ackee blockchain

Siren AMM

3 February 2022

by Ackee Blockchain

Contents

1. Document Revisions

DRAFT	Final Report	February 3, 2022
1.0	Final Report	February 15, 2022
1.1	Final Report • Add Appendix E, *Fix Review*	March 7, 2022

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain

Ackee Blockchain is an auditing company based in Prague, Czech Republic, specialized in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run a free certification course Summer School of Solidity and teach at the Czech Technical University in Prague. Ackee Blockchain is backed by the largest VC fund focused on blockchain and DeFi in Europe, Rockaway Blockchain Fund.

2.2. Audit Methodology

1. **Technical specification/documentation** - a brief overview of the system is requested from the client and the scope of the audit is defined.
2. **Tool-based analysis** - deep check with automated Solidity analysis tools and Slither is performed.
3. **Manual code review** - the code is checked line by line for common vulnerabilities, code duplication, best practices and the code architecture is reviewed.
4. **Local deployment + hacking** - the contracts are deployed locally and we try to attack the system and break it.
5. **Unit and fuzzy testing** - run unit tests to ensure that the system works as expected, potentially write missing unit or fuzzy tests.

2.3. Review team

Member’s Name	Position
Dominik Teiml	Lead Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.4. Disclaimer

We’ve put our best effort to find all vulnerabilities in the system, however our findings shouldn’t be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

Siren is a protocol that allows users to mint, trade, and exercise options for ERC20-compliant Ethereum tokens. Siren AMM is a subcomponent that allows to buy and sell option tokens in the form of an AMM.

Between Jan 3 and Feb 3, 2022, Siren Markets engaged ABCH to conduct a security review of the AMM subcomponent. This engagement followed up our previous review of the `/series` subdirectory from November 2021. Working commit 0329d49e58, we paid special attention to:

1. the `/amm` subdirectory,
2. the Welford library,
3. the VolatilityOracle,
4. the AddressesProvider, the SeriesDeployer and the WTokenVault.

Where appropriate, we also reviwed the dependencies of these contracts, such as SeriesController and Proxiable. In the period mentioned above, we were allocated 22 enginering days and the lead auditor was Dominik Teiml.

We began our review by using static analysis tools, namely Slither and the solc compiler. This yielded several issues such as 6.11 and 6.7, as well as possible code quality improvements, outlined in Appendix C. We then took a deep dive into the logic of the contracts. During the review, we paid special attention to:

- ensuring the arithmetic of the system is correct,
- detecting possible reentrancies in the code,
- ensuring access controls are not too relaxed or too strict,
- looking for common issues such as data validation.

We also created a `model` of the arithmetic in a higher-level language, that we then used to test the system. While easier to work with than the low-level implementations, we weren't able to find a vulnerability in the context of the arithmetics employed.

Our review resulted in 19 findings, ranging from Informational to High severity. The most critical issue was that related to re-entrancy protection (see Possibility of re-entrancy). While we weren't able to find an exploit scenario for this issue in the allocated time, we believe the protocol should have stricter re-entrancy protections.

Ackee Blockchain recommends Siren:

- extend the AddressesProvider to implement a global re-entrancy lock with modifiers on all non-view public entrypoints,
- address all reported issues,
- use Slither for vulnerability detection. Slither was able to detect several findings including 6.19, 6.11 and 6.7.

4. System Overview

This section contains an outline of the audited contracts. Note that this is meant for understandibility purposes and does not constitute a formal specification.

4.1. Contracts

AddressesProvider

The AddressesProvider is the central registry. It is used by the other modules to get addresses of other contracts.

ChainlinkEthUsdProxy

The ChainlinkEthUsdProxy converts prices from ETH/USD and ERC20/ETH to ERC20/USD.

Welford

Welford is a Solidity library that can compute volatility of price with an on-line, iterative manner. It is based on a contract by Ribbon Finance (see 6.8).

VolatilityOracle

The VolatilityOracle is used to compute the volatility of a token over a number of price readings. It stores the accumulated values for a token in an `accumulator`, and uses Welford to update these values.

BlackScholes

BlackScholes is a contract to comute the estimate of the price of an option token. It is based on a contract by Lyra Protocol (see 6.8).

SeriesDeployer

The SeriesDeployer allows anyone to create a series. The only pre-conditions are that the number of series per expiration date has not yet been reached, that an AMM exists for that (underlying token, price token, collateral token) triple, and, finally, that the strike price is one of the allowed values.

WTokenVault

The WTokenVault allows the MinterAmm to lock and redeem collateral and active writer option tokens.

AmmDataProvider

The AmmDataProvider provides helper functions for the MinterAmm, SirenExchange, and the WTokenVault. It doesn't have any publicly-accessible or internal non-view functions. It calls other contracts to gather information, and computes results based on those values.

AmmFactory

The AmmFactory allows the `owner` to create new MinterAmms.

MinterAmm

The MinterAmm is an implementation of an AMM to trade Siren option assets. Each AMM is parameterized by a (underlying token, price token, collateral token) triple and uses a combination of the SeriesController, AmmDataProvider, VolatilityOracle and BlackScholes to mint option tokens and offer them to users in the form of an automated market maker.

SirenExchange

The SirenExchange allows exchanging buying and selling bTokens for arbitrary user tokens. It does this by using the MinterAmm together with a

Uniswap V2 implementation.

4.2. Actors

This part describes actors of the system, their roles and permissions.

AddressesProvider

The [AddressesProvider](#) has an `owner` that is the initializer by default. The `owner` may transfer ownership, udpate the imlementation pointer, set address in a low-level manner for any `bytes32 id`, and set address registries for all system contracts.

ChainlinkEthUsdProxy

The [ChainlinkEthUsdProxy](#) does not have an `owner` or any other privileged actor.

