PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For Optix Protocol

06 November 2022

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.

Paladin retains full rights over all intellectual property (including expertise and new attack or exploit vectors) discovered during the audit process. Paladin is therefore allowed and expected to re-use this knowledge in subsequent audits and to inform existing projects that may have similar vulnerabilities. Paladin may, at its discretion, claim bug bounties from third-parties while doing so.

1 Overview

This report has been prepared for the Optix Protocol on the Polygon 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.

1.1 Summary

Project Name	Optix Protocol
URL	https://optixprotocol.com/
Platform	Polygon
Language	Solidity

1.2 Contracts Assessed

Name	Contract	Live Code Match
`OptionsERC721`	`0x8740dbCCCc2879d2FbC3A27f6dE06fD202aCE6E7`	MATCH
`OptionsVaultERC20`	`0x975c26bEEdA7De113D522183ae0ea50d07D16732`	MATCH
`OptionsVaultFactory`	`0xcbc06887c6Ee94676988952353370EA68197E843`	MATCH
`SimpleSeller`	`0x54b82190C3c89AEc027971908d4c941C0e107Af2`	MATCH
`Referrals`	`0x9fdD30aE10CCDEC6B1021C7f1c36b55A4814fDA4`	MATCH
`OptionsLib`	Dependency	MATCH
`Interfaces`	Dependency	MATCH

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Failed Resolution	Acknowledged (no change made)
High	9	8	-	1	-
Medium	7	4	3	-	-
Low	26	21	4	-	1
Informational	25	17	6	-	2
Total	**67**	**50**	**13**	**1**	**3**

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 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 Global Issues

ID	Severity	Summary	Status
01	INFO	Typographical errors	✓ RESOLVED

1.3.2 OptionsERC721

ID	Severity	Summary	Status
02	HIGH	Anyone can steal collateral tokens from any approved users' wallet as create allows an exploiter to specify and target the user	✓ RESOLVED
03	HIGH	Underlying NFT is never transferred when the transfer function is called, and transferring the underlying NFT does not actually allow the new owner to exercise the option	✓ RESOLVED
04	MEDIUM	Lack of nuanced safeguards with regards to option creation increases edge-case risks in case of oracle failure, option mispricing, etc.	PARTIAL
05	LOW	Governance risk: The protocol governance is free to block option purchases and can make them ridiculously expensive	PARTIAL
06	LOW	Exercise check is insufficiently strict as it allows exercising while payoff is still zero	✓ RESOLVED
07	LOW	Phishing risk: There is no way for the user to enforce the slippage/ price limits of their option purchase	✓ RESOLVED
08	LOW	DoS: create lacks a minimum option size allowing users to create minuscule positions that would be tedious and expensive to unlock	ACKNOWLEDGED
09	LOW	Lack of reentrancy guards	✓ RESOLVED
10	LOW	Lack of validation	PARTIAL
11	LOW	The referrals contract does not receive the necessary inputs to properly calculate a referral percentage	✓ RESOLVED
12	INFO	Returning fees as a percentage of optionSize is an unnecessary complication that might require extra care for fee calculation implementations with regards to rounding	✓ RESOLVED
13	INFO	EIP-165 `supportsInterface` does not return true for all relevant supported interfaces	✓ RESOLVED
14	INFO	Gas optimizations	✓ RESOLVED
15	INFO	Typographical errors	✓ RESOLVED

