PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Internal Report

For Trader Joe Bank

04 October 2021

paladinsec.co

info@paladinsec.co

Table of Contents

Disclaimer

Paladin Blockchain Security ("Paladin") has conducted an independent audit to verify the integrity of and highlight any vulnerabilities or errors, intentional or unintentional, that may be present in the codes that were provided for the scope of this audit. This audit report does not constitute agreement, acceptance or advocation for the Project that was audited, and users relying on this audit report should not consider this as having any merit for financial advice in any shape, form or nature. The contracts audited do not account for any economic developments that may be pursued by the Project in question, and that the veracity of the findings thus presented in this report relate solely to the proficiency, competence, aptitude and discretion of our independent auditors, who make no guarantees nor assurance that the contracts are completely free of exploits, bugs, vulnerabilities or deprecation of technologies. Further, this audit report shall not be disclosed nor transmitted to any persons or parties on any objective, goal or justification without due written assent, acquiescence or approval by Paladin.

All information provided in this report does not constitute financial or investment advice, nor should it be used to signal that any persons reading this report should invest their funds without sufficient individual due diligence regardless of the findings presented in this report. Information is provided 'as is', and Paladin is under no covenant to the completeness, accuracy or solidity of the contracts audited. In no event will Paladin or its partners, employees, agents or parties related to the provision of this audit report be liable to any parties for, or lack thereof, decisions and/or actions with regards to the information provided in this audit report.

Cryptocurrencies and any technologies by extension directly or indirectly related to cryptocurrencies are highly volatile and speculative by nature. All reasonable due diligence and safeguards may yet be insufficient, and users should exercise considerable caution when participating in any shape or form in this nascent industry.

The audit report has made all reasonable attempts to provide clear and articulate recommendations to the Project team with respect to the rectification, amendment and/or revision of any highlighted issues, vulnerabilities or exploits within the contracts provided. It is the sole responsibility of the Project team to sufficiently test and perform checks, ensuring that the contracts are functioning as intended, specifically that the functions therein contained within said contracts have the desired intended effects, functionalities and outcomes of the Project team.

1 Overview

This report has been prepared for Trader Joe's Bank contracts on the Avalanche network. Paladin provides a user-centred examination of the smart contracts to look for vulnerabilities, logic errors or other issues from both an internal and external perspective.

Throughout the Trader Joe Lending audit, Paladin will include deployment tips & recommendations sections for the client to consider. We understand that managing a lending protocol can be difficult and overwhelming so have decided to include these sections to reduce the risk of misconfiguration or important aspects of the system being overlooked.

1.1 Summary

Project Name	Trader Joe
URL	https://traderjoexyz.com
Platform	Avalanche
Language	Solidity

1.2 Contracts Assessed

Name	Contract	Live Code Match
PriceOracle	inherited by PriceOracleProxyUSD	MATCH
PriceOracleProxyUSD	0xe34309613B061545d42c4160ec4d64240b114482	MATCH
SimplePriceOracle	Not used	
V1PriceOracle	Not used	
Joetroller	0x1Ed8368Ca83437DbF43E50e85E6f82342e92CFCB	MATCH
Unitroller	0xdc13687554205E5b89Ac783db14bb5bba4A1eDaC	MATCH
RewardDistributor	0x2274491950B2D6d79b7e69b683b482282ba14885	MATCH
FlashLoanLender	Not used	
Exponential	Library	
SafeMath	Library	
CarefulMath	Library	
ErrorReporter	Library	
JumpRateModelV2	Not used	
TripleSlopeRateModel	0xe6FfD92B9F77FBF5BfEc0F3D9c9d027C4CF3bA6e 0x3C5486b85fAAE29B071F2a616a59cA7bF8F73682 0x82Ea6f7bf853A199AB921137B119B6D41F08038e	MATCH
JAvaxDelegator	0xC22F01ddc8010Ee05574028528614634684EC29e	
JAvaxDelegate	0xd915Fdb10530eF2A8337B4b0bb33F1B0Bc015531	
JUsdcDelegator	0xEd6AaF91a2B084bd594DBd1245be3691F9f637aC	
JUsdcDelegate	0x737Fdfb2365973474beFA244953954c5B6Fddf34	
JUsdtDelegator	0x8b650e26404AC6837539ca96812f0123601E4448	
JUsdtDelegate	0x5f2a43EeB6D624E145F2d7eFEBD13CADe7083Ae6	
JDaiDelegator	0xc988c170d0E38197DC634A45bF00169C7Aa7CA19	
JDaiDelegate	0xF65A0817D7C5b78C97B4265576aFBd9535503d42	
JLinkDelegator	0x585E7bC75089eD111b656faA7aeb1104F5b96c15	

JLinkDelegate	0x6Caf4068ADC5766447205C9e51488586219d51C5
JMimDelegator	0xcE095A9657A02025081E0607c8D8b081c76A75ea
JMimDelegate	0x647Da94Ae8eC35E9627DF11BFFad19513892CF2B
JWethDelegator	0x929f5caB61DFEc79a5431a7734a68D714C4633fa
JWethDelegate	0xEb41c98513ff3f975016b26e16cbf26E2F1B1DF8
JWbtcDelegator	0x3fE38b7b610C0ACD10296fEf69d9b18eB7a9eB1F
JWbtcDelegate	0x3b34E169438FC65Ed1c018655d04e5b0f3185ecc

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	4	2	-	2
Medium	6	1	-	5
Low	3	1	1	1
Informational	30	19	1	10
Total	**43**	**23**	**2**	**18**

Classification of Issues

Severity	Description
High	Exploits, vulnerabilities or errors that will certainly or probabilistically lead towards loss of funds, control, or impairment of the contract and its functions. Issues under this classification are recommended to be fixed with utmost urgency.
Medium	Bugs or issues with that may be subject to exploit, though their impact is somewhat limited. Issues under this classification are recommended to be fixed as soon as possible.
Low	Effects are minimal in isolation and do not pose a significant danger to the project or its users. Issues under this classification are recommended to be fixed nonetheless.
Informational	Consistency, syntax or style best practices. Generally pose a negligible level of risk, if any.

1.3.1 PriceOracle

No issues found.

1.3.2 PriceOracleProxyUSD

ID	Severity	Summary	Status
01	HIGH	The price oracle does not work for ChainLink aggregators with over 18 decimals or underlying assets with over 18 decimals	RESOLVED
02	INFO	Gov privilege: Admin and guardian accounts can change the price aggregators to potentially malicious accounts to manipulate prices and provide bad collateral	ACKNOWLEDGED
03	INFO	`getUnderlyingPrice` does not revert if the oracle price returned by `v1PriceOracle` is 0	RESOLVED
04	INFO	Redundant mappings in PriceOracleProxyUSD	RESOLVED
05	INFO	`CurveSwapInterface`, `YVaultInterface`, `yVaults` and `curveSwap` are unused	RESOLVED

1.3.3 v1PriceOracle

ID	Severity	Summary	Status
06	MEDIUM	Anchoring system could lead to outdated prices if not properly managed	RESOLVED
07	LOW	Usage of block height to keep track of time is not recommended due to the volatile block rate on Avalanche	RESOLVED
08	INFO	`pendingAnchor` will not override the anchor if it is the price update is called in the same period as the previous anchor	RESOLVED
09	INFO	Comments still indicate that the oracle uses ETH as a numeraire	RESOLVED
10	INFO	Outdated comment about an admin role	RESOLVED
11	INFO	Poster role cannot be transferred	RESOLVED
12	INFO	`maxSwing` and `poster` can be made `immutable`	RESOLVED
13	INFO	`_setPendingAnchor`, `_setPaused`, `_setPendingAnchorAdmin`, `_acceptAnchorAdmin`, `getPrice`, `setPrice` and `setPrices` can be made `external`	RESOLVED

1.3.4 SimplePriceOracle

Not within audit scope.