VolatilityOracle

The [VolatilityOracle](#) has an `owner` that is the deployer. The `owner` has the opportunity to transfer ownership per `OwnableUpgradeable.transferOwnership`, and can add token pairs, set accumulator values and last prices for token pairs.

BlackScholes

[BlackScholes](#) does not have any `owner` or any other privileged actor.

SeriesDeployer

The [SeriesDeployer](#) uses [AccessControlUpgradeable](#), however, it only uses one role (the `DEFAULT_ADMIN` role). This is by default the initializer, and is also called the `owner` throughout the contract, and can be transferred. The `owner` may update the implementation contract, the `AddressesProvider` registry and may arbitrarily change the maximum number of series for each expiration date. Finally, the `owner` may update the allowed strike price ranges for each underlying token.

WTokenVault

The WTokenVault has an `owner`, by default the initializer. The `owner` may be transferred to a different address through `OwnableUpgradeable.transferOwnership`. There are currently no other features that only the `owner` can execute.

AmmDataProvider

The AmmDataProvider does not have any `owner` or any other privileged actor.

AmmFactory

The AmmFactory has an `owner` which is by default the initializer. The `owner` may update the token and amm implementation that new AMMs are proxied to. They can also update the AmmFactory implementation, transfer their ownership and create AMMs.

MinterAmm

The MinterAmm has an `owner`, by default the initializer. The `owner` may transfer their ownership, udpate the implementation and AddressesProvider pointer. They can also set the trading and maxium fees fees, their destination address, and set the config values used for price calculation, namely `ivShift`, `dynamicIvEnabled` and `ivDriftRate`.

SirenExchange

The SirenExchange does not have any `owner` or any other privileged actor.

5. Vulnerabilities risk methodology

Each finding contains an `Impact` and `Likelihood` ratings.

If we have found a scenario in which the issue is exploitable, it will be assigned an impact of `Critical`, `High`, `Medium`, or `Low`, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of `Warning` or `Informational`.

`Low` to `Critical` impact issues also have a `Likelihood` which measures the probability of exploitability during runtime.

5.1. Finding classification

The full definitions are as follows:

Impact

High

Code that activates the issue will lead to undefined or catastrophic consequences for the system.

Medium

Code that activates the issue will result in consequences of serious substance.

Low

Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.

Warning

The issue cannot be exploited given the current code and/or configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.), but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as "Warning" or higher, based on our best estimate of whether it is currently exploitable.

Informational

The issue is on the border-line between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or configuration (see above) was to change.

Likelihood

High

The issue is exploitable by virtually anyone under virtually any circumstance.

Medium

Exploiting the issue currently requires non-trivial preconditions.

Low

Exploiting the issue requires strict preconditions.

6. Findings

This section contains the list of discovered findings. Unless overriden for purposes of readability, each finding contains:

- a *Description*,
- an *Exploit scenario*, and
- a *Recommendation*

Many times, there might be multiple ways to solve or alleviate the issue, with varying requirements in terms of the necessary changes to the codebase. In that case, we will try to enumerate them all, making clear which solve the underlying issue better (albeit possibly only with architectural changes) than others.

Summary of Findings

Id		Type	Impact	Likelihood
1	Possibility of re-entrancy	Re-entrancy	High	Medium
2	Pitfalls of upgradeability	Access controls, Upgradeability	Warning	N/A
3	`SeriesDeployer.autoCreateSeriesAndBuy` contains unchecked `transfers`	Data validation	High	Medium
4	WToken Vault has no access controls	Access controls	Medium	High
5	`MinterAmm.claimAllExpiredTokens` contains a `for` loop with a dynamic condition	Code maturity	Informational	N/A

Id		Type	Impact	Likelihood
6	Use `_msgSender` over `msg.sender`	Builtin variables	Informational	N/A
7	Missing zero-address checks	Data validation	High	Low
8	Contracts used as dependencies don't track upstream changes	Dependencies	High	Low
9	Code layout can be improved	Code maturity	Informational	N/A
10	Use of semantic values as defaults in enums	Data validation	Warning	Medium
11	No return parameter in `SeriesController.setSettlementPrice`	Return parameters	Warning	High
12	`SeriesController.state` doesn't have data validation	Data validation	Warning	High
13	Usage of `solc` optimizer	Compiler configuration	High	Low
14	System is lacking in documentation	Specification & documentation	Informational	N/A
15	OpenZeppelin's upgradeable contracts are used in non-upgradeable contracts	Dependencies	Medium	Low

Id		Type	Impact	Likelihood
16	`ChainlinkEthUsdProxy.latestRoundData` can contain uninitialized return parameters	Uninitialized variables	Warning	N/A
17	Initialization functions are inconsistently named	Code maturity	Informational	N/A
18	Log old values in logs	Logging	Informational	High
19	`MinterAmm.updateVolatility`'s return parameter is never initialized	Return parameters	Low	High

Table 1. Table of Findings

6.1. Possibility of re-entrancy

Impact:	High	Likelihood:	Medium
Target:	`/**/*`	Type:	Re-entrancy

Background

Re-entrancy vulnerabilities come in various forms. The vulnerability might occur when an external call separates two code blocks, and somewhere on the network there is code that is contingent on both blocks executing without interruption.

A special case of this is when the re-entrancy is the same function. However, it could be a different function, or a different contract, or even a different protocol altogether. Whenever there exists logic on the network that is contingent on the second code block, it could be possible to utilize a code injection to violate their atomicity.

Re-entrancies most commonly occur in:

- ether transfers,
- transfers of ERC20s that are also ERC223s or ERC777s,
- transfers of ERC1155s.

Protection

One way to prevent re-entrancies is to use the checks-effects-interactions pattern. However, this is not always possible. A function's semantics may include:

- state mutations to the current contract based on external interactions,
- multiple external interactions (code elsewhere may depend the atomicity of these multiple interactions).

