DEFIMOON

be secure

Smart Contract Audit Report

March, 2023

Safe Yields

DEFIMOON PROJECT

Audit and Development

CONTACTS

defimoon.org
audit@defimoon.org
defimoon_org
defimoonorg
defimoon
defimoonorg

DEFIMOON

be secure

27 March 2023

This audit report was prepared by DefiMoon for Safe-Yields protocol.

Audit information

Description	DeFi protocol on Arbitrum blockchain implementing a deflationary token mathematically designed to sustain positive price action.
Website	https://www.safeyields.io/
Audited files	interfaces: ISafeToken.sol, ISafeVault.sol SafeToken.sol, SafeVault.sol, Wallets.sol
Timeline	15 March 2023 - 27 March 2023
Audited by	Ilya Vaganov
Approved by	Artur Makhnach, Kirill Minyaev
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Manual Review
Source code	https://github.com/SafeYields/safe-yields-contracts/tree/01b281ac9ed027fba929211eb9e5dbdbb06fedb8/contracts https://github.com/SafeYields/safe-yields-plus-contracts/tree/30bc65672a92c981c83f820bd4cf0ee8388fffef/contracts
Chain	Arbitrum
Status	Passed

All issues have been resolved
0

🟥	High Risk	A fatal vulnerability that can cause the loss of all Tokens / Funds.
🟧	Medium Risk	A vulnerability that can cause the loss of some Tokens / Funds.
🟩	Low Risk	A vulnerability which can cause the loss of protocol functionality.
❗	Informational	Non-security issues such as functionality, style, and convention.

Disclaimer

Audit Information

Defimoon utilizes both manual and automated auditing approach to cover the most ground possible. We begin with generic static analysis automated tools to quickly assess the overall state of the contract. We then move to a comprehensive manual code analysis, which enables us to find security flaws that automated tools would miss. Finally, we conduct an extensive unit testing to make sure contract behaves as expected under stress conditions.

In our decision making process we rely on finding located via the manual code inspection and testing. If an automated tool raises a possible vulnerability, we always investigate it further manually to make a final verdict. All our tests are run in a special test environment which matches the "real world" situations and we utilize exact copies of the published or provided contracts.

While conducting the audit, the Defimoon security team uses best practices to ensure that the reviewed contracts are thoroughly examined against all angles of attack. This is done by evaluating the codebase and whether it gives rise to significant risks. During the audit, Defimoon assesses the risks and assigns a risk level to each section together with an explanatory comment.

Audit overview

Major issues have been found.

Smart contracts contain both small inaccuracies and errors in calculations, and serious vulnerabilities in the logic of the protocol.

Smart contracts assume the use of proxies, but follow insecure approaches - using constructors and not using upgradeable versions of inherited contracts.

The code contains a sufficient number of small bugs, which could be avoided if unit tests were written for smart contracts. We want to draw your attention to the fact that unit tests are a very important part of development. They help to check the operation of the smart contract logic and find minor bugs that are often difficult to notice when reviewing the code.

In addition, the code contains very few useful descriptions of functions, we recommend that you write more detailed comments and notes in NatSpec format, as they help you understand faster and better navigate the code.

Reaudit overview

All vulnerabilities in smart contract logic have been resolved.
Unit tests were written for the functionality of smart contracts.

Summary of findings

ID	Description	Severity	Status
DFM-1	Possible loss of contract	High Risk	Resolved
DFM-2	Token price exploit	High Risk	Resolved
DFM-3	Potential different decimals	Medium	Acknowledged
DFM-4	Users cannot dispose of their tokens	Medium	Resolved
DFM-5	Pause logic not working	Low Risk	Resolved
DFM-6	Admin cant burn	Low Risk	Resolved
DFM-7	Arithmetic miscalculation	Low Risk	Resolved
DFM-8	Redundant condition	Low Risk	Resolved
DFM-9	Empty function	Low Risk	Resolved
DFM-10	Tax calculation error	Low Risk	Resolved
DFM-11	Reserves calculation error	Low Risk	Resolved
DFM-12	Pointless use of SafeMath	Informationa	Resolved
DFM-13	Missing a multiplier	Low Risk	Resolved
DFM-14	Incorrect storage pointer	Informationa	Acknowledged
DFM-15	Upgradeable versions of contracts and initialization	High Risk	Acknowledged
DFM-16	Unit tests	Informationa	Resolved
DFM-17	Using events	Informationa	Partially Resolved

