furthermore, other parts of the contract could benefit from such a cache. As a result, we recommend that the ubinetic team reconsider implementing the suggested remediation.

Verification

Unresolved.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. Although we are a small team, we believe that we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.