Another way to protect against reentrancies is by introducing a re-entrancy lock. Based on Background, a re-entrancy lock will only work if:

1. It protects *all* public entrypoints of a contract.

   It is not enough to protect just *publicly-accessible* functions. An `onlyOwner` function may, for example, transfer tokens, and those may call callbacks. If that is the case, the atomicity of `onlyOwner` function may be violated.

2. It protects *all* public entrypoints of *all* contracts.

   Other modules may rely on the contract's state. If an attacker calls these modules, they may perform a *dirty read*.

3. The lock can be read by *any* network contract.

   Similarly, other projects may rely on the contract's state.

Note that it is only necessary to protect mutating functions. View functions might give incorrect results if injected, but they will be relevant only if called by a function that is non-view.

Siren

Siren added the `nonReentrant` modifier to all relevant publicly-accessible non-view functions.

Based on Background, it is not possible to perform all external calls atomically at the end. Hence a re-entrancy lock really is necessary. However, based on the analysis in Protection, it is not enough to just protect publicly-accessible functions.

Vulnerability scenario

An Ethereum protocol `P` relies on `Siren`. Mallory calls one of the publicly-accessible functions. One of the token transfers allows her to execute arbitrary code. She re-enters into `P`, which calls `Siren`. At this stage, `Siren` is in an uncommitted state. She is able to exploit that to attack protocol `P`.

Alternativaley, the same exploit, but for a different module in `Siren` rather than for a different protocol.

Recommendation

Add a system-wide re-entrancy lock in AddressesProvider by declaring a state variable representing a lock. When any mutating function in the system is called, there will be a switch on the caller (`msg.sender`):

- if it is any contract in the system, the call will proceed,
- if it is not and the lock has been acquired, the call will revert,
- if it is not any contract in the system and the lock has not been acquired, it will be acquired.

This will ensure the project is resilient against the re-entrancy attacks outlined above.

Go back to Findings Summary

6.2. Pitfalls of upgradeability

Impact:	Warning	Likelihood:	N/A
Target:	`/**/*`	Type:	Access controls, Upgradeability

Description

Many contracts in the system are upgradeable per Proxiable. Currently, the initialization process for these contracts is:

1. Deploy the (logic) contract.
2. Deploy the proxy, pointing to the logic contract.
3. Call the initialization function on the proxy.

There are three issues with the current upgradeability process:

1. The logic contracts have no access controls to prevent malicious actors from interacting with them directly. Note that this is only a problem insofar as they could change the logic contract's code.
2. An attacker could front-run one of the initialization functions.
3. An attacker could call other functions on the proxy before initialize is called on it.

Analysis of Requirement #1

Contract code invariant	A contract that doesn't use `callcode`, `delegatecall` or `selfdestruct` instructions cannot be selfdestructed. Moreover, its code cannot change.

Based on the Contract code invariant, the only way to change a contract's code is through the use of `callcode`, `delegatecall` or `selfdestruct`. We checked that none of the logic contracts uses one of these statements. For a fully generic discussion, see Appendix D.

Recommendation

To protect against Requirement #2, consider:

- requiring in your deployment scripts that the initialization call is successful. OpenZeppelin's `initializer` modifier will ensure that if the initialization has been called, the call to it will fail.
- modifying Proxy's constructor to take an additional `bytes` parameter. Delegatecall this on the target, requiring the call be successful. This will ensure an atomic construction and initialization of the Proxy.

To protect against Requirement #3, refer to Appendix D.

Go back to Findings Summary

6.3. SeriesDeployer.autoCreateSeriesAndBuy contains unchecked transfers

Impact:	High	Likelihood:	Medium
Target:	SeriesDeployer	Type:	Data validation

Listing 1. SeriesDeployer.sol#L261-L266

```
        // Move the collateral into this address and approve the AMM
        IERC20(ammTokens.collateralToken).transferFrom(
            msg.sender,
            address(this),
            _collateralMaximum
        );
```

Listing 2. SeriesDeployer.sol#L291-L297

```
        // Send any unused collateral back to buyer
        if (IERC20(ammTokens.collateralToken).balanceOf(address(this))
    > 0) {
            IERC20(ammTokens.collateralToken).transfer(
                msg.sender,
                IERC20(ammTokens.collateralToken).balanceOf(address
    (this))
            );
        }
```

Description

The SeriesDeployer is a module that allows anyone to create a series whose (`underlying`, `price`, `collateral`) token triple already exists in a AMM. The `autoCreateSeriesAndBuy` implements this, and also allows users to directly buy bTokens of the series from the corresponding AMM.

To do this, it uses `transferFrom` to transfer the collateral tokens from the user, and `transfer` to send any unused collateral back.

However, in these instances, it doesn't use `SafeERC20`, nor does it appropriately handle the case of tokens returning `false` (rather than reverting) on failure conditions (such as insufficient allowance).

Vulnerability scenario

An ERC20 token with the above behavior is used as a collateral token in an AMM. Since one of these functions may not have the expected behavior of transferring tokens, undefined behavior may occur.

For example, the `transfer` function used to send leftovers may fail, for whatever reason internal to the token at hand. As a result, the AMM will keep the leftover of the user's collateral tokens.

Recommendation

Short term, use `SafeERC20` in these instances.

Long term, always use `SafeERC20` when interacting with external tokens. This will ensure the maximum support range for variously-behaving ERC20 tokens.