Application security checklist

Compiler errors	Passed
Possible delays in data delivery	Passed
Timestamp dependence	Passed
Integer Overflow and Underflow	Passed
Race Conditions and Reentrancy	Passed
DoS with Revert	Passed
DoS with block gas limit	Passed
Methods execution permissions	Passed
Private user data leaks	Passed
Malicious Events Log	Passed
Scoping and Declarations	Passed
Uninitialized storage pointers	Passed
Arithmetic accuracy	Passed
Design Logic	Passed
Cross-function race conditions	Passed

Detailed Audit Information

Contract Programming

Solidity version not specified	Passed
Solidity version too old	Passed
Integer overflow/underflow	Passed
Function input parameters lack of check	Passed
Function input parameters check bypass	Passed
Function access control lacks management	Passed
Critical operation lacks event log	Passed
Human/contract checks bypass	Passed
Random number generation/use vulnerability	Passed
Fallback function misuse	Passed
Race condition	Passed
Logical vulnerability	Passed
Other programming issues	Passed

Code Specification

Visibility not explicitly declared	Passed
Variable storage location not explicitly declared	Passed
Use keywords/functions to be deprecated	Passed
Other code specification issues	Passed

Gas Optimization

Assert () misuse	Passed
High consumption ‘for/while’ loop	Passed
High consumption ‘storage’ storage	Passed
“Out of Gas” Attack	Passed

Findings /SafeToken.sol

DFM-1 «Possible loss of contract»

Severity: High Risk

Status: Resolved

Description: The administrator can be deprived of access rights.

The SafeToken::rely() and SafeToken::deny() functions allow you to remove existing administrators or even remove your own rights. In this case, you may lose access due to the fact that another administrator deprives you of the rights or the smart contract may be left without an administrator at all if you yourself deprive yourself of the rights through negligence.

Recommendation: Remove the ability to remove your own rights and change the logic for adding and removing administrators if the described finding poses a threat to you.

You can use the OpenZeppelin AccessControl contract, which includes a default owner and allows you to add roles for other members.

Client Comment:

There might be a missunderstanding.
If you could have had a chance to take a look at the code in a deeper detail you'll find out the implementation containing rely() and deny() lyes on top of the ERC-173 compliant proxy which is a widely accepted standard of ownership management. Meaning, rely() and deny() reside on the implementation contract level only (the same is used by Maker DAO Dai stable coin).
We enhanced with ERC-173 which is on proxy contract implementing ownership management.
https://eips.ethereum.org/EIPS/eip-173

Defimoon's reply:

We would like to bring to your attention a potential vulnerability in the implementation contract. While it is true that access to the proxy contract will remain, allowing for updates to the implementation contract in the future, we believe it is important to rectify the situation at the get-go. Updating the implementation contract post-deployment to resolve issues that were flagged before the deployment may cause inconveniences, delays and extra funds being spent, which we consider to be redundant.

In reference to your comment, relying solely on someone else's code as a solution may not be sufficient. It is important to familiarize oneself with the latest updates in the Solidity language and it's documentation, including updates such as the obsolescence of SafeMath library two years ago. We would have taken the same approach with the DAI developers in this situation.

DFM-2 «Token price exploit»

Severity: High Risk

Status: Resolved

Description: Users can manipulate the price of the token in order to make a profit and dump the price of the token.

Using SafeVault:deposit() function, a number of actions can be taken to artificially increase the token price and profit from it. To do this, it is sufficient to:
1) Buy tokens on the SafeToken contract.
2) Make a deposit to the SafeVault contract oneself, which will artificially increase the token price.
3) Sell tokens on the SafeToken contract at an inflated price.
4) Make a withdraw on the SafeVault contract.

For example:
- User buy 1000 tokens: token price = 1.001253
- Attacker buy 100 tokens: token price = 1.001367
- Attacker deposit 1000 usd to SafeVault: token price = 1.912840
- Attacker sell 100 tokens overpriced: received 190 usd
- Attacker withdraw 1000 usd from SafeVault: token price = 0.910572

Such a situation can only occur if other users have already purchased Safe tokens, thereby replenishing the Vault. Due to this vulnerability, other users will suffer losses.