1.3.5 Joetroller

ID	Severity	Summary	Status
14	MEDIUM	Gov privilege: Multiple privileged roles which can modify the behavior of the protocol	ACKNOWLEDGED
15	MEDIUM	Gov privilege: Guardians should probably not be able to increase the borrow and supply caps	ACKNOWLEDGED
16	MEDIUM	Lack of validation on important state variables	ACKNOWLEDGED
17	MEDIUM	Gov privilege: Iron bank allows the admin to give accounts or protocols uncollateralized loans	ACKNOWLEDGED
18	LOW	Supply and borrow cap are initially set to infinity when a new market is added	PARTIAL
19	LOW	Adding too many assets might cause DoS for users that participate in all of them	ACKNOWLEDGED
20	INFO	There are several typographical errors in the contract	RESOLVED

1.3.2 JoetrollerInterface and JoetrollerStorage

ID	Severity	Summary	Status
21	INFO	Inefficient array storage types can lead to running out of gas on certain functions if many tokens are added	ACKNOWLEDGED

1.3.3 Unitroller

ID	Severity	Summary	Status
22	HIGH	Admin has privileges to change the proxy implementation	ACKNOWLEDGED
23	INFO	Unnecessary comparison of `msg.sender` with the zero address	RESOLVED

1.3.4 RewardDistributor

ID	Severity	Summary	Status
24	MEDIUM	`_grantJoe` allows admin to transfer any JOE token amount to an arbitrary address	ACKNOWLEDGED
25	INFO	AVAX rewards cannot be claimed by a supplier/borrower that is a contract without a fallback function	ACKNOWLEDGED
26	INFO	`setJoeAddress`, `setJoetroller`, `setAdmin` and `claimReward` does not emit events	ACKNOWLEDGED
27	INFO	RewardDistributor will stop working in the year 2106 due to overflow reverting	ACKNOWLEDGED

1.3.5 FlashLoanLender

No issues found.

1.3.6 Exponential

ID	Severity	Summary	Status
28	INFO	Inconsistent rounding behavior between `mulExp` and `mul_`: `mulExp` rounds to the nearest integer while `mul_` does not	ACKNOWLEDGED
29	INFO	Gas optimization: Unnecessary memory allocation for error messages in `add_`, `sub_`, `mul_` and `div_` functions	ACKNOWLEDGED
30	INFO	Gas optimization: No need to handle error pro-actively in certain functions	ACKNOWLEDGED

1.3.7 SafeMath

ID	Severity	Summary	Status
31	INFO	Gas optimization: Unnecessary memory allocation for error messages in `add_`, `sub_`, `mul_` and `div_` functions	ACKNOWLEDGED

1.3.8 CarefulMath

No issues found.

1.3.9 ErrorReporter

No issues found.

1.3.10 JumpRateModelV2

ID	Severity	Summary	Status
32	INFO	`accrueInterest()` is not called before the interests rate model is manually adjusted by governance causing the adjustment to work slightly in retrospect	RESOLVED
33	INFO	Lack of validation within `updateJumpRateModelInternal`	RESOLVED
34	INFO	Comments and variables still indicate that the model is based in block height time	RESOLVED

1.3.11 TripleSlopeRateModel

ID	Severity	Summary	Status
35	INFO	`accrueInterest()` is not called before the interests rate model is manually adjusted by governance causing the adjustment to work slightly in retrospect	PARTIAL
36	INFO	Lack of validation within `updateTripleRateModelInternal`	ACKNOWLEDGED
37	INFO	Comments and variables still indicate that the model is based in block height time	RESOLVED

1.3.12 JToken

ID	Severity	Summary	Status
38	HIGH	Admin holds privileges to change state of JTokens	ACKNOWLEDGED
39	HIGH	Reentrancy is possible from one JToken to another JToken	RESOLVED
40	INFO	Token transfers do not revert if the balance is insufficient	RESOLVED

1.3.13 Token Implementations: JCollateralCapERC20

ID	Severity	Summary	Status
41	INFO	`updateJTokenVersion` is only called when a token is upgraded to the CollateralCap version but this is not on unset on downgrades	RESOLVED
42	INFO	Unnecessary arithmetic in `increaseUserCollateralInternal`	RESOLVED

1.3.14 Token Implementations: JWrappedNative

ID	Severity	Summary	Status
43	INFO	`updateJTokenVersion` is only called when a token is upgraded to the JWrappedNative version but this is not on unset on downgrades	RESOLVED

1.3.15 Token Implementations: JErc20

No issues found.

2 Oracle

Operational tips & recommendations

The Price Oracle is responsible for providing each asset with a price. In the case of Trader Joe, the PriceOracleProxyUSD contract uses ChainLink to derive a USD price for relevant assets. There are a few important things to note however.

The precision requirements of the prices are unique and must be considered when manually overriding prices

From time to time, the admin might need to override a price using the v1PriceOracle if there is an issue with ChainLink on said asset. However, this should be done with utmost care as the precision mechanism (the number of decimals used in the mantissa) of the PriceOracle is quite unique.

The basic requirement for the actual price is as follows:

```
valueUSD = amountToken * getUnderlyingPrice(jToken) / 1e18
```

The special requirement is that `valueUSD` must always be a decimal with a mantissa of exactly 18 digits. This means that $1 is always 1e18 in said notation. In other words, if `amountToken` has 17 decimals, then `getUnderlyingPrice` must be 10 times larger than when it would have 18 decimals.

Let us quickly go over an example: say we want to value 100 USDC with 6 decimals, than `amountToken` would be `100 *1e6`, the `underlyingPrice` of USDC must thus be `1*1e18*1e12`, or more generally `1 * 1e18 * 1e(18-decimals)`.
When setting the price manually for assets, especially when the number of decimals is not 18, the client should carefully consider the amount of decimal digits to include. If this is not considered, they may accidentally set the price orders of magnitude higher or lower than the actual price. To avoid this risk in the first place,

setting prices manually should be avoided. Furthermore, we recommend the client to setup a staging environment where they always test such overrides first.

Governance privileges

Being able to manually override prices can be used maliciously to extract a lot of money. First, doing so could be used to forcefully liquidate accounts but more importantly, this could be used to borrow assets while only providing a fraction of the real value as collateral. We recommend the client to seriously consider this privilege and try to safeguard it as much as possible. If private keys are ever stolen or the project team turns malicious, the fact that these prices could change should at least be detectable by the community. We thus recommend using at least a timelock for the admin account; however, multisig configurations or more advanced governance configurations are the more desired solution. We recommend to clearly state these safeguards in the protocol's documentation to allow the community to monitor these accounts and inform each other in the unlikely event that a manual mispricing does occur.

The oracle allows using aggregators with ETH as base

The oracle allows for aggregators that use ETH as the base unit (the numeraire). However, since this protocol is deployed on Avalanche, the operations team might accidentally use aggregators with AVAX as the base. It might be desirable to add AVAX as an explicit numeraire option alongside ETH and USD as to avoid confusion internally. Furthermore, we recommend clearly documenting the steps that should be taken to add a new aggregator with the different bases. It would be highly unfortunate if ever an asset is added with an AVAX-based pricing while the oracle then converts it as if it was priced in ETH.

2.1 PriceOracle

The PriceOracle is a simple interface contract that exposes one single function, `getUnderlyingPrice`. The purpose of the Price Oracle contracts is to provide the real world price for each asset. The rest of the system then uses this price to calculate collateral requirements and liquidation variables. It is thus one of the essential components of any lending system.

Given that it is so essential, it is also thus necessary from a security point of view that the price oracle cannot be manipulated, will remain available ,and will remain accurate. Furthermore, the governance should not be able to quickly manipulate the oracle for their own benefit as this could pose a risk if their private keys ever get stolen.

During the first iterations of lending protocols, often the price of the AMM pair was used to price assets. However, this approach had several fallbacks: first of all, this price can be easily manipulated at a low cost through flash-loans. Second of all, the on-chain liquidity is often only a fraction of the total liquidity which could cause prices to diverge if there is no active bridging possible (eg. bridges go offline) and thus the price does not exactly reflect the real world price. Relying on such on-chain prices therefore contains risk. Hence, any significant protocol, including Joe Lending, will opt for protocols like ChainLink to provide pricing information. ChainLink provides a volume-weighted average of the price across multiple exchanges which makes it more robust. It also has a mechanism where the prices are not generated by a single party, but instead are taken from a set of partially trusted parties to reduce centralisation risk.

```
function getUnderlyingPrice(CToken cToken) external view returns
(uint256);
```

The `getUnderlyingPrice` function takes the CToken as an argument and returns the scaled underlying price (1e18 places).

2.1.2 Issues & Recommendations

No issues found.

2.2 PriceOracleProxyUSD

It should be noted that the `ethUsdAggregator` parameter should provide the Ethereum price in USD and not the other way around.

2.2.1 Privileged Operations

The following functions can be called by the owner of the contract:

- `_setGuardian`
- `_setAdmin`
- `_setAggregators`

2.2.2 Issues & Recommendations

Issue #01	**The price oracle does not work for ChainLink aggregators with over 18 decimals or underlying assets with over 18 decimals**
Severity	HIGH SEVERITY

Location	Line 109
	`return mul_(price, 10**(18 - underlyingDecimals));`
	
	Line 127
	`return mul_(uint(price), 10**(18 - uint(aggregator.decimals())));`

Description	If a ChainLink aggregator is ever added with a precision higher than 18 decimal places, a severe underflow will occur that results in either a wildly inaccurate price or reversion.

