Primitive Audit

OPENZEPPELIN SECURITY | AUGUST 21, 2020

Security Audits

The Primitive team asked us to review and audit an early version of their smart contracts. We looked at the code and now publish our results.

The audited commit is `78a8e64b7618e9199203ab84042876c580ae1e90`, and the following files were in scope:

- contracts/Primitives.sol

- contracts/option/interfaces/IRedeem.sol
- contracts/option/interfaces/IFlash.sol
- contracts/option/interfaces/ITrader.sol
- contracts/option/extensions/Trader.sol
- contracts/option/libraries/TraderLib.sol

Other contracts of the repository outside the scope were explored to the extent they clarified intentions of the in-scope code, such as the `Registry` and the `OptionFactory` contracts, but those were not audited.

All external code and contract dependencies were assumed to work correctly. Additionally, during this audit, we assumed that the system administrators are available, honest, and not compromised.

Before overviewing the system and its privileged roles, and moving to full list of issues found during the audit, some introductory remarks about the project's current status are in order.

Project status

We audited an early version of the Primitive project that is a work-in-progress and not yet ready for production. In view of the project's maturity, this first security audit round should be taken as the initial step forward in the way to reach the highest levels of code quality and robustness demanded by systems intended to handle large sums of financial assets. We identified numerous opportunities for improvement in the project's documentation and code itself, which are highlighted throughout the report. They will require not only specific patches in several code segments, but also efforts in terms of testing and the redesign of how components are architectured. While these issues could be considered symptoms of the inherent difficulty of building a sustainable complex financial system, by no means are they to be taken lightly. Further security reviews of the entire protocol are in order, which along with our recommendations in this report, should help bring the project to a production-ready state.

Furthermore, while reviewing the activity of the development team in the public repository during our audit (and the subsequent review phase), we noticed shortcomings in the applied development practices. As we always recommend to all audited projects, the development team should strive to keep high standards in their software development life cycle with practices of continuous

Update

All issues listed below have been reviewed by the Primitive team, who applied several changes to the code base to address them. These were implemented in branch `hotfix/audit-fixes`. The latest commit in this branch as of the time of this update is `98060324ac6588b1d05748911325a4d39869e4ae`, which was used as reference to review the fixes.

In our initial report, we had raised a number of severe issues in the core `Option` contract stemming from the system's design (see **[C01] Anyone can steal funds from the Option contract before they are used**, **[C03] Deposited funds can be stolen or lost due to transaction ordering** and **[H01] Fragile internal accounting mechanism may cause loss of funds**). After conversations with the Primitive team, it became clearer that the attack vectors described in these issues are only present through an incorrect interaction with the Primitive contracts. In the Primitive team words:

> Features designed to support or protect traders are implemented with separate contracts that call into the `Option` contract. Attempting to transfer tokens to the `Option` contract directly, in a separate transaction, is incorrect and could result in loss of funds.

In summary, all interactions with Primitive's `Option` contract must be carried out via the `Trader` contract, or other similar wrapper contracts that implement all necessary safety checks.

System overview

Primitive is an options market platform that allows users to create new option types, to then mint, exercise, or close them during their entire life cycle. Because these options are fungible ERC20-compliant tokens, users can trade them in 3rd party exchanges to receive a premium.

The audited scope covered some of the smart contracts that implement these mechanisms:

- The `Primitives` contract defines how the parameters of each option type will be stored. After our initial report, this has been merged into the `Option` contract.
- The `Option` contract contains the core business logic of each option type. It is a ERC20-compliant contract that implements the minting, exercising, redeeming, and closing

option tokens. Redeem tokens are used during the closing and redeeming processes and are linked to only one option type.

- The `Trader` contract allows users to safely perform the multiple steps required by the `Option` contract's operations in a single transaction. **Interactions with the `Option` contract must always be carried out through this contract.**
- The `TraderLib` library implements the actual functionality used in the `Trader` contract.

The project extensively uses contracts from the OpenZeppelin Contracts library, such as the `ERC20`, `SafeERC20`, and `Ownable` contracts.

Privileged roles

In the original audited scope, a system owner had the power to pause and unpause `Option` contracts at will. When an option was paused, users would not be able to mint new option tokens (and the associated redeem tokens), nor exercise their right to exchange strike assets for underlying assets. We highlighted that users of the Primitive protocol must be aware that system administrators were allowed to lock funds at will, as addressed in issue **"[H03] Malicious owner can execute Denial of Service attacks"**.

After our initial report, the Primitive team decided to entirely remove the pausing functionality (and the related privileged role) from the `Option` contract.

Ecosystem dependencies

The protocol uses time-based logic to determine when options expire, which means it is dependent on the availability of the Ethereum network. If users are temporarily unable to submit transactions to exercise options (for instance, during a period of high Ethereum congestion, or due to miners delaying transactions purposely), they can lose their exercise rights.