Furthermore, thanks to this vulnerability, not only can the token price gradually dump, but it can also be reduced to less than 1 (0.910572 in the screenshot). This contradicts what is written in your WhitePaper about the token never dropping in price: "$SAFE is 100% collateralized by $USDC and its unique architecture makes it **mathematically impossible to drop in price**.»

Recommendation: We recommend reworking the user interaction logic with the SafeVault contract.

DFM-3 «Potential different decimals»

Severity: Medium Risk

Status: Acknowledged

Description: In contract SafeToken.sol, in addition to its own token, the usd token is also used. When calculating the price of a token, the SafeToken::price() function uses the values of both tokens, which can have different decimals, so it is important to take this into account when calculating. For example, USDT and USDC stablecoins has decimals of 6, while BUSD has decimals of 18.

Since we do not know which token you are going to use, we must warn you about this.

Recommendation: Check if your code takes into account the decimals of both tokens and make sure that if you suddenly want to change the token, then this will not break the code.

The best practice is not to use a static variable, but to get these values of both tokens dynamically when calculating. Like this: usd.decimals() for a third-party token and decimals for your own (or SafeToken::decimals() if called externally).

Client Comment:

Not quite sure what this is doing here as "Medium Risk" which you define as "A vulnerability that can cause the loss of some Tokens / Funds.".
We are perfectly aware of the difference in decimals between stables.
"Since we do not know which token you are going to use" - we state this in the whitepaper. USDC uses 6 decimals as well as the SAFE, which is not configurable as well.

Defimoon's reply:

We would like to clarify our previous message regarding the risk level descriptions. The descriptions of "High Risk" and "Medium Risk" are examples to illustrate the degree of threat and are primarily focused on the potential impact on funds. However, it is important to acknowledge that there may be other types of serious vulnerabilities that are not explicitly described.

In light of your comment, we understand the need for more detailed and comprehensive descriptions. We will consider incorporating additional examples and information in our future assessments.

Regarding the code, we would like to highlight that the specific token used is not set as a constant and is not referenced in any way in the code itself — contracts can be deployed with any token mimicking USDC, and therefore, it is our responsibility to bring any potential issues to your attention. Our comment serves as a reminder for future reference, should you choose to change the token. We trust that the WhitePaper also acknowledges the possibility of changes to the project, as stated "This is a draft based on the overall ideas of how we plan to develop the project, but all of them are subject to CHANGE".

Moreover, we would like to point out that this is a security engagement and even though we do take accompanying documentation, such as WhitePaper into the account, the final decision is made based on what is present in the codebase. The submitted code contains no documentation, almost no comments, no descriptions and no unit tests, which, combined with multiple issues and inconsistencies, make it impossible to determine the true intent of the protocol.

In other words, the claims made in the WhitePaper do not correlate with the provided code. The client claims that the provided code operates as intended. Since we are a security agency and our goal is to protect the protocol and the community, we cannot base our decisions on claims made in promotional materials, personal assurances in authenticity or promises — this information is taken into consideration, but the decision is made based on what is present in the code.

DFM-4 «Users cannot dispose of their tokens»

Severity: Medium Risk

Status: Resolved

Description: Weird logic in a SafeToken::transferFrom() function:

```
require(src == address(0) || dst == address(0) || admin[src] == 1 || admin[dst] == 1,
"SafeToken: transfer-prohibited");
```

This modifier says that token transfers can only be performed either by administrators or by address(0), which is equivalent to burning tokens.

In addition, your contracts also use the transfer() and transferFrom() functions, which means that either the contracts themselves will need to have an administrator role, or the recipients will have an administrator role.

Recommendation: Regardless of whether only administrators should be able to use the transfer, transfer must be disabled on address(0). This is due to the fact that it is used to burn the token, which means that when transferring to address(0) there should be a decrease in supply tokens, as happens when calling burn(). You can follow the example of OZ, whose code has been repeatedly audited and is used everywhere.

Also, please check if it was intended that only administrators can perform the transfer. After all, users still have access to buying and selling tokens, but not their transfer.

Client Comment:

This might be a typo, transferring FROM address(0) is minting but not burning, burning is transferring TO address(0), both of which are intended behavior.