Recommendation	Consider adding an if-else clause to handle the case where the aggregator has over 18 decimal places.
	
	`if (underlyingDecimals <= 18) {`
	`  return mul_(price, 10**(18 - underlyingDecimals));`
	`}else {`
	`  return div_(price, 10**(underlyingDecimals - 18)));`
	`}`

Resolution	RESOLVED The client has added logic to handle both cases similar to the logic in the recommendation.

Issue #02	**Gov privilege: Admin and guardian accounts can change the price aggregators to potentially malicious accounts to manipulate prices and provide bad collateral**
Severity	INFORMATIONAL
Description	If governance turns malicious or if their private keys are stolen, they can make substantial profit through setting some of the asset prices severely high and providing collateral to them so that they can loan all the other assets. It should be noted that the guardian wallet can only clear the configuration, which means that they can only make the PriceOracleProxyUSD fallback to the v1PriceOracle implementation and not update it to a malicious contract to directly manipulate the price.
Recommendation	In the long term, consider devising a governance structure that makes it difficult for individual keys to do damage to the protocol, for example, a multisig or DAO. In the short term, consider putting both the admin and guardian account behind a sufficiently long timelock which can ideally be vetoed by a trusted third-party.
Resolution	ACKNOWLEDGED The client has indicated that these privileges will be put behind a timelock. This issue can be marked as Resolved at the request of the client once they have actually deployed this timelock with a sufficiently long delay or have setup another reasonable governance structure.

Issue #03	**`getUnderlyingPrice` does not revert if the oracle price returned by v1PriceOracle is 0**
Severity	INFORMATIONAL

Description	For the prices provided by the ChainLink oracle, there is a check in `getPriceFromChainlink` that reverts if the price is 0 or lesser.
	
	`( , int price, , , ) = aggregator.latestRoundData();`
	`require(price > 0, "invalid price");`
	
	However, if the price by v1PriceOracle is used, `getUnderlyingPrice` will return the exact value returned by v1PriceOracle. In the case where v1PriceOracle is paused, has the asset price manually set to 0 or an inexistant asset, the price will be 0. This results in a slight behavior difference between the checks in place for the price from ChainLink and v1PriceOracle.
	
	All usages of the `getUnderlyingPrice` function in the Comptroller do check and return a price error if the `oraclePriceMantissa` is 0, but future code changes or additions could have some unintended consequences if the function is called but the zero check is not done.
	
	It should be noted that it might be desired behavior to give governance the freedom to enter this error state for certain variables by setting the price to zero. However, the zero price might not be appropriate for addresses that have actually hit zero value.

Recommendation	If the intended behavior is for a revert to occur in the PriceOracleProxyUSD contract if the price is 0, add a check to ensure that the price returned by v1PriceOracle is greater than 0.
Resolution	RESOLVED The check has been added.

Issue #04	Redundant mappings in PriceOracleProxyUSD
Severity	INFORMATIONAL

Description	The following mappings appear to be redundant in PriceOracleProxyUSD, as they are completely unused.
	`Line 72-...`
	`// @notice Mapping of crToken to y-vault token`
	`mapping(address => address) public yVaults;`
	`// @notice Mapping of crToken to curve swap`
	`mapping(address => address) public curveSwap;`
	Furthermore, the CurveSwapInterface and YVaultInterface included in the same contract file are not used and redundant as well.

Recommendation	The redundant mappings can be removed. In addition, the CurveSwapInterface and YVaultInterface can be removed as well.
Resolution	RESOLVED The redundant mapping has been removed.

Issue #05	`CurveSwapInterface`, `YVaultInterface`, `yVaults` and `curveSwap` are unused
Severity	INFORMATIONAL
Description	The code contains variables and interfaces which are not used. This could make reviewing the code more difficult for third-parties as it unnecessarily complicates the codebase.
Recommendation	Consider removing the unused code sections.
Resolution	RESOLVED The unused sections have been removed.

2.3 v1PriceOracle

The v1PriceOracle is a simple, centralised oracle. This means that the oracle owner – the client – is supposed to manually set the prices periodically for all relevant assets.

The oracle has a few safeguards: It has a simple "anchor" mechanism which ensures that from period-to-period, prices can change by at most 10%. A period is defined as a number of blocks (set to 1 hour according to the comments). However, this functionality can be overridden by the administration to re-anchor the price in case a sudden move larger than 10% occurs. Until it is manually overridden, no price updates occur which could make the system vulnerable since an outdated price could allow for arbitrage. Each time a period expires, the next price will be the new anchor for the subsequent period as long as it is within a 10% range of the previous anchor, otherwise the price is again stuck at the previous anchor.

The CarefulMath and Exponential libraries are directly embedded in this oracle, presumably so the Error enum can overlap. However, since the code is highly similar, we refer to the sections of these libraries for the issues on them. If the client resolves any of the issues there, these issues should likely be ported into the copies within this contract as well.

2.3.1 Privileged Operations

The following functions can be called by the owner of the contract:

- `_setPendingAnchor`
- `_setPaused`
- `_setPendingAnchorAdmin`
- `_acceptAnchorAdmin`
- `setPrice`
- `setPrices`

2.3.2 Issues & Recommendations

Issue #06	**Anchoring system could lead to outdated prices if not properly managed**
Severity	MEDIUM SEVERITY
Description	The `v1PriceOracle` periodically choses a price point as an anchor, and this is done about once every hour. However, if the price changes by more then 10% during this period, the price could get stuck around the anchor until the `anchorAdmin` manually updates it with `_setPendingAnchor`. In addition, as described in a later issue, this update will only properly update the anchor if the present period has passed. This all increases the likelihood of outdated prices being present which could be very detrimental to the quality of the collateral.
Recommendation	If the `v1PriceOracle` is used, consider explicitly automating the re-anchoring mechanism to ensure prices are updated again, furthermore consider having a small period timestamp as to not keep the stale price present for too long. Finally, consider not allowing `v1PriceOracle` tokens to be used as collateral or setting a very low collateral factor and limit on these assets.
Resolution	RESOLVED `v1PriceOracle` will not be used.

Issue #07	**Usage of block height to keep track of time is not recommended due to the volatile block rate on Avalanche**
Severity	LOW SEVERITY
Description	The `v1PriceOracle` periodically choses a price point as an anchor, and this is done about once every hour. However, this hour is currently defined as 240 blocks. This is firstly not the average block rate on Avalanche, and secondly, Avalanche has stated that there is no such average block rate and that developers should rely on the block timestamps instead. Not doing so could be considered risky in certain situations, as the block rate is manipulatable at a relatively low cost.
Recommendation	Consider moving the `numBlocksPerPeriod` variable to a `numSecondsPerPeriod` variable. All relevant functionality should then be adjusted to work with timestamps.
Resolution	RESOLVED `v1PriceOracle` will not be used.

Issue #08	**pendingAnchor will not override the anchor if it is the price update is called in the same period as the previous anchor**
Severity	INFORMATIONAL
Description	The governance of the contract can manually update the anchor variable, which is used as a price safeguard if it is ever stuck since the price moved more than 10%. However, due to the current design of the protocol, even though a single price change will be accepted, the anchor will not be re-anchored if it was called in the same period as the previous anchoring. We believe this might be desired behavior since moving the anchor manually might only be necessary if the anchor is in fact stuck, that means it has not changed for a few periods. However, it does not make sense to us that this behavior is implicit and that a price change is still allowed to happen without a re-anchoring.

Recommendation	Consider whether this behavior is intentional, if not, consider always re-anchoring when a `pendingAnchor` is set.
	
	`if (localVars.currentPeriod > localVars.anchorPeriod || pendingAnchor != address(0)) { // every period we set the current price as the new anchor`
	`anchors[asset] = Anchor({period: localVars.currentPeriod, priceMantissa: localVars.price.mantissa});`
	`}`
	
	`// Set pendingAnchor = Nothing`
	`// Pending anchor is only used once.`
	`if (pendingAnchors[asset] != 0) {`
	`pendingAnchors[asset] = 0;`
	`}`

Resolution	RESOLVED `v1PriceOracle` will not be used.

Issue #09	Comments still indicate that the oracle uses ETH as a numeraire
Severity	INFORMATIONAL

Location	Line 397
	`@dev Mapping of asset addresses and their corresponding price in terms of Eth-Wei`
	
	Line 497
	`// Price in ETH, scaled by 10**18`

Description	While the PriceOracleProxyUSD actually uses USD based prices, the comments in the v1PriceOracle indicate that it is used for ETH-based prices.
Recommendation	If any existing tooling is used to call the oracle, consider carefully inspecting the code to ensure it is called using USD as numeraire. Furthermore, consider adjusting the comments to indicate that the numeraire does not have to be ETH and will be USD in this implementation.
Resolution	RESOLVED `v1PriceOracle` will not be used.