Critical severity

[C01] Anyone can steal funds from the Option contract before they are used

balances of underlying and strike tokens. The caller can also select any redeem and option tokens in balance.

The `Option` contract assumes that users send funds prior to interacting with it. Given that the `take` function is marked as `external` and it does not implement any kind of access controls, it allows anyone to steal deposited funds before they are effectively used by their owners. It should be noted that other attacks can be carried out to steal funds from the `Option` contract, as described in the related issue **"[C03] Deposited funds can be stolen or lost due to transaction ordering"**. Moreover, the lack of clear docstrings makes it difficult to understand the `take` function's intention and the scenarios in which it should be called.

For this case in particular, it might be necessary to re-think the purpose of the exposed `take` function, which now offers an entry point for attackers to steal user funds from the contract. Yet this issue is not to be considered in isolation, but rather as part of bigger shortcomings in the contract's design. Any solution for this issue should also take into account what is described in the issues **"[H01] Fragile internal accounting mechanism may cause loss of funds"** and **"[C03] Deposited funds can be stolen or lost due to transaction ordering"**.

Update: *Fixed in PR #34. The Primitive team decided to remove the `take` function from the `Option` contract. The purpose of this function was to force sync the actual balances of the underlying, strike, option, and redeem tokens. In word of the Primitive team:*

> The consequence of this is that any erroneously transferred option or redeem tokens that are sent to `Option.sol` would be forever locked in the contract. Additionally, any erroneously transferred underlying or strike tokens will be forever locked in `Option.sol` once `updateCacheBalances` is called either externally or internally through one of the core functions.

To avoid erroneously sending tokens into the `Option` contract, users must use the `Trader` contract. The Primitive team warns users about this in the newly added docstrings above the `Option` contract.

[C02] Issued options might not be backed by underlying assets

block's timestamp is already greater than its expiry time (set in the option's `expiry` field).

Before expiration, the `Option` contract allows closing options via its `close` function. Closing options can be seen as the counterpart of minting new ones, and requires both option and redeem tokens to be sent in, which are burned to account for the extracted underlying tokens. However, the `close` function does not always correctly burn transferred option tokens when closing non-expired options.

Specifically, the function incorrectly validates whether an option is expired by using the `>` operator (see lines 327 and 338 of `Option.sol`), when it should use the `>=` operator instead (to match the `notExpired` modifier). As a consequence, `close` incorrectly considers expired options those whose `expiry` time equals the current block's timestamp, thus failing to burn option tokens deposited when closing non-expired options (but still transferring out the corresponding underlying tokens).

These unburned tokens will remain in the `Option` contract's balance until they are extracted by anyone calling the `take` function.

As we showcase in the attack vector below, the described flaw can be exploited by attackers to profit by issuing and selling options of the Primitive protocol that will not be backed by underlying assets – therefore breaking the fundamental invariant that the Primitive protocol should hold.

Let us consider an option with DAI and WETH as underlying and strike assets respectively, with a base of 200 DAI and a quote of 1 WETH.

1. Attacker deposits 100 DAI into the `Option` contract and receives 100 option tokens and 0.5 redeem tokens.
2. The attacker sells 80 option tokens in the open market.
3. Before expiration, when the current block's timestamp matches the option's expiry date, the attacker executes the following steps 5 times:

- Deposit 0.1 redeem tokens and 20 options in the `Option` contract.
- Call the `close` function, burning the deposited redeem tokens and receiving 20 DAI in return.
- Call the `take` function, collecting the 20 option tokens that were not burned.

that backed the 100 options minted, which are now not backed by deposits in the Primitive protocol.

All components of the protocol should use the same criteria to evaluate whether an option is expired. Therefore, consider applying the necessary changes in the `close` function so that its validations match what is stated in the `notExpired` modifier. Moreover, consider extracting the validation logic to an internal function that can be reused across the code base. Finally, related unit tests are in order to prevent re-introducing this critical flaw in future changes to the code base.

Update: *Fixed in PR #14. Suggested changes have been made.*

[C03] Deposited funds can be stolen or lost due to transaction ordering

The `Option` contract plays a fundamental role in the Primitive protocol. It contains the core business logic for operations such as minting new options, allowing option holders to exercise their right, letting the option writers to redeem the strike assets after an exercise, or closing options before expiration.

Essentially, there are two ways of performing these actions:

- Directly through the `Option` contract.
- Using the `Trader` contract.

The `Trader` contract is intended to increase usability of the contracts by checking balances, transferring previously approved funds into the `Option` contract, and finally calling the corresponding function of the `Option` contract to effectively execute the desired operation. These steps are executed in a single atomic transaction.