It's quite hard to accept " Weirdness" of the logic as a review conclusion. That's a subjective category. This is part of our token mechanics.
Token transfers are only allowed internally, minting and burning is the only way to buy or sell the token for a protocol participant.
Yes, the contract addresses are authorized with "rely" on deployment, that was one of the reason to using those above.
The core concept of SAFE is minting/burning mechanism to manage the supply as I stated above.
We're aware what transfer to 0 address means and yes, that's the whole intention.
It's also hard to follow the example of OZ.
I also have to remind that OZ is not a EIP and it never claimed to be the one, and being "used everywhere" is a nice but subjective category considering the dynamics of our token.

Defimoon's reply:

Let us rephrase the statement to better reflect our intention: "The logic in the SafeToken function deviates from the ERC20 token standard."

We acknowledge the central role of minting and burning in your protocol and appreciate your understanding of the transfer to address(0) functionality. Our aim with this finding was to highlight a potential vulnerability, or to disclose a potential malicious intent of the contract in the transferFrom() function to end users (in light of your claim), which allows for the bypassing of the burn() function. Through the transferFrom() function, any user can send tokens to address(0), but this will not result in a decrease in totalSupply, unlike burning through the burn() function. Moreover, the burn() function is only accessible for administrators or other contract functions, while any user can send tokens to address(0) through the transferFrom() function.

We provided the example from the OZ contract to demonstrate a secure and reliable implementation of this functionality, as OZ contracts are widely regarded as the standard in Solidity development. It is important to note that OZ serves as a guide and benchmark for security and reliability, utilizing Ethereum Improvement Proposals (EIPs) in its implementation.

DFM-5 «Pause logic not working»

Severity: Low Risk

Status: Resolved

Description: Contract SafeToken.sol can never be paused. The SafeToken::transferFrom() functions include the pause logic:

```
function transferFrom(

    address src,

    address dst,

    uint256 wad

) public nonReentrant returns (bool) {

    require(!paused(), "SafeToken:paused");
```

but the contract does not contain logic that allows you to change the pause state, which means that it will never be possible to pause the contract.

Remember that OZ Pausable contract only includes internal functions for managing pause.

Recommendation: Add administrative functions to control the pause. Do not forget to restrict access to this functionality, for example, with the auth modifier.

Client Comment:

Yes, it's not intended to be paused in its current implementation, which is not part of its interface in this implementation, although can be added later. The token is completely self-contained not possible to be listed on DEX and can't be dropped in price.

Overall good point, low risk accepted, and public function to be exposed requiring admin privileges.

DFM-6 «Admin cant burn»

Severity: Low Risk

Status: Resolved

Description: The _burn() function is only available to admins or calls from inside the contract, but it still contains allowance checks:

```
if (usr != _msgSender() && allowance[usr][_msgSender()] != type(uint256).max) {

    require(allowance[usr][_msgSender()] >= wad, "SafeToken:insufficient-allowance");

    allowance[usr][_msgSender()] = sub(allowance[usr][_msgSender()], wad);

}
```

This suggests that even administrators are not allowed to burn tokens if they have not received permission to do so. There's nothing wrong with that, we just want to make sure it fits your intent.

Recommendation: Pay attention to this and check if it matches your idea and make changes otherwise.

Client Comment:

Okay.
Not clear why it's called "admin can't burn" though.

Defimoons' reply:

We would like to bring to your attention the contradiction between the auth modifier in the burn() function and the requirement for user permission — in the current implementation, the admin cannot burn someone else's tokens without the user's permission. This is why we deemed it necessary to mention this issue in our analysis.

For comparison, we would like to draw your attention to the DAI contract implementation, which you have referenced in your comments, where the burn() function does not utilize any modifiers, avoiding any potential confusion or questions about its operation principle.