Issue #10	Outdated comment about an admin role
Severity	INFORMATIONAL
Location	Line 527 `// Check caller = anchorAdmin. Note: Deliberately not allowing admin. They can just change anchorAdmin if desired.`
Description	Line 527 of the contracts contains a comment saying that the "admin" role is deliberately not allowed, however, such a role is not present in the contract thus we believe this comment might have become stale.
Recommendation	Consider explaining what is meant with the admin role in this comment or removing it.
Resolution	RESOLVED `v1PriceOracle` will not be used.

Issue #11	Poster role cannot be transferred
Severity	INFORMATIONAL
Description	The poster within the v1PriceOracle is responsible for updating prices within the v1PriceORacle and is thus the wallet that calls the `setPrices` method. However, this privilege can not be transferred to a new wallet which means if having prices be set by an EOA is ever deemed to insecure, this privilege cannot be transferred to a more secure smart contract.
Recommendation	Consider adding `setPendingPoster` and `acceptPoster` methods.
Resolution	RESOLVED `v1PriceOracle` will not be used.

Issue #12	`maxSwing` and `poster` can be made immutable
Severity	INFORMATIONAL
Description	Variables that are only set in the constructor but never modified can be indicated as such with the `immutable` keyword. This is considered best practice since it makes the code more accessible for third-party reviewers.
Recommendation	Consider making the above variables explicitly `immutable`.
Resolution	RESOLVED `v1PriceOracle` will not be used.

Issue #13	**`_setPendingAnchor`, `_setPaused`, `_setPendingAnchorAdmin`, `_acceptAnchorAdmin`, `getPrice`, `setPrice` and `setPrices` can be made `external`**
Severity	INFORMATIONAL
Description	The contract contains functions that can be changed from `public` to `external`. Apart from being a best practice when the function is not used within the contract, this can lead to a lower gas usage in certain cases.
Recommendation	Consider making the above functions `external`.
Resolution	RESOLVED `v1PriceOracle` will not be used.

2.4 SimplePriceOracle

Although the SimplePriceOracle is not going to be included in Joe Lending, it is the most simple implementation of a Price Oracle and is useful for testing and understanding the mechanism. The SimplePriceOracle is extremely centralised as it simply allows the owner to set prices for assets through a function, then `getUnderlyingPrice` will return these prices. It should be noted that FTM is seen as the numeraire in this oracle, thus the price of FTM is 1. Since the source code was forked from Cream FTM and is not used, this numeraire was not adjusted.

This contract is excluded from the audit scope since it is not adjusted to Avalanche and is clearly a testing contract. It should absolutely not be used in any way, shape or form in production since anyone can set prices.

2.4.1 Issues & Recommendations

Not included in audit scope.

3 Joetroller

Operational tips & recommendations

Number of markets

Since the `getHypotheticalAccountLiquidityInternal` function has to loop over all assets a user participates in and then do various calls (including to ChainLink for every asset) and calculations, user transactions might start hitting gas limit issues if the user participates in too many markets. The client should be careful with adding new markets and should thoroughly test how many markets can be safely added without any issues.

As recommended in the relevant issue, it might be desirable for the client to simply implement safeguards that prevent users from entering in too many markets (for example, 20). This way, the total number of markets can still exceed this number greatly without the risk of locking users out of their account.

Governance privilege: Iron Bank

The iron bank features of Cream, or more specifically the feature allowing certain accounts to borrow without collateral, can be very valuable and economically efficient, but might also pose risks on the protocol when not carefully managed or when there is no governance. Currently, the admin is able to give any wallet a limit to borrow without collateral through the `setCreditLimit` function. If the admin is not thoroughly secured (such as having a timelock, multisig and/or governance), this might pose a significant risk to the protocol.

If the iron bank feature is ever put to use, the governance should first of all ensure that the admin account is again properly secured but also that no individual protocol or lender can loan a major amount. Ideally, Trader Joe should operate an insurance treasury that can rebalance the system if one or two ever protocols fail. A good rule of thumb could be to maintain a treasury large enough to cover two

failing protocols. In other words, the iron bank credit of an individual lender should not exceed half the treasury allocated to insuring this feature.

Quality of collateral

Collateral is what is used as insurance that a loan against the protocol will eventually be paid back. If a user wants to borrow USDC, they can do so but only if they supply a greater amount of another token to the protocol. This way, the protocol can maintain a positive balance sheet for each borrower. However, if this other token that is supplied is of low quality, it might not provide much insurance. An example of a low quality token is if the price of the token can be easily manipulated due to a high supply but low liquidity. Thus, when assets are added, the client should first and foremost set the supply, borrow and collateral caps of this asset to values that reflect the liquidity of these tokens. If it takes only $100,000 to triple the price of this asset, this is not a suitable asset for collateral and most definitely not for uncapped collateral.

Furthermore, the caps should also be adjusted to the odds and magnitudes of black-swan events, especially when these events are not correlated with the health of the protocol itself. An example of this would be a relatively new network token where the governance has minting privileges. Even though this token might have $10 million in liquidity as an example, it might not be suitable collateral since there is a non-negligible risk that the token might drop to zero in a single moment if the governance ever abuses their minting privilege.

In general, worthless collateral is something which the project team will eventually have to deal with as at the end of the day, managing a lending protocol comes down to risk management when more collateral and tokens started to get incorporated into the protocol. Apart from limiting the damages of this risk through cap management, the client should consider building up a significant treasury through the reserves of the protocol (an example target could be $1 million in reserves). These reserves should be used to mitigate events that create worthless collateral to keep the system healthy. One way lending protocols often collect reserves quickly is by adding high interest rates to lower quality tokens while these tokens are not

enabled as collateral. Inspiration can be taken from Cream Finance which has set the reserve factor to up to 40% on less quality assets. Although this might make the market less efficient, it might be a valuable approach in the long term when the reserves are kept to safeguard the system.

Market can become illiquid

Due to the way the incentives are structured, it is likely that the market can never reach full liquidity again as certain loans will just be too undercollateralized or too small to be worth liquidating. As there are no mechanisms in the system to rebalance this slow drift towards a less liquid market, we recommend the governance to maintain a strong treasury, as we have previously recommended, in case illiquidity is ever reached. When this happens, governance can supply assets to resolve the issue.

3.1 Joetroller

The Joetroller is the heart of the Trader Joe lending system. Every operation that happens within the system requires explicit permission by the Joetroller before it occurs, and an explicit validation after it occurred. The Joetroller thus ensures that no actions can be taken that would leave a user undercollateralized and that only liquidations can occur if the user is undercollateralized. In addition, the Joetroller is the entry point and manager of the reward contract.

Finally, the protocol governance can tweak many essential variables within the Joetroller and thus all interactions with it should be carefully considered and validated by the community.

If `_setRewardDistributor` is called and the distributor contract is changed, users will still be able to interact with RewardDistributor to claim any pending rewards for supply or borrow. Even after the `rewardDistributor` address in Joetroller has been changed, the old `rewardDistributor` will still be actively updating the reward emissions as `updateRewardSupplyIndex` and `updateRewardBorrowIndex` calculates the reward indexes based on the delta between the current timestamp and previous update timestamp, the `supplySpeed` and the total supply of the JToken.

To stop reward emissions, the admin of the `rewardDistributor` contract has to call `_setRewardSpeed` for a JToken, which will call `updateRewardSupplyIndex` and `updateRewardBorrowIndex` to update the indexes up to the latest timestamp.