In must be noted that users can still interact with the `Option` contract directly, as long as the necessary funds are deposited into it prior to the execution of the desired operation. Nevertheless, this has been found to be extremely dangerous, as most scenarios can lead to loss or theft of funds. This is due to the fact that:

1. The process of interacting with the `Option` contract usually involves two or more steps. If these are executed in separate transactions, they can be frontrun or backrun, potentially

described in issue "**[low] Fragile internal accounting mechanism might cause loss of funds**").

As an example, consider the scenario where Alice wants to mint new option tokens. To do it, Alice first deposits some underlying assets into the `Option` contract. When Alice calls the `mint` function, the contract will check its balance of underlying assets and it will use the difference between the old cached balance and the newer one to calculate how many option tokens must be minted. If Bob sends a transaction to deposit underlying assets which is placed *before* Alice's call to `mint`, deposits from Alice and Bob will be merged, and whoever gets mined the transaction calling `mint` first will receive all option and redeem tokens. Ultimately, miners in full control of transaction ordering can leverage this scenario to place all deposit transaction first and later place a single transaction to steal all deposited funds. It must be highlighted that similar scenarios can be replicated across different operations of the `Option` contract. This is not particular to any specific function, but rather a fundamental problem in the contract's design.

A similar attack vector allows rendering funds unusable. In this case, any malicious user who backruns transactions that solely deposit funds into the `Option` contract can call the `update` function. This will trigger an internal synchronization of cached and actual balances in the contract, and deposited funds will no longer be available for the victim to operate in the Primitive protocol.

Considering the described attack vectors affect several operations of the `Option` contract, it might be necessary to re-design how users are expected to interact with it. The system should be designed in such a way that users are not exposed to serious transaction ordering attacks when using the protocol. A plausible solution might be to entirely prevent users from executing operations in the `Option` contract. This would greatly reduce the system's attack surface, and interactions with the `Option` contract should only be carried out via another contract (such as the `Trader`) that correctly implements the necessary checks and safely executes operations.

Update: *By design, users must never directly interact with the* `Option` *contract. Doing so could result in loss of funds. The attack vector described is only present through an incorrect interaction with the contracts, as described in PR #35. Relevant docstrings and warnings have been added above the* `Option` *contract, and the* `updateCacheBalances`, `mintOptions`, `exerciseOptions`, `redeemStrikeTokens` *and* `closeOptions` *functions.*

The `Option` contract does not keep track of individual asset deposits done by users. Instead, it uses two state variables to track the last known total of underlying and strike token balances (known as “cached balances”). When a user interacts with functions of the `Option` contract that require having deposited funds (such as `mint`, `exercise`, `redeem`, or `close`), the contract assumes that any difference between the actual balance (obtained querying the token) and the latest cached balance belongs to the calling account.

Nevertheless, this way of handling deposited funds has several shortcomings that render the entire internal accounting too fragile.

To begin with, in a scenario where multiple users are interacting with the contract, deposited assets will not be differentiated on a per-user basis. Upon any call to a function that requires having deposited assets, the difference between the actual and cached balance can be greater than the funds sent by the caller, resulting in loss of funds for everyone else.

Secondly, functions such as `close` only check whether a minimum amount of assets were deposited prior to the call, thus allowing for deposits to be actually greater than those to be effectively used. This could potentially lock these funds into the contract without possibility of recovering them. As an example, if a user sends more option tokens than needed along with the necessary redeem tokens, then the extra option tokens will be burned instead of returning the remaining ones to the user.

A similar case occurs in the `exercise` function when a user returns all underlying assets, having deposited more strike tokens than they should. These strike assets will be included into the cached balance, and the user will not be able to withdraw them from the contract.

Finally, the owner of the system can pause the `Option` contract without warning users with enough time in advance. If they are already depositing funds in order to operate in the protocol in subsequent operations, pausing the contract will prevent minting or exercising options. As a consequence, funds would start accumulating until the contract is suddenly unpaused, and the first user that mines a call to the `take` function would withdraw all the non-cached balance for all deposited assets.

that the problems here described are not to be considered in isolation, but rather as part of bigger shortcomings in the contract's design. Related problems are presented in the reported issues **"[C01] Anyone can steal funds from the Option contract"** and **"[C03] Deposited funds can be stolen or lost due to transaction ordering"**, and the solution should take into account all these issues as a whole.

Update: *By design, users must never directly interact with the* `Option` *contract. Doing so could result in loss of funds. The attack vector described is only present through an incorrect interaction with the contracts, as described in PR #36. Relevant docstrings wherever necessary have been added to warn users.*

[H02] Flash loans of underlying tokens cannot be executed safely

The `Trader` contract implements a set of functions that allow users to safely interact with the `Option` contract, executing all necessary steps to operate in the protocol in a single transaction. Additionally, it includes security checks to ensure asset balances involved are between expected boundaries and it calculates the exact amount of assets needed to perform a successful operation.