Go back to Findings Summary

6.4. WToken Vault has no access controls

Impact:	Medium	Likelihood:	High
Target:	WTokenVault	Type:	Access controls

Listing 3. WTokenVault.sol#L66-L77

```
function lockActiveWTokens(
    uint256 lpTokenAmount,
    uint256 lpTokenSupply,
    address redeemer,
    uint256 volatility
) external override {
    LocalVars memory vars;

    ISeriesController seriesController = ISeriesController(
        addressesProvider.getSeriesController()
    );
    IMinterAmm amm = IMinterAmm(msg.sender);
```

Listing 4. WTokenVault.sol#L243-L248

```
function lockCollateral(
    uint64 seriesId,
    uint256 collateralAmount,
    uint256 wTokenAmount
) external override {
    address ammAddress = msg.sender;
```

Description

WTokenVault has many code locations where it assumes the caller is an AMM (see Listing 3 and Listing 4). Undefined behavior could occur if these publicly-accessible functions are called by other parties.

Recommendation

Short term, either remove the assumptions that the caller is an AMM, and document the expected behavior for a generalized caller, or, alternatively, add access controls to forbid other callers.

Long term, avoid unclear assumptions; this will make the code more readable.

Go back to Findings Summary

6.5. `MinterAmm.claimAllExpiredTokens` contains a `for` loop with a dynamic condition

Impact:	Informational	Likelihood:	N/A
Target:	MinterAmm	Type:	Code maturity

Listing 5. MinterAmm.claimAllExpiredTokens

```
    /// @notice Claims any remaining collateral from all expired series whose wToken is held by the AMM, and removes
    /// the expired series from the AMM's collection of series
    function claimAllExpiredTokens() public {
        for (uint256 i = 0; i < openSeries.length(); i++) {
            uint64 seriesId = uint64(openSeries.at(i));
            while (
                seriesController.state(seriesId) ==
                ISeriesController.SeriesState.EXPIRED
            ) {
                claimExpiredTokens(seriesId);

                // Handle edge case: If, prior to removing the Series, i was the index of the last Series
                // in openSeries, then after the removal `i` will point to one beyond the end of the array.
                // This means we've iterated through all of the Series in `openSeries`, and we should break
                // out of the while loop. At this point i == openSeries.length(), so the outer for loop
                // will end as well
                if (i == openSeries.length()) {
                    break;
                } else {
                    seriesId = uint64(openSeries.at(i));
                }
            }
        }
    }
```

Listing 6. MinterAmm.sol#L599-L601

```
        // Remove the expired series to free storage and reduce gas fee
        // NOTE: openSeries.remove will remove the series from the i th
position in the EnumerableSet by
        // swapping it with the last element in EnumerableSet and then
calling .pop on the internal array.
```

Description

MinterAmm contains a function `claimAllExpiredTokens`. The intended behavior of this function is to loop over all series' tokens held by the AMM, and claim collateral for those that are past the expiry date.

The implementation of this function is unsatisfactory:

1. Even though it is looping over one list, it uses two loops to accomplish that
2. It relies on `claimExpiredTokens`' behavior, which uses implementation-specific details of Enumerable Set (see Listing 6).
3. The bound of the condition in the for loop (`openSeries.length()`) is dynamic and might change on every iteration. Dynamic conditions are very difficult to read, debug, and reason about.

Recommendation

Short term, investigate the option of using `MinterAmm.allSeries` to fetch all `id`s, and then claiming tokens and evicting series appropriately. Unless are other disadvantages, this should solve all three of the above issues.

Long term, avoid anti-patterns such as loop conditions with side-effects or ones that are dynamic. This will make the code much easier to reason about.

Go back to Findings Summary

6.6. Use `_msgSender` over `msg.sender`

Impact:	Informational	Likelihood:	N/A
Target:	VolatilityOracle, SeriesDeployer, AddressesProvider, WTokenVault, MinterAmm, AmmFactory	Type:	Builtin variables

Description

The following contracts have ContextUpgradeable in its inheritance chain:

1. VolatilityOracle
2. SeriesDeployer
3. AddressesProvider
4. WTokenVault
5. MinterAmm
6. AmmFactory

ContextUpgradeable defines `_msgSender` and `_msgData` functions. This makes it easy to switch their semantics, e.g. if Siren decides to support metatransactions in the future. If a contract inherits from ContextUpgradeable, uses of `msg.data` and `msg.sender` should be replaced by `internal` calls to `_msgData` and `_msgSender`, respectively. This will ensure that if the semantics is changed in the future, the codebase will remain consistent.

Recommendation

Short term, replace all instances of `msg.sender` with `_msgSender()` in the contracts that inherit from ContextUpgradeable. This will ensure futureproofness against future code changes.

Long term, ensure that all contracts’ code is consistent with the code of their inherited contracts.

6.7. Missing zero-address checks

Impact:	High	Likelihood:	Low
Target:	/**/*	Type:	Data validation

Listing 7. MinterAmm.setTradingFeeParams

```
    /// The owner can set the trade fee params - if any are set to 0/0x0 then trade fees are disabled
    function setTradingFeeParams(
        uint16 _tradeFeeBasisPoints,
        uint16 _maxOptionFeeBasisPoints,
        address _feeDestinationAddress
    ) public onlyOwner {
        tradeFeeBasisPoints = _tradeFeeBasisPoints;
        maxOptionFeeBasisPoints = _maxOptionFeeBasisPoints;
        feeDestinationAddress = _feeDestinationAddress;
        emit TradeFeesUpdated(
            tradeFeeBasisPoints,
            maxOptionFeeBasisPoints,
            feeDestinationAddress
        );
    }
```

Listing 8. VolatilityOracle.sol#L83-L93

```
    constructor(
        uint32 _period,
        IPriceOracle _priceOracle,
        uint256 _windowInDays
    ) {
        require(_period > 0, "!_period");
        require(_windowInDays > 0, "!_windowInDays");

        period = _period;
        priceOracleAddress = _priceOracle;
        windowSize = _windowInDays.mul(uint256(1 days).div(_period));
```

Description

There are multiple places in the system where zero-address checks are not present (see [Listing 7](#) and [Listing 8](#)). While not a perfect method of data validation, zero-address checks are the first line of defense against incorrectly supplied input arguments.