3.1.1 Privileged Operations

The following functions can be called by the owner of the contract:

- `_setRewardDistributor [ Dangerous ! ]`
- `_setPriceOracle [ Dangerous! ]`
- `_setCollateralFactor [ Dangerous ! ]`
- `_setLiquidationIncentive [ Dangerous ! ]`
- `_supportMarket [ Dangerous ! ]`
- `_setCreditLimit [ Dangerous !]`
- `_delistMarket`
- `_setSupplyCapGuardian`
- `_stMarketSupplyCaps`
- `_setMarketBorrowCaps`
- `_setBorrowCapGuardian`
- `_setPauseGuardian`
- `_setMintPaused`
- `_stBorrowPaused`
- `_setFlashloanPaused`
- `_setTransferPaused`
- `_setSizePaused`
- `_become`

3.1.2 Issues & Recommendations

Issue #14	**Gov privilege: Multiple privileged roles which can modify the behavior of the protocol**
Severity	MEDIUM SEVERITY

Description	The following are the privileged roles and their capabilities in the JoeTroller:
	
	`Admin`
	`_setPriceOracle`
	`_setCloseFactor`
	`_setCollateralFactor`
	`_setLiquidationIncentive`
	`_supportMarket`
	`_delistMarket`
	`_setSupplyCapGuardian`
	`_setMarketSupplyCaps`
	`_setMarketBorrowCaps`
	`_setBorrowCapGuardian`
	`_setPauseGuardian`
	`_setLiquidityMining`
	`_setMintPaused`
	`_setBorrowPaused`
	`_setFlashloanPaused`
	`_setTransferPaused`
	`_setSeizePaused`
	`_setCreditLimit` (IronBank)
	
	`borrowCapGuardian`
	`_setMarketBorrowCaps`
	
	`supplyCapGuardian`
	`_setMarketSupplyCaps`
	
	`pauseGuardian`
	`_setMintPaused` - only pause
	`_setBorrowPaused` - only pause
	`_setFlashloanPaused` - only pause
	`_setTransferPaused` - only pause
	`_setSeizePaused` - only pause

Recommendation	Consider how these roles will be securely managed. The leakage or loss of the private key of any of these roles can result in abuse of privileges and a compromise in the protocol. It is recommended to at least use a multisig solution for the guardian roles, and a timelock contract with a reasonable delay as the admin. The timelock contract should also be owned by a multisig solution.
Resolution	ACKNOWLEDGED The client has indicated that these privileges will be put behind a timelock owned by a multisig. This issue can be resolved once we confirm that this has actually been added in the deployed contracts at the request of the client.

Variable	Setter	Explanation	Recommended limit
`closeFactor`	`_setCloseFactor`	The percentage of the position that is allowed to be closed on liquidation	[0, 1e18]
`liquidationIncentiveMantissa`	`_setLiquidationIncentive`	The liquidation incentive is how much of the liquidation should go to the liquidator.	[0, 0.5 * 1e18]

Issue #15 Gov privilege: Guardians should probably not be able to increase the borrow and supply caps

Severity	MEDIUM SEVERITY
Description	Throughout the Joetroller, guardians are used to decrease functionality (eg. they can pause functionality but the admin is necessary to unpause). However, for the supply and borrow caps, the guardian can increase them as well which might be an excessive privilege for this account. Allowing a less trusted account to increase the caps could lead to abuse by this account if they increase the supply or borrow cap for an asset that is less trusted.
Recommendation	Consider only allowing guardians to reduce the supply and borrow cap.
Resolution	ACKNOWLEDGED

Issue #16	Lack of validation on important state variables
Severity	MEDIUM SEVERITY
Description	The contract does not validate that important governance variables are set within reasonable bounds. Although there is already a high degree of governance power to the point that each action should be considered extremely carefully, it might still be good to add safeguards on variables which should never be set to unreasonable values. Such variables could be:
Recommendation	Consider carefully considering if the variable should ever exceed the recommended limit. If not, consider applying said limit.
Resolution	ACKNOWLEDGED

Issue #17	Gov privilege: Iron bank allows the admin to give accounts or protocols uncollateralized loans
Severity	MEDIUM SEVERITY
Description	The `_setCreditLimit` allows the Joetroller admin to give an account credit limit for borrowing loans without any collateral provided. If misused, it allows such accounts to take out funds from the protocol without the need to return these funds.
Recommendation	Consider implementing a thorough governance setup with a timelock, multisig and/or DAO to manage governance privileges over the protocol. Furthermore, consider implementing a thorough treasury setup to reimburse losses in case an iron-bank protocol fails. Individual protocols should not be given allocations larger than what Trader Joe can reimburse.
Resolution	ACKNOWLEDGED The client has indicated that these privileges will be put behind a timelock owned by a multisig. This issue can be resolved once we confirm that this has actually been added in the deployed contracts at the request of the client.

Issue #18	**Supply and borrow cap are initially set to infinity when a new market is added**
Severity	LOW SEVERITY
Description	When a market is added, the supply and borrow cap of the token are not initialized which leaves them at "0". This zero value is interpreted by the contract that there is no limit on what can be supplied and borrowed. Oftentimes, the initial hours after a deployment are the most uncertain and the caps might be the most useful in this period. Furthermore if a bad actor sees a token deployment incoming, they might abuse this uncapped period if there is some way to manipulate the price or they realize the fact that this token is highly illiquid. `increaseUserCollateralInternal` will be called even if the collateral factor is initially 0. For example, before the collateral cap is set, if the `totalCollateralTokens` for a JToken is 1000, and the new collateral cap is 900, `increaseUserCollateralInternal` will not increase a user or the total collateral tokens and return 0. The existing `totalCollateralTokens` would still be greater than the new collateral cap. For the supply cap, the existing total supply amount would still be greater than the new supply cap set, and no one can supply any more of the underlying token until the total supply amount goes below the new supply cap. Similarly, for the borrow cap, the existing total borrow amount would still be greater than the new borrow cap set, and no one can borrow any more of the underlying token until the total borrow amount goes below the new borrow cap. In short, existing collateral, supplies and borrows would still be considered valid even if they exceed the caps, as long as they are done before the change to the cap values.
Recommendation	Consider adding an initial supply and borrowing cap to new markets.
Resolution	PARTIALLY RESOLVED The client will take this behavior in consideration.

Issue #19	Adding too many assets might cause DoS for users that participate in all of them
Severity	LOW SEVERITY
Description	The `getHypotheticalAccountLiquidityInternal` function is an essential function within the contract. Furthermore, it loops over and queries ChainLink for every asset the user is participating in. This means if a user participates in a very large number of markets (eg. supplies and borrows many tokens simultaneously), they could lock themselves out of their account if `getHypotheticalAccountLiquidityInternal` ever starts hitting the block limit. Furthermore, on many RPCs there's an even lower limit than the block limit imposed on transactions, 1 AVAX token, which might be more easily reached.
Recommendation	In the short term, consider testing at which point the number of assets becomes problematic and adding a per-wallet maximum. The total number of markets can be unbound, especially when an EnumerableSet is used for the underlying data structures as is recommended in the JoetrollerInterface section. This requirement can be added to `addToMarketInternal` which would then return a new error (make sure to add this error at the end of the error list however, otherwise the error indices might cause a forked frontend to mess up).
Resolution	ACKNOWLEDGED

Issue #20	There are several typographical errors in the contract
Severity	INFORMATIONAL

Description	The contract contains the following typographic errors.
	
	Line 240
	`@notice Return a specific market is listed or not`
	
	This sentence should start with "Return whether"
	
	Line 484
	`@param borrower The account which would borrowed the asset`
	
	This should be the account which would have borrowed the asset.
	
	Line 919
	`// Unlike joeound protocol, getUnderlyingPrice is relatively expensive because we use ChainLink as our primary price feed.`
	
	Due to automatic text replacement, compound has been replaced with joeound while the reference is in fact still to compound.

Recommendation	Consider fixing the typographical mistakes.
Resolution	RESOLVED

3.2 JoetrollerInterface and JoetrollerStorage

Since the Joetroller is upgradeable, the variables are stored in a separate dependency to easily maintain the order of them. The main Joetroller contract inherits from this contract so it is as if these variables are directly saved there.

3.2.1 Issues & Recommendations

Issue #21	**Inefficient array storage types can lead to running out of gas on certain functions if many tokens are added**
Severity	INFORMATIONAL
Description	The JoetrollerInterface uses an array storage type for the allMarkets and accountAssets variable which is inefficient on delete and contains calls. This is seen through the many unbound for-loops in the Joetroller. Although there should be a limit on the amount of markets a user can participate in since this needs to be iterated over during the operations of the contract, the number of tokens/markets that are added in general should not have to be limited.
Recommendation	In case many markets will be added (hundreds), consider using OpenZeppelin's EnumerableSet instead for these arrays. This data structure allows for O(1) complexity of `delete` and `contains` calls.
Resolution	ACKNOWLEDGED

3.3 Unitroller

The unitroller is the upgradeable proxy that delegates calls to the Joetroller. All state is thus stored in the Unitroller.

3.3.1 Privileged Operations

The following functions can be called by the owner of the contract:

- `_setPendingImplementation`
- `_acceptImplementation`
- `_setPendingAdmin`
- `_acceptAdmin`

3.3.2 Issues & Recommendations

Issue #22	Admin has privileges to change the proxy implementation
Severity	HIGH SEVERITY

Description	The admin holds the privileged role of being able to change the implementation of the Unitroller. By doing so, the code for the Joetroller can be arbitrarily changed, and can result in unexpected behavior. This is a very powerful but intended privilege as upgradability allows further development of the protocol without having users to migrate, but it will affect the whole protocol's behavior.
	