However, the `Trader` contract does not include any function to safely execute flash loans of underlying tokens through the `exercise` function of the `Option` contract. As a consequence, users willing to take advantage of flash loans in the Primitive protocol are exposed to the risk of losing the deposited fees (as reported in issues **"[C01] Anyone can steal funds from the Option contract before they are used"** and **"[C03] Deposited funds can be stolen or lost due to transaction ordering"**) before being able to call the `exercise` function. Alternatively, they would need to code their own custom solutions to execute the entire operation in a single transaction, which could only be done by technically skilled users proficient in Solidity.

Consider implementing a secure method to perform flash loan operations in a single transaction, as it was done for the rest of the protocol's operations in the `Trader` contract. Any particular solution to this issue should also take into account what is described in **"[M07] Convoluted implementation of exercise and flash loan features"**.

Update: *Acknowledged in PR #37, and will not fix. According to the Primitive team, the system design relies on the fact that this feature is to be only used by other smart contracts that implement*

The `Option` contract inherits the pausing feature from the OpenZeppelin Contract's `Pausable` contract. In each `Option` contract, only the `factory` address is allowed to call the `kill` function that can effective pause or unpause the contract. Following the access control chain, it can be noted that the `owner` account of the `Registry` contract can call the `kill` function and trigger a pause. It must be highlighted that pausing and unpausing the `Option` contract is an immediate action, since there is no timelock mechanism in place. Moreover, the private key for the administrator's account is expected to be in control of the Primitive team, so users should fully trust them to execute a pause / unpause action under the right circumstances.

Pausing only affects the `mint` and the `exercise` functions. If a the private key controlling the owner account is compromised, the attacker can indefinitely prevent new options from being minted, and all issued options from being effectively exercised.

This kind of simple fail-safe mechanism are reasonable for early versions of projects under development and testing. However, they become more relevant as projects launch to mainnet, gain importance in the space, or users start depositing large sums of money into them.

In the short term, consider adding extensive, user-friendly, documentation related to the implemented pausing mechanism. The documentation should highlight under which circumstances the system is to be paused or unpaused, what are the specific consequences of pausing over the system's mechanics, and who is allowed to trigger this feature. For future versions of the protocol, it is advisable to start devising and implementing mechanisms that favor decentralization, giving users the possibility to exit the system before its behavior changes.

Update: *Fixed in PR #20. The Primitive team has removed the entire pausing functionality from the `Option` contract, by removing the inheritance from `Pausable`, the `kill` function and the `whenNotPaused` modifier. Note that these changes are not inline with our recommendation, even though they do mitigate the described attack vector.*

Medium severity

[M01] Fees cannot be collected

underlying tokens sent out by the contract, and is expected to be paid by the caller in strike tokens.

Before finishing execution of the `exercise` function, the cached balances of strike and underlying tokens are updated to the latest balance queried during the transaction. However, the logic does not keep track of the added fees. Thus they will be lost without any possibility of collecting them.

Consider modifying the way in which fees are tracked so that they can be effectively collected. Related issue **"[H01] Fragile internal accounting mechanism may cause loss of funds"** should be taken into account to solve this particular issue. Alternatively, if fees do not have a clear purpose in the Primitive protocol, consider removing them from the system altogether.

Update: *Fixed in PR #16. The concept of fee has been removed from the system altogether. The docstrings above the `exercise` function must be updated to reflect this.*

[M02] Exercises may avoid fee payments

The Primitive protocol intends to charge fees when an option is exercised. These fees are calculated as a percentage of the amount of underlying tokens the `Option` sends out in the exercise, and are expected to be paid by the caller in strike tokens.

While all exercises are expected to pay fees, callers can avoid paying them by exercising extremely low amount of tokens. In particular, in all exercises where the amount of underlying tokens to send out (which is a user-controlled parameter) is lower than 1000, no fees will be charged. This is due to the fact that the fee is calculated using Solidity's integer division, which rounds down to zero any value lower than 1.

Considering most tokens have 18 decimals, exercises with amounts lower than 1000 could be considered unlikely, since it might not be profitable for anyone (accounting for gas costs) to exercise such amount. However, the scenario described becomes more important as the number of decimals of the underlying token shrinks. For example, traders using tokens with less decimals such as USDC (which has 6), are more likely to take advantage of this issue to avoid fee payments.

case the calculated fee is zero. Otherwise, consider clearly documenting and testing this behavior to prevent unexpected outcomes.