Vulnerability scenario

Bob is an employee of Siren or a project cloning Siren. They call `setTradingFeeParams`, but because of a bug in the scripting library, the abi values are incorrectly encoded. The contract interprets the `_feeDestinationAddress` argument as `0x0` and sets the state variable accordingly. Fees are sent to this external address, resulting in loss of funds.

Recommendation

Short term, add a zero-address check for all addresses and contracts used as inputs to the system.

Long term, investigate more stringent method of data validation, such as through a specific id, to catch even more instances of machine or human error.

[Go back to Findings Summary](#)

6.8. Contracts used as dependencies don’t track upstream changes

Impact:	High	Likelihood:	Low
Target:	EnumerableSet, DSMath, Math, PRBMathSD59x18, Welford	Type:	Dependencies

Description

The system relies on several dependencies, such as `OpenZeppelin`’s `contracts-upgradeable` library, and `Maker`’s `DSMath` library. Some of these dependencies are managed through npm, a package manager. However, the following third-party contracts are not:

- EnumerableSet
- DSMath
- Math
- PRBMathSD59x18
- Welford

Vulnerability scenario

A vulnerability is discovered in one of these contracts. A hotfix is immediately pushed, as well as security alerts into all package managers, where it is distributed. Since these dependencies are used without a package manager, the following consequences may occur:

- developers installing these dependencies will not be alerted through their package manager
- a vulnerable version of the contract will be used during deployment

- the team will not find out about the vulnerability

Recommendation

Short term, track upstream changes of these dependencies using package managers that they are officially distributed in.

Long term, track all upstream changes of all dependencies. This will ensure the latest version of the contracts is used, which is the least likely to contain security concerns, as well as the least likely to contain safety concerns.

Go back to Findings Summary

6.9. Code layout can be improved

Impact:	Informational	Likelihood:	N/A
Target:	/**/*	Type:	Code maturity

Listing 9. [BlackScholes.sol#L179-L189](#)

```
        uint256 prob = uint256(
            (d *
                (3193815 +
                    ((-3565638 +
                        ((17814780 +
                            ((-18212560 + (13302740 * 1e7) / t1) * 1e7)
    /
                            t1) * 1e7) /
                        t1) * 1e7) /
                    t1) *
                1e7) / t1
        );
```

Description

There are many instances in the codebase when arithmetic operations span up to 11 LoC (see [Listing 9](#)). This hinders readability for users, auditors, developers, and other stake-holders. Splitting this onto multiple arithmetic expressions, separated by assignment operations, will significantly improve readability of the code.

Recommendation

Introduce local variables in places such as [Listing 9](#). This will improve readability while not compromising performance.

[Go back to Findings Summary](#)

6.10. Use of semantic values as defaults in enums

Impact:	Warning	Likelihood:	Medium
Target:	SeriesController	Type:	Data validation

Listing 10. ISeriesController.SeriesState

```
    /// @notice All possible states a Series can be in with regard to
its expiration date
    enum SeriesState {
        /**
         * New option token cans be created.
         * Existing positions can be closed.
         * bTokens cannot be exercised
         * wTokens cannot be claimed
         */
        OPEN,
        /**
         * No new options can be created
         * Positions cannot be closed
         * bTokens can be exercised
         * wTokens can be claimed
         */
        EXPIRED
    }
```

Listing 11. ISeriesController.FeeType

```
    /** Enum to track Fee Events */
    enum FeeType {
        EXERCISE_FEE,
        CLOSE_FEE,
        CLAIM_FEE
    }
```

Description

There are multiple times in the system when Solidity enums are used (see, for example, Listing 10 and Listing 11).

Solidity enums are implemented as integers. As such, the default value on the Ethereum Virtual Machine of 0 will be interpreted as the first enum option. This can be dangerous, for example if enums are used as state variables.

Vulnerability scenario

One of the two enums above is used in a state variable, for example in a mapping, or a top-level state variable. When an unitialized value is read, it will default to the first enum option even though that might not be desired behavior. Since this will not raise compiler warnings, and current tools such as Slither do not warn about this, it can be difficult to catch.

Recommendation

Short term, add a `NULL` or `DEFAULT` value to the above enums. This will ensure that whenever one of the other options is read, it doesn't correspond to the default value.

Long term, always assume that the first value could be an unitialized one. This will prevent future bugs.

Go back to Findings Summary

6.11. No return parameter in `SeriesController.setSettlementPrice`

Impact:	Warning	Likelihood:	High
Target:	SeriesController	Type:	Return parameters

Listing 12. SeriesController.setSettlementPrice

```
    /// @notice Sets the settlement price for all settlement dates prior to the current block timestamp
    /// for the given <underlyingToken>-<priceToken> pair
    /// @param _seriesId The specific series, accessed by its index
    function setSettlementPrice(uint64 _seriesId) internal {
        Series memory currentSeries = allSeries[_seriesId];

        return
            IPriceOracle(priceOracle).setSettlementPrice(
                address(currentSeries.tokens.underlyingToken),
                address(currentSeries.tokens.priceToken)
            );
    }
```

Description

SeriesController has an internal function `setSettlementPrice` that calls the PriceOracle to set the relative price of two assets (see Listing 12). The function has a return statement even though it doesn't have a return parameter.

Vulnerability scenario

Inspecting the code, a developer believes that `setSettlementPrice` returns the set price. It does not, leading to bad consequences.

Recommendation

Short term, either remove the return statement from `SeriesController.setSettlementPrice`, or add a return parameter and return statement to `PriceOracle.setSettlementPrice` and a return parameter to `SeriesController.setSettlementPrice`. This will ensure consistency and expected behavior of the system.