	The following is the flow of how the Unitroller would change the implementation:
	
	`Unitroller._setPendingImplementation("{new implementation address}")`
	`Joetroller._become("{new implementation address}")`

Recommendation	It is recommended that the admin address should be a timelock with a reasonable delay, which is owned by a multisig solution.
Resolution	ACKNOWLEDGED The client has indicated that these privileges will be put behind a timelock owned by a multisig. This issue can be resolved once we confirm that this has actually been added in the deployed contracts at the request of the client.

Issue #23	Unnecessary comparison of `msg.sender` with the zero address
Severity	INFORMATIONAL
Location	Line 110 `if (msg.sender != pendingAdmin || msg.sender == address(0)) {`
Description	There is a conditional check `if msg.sender == address(0)`, which is unnecessary.
Recommendation	The unnecessary conditional can be removed.
Resolution	RESOLVED The recommendation has been implemented.

3.4 RewardDistributor

The RewardDistributor contract holds the rewards in the form of AVAX and JOE tokens. The functions to update the supply and borrow rewards for an address, as well as the overall rewards emission, can only be called by the admin or Joetroller, while rewards can be claimed by the claimReward function. Claiming the rewards can be done on the behalf of the supplier borrower, and the rewards will be sent to the rightful supplier/borrower address.

3.4.1 Privileged Operations

The following functions can be called by the owner of the contract:

- `_setRewardSpeed`
- `updateAndDistributeSupplierRewardsForToken`
- `updateAndDistributeBorrowerRewardsForToken`
- `_grantJoe`
- `setJoeAddress`
- `setJoetroller`
- `setAdmin`

3.4.2 Issues & Recommendations

Issue #24	**_grantJoe allows admin to transfer any JOE token amount to an arbitrary address**
Severity	MEDIUM SEVERITY
Description	_grantJoe allows the admin or Joetroller (which does not have any functionality to call this function) to any address as long as the RewardDistributor has sufficient balance to fulfill the amount.
Recommendation	It is recommended that the admin address should be a timelock with a reasonable delay, which is owned by a multisig solution.
Resolution	ACKNOWLEDGED The client has indicated that these privileges will be put behind a timelock owned by a multisig. This issue can be resolved once we confirm that this has actually been added in the deployed contracts at the request of the client.

Issue #25	**AVAX rewards cannot be claimed by a supplier/borrower that is a contract without a fallback function**
Severity	INFORMATIONAL
Description	As the protocol does not limit smart contracts from interaction, a smart contract such as a vault would be able to supply and borrow, and in turn, earn rewards from the protocol. As one of the rewards is in AVAX native tokens, the `transfer` function call to transfer the AVAX rewards would not succeed if the address claiming is a contract but did not include a fallback function as such: `receive() external payable {}`. Additionally, as the `transfer` function is used here, and there is no guarantee that gas costs are always constant, and this could cause possible issues as `transfer` uses a hardcoded gas amount. https://consensys.net/diligence/blog/2019/09/stop-using-soliditys-transfer-now/
Recommendation	It is recommended to use the wrapped version of wAVAX, which is an ERC20 version of AVAX, for the rewards.
Resolution	ACKNOWLEDGED The client will take this into consideration and all their derivative contracts will have fallback functions.

Issue #26	**`setJoeAddress`, `setJoetroller`, `setAdmin` and `claimReward` does not emit events**
Severity	INFORMATIONAL
Description	Functions that change important state variables should emit events for third-parties to be able to listen to.
Recommendation	Consider adding events for the above functions.
Resolution	ACKNOWLEDGED

Issue #27	**`rewardDistributor` will stop working in the year 2106 due to overflow reverting**
Severity	INFORMATIONAL
Description	Within the `rewardDistributor`, block timestamps are cast to uint32 which requires they can be at most 2^32 large, or 4,294,967,295. This corresponds to the date Sunday, February 7, 2106. Once this date is reached, the protocol will stop functioning since these casts will revert.
Recommendation	Since this year is still extremely distant, derivative protocols and the Trader Joe Lending protocol should just keep this in mind when this year approaches and do the necessary upgrades before it.
Resolution	ACKNOWLEDGED

4 Utilities

4.1 FlashLoanLender

The FlashloanLender contract is a utility contract that makes the flash loan functionality of the JCapableERC20 and JCollateralCapErc20 tokens. It acts as a wrapper to allow viewing values of the various JTokens' maximum flash loan amount and flash loan fee. Users can use this contract to make a flash loan by specifying the underlying token address.

While the list of supported tokens is maintained by the owner using the underlyingToJToken mapping, even if a token is not listed in this contract, as long as the JToken contract supports flash loans, it would still be possible to make a flash loan through direct interaction with the JToken contract.

4.1.1 Issues & Recommendations

No issues found.

4.2 Exponential

The exponential utility contract provides the Exp structure which represents a decimal number. Each Exp contains a single variable "mantissa" which is just the decimal number scaled by 18 zeros and rounded. This way the decimal can be stored internally as a whole number.

4.2.1 Issues & Recommendations

Issue #28	**Inconsistent rounding behavior between `mulExp` and `mul_`: `mulExp` rounds to the nearest integer while `mul_` does not**
Severity	INFORMATIONAL
Description	The `mulExp` function rounds to the nearest mantissa integer while the reverting counterpart function `mul_` does not. Although we could not find any issues with this, this behavior is inconsistent which might indicate the rounding behavior was not thought through.
Recommendation	Consider why this inconsistency is present and consider whether it needs to be rectified by using the same rounding approach for both functions.
Resolution	ACKNOWLEDGED

Issue #29	**Gas optimization: Unnecessary memory allocation for error messages in `add_`, `sub_`, `mul_` and `div_` functions**
Severity	INFORMATIONAL

Location	Line 360 (example)
	`function mul_(`
	`  uint256 a,`
	`  uint256 b,`
	`  string memory errorMessage`
	`) internal pure returns (uint256) {`

Description	The `mul_` function allocates memory for the error message even when it is never invoked. This results in the `mul_(uint256 a, uint256 b)` function using more gas than is strictly necessary.
Recommendation	Consider relying on the careful math library for the `mul_(uint256 a, uint256 b)` and relying on the return value to then throw the error message. This will likely be more gas efficient. Directly hardcoding `mul_(uint256 a, uint256 b)` is likely most efficient but results in repeated code. The same recommendations hold for the other operations.
Resolution	ACKNOWLEDGED

Issue #30	Gas optimization: No need to handle error pro-actively in certain functions
Severity	INFORMATIONAL

Location	Lines 68-75 (example)
	`function mulScalar(Exp memory a, uint256 scalar) internal pure returns (MathError, Exp memory) {`
	`    (MathError err0, uint256 scaledMantissa) = mulUInt(a.mantissa, scalar);`
	`    if (err0 != MathError.NO_ERROR) {`
	`        return (err0, Exp({mantissa: 0}));`
	`    }`
	
	`    return (MathError.NO_ERROR, Exp({mantissa: scaledMantissa}));`
	`}`

Description	Certain functions check whether the error is not `NO_ERROR` and then return an explicit response. This extra check might lead to extra gas usage. Other functions avoid this check by simply forwarding the error directly, which should save gas. An example of such a function is provided in the recommendation. The following functions can be simplified: `getExp`, `mulScalar` and `divScalar`.

Recommendation	In case gas optimization is important, consider checking whether simply forwarding the error with the wrongful mantissa could have side-effects. Furthermore consider doing a before-after check on the gas usages of these functions to validate that they have gone down as expected.
	
	An example of such a simplification is already provided in some of the functions like the following. It should be noted that through this simplification, a wrongful mantissa is returned, which could have side-effects if it is handled. This should be carefully checked throughout the codebase if implemented.
	
	Lines 59-63
	`function subExp(Exp memory a, Exp memory b) internal pure returns (MathError, Exp memory) {`
	`    (MathError error, uint256 result) = subUInt(a.mantissa, b.mantissa);`
	
	`    return (error, Exp({mantissa: result}));`
	`}`

Resolution	ACKNOWLEDGED

4.3 SafeMath

SafeMath is a simple math library to do operations like addition, subtraction, multiplication and division without the risk of them overflowing. Instead of overflowing, the operations will revert. It is directly based on OpenZeppelin's SafeMath.