Update: *Fixed in PR #16. The concept of fee has been removed from the system altogether.*

[M03] Anyone can trigger a flash loan for unprotected receivers

The `exercise` function of the `Option` contract allows anyone to execute a flash loan of underlying tokens deposited in the contract. The caller can specify in the `receiver` parameter any contract address that implements the `IFlash` interface.

Should the receiver contract not implement the necessary validations to identify who originally triggered the transaction, it may be possible for an attacker to force any `IFlash` contract to open arbitrary flash loans in the Primitive protocol that would inevitably pay the corresponding fees (in strike tokens). This can potentially drain all tokens in balance from the vulnerable contract implementing the `IFlash` interface. It should be highlighted that the `Flash` contract (used for testing purposes) is the only available example of an implementation of the `IFlash` interface, and it does not include any security measure, nor warning documentation, to prevent this issue.

To reduce the attack surface, it is advisable to modify the flash loan logic so that only the actual receiver of the loan can execute it. If opening flash loans on behalf of `IFlash` contracts is an intended feature, then consider adding user-friendly documentation to raise awareness, along with sample implementations showcasing how to defend from attackers that attempt to open flash loans on behalf of unprotected `IFlash` contracts. Finally, any solution for this issue should take into consideration what is described in related issue **"[M07] Convoluted implementation of exercise and flash loan features"**.

Update: *Partially fixed in PR#17. The call to the receiver's `primitiveFlash` function now passes the `msg.sender` address as argument, so as to inform the receiver who the caller of the `exercise` function is. However, the project is still lacking user-friendly documentation and examples to guide the development of secure implementations of receiver contracts.*

[M04] Mismatches between contracts and interfaces

counterpart contracts that are not defined, as listed below:

The `IRedeem` interface is missing:

- The `initialize` function.

The `IOption` interface is missing:

- The `take` function.
- The `kill` function.
- The `update` function.

Moreover, the definition of the `getParameters` function in the `IOption` interface does not match its counterpart implementation `getParameters` of the `Option` contract, which returns the first two values in the wrong order. This can potentially lead to unexpected errors and loss of funds, since anyone that uses the `IOption` interface will expect to receive token addresses in a different order.

Finally, line 22 in `ITrader.sol` does not match with the returned values in line 82 of `Trader.sol`.

Consider applying the necessary changes in the mentioned interfaces and contracts so that definitions and implementations fully match.

Update: *Fixed in PR #23. Suggested changes have been made.*

[M05] Lack of event emission after sensitive actions

In several parts of the code base there are sensitive functions that lack event emissions.

- The `initRedeemToken` function in the `Option` contract does not emit an event when the redeem token's address is set.
- The `take` function in the `Option` contract does not emit an event when callers withdraw underlying, strike, redeem, or option tokens.

Update: *Fixed in PR #19. An event is emitted in* `initRedeemToken` *function when the redeem token's address is initialized. Note that while the mentioned PR adds an event to the* `take` *function, the function was later renamed to* `withdrawUnusedFunds` *and finally removed from the code base in PR #34.*

[M06] Misleading comments and docstrings

Several docstrings and inline comments throughout the code base were found to be erroneous and / or incomplete and should be fixed. In particular:

- In line 140 of `Option.sol`, docstrings do not mention that the function also mints redeem tokens.
- In line 177 of `Option.sol`, the documentation says `exerise` instead of `exercise`.
- In line 195 of `Option.sol`, the inline comment does not reflect the actual conditions being checked in the `require` statement in lines 197 to 200.
- Line 237 of `Option.sol` states that *"Assumes the cached optionToken balance is 0, which is what it should be"*, but actually this is not true if another user sents — near the same time — option tokens to the contract. The contract does not check this. The same problem happens in line 324 of `Option.sol`.
- Line 275 of `Option.sol` states that the strike tokens are sent to the `msg.sender` address but actually it is sent to the `receiver` parameter.
- In line 289 of `Option.sol`, the docstrings state that the function burns option tokens, even though this is not always true.
- In line 181 of `Trader.sol` and in line 166 of `TraderLib.sol` the documentation states that the `unwindQuantity` parameter represents the `Quantity of redeemTokens to burn`, although it is actually the amount of option tokens used to calculate the amount of redeem tokens to be burned.
- In line 164 of `TraderLib.sol`, docstrings state that strike tokens are withdrawn during the execution of the function, which is not true.
- In line 166 of `TraderLib.sol`, docstrings state that `unwindQuantity` of redeem tokens are burned, but actually `unwindQuantity * quote / base` of redeem tokens are burned.

Clear docstrings are fundamental to outline the intentions of the code. Mismatches between them and the implementation can lead to serious misconceptions about how the system is expected to behave. Therefore, consider fixing these errors to avoid confusions in developers, users, auditors alike.