DFM-7 «Arithmetic miscalculation»

Severity: Low Risk

Status: Resolved

Description: The SafeToken::buySafeForExactAmountOfUSD() function may have errors in its calculations. Solidity works only with integers, so the result of dividing small numbers by large ones can be zero, and the sum of the result of dividing two numbers by another number will not always be equal to the result of dividing the sum of two numbers by another. This must always be taken into account. In such a case, SafeToken::buySafeForExactAmountOfUSD() can be changed like this:

```
function buySafeForExactAmountOfUSD(uint256 _usdToSpend) public nonReentrant returns
(uint256) {

    uint256 usdTax = _usdToSpend * BUY_TAX_PERCENT / HUNDRED_PERCENT;

    uint256 usdToSwapForSafe = _usdToSpend - usdTax;

    uint256 safeTokensToBuy = (usdToSwapForSafe * 1e6) / price();

    _mint(_msgSender(), safeTokensToBuy);

    usd.transferFrom(_msgSender(), address(this), _usdToSpend);

    uint256 paid = _distribute(usd, usdTax, taxDistributionOnMintAndBurn);

    safeVault.deposit(_usdToSpend - paid);

    return _usdToSpend - paid;

}
```

Recommendation: Fix the function and always pay attention to the peculiarities of calculations in Solidity.

Client Comment:

Accepted. Our calculation could have been made neater, although you also noticed the percent values are increased to mitigate the precision drop.

Defimoons' reply:

Please note that the name of the issue is chosen to demonstrate potential consequences of the issue present. This means that the issue we have described can lead to the miscalculations in the protocol.

It is also important to note that no unit tests, test outputs, any descriptions or any other information that can be used to verify client's claims were provided.

DFM-8 «Redundant condition»

Severity: Low Risk

Status: Resolved

Description: The SafeToken::buyExactAmountOfSafe() function contains an unnecessary if condition:

```
uint256 paid = _distribute(usd, usdTax, taxDistributionOnMintAndBurn);

if (usdTax - paid > 0) {

    safeVault.deposit(usdToSpend - paid);

}
```

Recommendation: The if condition should be removed as it is unnecessary and doesn't match the logic in the implementation block.

DFM-9 «Empty function»

Severity: Low Risk

Status: Resolved

Description: Unused empty function SafeToken::estimateBuyExactAmountOfSafe():

```
function estimateBuyExactAmountOfSafe(uint256 _safeTokensToBuy) public {}
```

Empty functions in a contract can be bad for other contracts that might try to use them, since calling such a function will not return an error. In the future, this may also affect your contracts if you decide to expand the ecosystem.

Recommendation: Delete this function.

DFM-10 «Tax calculation error»

Severity: Low Risk

Status: Resolved

Description: The SafeToken::sellSafeForExactAmountOfUSD() function contains an error in the calculation of tax. The function multiplies by SELL_TAX_PERCENT twice. The function can be changed like this:

```
uint256 usdTax = _usdToGet * SELL_TAX_PERCENT / HUNDRED_PERCENT;

uint256 usdToSpend = _usdToGet + usdTax;
```

```
uint256 safeTokensToSell = (usdToSpend * 1e6) / price();
```

Recommendation: Correct the calculations in the function.

DFM-11 «Reserves calculation error»

Severity: Low Risk

Status: Resolved

Description: The SafeToken::getUsdReserves() function uses the value of SafeVault::totalSupply() to calculate the price. It is assumed that the value of SafeVault::totalSupply() should correspond to the value of the deposit, although this is not the case.

Tokens can simply be sent to the SafeVault.sol contract, resulting in a different SafeVault::totalSupply() and deposit value. Thus, it is possible to achieve an increase in the price of the token without changing the variable that stores the value of the deposit.

Recommendation: Best practice would be to use SafeVault::deposited() instead of SafeVault::totalSupply().

Client's comments:

This is how the token works.

Defimoon's reply:

We would like to provide a demonstration to help clarify this finding in more detail. Using the following example (please note that taxes and multipliers are not taken into account for simplicity):

Consider a scenario where User1 buys 100 SAFE tokens:

usd.balanceOf(SafeVault) = 100
SafeVault.totalSupply() = 100
SafeVault.deposited() = 100
SafeVault.balances(SafeToken) = 100
SafeToken.price() = 1

Then, another user, User2, sends additional 100 usd tokens to the SafeVault contract:

usd.balanceOf(SafeVault) = 200
SafeVault.totalSupply() = 200
SafeVault.deposited() = 100
SafeVault.balances(SafeToken) = 100
SafeToken.price() = 2

In this scenario, if User1 decides to sell their SAFE tokens, they will only be able to sell 50 tokens to retrieve 100 usd. This means that they will have 50 tokens left that they cannot sell, and these tokens will not be backed by anything, despite the fact that the SafeVault contract has usd tokens (as these tokens will be impossible to obtain due to a greedy contract vulnerability).