4.3.1 Issues & Recommendations

Issue #31	**Gas optimization: Unnecessary memory allocation for error messages in `add_`, `sub_`, `mul_` and `div_` functions**
Severity	INFORMATIONAL

Location	Lines 117-121 (example)
	`function mul(`
	`  uint256 a,`
	`  uint256 b,`
	`  string memory errorMessage`
	`) internal pure returns (uint256) {`

Description	The `mul_` function allocates memory for the error message even when it is never invoked. This results in the `mul(uint256 a, uint256 b)` function and other functions using more gas than is strictly necessary.
Recommendation	Consider taking inspiration from more recent SafeMath versions where OpenZeppelin opts for alternatives like `tryMul`. The CarefulMath library included within this audit is very similar to such versions.
Resolution	ACKNOWLEDGED

4.4 CarefulMath

CarefulMath is a simple math library derived from OpenZeppelin's SafeMath, with the goal of handling potential overflows, underflows and division by zero issues. However, instead of reverting, it returns an error status as the first argument which can then be processed.

4.4.1 Issues & Recommendations

No issues found.

4.5 ErrorReporter

The ErrorReporter provides some basic utilities to emit error events for interpretation by the frontend. It should be noted that none of the `failure` functions actually revert so an error handling should always be followed after an error is reported.

4.5.1 Issues & Recommendations

No issues found.

5 Interest Rate Models

Within Trader Joe Lending, borrowers pay interest to suppliers. This interest rate is directly based on the utilization rate of the supply. If more of the supply is being lent out, a higher interest rate is paid. This is based on the economical principle of supply and demand to achieve a lending market in equilibrium. The degree to which the interest rate adjusts is specified in the interest rate models which should be carefully adjusted asset-by-asset. For example, a token which already has a risk-free interest rate when being staked should likely have a higher interest rate and faster climbing rate than a coin with less staking opportunities and more intrinsic value.

The interest rate models are responsible for calculating both the supply and borrow rate and should properly account for the fact that the reserve rate, which is how much of the interest goes to the reserve, is the difference between these two rates (adjusted for the fact that not the whole supply is lent out).

Operational tips and recommendations

Timestamps instead of blocks

Within the Cream framework, block height is used to signal time passing. However, due to the modifications made within Trader Joe, time is in fact measured in seconds since the unix timestamp. Although this is correctly setup within the interest rate models, the models and interface still indicate that these values are per block. If the operation is ever transferred to another party or delegated, there's a risk that they might assume that they need to configure these contracts using block height while in fact they use seconds.

The client should consider keeping an internal operational documentation that contains information on how the interest rate models function and that they are in fact configured in seconds.

Roof parameter

The interest rate models contain a roof parameter which must be set to an utilization ratio of at least 100%. For example, on certain Cream models, we saw it was set to 150% (1.5 * 1e18), what this means is that if through some unlikely cycle of (lack of) liquidations the utilizations reaches 150%, the interest rate becomes capped after this percentage instead of further rising with utilization. We do not see this parameter as highly essential in the functioning of the protocol but were initially confused by its usage. It can be set anywhere between 100% and 150% without any issue. The code will revert in case it is set lower than 100%, if needed, that minimum needs to be lowered within the code.

Model effectiveness

Even for deployments using the exact same contract code, it is important for operations to define the parameters for the interest rates to work effectively. Without appropriate values of variables such as `multiplierPerYear` and `jumpMultiplierPerYear`, the borrow interest rate could cap out at a rate where it is advantageous for borrowers to not return the funds, even at extremely high levels of utilization. This could result in a liquidity crisis. It is recommended to start off with tried and tested values such as the ones stated here. The behavior of the system should be closely monitored and adjusted accordingly to avoid the situation where there is a constant lack of liquidity.

5.1 JumpRateModelV2

The JumpRateModelV2 is an interest model that depends on the utilization rate. It has a starting interest rate called `baseRatePerBlock` and goes up linearly with the utilization according to the `multiplierPerBlock` value. Once the kink utilization is reached, the rate can grow faster or slower according to `jumpMultiplierPerBlock`.

The model thus has a base rate and two linear components.

5.1.1 Issues & Recommendations

Issue #32	**`accrueInterest()` is not called before the interests rate model is manually adjusted by governance causing the adjustment to work slightly in retrospect**
Severity	INFORMATIONAL
Description	The governance can manually adjust the slope and the two rates of the jump rate model through the `updateJumpRateModel` function. However, since no `accrueInterest()` is called on the related token before this, this might cause the new model to work slightly in retrospect. This might cause stakers to unexpectedly pay a higher rate. This issue is marked as informational risk since it might not be trivial to add this requirement in since it would require either having the update go through the token or making the interest model aware of the token which might not be worth the trade-off.
Recommendation	This issue will be marked as partially resolved if the client can confirm that they will call `accrueInterest()` manually before updating the model. If the client wants to go for complete resolution, they could use a wrapper contract that is the owner of the models and always calls `accrueInterest` first within the same transaction.
Resolution	RESOLVED This model will not be used.

Issue #33	Lack of validation within `updateJumpRateModelInternal`
Severity	INFORMATIONAL
Description	The `updateJumpRateModelInternal` function does not validate that the parameters are within the expected ranges. Through user error, this might lead to undesirable effects.
Recommendation	Consider adding a minimum to the kink and sanity checks to the multipliers.
Resolution	RESOLVED This model will not be used.

Issue #34	Comments and variables still indicate that the model is based in block height time
Severity	INFORMATIONAL
Description	Avalanche has stated that there is no such average block rate and that developers should rely on the block timestamps instead. Not doing so could be considered dangerous in certain situations, as the block rate can be manipulated at a relatively low cost. The comments and variable names still indicate that block height is used. However, we noticed that `blockPerYear` has been correctly adjusted to `secondsPerYear` so this is not a practical issue but just misleading for third-party reviewers.
Recommendation	Consider renaming the variables and comments to indicate that seconds are used.
Resolution	RESOLVED This model will not be used.

5.2 TripleSlopeRateModel

The TripleSlopeRateModel is another model with linear components. First the interest rate increases linearly with the utilization according to the `multiplierPerBlock` rate. Once it reaches the first kink, the interest rate is stable until utilization reaches the second kink after which it can increase again.

5.2.1 Issues & Recommendations

Issue #35	**`accrueInterest()` is not called before the interests rate model is manually adjusted by governance causing the adjustment to work slightly in retrospect**
Severity	INFORMATIONAL
Description	The governance can manually adjust the slope and the two rates of the jump rate model through the `updateTripleRateModelInternal` function. However, since no `accrueInterest()` is called on the related token before this, this might cause the new model to work slightly in retrospect. Thus, stakers might unexpectedly pay a higher rate. This issue is marked as informational risk since it might not be trivial to add this requirement in since it would require either having the update go through the token or making the interest model aware of the token which might not be worth the trade-off.
Recommendation	This issue will be marked as partially resolved if the client can confirm that they will call `accrueInterest()` manually before updating the model. If the client wants to go for complete resolution, they could use a wrapper contract that is the owner of the models and always calls `accrueInterest` first within the same transaction.
Resolution	PARTIALLY RESOLVED `accrueInterest` will be called before updates.

Issue #36	Lack of validation within `updateTripleRateModelInternal`
Severity	INFORMATIONAL
Description	The `updateTripleRateModelInternal` function does not validate that the parameters are within the expected ranges. Through user error, this might lead to undesirable effects.
Recommendation	Consider adding limits to the kinks and sanity checks to the multipliers.
Resolution	ACKNOWLEDGED

Issue #37	Comments and variables still indicate that the model is based in block height time
Severity	INFORMATIONAL
Description	Avalanche has stated that there is no such average block rate and that developers should rely on the block timestamps instead. Not doing so could be considered dangerous in certain situations, as the block rate can be manipulated at a relatively low cost. The comments and variable names still indicate that block height is used. However, we noticed that `blockPerYear` has been correctly adjusted to `secondsPerYear` so this is not a practical issue but just misleading for third-party reviewers.
Recommendation	Consider renaming the variables and comments to indicate that seconds are used.
Resolution	ACKNOWLEDGED

6 JTokens

Each JToken represents an asset that can be loaned and borrowed within the Joe Banker system. This is referred to as a "market" since it allows lenders and borrowers to come together and transact the token. The Joe Banker system, based on Cream, allows for various implementations of the JToken interface. The team has indicated that the JCollateralCapERC20 token will be the token which will initially be used in production, since this is the token which is also actively used by Cream.

When users supply underlying tokens (eg. DAI.e) for lending, JToken will be minted as a receipt token. The underlying tokens will be stored in the JToken contract, where others who have provided collateral are able to borrow the supplied tokens.

Users who have exceeded the collateral ratio, defined as how much value they have borrowed over the value they have provided to the system, can be subjected to a liquidation of a portion of their position with a liquidation penalty. Timely liquidations are required to keep the protocol healthy with healthy debt (defined as properly collateralized debt).