Update: *Partially fixed in PR #24. The line 131 of* `Option.sol` *incorrectly states that Redeem tokens are minted at a* `base:quote` *ratio, however, they are minted at* `quote:base` *ratio. Additionally, the line 159 of* `TraderLib.sol` *still incorrectly states that strike tokens are withdrawn during the execution of the function.*

[M07] Convoluted implementation of exercise and flash loan features

The `exercise` function of the `Option` contract is intended to allow option-token holders to exercise the right of exchanging strike assets for underlying assets at a certain strike price.

Nevertheless, depending on the arguments passed to the function, the `exercise` function can change its behavior to allow flash loans of underlying tokens.

Exercising an option and taking out flash loans of underlying tokens are two different use cases of the Primitive protocol. Merging these two functionalities under a single function renders the code more difficult to read and understand by users, developers and auditors alike. Additionally, the added complexity in the business logic to handle both features makes the implementation more error-prone and harder to test.

Consider splitting the exercise and flash loan features into two separate, independent, functions.

Update: *Acknowledged in PR #37, and will not fix. The Primitive team has decided not to apply the suggested changes, arguing that the proposed separation of functionalities would lead to code duplication.*

[M08] Expired and/or paused options can still be traded

Every `Option` contract has an expiry date, set during initialization, after which the contract will no longer allow minting new options nor exercising existing ones. Similarly, minting and exercising are also prevented when the `Option` contract is paused by a privileged account in the system.

cannot be exercised in the open market to exchanges and users who do not take the necessary precautions before buying an option minted by the Primitive protocol. Particularly, decentralized exchanges willing to trade Primitive's option tokens should implement the necessary logic in their contracts to avoid accepting paused or expired options into their pools.

Should this be the system's expected behavior, consider clearly documenting it in user-friendly documentation so as to raise awareness in option sellers and buyers. Moreover, the documentation should include a specific section aimed for developers willing to integrate the Primitive protocol into their platform, providing the necessary guidance on these particular scenarios. Lastly, it should be noted that the described behavior is not being covered with unit tests.

Alternatively, if the described behavior is not intended, consider implementing the necessary logic in the `Option` contract to prevent transfers of tokens during pause and after expiration. Note that the implementation should take into account that option tokens might be needed to be transferred back to the `Option` contract, even during pause or after expiration, to be able to still use other features of the protocol.

Update: *Not fixed. Expired options can still be transferred, and no additional documentation has been added to highlight it. Note that the issue no longer applies for paused options, since the pausing functionality has been removed altogether from the* `Option` *contract.*

[M09] Untested, undocumented, support for tokens with different decimals

The underlying and strike assets used to create options in the Primitive protocol are essentially the addresses of two different ERC20 tokens. While some popular ERC20 tokens such as DAI have 18 decimals, others such as USDC or USDT are represented with less decimals (6 in these cases).

Even though no security-related issues were identified in the code base regarding the creation of options with tokens of different decimals (as long as the quote and base parameters are chosen sensibly), it is unclear to what extent the Primitive protocol is expected to support this type of options. It must be highlighted that no specific documentation was found related to this particular

Consider including unit tests to ensure the system behaves as expected when options with tokens of a varying number of decimals are created. This should also help increase coverage and improve the overall quality of the testing suite. Related documentation is in order too, so as to clearly specify the intended behavior of the protocol.

Update: *Not fixed.*

Low severity

[L01] Erroneous data logged in events

- The `Close` event should emit the caller's address and the number of options sent in, but it is actually logs the receiver address and the number of underlying tokens sent out.
- The `Redeem` event should emit the caller's address, but it actually logs the receiver address.
- The `Exercise` event should emit the caller's address, but it actually logs the receiver address.
- The `Mint` event should emit the caller's address, but it actually logs the receiver's address.

Update: *Fixed in PR #25. Event definitions and logged data now match.*

[L02] External call does not check if target is contract

The `Option` contract performs external function calls during its execution to user-controlled addresses. An example can be seen in the `exercise` function, where the `receiver` argument is implicitly assumed to be a contract.

While this issue does not pose a security risk, consider always using the `Address` library from OpenZeppelin Contracts to explicitly validate whether user-controlled targets of external calls are indeed contracts.

Update: *Not fixed.*

[L03] Naming issues hinder code understanding and readability

- `parameters` to `optionParameters`.
- `take` to `withdrawUnusedFunds`.
- `update` to `updateCacheBalances`.
- `tokens` to `getAssetAddresses`.
- `redeem` to `redeemStrikeTokens`.
- `exercise` to `exerciseOptions`.
- `mint` to `mintOptions`.
- `_fund` to `_updateCacheBalances`.
- `caches` to `getCacheBalances`.
- `strikeToken` to `getStrikeTokenAddress`.
- `underlyingToken` to `getUnderlyingTokenAddress`.
- `base` to `getBaseValue`.
- `quote` to `getQuoteValue`.
- `expiry` to `getExpiryTime`.