Long term, avoid returning return values from functions that don't return anything. While not currently enforced by the compiler, it is poor practice and can lead to unintended consequences.

Go back to Findings Summary

6.12. `SeriesController.state` doesn’t have data validation

Impact:	Warning	Likelihood:	High
Target:	[SeriesController](#)	Type:	Data validation

Listing 13. [SeriesController.state](#)

```
    /// @notice Returns the state of a Series, which can be OPEN or
EXPIRED. The OPEN state
    /// means the block timestamp is still prior to the Series'
expiration date, and so option
    /// tokens can be minted or closed. The EXPIRED state means the
block timestamp is after
    /// the expiration date, and now the bTokens can be exercised and
the wTokens claimed
    /// @param _seriesId The index of this Series
    /// @return The state of the Series
    function state(uint64 _seriesId)
        public
        view
        override
        returns (SeriesState)
    {
        // before the expiration
        if (block.timestamp < allSeries[_seriesId].expirationDate) {
            return SeriesState.OPEN;
        }

        // at or after expiration
        return SeriesState.EXPIRED;
    }
```

Description

[SeriesController](#) has a function `state(uint64)` that returns the current state a series is in (see [Listing 13](#)). If the series does not currently exist, it returns `SeriesState.EXPIRED`. External parties can make the conclusion that the series has expired, when in fact a series with that id does not exist.

Note that fixing 6.10 would make this issue less severe. However, we still recommend making partial functions (functions that revert on invalid inputs) unless there is a good reason to work with uninitialized series.

Vulnerability scenario

A function in the system calls `SeriesController.state` with a non-existent series' id. The return value is interpreted as an expired series, which can lead to unintended consequences.

Recommendation

Short term, unless there arises a situation when `state` returning an uninitialized series' state would be semantically significant, add validation to `state` that if `expirationDate == 0`, the function should revert.

Long term, add more data validation to getter functions. While this might have a gas cost trade-off, it will improve the security of your system.

Go back to Findings Summary

6.13. Usage of `solc` optimizer

Impact:	High	Likelihood:	Low
Target:	`/**/*`	Type:	Compiler configuration

Description

The project uses the `solc` optimizer. Enabling the `solc` optimizer may lead to unexpected bugs.

The Solidity compiler was audited in November 2018 and the audit concluded that the optimizer may not be safe.

Vulnerability scenario

A few months after deployment, a vulnerability is discovered in the optimizer. As a result, it is possible to attack the protocol.

Recommendation

Until the `solc` optimizer undergoes more stringent security analysis, opt out using it. This will ensure the protocol is resilient to any existing bugs in the optimizer.

Go back to Findings Summary

6.14. System is lacking in documentation

Impact:	Informational	Likelihood:	N/A
Target:	`/**/*`	Type:	Specification & documentation

Description

The system is currently lacking high-level documentation. While some contracts, such as the MinterAmm, have contract-level natspec, many contracts and many functions lack natspec documentation. This hinders readability and makes onboarding onto the system more difficult.

Recommendation

Short term, add contract-level and function-level natspec to all contracts and functions, respectively. Additionally, add a high-level overview of the system.

Long term, document the codebase while writing it. This will ensure maximum transparency and ease of use for developers, users, and auditors.

Go back to Findings Summary

6.15. OpenZeppelin’s upgradeable contracts are used in non-upgradeable contracts

Impact:	Medium	Likelihood:	Low
Target:	VolatilityOracle	Type:	Dependencies

Listing 14. VolatilityOracle.sol#L12-L12

```
contract VolatilityOracle is DSMath, OwnableUpgradeable {
```

Description

VolatilityOracle is not meant to be upgraded, as it doesn’t inherit from Proxiable and hence lacks the `_updateCodeAddress` method. However, it still inherits from OwnableUpgradeable (see Listing 14). OwnableUpgradeable is meant to be used for upgradeable contracts.

Recommendation

Short term, inherit from Ownable instead.

Long term, use all dependencies in the way they are intended to be used.

Go back to Findings Summary

6.16. ChainlinkEthUsdProxy.latestRoundData can contain uninitialized return parameters

Impact:	Warning	Likelihood:	N/A
Target:	ChainlinkEthUsdProxy	Type:	Uninitialized variables

Listing 15. ChainlinkEthUsdProxy.latestRoundData

```
    function latestRoundData()
        external
        view
        override
        returns (
            uint80 roundId,
            int256 answer,
            uint256 startedAt,
            uint256 updatedAt,
            uint80 answeredInRound
        )
    {
        (, int256 ethUsdPrice, , , ) = ethUsdOracle.latestRoundData();
        (, int256 assetEthPrice, , , ) = assetEthOracle.
latestRoundData();

        require(ethUsdPrice > 0, "ETH/USD price is 0");
        require(assetEthPrice > 0, "ASSET/ETH price is 0");

        answer = (ethUsdPrice * assetEthPrice) / priceDivisor;
    }
```

Description

ChainlinkEthUsdProxy is a module that aggregates two chainlink oracles to produce asset prices in USD. It exposes the same API as Chainlink aggregators, including the function `latestRoundData()`.

The issue is that since the prices can come from multiple Chainlink rounds, it is difficult to determine the appropriate values for the following return parameters:

- `roundId`,
- `startedAt`,
- `updatedAt`,
- `answeredInRound`

In the current code (see Listing 15), these return parameters remain uninitialized. As they are all integers, they will be 0.

Vulnerability scenario

Bob is a developer building on top of the `ChainlinkEthUsdProxy`. Based on the contract API, he assumes that `latestRoundData` returns the above parameters. He uses them in the code, but they are 0, leading to unintended consequences.