1.3.3 OptionsVaultERC20

ID	Severity	Summary	Status
16	HIGH	An exploiter can drain any account by providing or withdrawing tokens for that account	✓ RESOLVED
17	HIGH	Shares are vulnerable to frontrunning risk	✓ RESOLVED
18	HIGH	An exploiter can completely bypass existing share frontrunning protection through carefully pre-requesting withdrawals with dummy accounts	✓ RESOLVED
19	HIGH	Option holders cannot exercise their options unless they have their permitted withdrawal period active	✓ RESOLVED
20	MEDIUM	Centralization risk: Vault owners can adjust crucial parameters including the option pricing algorithm, the withdrawal delay for suppliers which allows the owner to prevent any supplier from ever withdrawing and price options too cheaply	PARTIAL
21	MEDIUM	`provideAndMint` does not adhere to checks-effects-interactions, allowing an exploiter to bypass certain checks through a reentrancy-exploit if the collateral token permits reentrancy	✓ RESOLVED
22	MEDIUM	Options are no longer permitted to be sold once the collateral limit is reached	✓ RESOLVED
23	MEDIUM	`vaultCollateralAvailable` can revert, causing various other functions to unnecessarily revert	✓ RESOLVED
24	LOW	The AccessControl from OpenZeppelin is not used as it should be	✓ RESOLVED
25	LOW	`vault` does not explicitly handle the case where an undercollateralized vault becomes insolvent	✓ RESOLVED
26	LOW	Centralization risk: The operator can change any state to an immutable boolean	✓ RESOLVED
27	LOW	Role management functions use AccessControl's public functions which already have authentication	✓ RESOLVED
28	LOW	`provideAndMint` unnecessarily validates that mintTokens is larger than 0 and emits a mint event even if the function is called without requiring a share mint	✓ RESOLVED
29	LOW	The default value of the `vaultFee` is ignored completely due to the contract implementing the proxy pattern	✓ RESOLVED
30	LOW	Collateralization ratio calculations are inverted compared to what they are supposed to be	✓ RESOLVED
31	LOW	`withdrawAndBurn` unnecessarily rounds the withdraw amount twice when it is called via `send`	✓ RESOLVED
32	LOW	Put options do not follow the traditional put option payoffs	PARTIAL
33	LOW	Silently checking for allowance on `initiateWithdraw` makes little sense	✓ RESOLVED
34	LOW	`readonly` parameter still allows for collateral deposits	✓ RESOLVED

35	LOW	The `setVaultFeeRecipient` function lacks an `address(0)` check	✓ RESOLVED
36	INFO	Vault token balances will be extremely high due to the configured decimals being too low	✓ RESOLVED
37	INFO	Gas optimizations	✓ RESOLVED
38	INFO	Contract does not use upgradeable OpenZeppelin dependencies	ACKNOWLEDGED
39	INFO	Typographical errors	✓ RESOLVED
40	INFO	The protocol does not support tokens with a fee on transfer and tokens that rebase	✓ RESOLVED

1.3.4 OptionsVaultFactory

ID	Severity	Summary	Status
41	HIGH	Governance risk: Governance can reduce the collateralization ratio of vaults to zero which can pose significant risks for people who have active options purchased	FAILED
42	LOW	Lack of validation	PARTIAL
43	INFO	Typographical errors	✓ RESOLVED
44	INFO	Gas optimizations	PARTIAL
45	INFO	`createVault` does not strictly adhere to checks-effects-interactions	✓ RESOLVED
46	INFO	Lack of events for and `initialize` and `setCollateralizationRatio`	PARTIAL

1.3.5 SimpleSeller

ID	Severity	Summary	Status
47	HIGH	The `getFactor` function is severely flawed	✓ RESOLVED
48	HIGH	`callPrices` or `putPrices` cannot be reset	✓ RESOLVED
49	MEDIUM	The different periods may not be in decreasing order while strikes and factors might not be in increasing order	✓ RESOLVED
50	MEDIUM	The pricing often rounds in the user's favor	PARTIAL
51	LOW	The `findStrikeAndMatchPeriod` function does not always revert on a currentPrice of 0	✓ RESOLVED
52	LOW	The `findMatchFee` function does not revert if an incorrect `optionType` is provided	✓ RESOLVED
53	LOW	The different stored arrays lack a getter for their length	✓ RESOLVED
54	INFO	Lack of events for the `setFactor`, `deletePricePoints` and `pushPricePoints` functions	✓ RESOLVED
55	INFO	`setFactor`, `deletePricePoints` and `pushPricePoints` can be made `external`	PARTIAL
56	INFO	Lack of validation	PARTIAL
57	INFO	Typographical errors	✓ RESOLVED
58	INFO	Gas optimizations	PARTIAL