Update: *Fixed in PR #26.*

[L04] Incomplete IRedeem and IOption interfaces

Although the `Redeem` and `Option` contracts inherit from OpenZeppelin Contract's `ERC20` contract, the corresponding interfaces `IRedeem` and `IOption` do not inherit from the `IERC20` interface.

To favor explicitness and avoid unexpected errors when using the `IRedeem` and `IOption` interfaces, consider inheriting from the `IERC20` interface in the `IRedeem` and `IOption` interfaces.

Update: *Fixed in PR #27.*

[L05] Lack of input validation in Option contract's initialization

The `initialize` function of the `Option` contract does not validate the initialization parameters passed. In particular:

division-by-zero errors in other operations.

- The `expiry` time can be set in the past (that is, lower than the current block's timestamp).

To avoid errors and explicitly restrict the type of options that can be created, consider implementing `require` clauses where appropriate to validate all user-controlled input.

Update: *Fixed in PR #28.*

[L06] Missing check in initRedeemToken function

The Primitive protocol intends to have a single redeem token associated with each `Option` contract. This redeem token can only be set once. Only the owner of the `OptionFactory` contract can call the `initialize` function which in turns calls the `initRedeemToken` function in the `Option` contract. The owner of the `OptionFactory` contract is the `Registry` contract which deploys the factory contract by calling the `deployOption` function.

In summary, `Option` contracts are deployed via the factory, and the factory is owned by the registry, and there does not seem to be a way of setting the redeem token for an option twice. However, future changes to the code base might introduce viable ways of doing it.

Since setting the redeem token is an important functionality of the Primitive protocol, in order to reduce the attack surface, consider adding an explicit check in the `initRedeemToken` function that verifies that the address of `redeemToken` is zero before setting it to the passed `_redeemToken` address.

Update: *Fixed in PR #29.*

[L07] Missing docstrings

Some of the contracts and functions in Primitive's code base lack documentation. This hinders reviewers' understanding of the code's intention, which is fundamental to correctly assess not only security, but also correctness. Additionally, docstrings improve readability and ease maintenance. They should explicitly explain the purpose or intention of the functions, the scenarios under which they can fail, the roles allowed to call them, the values returned and the events emitted.

documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec), as addressed in the issue **"[N04] Not following the Ethereum Natural Specification Format"**.

Update: *Partially fixed. While there is no PR covering this particular issue, overall improvements have been made in terms of docstrings. However, there are still exposed functions without docstrings (see the* `initialize` *function of* `Redeem` *contract as an example).*

[L08] Multiple conditions in a single require statement

There is a single require statement with multiple conditions in lines 241 to 243 of `Option.sol`. Consider isolating each condition in its own require statement, so as to be able to include specific user-friendly error messages for every required condition.

Update: *Fixed in PR #30.*

[L09] Repetitive getters and public state variables

The `Option` contract declares a struct `parameters` which is marked as `public` and can be accessed by anyone. The contract also defines a `public` getter function `getParameters` which returns the value of `parameters` struct along with the address of `redeemToken`.

Additionally, there are other getter functions such as `tokens`, `strikeToken`, `underlyingToken` (and more) that return the value of individual variables of the `parameters` struct or a combination of them.

Similarly, the variables `underlyingCache`, `strikeCache`, `redeemToken` in `Option` contract are marked as public and can also be are accessed via the `caches` and `tokens` functions.

Moreover, other public variables in the code base that do not have explicit associated getters are:

- `factory` in the `Option` contract.
- `factory`, `optionToken` and `redeemableToken` in the `Redeem` contract.

Update: *Not fixed.*

[L10] Unified pausing and unpausing functionality

The `Option` contract allows the `factory` account to perform important actions over the contract. Among them, this privileged account can pause and unpause certain operations in the `Option` contract via the `kill` function.

Given pausing and unpausing are two different use cases, and they might involve specific business logic particular to each one in future versions of the protocol (such as different timelock mechanisms), consider splitting the pausing and unpausing logic into two different functions. Additionally, the Primitive team might need to consider whether pausing and unpausing should be executed by the same privileged account, or by different accounts that do not share privileges.

Update: *Fixed by removing the pausing functionality altogether in PR #20.*

[L11] Duplicate variables and unnecessary lines of code

There are occurrences in the code of the `Option` contract where a variable simply duplicates the value of another variable. In most of these cases, one of the two variables are useful in the function, making the other redundant. One such example is in the `redeem` function where the variable `inRedeems` is assigned the value of the variable `redeemBalance`. While `inRedeems` is used further in the function, `redeemBalance` has no purpose after line 271.