This demonstrates that the current methodology for calculating the price of the SAFE token is incorrect and should be revised to provide more accurate results. For example, one can use something like this:

SafeToken.price() = SafeVault.deposited() / SafeToken.totalSupply();

DFM-12 «Redundant use of SafeMath»

Severity: Informational

Status: Resolved

Description: Since version 0.8.0, the definition of overflow and underflow of variables is built into the Solidity compiler and the use of the SafeMath library does not make sense, but only takes up the contract bytecode. You are using version 0.8.17.

Recommendation: You can replace using the SafeMath library with regular arithmetic operations.

Findings / Fork SafeToken.sol

DFM-13 «Missing a multiplier»

Severity: Low Risk

Status: Resolved

Description: In the token fork contract, the price calculation is missing a multiplier 1e18 that should be multiplied with the price.

Recommendation: Add a multiplier, as implemented in the original contract.

Findings /Others

DFM-15 «Upgradeable versions of contracts and initialization»

Severity: High Risk

Status: Acknowledged

Description: Your smart contracts assume a proxy implementation, but still use constructors and regular versions of inherited contracts, which can cause contracts to be deployed and initialized incorrectly.

Recommendation: When using an implementation proxy, it is better to use an upgrade version of contracts (analogues from OZ), since such contracts do not take into account the logic written in the constructor. Upgradeable versions of contracts provide for this and use initialization functions.

Also, we highly recommend using OpenZeppelin's Initializable.sol contract (instead of "hardhat-deploy/solc_0.8/proxy/Proxied.sol") in conjunction with the hardhat-upgrades plugin, which does all the work with TransparentProxy and does the interactions with a smart contract simple and convenient.

Client's comments:

These are not regular contracts, there's a "proxied" modifier for constructor/initializer pair making them proxy implementations.

This doesn't cover disadvantages of ERC-1967 transparent proxy here which is currently used and part of hardhat-deploy. It's true that ERC-1967 doesn't have an initialization mechanism.
This also doesn't include advantages of OZ proxy.
Also, ERC-1967 is a widely adopted standard, and OZ Proxy.sol uses it as well.

Defimoon's reply:

We would like to clarify that we are not mandating the use of Initializable.sol and hardhat-upgrades.

The reason for our recommendation is that the current implementation does not use upgradeable versions of contracts, which can potentially lead to improper initialization of inherited contracts. For example, in the SafeNFT contract, the methods of inherited contracts are independently called, rather than being invoked within their constructors.

By using upgradeable versions of contracts, the risk of improper initialization can be reduced and the integration of ready-made solutions can be simplified for better compatibility.

DFM-16 «Unit tests»

Severity: Informational

Status: Resolved

Description: For contracts, there are not even minimal unit tests that could prevent the occurrence of most logical errors, and with the proper approach, some vulnerabilities. Unit tests are an integral part of the development of any application, so they should not be ignored.

Recommendation: We do not recommend launching into production until unit tests are written for all smart contracts that pass without errors.

Client's comment: Accepted.

DFM-17 «Using events»

Severity: Informational

Status: Partially Resolved

Description: Use events in all major functions.

Events help you view the event log for the necessary topics and make a convenient search for them. In addition, events will be useful for collecting statistics and programming interactions with the contract.

Recommendation: The best practice would be to use events in all core functions.

Client's comment:

State changing functions are only transfers here which are being emitted already.

Defimoon's reply:

The functions of selling and buying tokens change the state of the contract and are an important part of the protocol.

Automated Analyses

Slither
Slither's automatic analysis not found vulnerabilities, or these false positives results .

Methodology

Manual Code Review

We prefer to work with a transparent process and make our reviews a collaborative effort. The goal of our security audits is 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.

Vulnerability Analysis

Our audit techniques include 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, review open issue tickets, and investigate details other than the implementation.

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 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 to make a final decision.

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.

Appendix A — Finding Statuses

Resolved	Contracts were modified to permanently resolve the finding
Mitigated	The finding was resolved by other methods such as revoking contract ownership or updating the code to minimize the effect of the finding
Acknowledged	Project team is made aware of the finding
Open	The finding was not addressed