Recommendation

Short term, either change the API of the contract to exclude these parameters, or add developer natspec to the method stating that the four parameters should not be used. This will ensure that developers know the limitations of using this method.

Long-term, document all locations where return values may be different than expected by other developers. This will prevent potentially costly mistakes such as that outlined above.

Go back to Findings Summary

6.17. Initialization functions are inconsistently named

Impact:	Informational	Likelihood:	N/A
Target:	/**/*	Type:	Code maturity

Listing 16. AmmFactory.initialize

```
    function initialize(
        address _ammImplementation,
        address _tokenImplementation,
        ISeriesController _seriesController,
        IAddressesProvider _addressesProvider
    ) external {
        __AmmFactory_init(
            _ammImplementation,
            _tokenImplementation,
            _seriesController,
            _addressesProvider
        );
    }
```

Description

Upgradeable functions in the system contain initialization methods to be able to run in the context of proxy contracts. However, these functions are inconsistently named.

Consider these examples:

- the AmmFactory contains an `initialize` and `__AmmFactory_init` functions (see Listing 16),
- the SeriesVault and SeriesDeployer contain only a `__SeriesVault_init` function
- the MinterAmm contains only an `initialize` function

Recommendation

Short term, ensure consistency in the naming of the initialization functions - this will ensure no surprises for developers building on top the code.

Long term, ensure consistency in all areas of the codebase.

Go back to Findings Summary

6.18. Log old values in logs

Impact:	Informational	Likelihood:	High
Target:	/**/*	Type:	Logging

Listing 17. AmmFactory.sol#L116-L131

```
    /// @notice The owner can update the token implementation address
that will be used for future AMMs
    function updateTokenImplementation(address newTokenImplementation)
        external
        onlyOwner
    {
        require(
            newTokenImplementation != address(0x0),
            "Invalid newTokenImplementation"
        );

        // Update the address
        tokenImplementation = newTokenImplementation;

        // Emit the event
        emit TokenImplementationUpdated(tokenImplementation);
    }
```

Description

When logging important state changes, currently the codebase usually logs only the new value (see Listing 17). This might make incident analysis and other analyses of runtime behavior difficult.

Recommendation

Short term, log old values for very important operations such as updating implementation pointers. This will ensure the most possible information is available for someone analyzing runtime behavior.

Long term, log any values that on-chain and off-chain observers might be interested in. This ensures the maximum transparency of the protocol to its users, developers and other stakeholders.

Go back to Findings Summary

6.19. MinterAmm.updateVolatility's return parameter is never initialized

Impact:	Low	Likelihood:	High
Target:	MinterAmm	Type:	Return parameters

Listing 18. MinterAmm.updateVolatility

```
    /// Each time a trade happens we update the volatility
    function updateVolatility(
        uint64 _seriesId,
        int256 priceImpact,
        uint256 currentIV,
        uint256 vega
    ) internal returns (uint256) {
        int256 newIV = int256(currentIV) + (priceImpact * 1e18) /
int256(vega);

        // TODO: ability to set IV range
        int256 MAX_IV = 4e18; // 400%
        int256 MIN_IV = 5e17; // 50%
        if (newIV > MAX_IV) {
            newIV = MAX_IV;
        } else if (newIV < MIN_IV) {
            newIV = MIN_IV;
        }
        SeriesVolatility storage seriesVolatility = seriesVolatilities[
            _seriesId
        ];
        seriesVolatility.volatility = uint256(newIV);
        seriesVolatility.updatedAt = block.timestamp;
    }
```

Description

`MinterAmm.updateVolatility` declares an anonymous return parameter (see Listing 18). Since there are no return statements in the method, the function will always return 0.

Vulnerability scenario

Alice, a developer building on top of Siren, observes the function signature of `MinterAmm.updateVolatility` and assumes that the function returns some semantically meaningful unsigned integer. However, it returns 0 instead.

Recommendation

Short term, either remove the return parameter or return a value from a function.

Long term, avoid declaring return parameters when there is no way the function returns anything but the default value. This will improve the quality of the code and avoid future bugs.

Go back to Findings Summary

Endnotes

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, ”Report template”, January 22, 2022.

If an individual issue is referenced, please use the following identifier:

`ABCH-{project_identifer}-{finding_number}`,

where `{project_identifier}` for this project is `SIREN-AMM` and `{finding-number}` is the integer corresponding to the section number aligned to three digits. For example, to cite Possibility of re-entrancy, we would use `ABCH-SIREN-AMM-001`.

Appendix B: Glossary of terms

The following terms might be used throughout the document:

Public entrypoint

An `external` or `public` function.

Publicly-accessible function/entrypoint

An `external` or `public` function that can be successfully executed by any network account.

Appendix C: Non-Security-Related Recommendations

C.1. Lack of `emit`

`MinterAmm.provideCapital` contains an event emission without `emit`

Listing 19. MinterAmm.sol#L390-L391

```
            // Emit event
            LpTokensMinted(msg.sender, collateralAmount,
    collateralAmount);
```

Use the `emit` keyword in Listing 19. This will improve readability and automated static analysis of the codebase.

C.2. Dead code

`MinterAmm.WTokensSold` is never used

In MinterAmm, the event `WTokensSold` is defined but never used throughout the codebase.

`SirenExchange._status` is never used

Listing 20. SirenExchange._status

```
    uint256 private _status;
```

SirenExchange defines a uint `_status` (see Listing 20). This state variable is never used throughout the codebase.

C.3. Conditionals

`AmmDataProvider.getPriceForSeriesInternal` contains an unnecessary boolean comparison

Listing 21. AmmDataProvider.sol#L423-L427

```
        if (series.isPutOption == true) {
            return ((put * 1e18) / underlyingPrice);
        } else {
            return ((call * 1e18) / underlyingPrice);
        }
```

AmmDataProvider's `getPriceForSeriesInternal` uses a boolean comparison for control flow. This boolean can be used directly and the boolean comparison removed.