Each token implementation has two other related contracts: A Delegator and Delegate contract. This is because JTokens are upgradeable, meaning that the token admin has the complete freedom to change their behavior over time. This was for example done by Cream to upgrade their tokens to the CollateralCap version. The Delegator contract is the proxy contract while the Delegate contract is the implementation contract that wraps the normal contract.

As the protocol will be running on the Avalanche network, where block production times are not fixed, Joe Banker uses block timestamps instead of block numbers for time accounting.

6.1 JToken

The JToken contract is the base contract which all other token implementations extend. It contains the core logic for a token market including the management of the lending and borrowing rates, liquidations and borrowing and lending interfaces for protocol users. Along with the Comptroller, they are the most central component of the system and the compromise of a single JToken could be detrimental to the protocol.

6.1.1 Issues & Recommendations

Issue #38	Admin holds privileges to change state of JTokens
Severity	HIGH SEVERITY
Description	The admin of JTokenAdmin possesses the ability to make state changes to any JToken owned by it. Such changes are sensitive and can cause behavioral changes in the protocol. If the admin account is compromised, there would be devastating consequences including loss of all staked funds.
Recommendation	Consider carefully designing the governance structure to minimize the risk of the governance account ever being compromised either by theft or malice. During the initial months of the protocol, this could be done by ensuring that the admin of JTokenAdmin is a TimeLock owned by a multisig solution.
Resolution	ACKNOWLEDGED The client has indicated that these privileges will be put behind a timelock. This issue can be resolved once we confirm that this has actually been added in the deployed contracts at the request of the client.

Issue #39	Reentrancy is possible from one JToken to another JToken
Severity	HIGH SEVERITY
Description	Certain tokens allow the user to execute code whenever they send or receive these tokens. When this code includes calling a function within the system again, that is called reentrancy. Because the lending system interacts between multiple tokens for liquidations and other features, it is very difficult to implement protection against such reentrancy risk. This is why many of the previous protocols like Compound and Cream have decided to simply not add any tokens which allow for reentrancy (most famously ERC-777 based tokens). Cream has famously added such a malicious token at the beginning of September 2021, allowing for this exploit to occur. As the reentrancy prevention is only done at the JToken level, it is possible to re-enter from one JToken to another if the underlying token has some sort of callback functionality during transfers. This reentrancy allows for further borrowing in another token while the state has not yet fully adjusted to the first borrow within the first token.
Recommendation	Verify that a token has no callback behavior (unsafe external calls during transfers) before listing it in the protocol. A further step can be taken to implement reentrancy protection at the comptroller level, but this would require some big changes that are not present in the current code. ! Furthermore, these are some types of tokens that should not be added to the protocol: - Rebase tokens or tokens that dynamically change the balance of tokens in an account without any transfers done. - Tokens that have callback functionality (e.g ERC777) which will allow reentrancy into another JToken contract.
Resolution	RESOLVED The client has indicated that the protocol is not designed for these tokens and will never add them. It should be noted that if this is ever done by accident, the protocol will be compromised.

Issue #40	Token transfers do not revert if the balance is insufficient
Severity	INFORMATIONAL
Description	In most ERC-20 tokens, transfers are reverted in case they failed. Some less competent developers have thus stopped checking the return value of the transfer function. However, the JTokens still simply return false instead of reverting. Although this is completely acceptable by the ERC-20 specification, third-party developers should take this into consideration and never forget to check the return value of JToken transfers.
Recommendation	Third-party developers and derivative contracts should never forget to interpret the return value of the JToken transfer, since the transaction will not revert if the transaction failed.
Resolution	RESOLVED Although no change has been made, the client has indicated that derivative protocols should simply check the return value on transfers as is required by the ERC-20 standard.

6.2 Token Implementations: JCollateralCapERC20

The JCollateralCapErc20 token is an extended feature version of JCapableErc20. This contract also introduces the concept of collateral and buffer tokens, where the sum of collateral and buffer tokens give the total balance of an account. When transferring tokens, only the collateral amount of the source account is checked to see if transfer is possible, and the collateral amount of the source is transferred to the destination.

When a supply is done, if the amount + totalCollateral > capCollateral, only capCollateral - totalCollateral will be added as collateral. Any excess of the supplied amount will not be able to be used as collateral if the collateral cap is exceeded. The admin can change the collateral cap. This would be the implementation used for ERC20 tokens in Joe Lending.

Finally, the JCollateralCapErc20 token also provides third parties to use the idle capital within the token for flashloan purposes, and a small fee linear to the amount of tokens borrowed is levied for each flashloan.

JCollateralCapErc20Delegator

Th JCollateralCapErc20Delegator is a proxy contract that is supposed to manage a JCollateralCapErc20Delegate and can upgrade between implementations.

JCollateralCapErc20Delegate

The JCollateralCapErc20Delegate is a proxy implementation that is supposed to be managed by the JCollateralCapErc20Delegator.

6.2.1 Issues & Recommendations

Issue #41	**`updateJTokenVersion` is only called when a token is upgraded to the CollateralCap version but this is not on unset on downgrades**
Severity	INFORMATIONAL
Description	When an old token is upgraded to JCollateralCapErc20, the delegate sets the version in the JoeTroller to COLLATERALCAP. However, if at a later date this token is downgraded again, this version is not reset by the other delegates. The client should thus never downgrade and be careful of side-effects if they change the implementation.
Recommendation	The client should not downgrade to an old token implementation once they upgrade or set their token implementation to the CollateralCap version.
Resolution	RESOLVED The client has confirmed they will never downgrade after the token is set to the collateral cap variant.

Issue #42	Unnecessary arithmetic in `increaseUserCollateralInternal`
Severity	INFORMATIONAL
Description	`totalCollateralTokens` can be set as the `collateralCap` as it is the sum of gap and `totalCollateralTokens`. `else if (collateralCap > totalCollateralTokens) { // If the collateral cap is set but the remaining cap is not enough for this user, // give the remaining parts to the user. uint256 gap = sub_(collateralCap, totalCollateralTokens); totalCollateralTokens = add_(totalCollateralTokens, gap);`
Recommendation	Set `totalCollateralTokens = collateralCap;`, which is the equivalent of `add_(totalCollateralTokens, gap)`.
Resolution	RESOLVED The client has simplified the arithmatic as was recommended.

6.3 Token Implementations: JWrappedNative

The JWrappedNative token is a wrapper for the AVAXnative token which allows minting and redeeming either by the wrapped native token, or the native token.

JWrappedNativeDelegator

Th JWrappedNativeDelegator is a proxy contract that is supposed to manage a JWrappedNativeDelegate and can upgrade between implementations.

JWrappedNativeDelegate

The JWrappedNativeDelegate is a proxy implementation that is supposed to be managed by the JWrappedNativeDelegator.

6.3.1 Issues & Recommendations

Issue #43	**`updateJTokenVersion` is only called when a token is upgraded to the JWrappedNative version but this is not on unset on downgrades**
Severity	INFORMATIONAL
Description	When an old token is upgraded to JWrappedNative, the delegate sets the version in the JoeTroller to `WRAPPEDNATIVE`. However, if at a later date this token is downgraded again, this version is not reset by the other delegates. The client should thus never downgrade and be careful of side-effects if they change the implementation.
Recommendation	The client should not downgrade to an old token implementation once they upgrade or set their token implementation to the JWrappedNative version.
Resolution	RESOLVED The client has confirmed they will never downgrade the token once it is set to the wrapped native variant.

6.4 Token Implementations: JErc20

The JErc20 is the most standard contract that does not have any of the features like flash loans and collateral caps. It is therefore no longer used within most lending systems.

JErc20Delegator

Th JErc20Delegator is a proxy contract that is supposed to manage a JErc20Delegate and can upgrade between implementations.

JErc20Delegate

The JErc20Delegate is a proxy implementation that is supposed to be managed by the JErc20Delegator.

JErc20Immutable

The JErc20Immutable contract is a simple implementation of the JErc20 contract which is not intended to be upgradeable.

JCapableErc20

The JCapableErc20 is an extension of the JErc20 contract which most prominently adds flash loan functionality. It furthermore adds internal cached deposit amounts.

JTokenAdmin

This is a wrapper contract that is the admin for JTokens, which allows making privileged function calls to the JToken. There are 2 types of privileged roles:

Admin

- `_setPendingAdmin`
- `_acceptAdmin`
- `_setJoetroller`
- `_setReserveFactor`
- `_reduceReserves`
- `_setInterestRateModel`
- `_setCollateralCap`
- `_setImplementation`
- `seize`
- `setAdmin`
- `setReserveManager`

ReserveManager

- `setReserveManager`

6.4.1 Issues & Recommendations

No issues found.


PALADIN
BLOCKCHAIN SECURITY