Also, in line 281, the variable `redeemBalance` is assigned a new value which remains unused. Since this line of code is unnecessary, it can be removed.

Similar issues can be found in the `close` function of the `Option` contract.

To favor readability and avoid confusions, consider removing all unnecessary assignments, and avoiding using multiple variables for the same purpose.

Update: *Fixed in PR #31.*

Notes & Additional Information

production soon, it should strive to always follow best practices of code cleanliness to avoid future errors.

In particular, consider removing any lines of code from the contracts that were included for testing purposes. These leftovers from tests in the contracts could lead to the deployment of undesired code. For instance, it seems that the use of `weth` in the constructor of the `Trader` contract was solely intended for testing purposes.

Additionally, for improved readability, consider using smaller names and symbols for the `Option` and `Redeem` tokens.

Update: *Not fixed.*

[N02] Inconsistent coding style

The code base does not always follow a consistent coding style. Some cases identified in the `Option` contract are:

- It inherits the properties from the OpenZeppelin's `ERC20` contract and with it, the `balanceOf` function. Nevertheless, the `Option` contract calls this function in 2 different ways in line 124 and in line 238.
- The revert reasons, such as the one in line 222, display in capital letters and underscores a brief error message. However, given the contracts also use contracts from OpenZeppelin Contracts, such as `Pausable`, the style of revert reasons are not going to be consistent across the code base.
- Some functions declare the return variable in the function definition, and then copy a state variable's value into the implicit return variables instead of directly returning the value of the state variables. Furthermore, this style is not consistent across all functions, where getters from the same contract return directly the state variable instead of copying into a new one.

Consider fixing this inconsistent styles to improve the project's readability. As reference, consider always following the style proposed in Solidity's Style Guide. Taking into consideration how much value a consistent coding style adds to the project's readability, enforcing a standard coding style with help of linter tools such as Solhint is recommended.

Consider removing all named return variables, explicitly declaring them as local variables, and adding the necessary return statements where appropriate. This would improve both explicitness and readability of the code, and may also help reduce regressions during future code refactors.

Update: *Not fixed.*

[N04] Not following the Ethereum Natural Specification Format

Most of the docstrings in the code base are not fully following the Ethereum Natural Specification Format (NatSpec). Consider following this specification on everything that is part of a contracts' public API.

Update: *Not fixed.*

[N05] OpenZeppelin Contract's dependency is not pinned

To prevent unexpected behaviors in case breaking changes are released in future updates of the OpenZeppelin Contracts's library, consider pinning the version of this dependency in the package.json file.

Update: *Not fixed.*

[N06] Solidity compiler version is not pinned

Throughout the code base, consider pinning the version of the Solidity compiler to its latest stable version. This should help prevent introducing unexpected bugs due to incompatible future releases. To choose a specific version, developers should consider both the compiler's features needed by the project and the list of known bugs associated with each Solidity compiler version.

Update: *Not fixed.*

[N07] Unnecessary library

The `Primitives` library consists of only a struct `Option` and has no other functions or variables in it. This struct is solely used in the `Option` contract.

Update: *Fixed in PR #32.*

[N08] Unused import statements

In the code base, there are places in which contracts are imported but are never used. Consider removing the following unused imports:

- `IERC20` in the `Redeem` contract.
- `ERC20`, `IERC20`, and `SafeERC20` in the `Trader` contract.

Additionally, the `TraderLib` library imports the `IERC20` interface from the `ERC20.sol` file, while it could directly import the `IERC20` interface from the `IERC20.sol` file.

Update: *Fixed in PR #33.*

[N09] Unused state variables

The `Redeem` contract declares a variable `redeemableToken` which is later assigned in the `initialize` function. The `deployOption` function in the `Registry` contract calls the `deploy` function present in the `RedeemFactory` contract, which in turn calls the `initialize` function present in the `Redeem` contract and sets the value of `redeemableToken` as the address of `strikeToken`.
However, after the initialization, this variable is not used anywhere in the code base.

Additionally, the `Trader` contract declares a variable `weth` which is initialized in the constructor but is not used anywhere in the code.

In order to increase code readability and favor simplicity, consider removing the state variable `redeemableToken` from the `Redeem` contract (making the subsequent necessary code changes). Similarly, consider removing the `weth` state variable from the `Trader` contract.

Update: *Not fixed.*

Conclusions

considering them inherent to the system's design) or correctly fixed.

We audited an early version of the Primitive project that is a work-in-progress and not yet ready for production. This first audit round has been Primitive's initial step on its way to reach the needed level of maturity for projects intended to handle large sums of financial assets. To further help the project reach a production-ready state, we highly advise additional rounds of security reviews.

Related Posts

Beefy Zap Audit

BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

Security Audits

OpenBrush Contracts Library Security Review

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits

Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits

OpenZeppelin