C.4. State variables

`VolatilityOracle.priceOracleAddress` could be made `immutable`

VolatilityOracle has a state variable called `priceOracleAddress`. Since it is assigned to in the constructor and nowhere else, it could be made `immutable`.

Appendix D: Upgradeability

In 6.2, we assumed that logic contracts cannot be selfdestructed and that calling functions on logic contracts before they are initialized is safe. For reference, we are relaxing our assumptions to illustrate how to protect against those threats in a general sense.

For a discussion of how to accomplish (3) of the Requirements, see 6.2.2.

The best way to accomplish both (1) and (2) (while preserving (3)) is to:

1. Ensure that no function on the logic contract can be called until its initialization function is called.
2. Make sure that once the logic contract is constructed, its initialization function cannot be called.
3. Ensure that the initialization function can be called on the Proxy.
4. Ensure that all functions can be called on the Proxy once it has been initialized.

If we are able to accomplish these (and only these) constraints, then the only risk will be the front-running of the initialization function by an attacker; we'll inspect that later.

The initialization function can only currently be called once. Hence the way to accomplish the above (and only the above) constraints is to:

1. Add the `initializer` modifier to the constructor of the logic contract. The constructor will be called on the logic, but not on the proxy contract (see Listing 22)
2. Add a `initialized` storage slot that gets set to `true` on initialization (see Listing 23). Note that we have to define a new variable, since OpenZeppelin's `_initialized` is marked as `private`.

3. Add a require to every non-view public entrypoint in the logic contract that it has been initialized (see Listing 24).

Listing 22. To be added to the logic contract

```
bool public initialized;

constructor() initializer {}
```

Listing 23. To be added to `initialize` *on the logic contract*

```
initialized = true;
```

Listing 24. To be added to every non-view public entrypoint on the logic contract

```
modifier onlyInitialized() {
    require(initialized);
    _;
}
```

In summary, the process would be to:

1. Add a requirement to every non-view public entrypoint that the contract has been initialized.
2. Add a requirement to the initialization function that it cannot be called on the logic contract.

Together, these will accomplish both (1) and (2) of the upgradeability requirements.

Appendix E: Fix Review

On Mar 7, 2022, ABCH reviewed Siren Market's fixes for the issues identified in this report. The fixes were spread across the following pull requests:

Id		PR
3	`SeriesDeployer.autoCreateSeriesAndBuy` contains unchecked `transfers`	#124, #127, #134
5	`MinterAmm.claimAllExpiredTokens` contains a `for` loop with a dynamic condition	#125
7	Missing zero-address checks	#120
11	No return parameter in `SeriesController.setSettlementPrice`	#121
12	`SeriesController.state` doesn't have data validation	#123
16	`ChainlinkEthUsdProxy.latestRoundData` can contain uninitialized return parameters	#138
17	Initialization functions are inconsistently named	#119
19	`MinterAmm.updateVolatility`'s return parameter is never initialized	#122

Table 2. Pull requests for issues

At the time of this writing, all pull requests have been merged into the `v3` branch.

Siren Markets has fixed or partially fixed 8 issues. We reviewed the fixes to ensure they were effective.

Id		Impact	Likelihood	Status
1	Possibility of re-entrancy	High	Medium	Not fixed
2	Pitfalls of upgradeability	Warning	N/A	Not fixed

Id		Impact	Likelihood	Status
3	`SeriesDeployer.autoCreateSeriesAndBuy` contains unchecked `transfers`	High	Medium	Fixed
4	WToken Vault has no access controls	Medium	High	Not fixed
5	`MinterAmm.claimAllExpiredTokens` contains a `for` loop with a dynamic condition	Informational	N/A	Fixed
6	Use `_msgSender` over `msg.sender`	Informational	N/A	Not fixed
7	Missing zero-address checks	High	Low	Partially fixed
8	Contracts used as dependencies don't track upstream changes	High	Low	Not fixed
9	Code layout can be improved	Informational	N/A	Not fixed
10	Use of semantic values as defaults in enums	Warning	Medium	Not fixed
11	No return parameter in `SeriesController.setSettlementPrice`	Warning	High	Fixed
12	`SeriesController.state` doesn't have data validation	Warning	High	Fixed
13	Usage of `solc` optimizer	High	Low	Not fixed
14	System is lacking in documentation	Informational	N/A	Not fixed

Id		Impact	Likelihood	Status
15	OpenZeppelin's upgradeable contracts are used in non-upgradeable contracts	Medium	Low	Not fixed
16	`ChainlinkEthUsdProxy.latestRoundData` can contain uninitialized return parameters	Warning	N/A	Fixed
17	Initialization functions are inconsistently named	Informational	N/A	Fixed
18	Log old values in logs	Informational	High	Not fixed
19	`MinterAmm.updateVolatility`'s return parameter is never initialized	Low	High	Fixed

Table 3. Fix log

E.1. Detailed fix log

Missing zero-address checks

Listing 25. VolatilityOracle.sol#L86-L95

```
    function initialize(
        uint32 _period,
        IAddressesProvider _addressesProvider,
        uint256 _windowInDays
    ) external initializer {
        require(_period > 0, "!_period");
        require(_windowInDays > 0, "!_windowInDays");

        period = _period;
        addressesProvider = _addressesProvider;
```

Partially fixed. At the time of this writing the latest commit on `v3` is 45c6c999c4. As of that commit, there are still many places in the code when zero-address checks are missing (see Listing 25).

Thank You

Ackee Blockchain a.s.

Prague, Czech Republic

hello@ackeeblockchain.com

https://discord.gg/z4KDUbuPxq