1.3.6 Referrals

ID	Severity	Summary	Status
59	LOW	Governance risk: Governance can override referrals at any point in time and gives themselves this privilege by default	✓ RESOLVED
60	LOW	First address can be added multiple times to the referrers array	✓ RESOLVED
61	LOW	The `setReferralFeeRecipient` function lacks an `address(0)` check	✓ RESOLVED
62	INFO	Gas optimizations	PARTIAL
63	INFO	Typographical errors	✓ RESOLVED

1.3.7 OptionsLib

ID	Severity	Summary	Status
64	INFO	Code style improvement: Attaching the library to the `BoolState` struct	✓ RESOLVED

1.3.8 Interfaces

ID	Severity	Summary	Status
65	INFO	Unused portions of code	✓ RESOLVED
66	INFO	`IStructs` interface is too broad which might cause unused events to show up in the contracts their ABI specification	ACKNOWLEDGED
67	INFO	The different interfaces do not define the functions of their contracts	✓ RESOLVED

2 Findings

2.1 Global Issues

The issues listed within this section are global issues that apply to the entire codebase.

2.1.1 Issues & Recommendations

Issue #01	Typographical errors
Severity	INFORMATIONAL

Description	We have consolidated typographical errors into a single issue in an effort to keep the report size reasonable.
	
	Top section of most contracts:
	`*  * SPDX-License-Identifier: GPL-3.0-or-later`
	
	Generally the SPDX-License-Identifier is structured as a single comment at the very top of the codebase:
	`// SPDX-License-Identifier: GPL-3.0-or-later`
	
	OptionsERC721::9
	OptionsVaultERC20::6
	OptionsVaultFactory::9
	SimpleSeller::7
	`import "./Referrals.sol";`
	`import "./OptionsVaultFactory.sol";`
	`import "./OptionsVaultERC20.sol";`
	`import "./OptionsVaultFactory.sol";`
	
	Within several locations of the code, a whole contract is imported instead of just the contract's interface. This has a few minor disadvantages. First, the verified code in the explorer will need to include the code of all these contracts when only the interface is used. Second, when a change is made to a single contract, a deploy script might need to recompile and deploy everything instead of just the single contract. This latter issue might be cumbersome once the client decides to update `Referrals.sol`.
	
	Throughout the codebase, almost all of the code is not formatted with a formatter. Many lines end in a number of spaces. Keeping the code within a standard format is generally considered best practice.

	Throughout the codebase, both `_msgSender()` and msg.sender are used. This makes the usage of `_msgSender()` rather futile. Consider sticking to a single method.
Recommendation	Consider fixing the typographical errors.
Resolution	RESOLVED

2.2 OptionsERC721

`OptionsERC721` is an NFT token which represents individual option positions. It is shared across all vaults created by the `OptionsVaultFactory` and can represent both CALL and PUT options.

This contract not only acts as the token for the option positions, it also acts as the primary interface between the users who want to purchase options and the underlying vaults which can sell them.

The main function to purchase an option is the `create` function. When called, the user can provide all details of the option they are interested in purchasing: the strike price, the oracle to use (in which currency the option is denominated), whether the option is a call or put and when it expires (the maturity date). When creating an option, the user needs to transfer in collateral tokens of the vault as payment for the option. These tokens are then distributed to various recipients as fees and as compensation for the vault. The distribution locations are as follows:

- protocol fee (configured by the governance) to the `protocolFeeRecipient` (governance wallet)
- vault fee (payment to the vault manager) to `vaultFeeRecipient` of the vault
- referral fee to the referrer of the person who called `create`
- intrinsic and extrinsic fees (calculated by the `SimpleSeller`) are sent to the vault as value for the vault

Once all these fees have been transferred from the creator to the destinations, the option details are registered within this contract, the collateral is locked within the vault and a receipt token linking to the option is minted.

The option can be exercised at any point in time up to the expiration timestamp via the `exercise` function. Exercising is only allowed if the option is in-the-money. The option holder is the only account that can exercise their option. However, for the

last 30 minutes of the option, anyone can exercise the option for the holder as the option would have no value anyway after it expires. In this case, the profits are still sent to the original option holder. This 30 minute duration can be freely scaled down by the vault owner.

Profits are always capped to the `optionSize`, which is the number of contracts the holder purchased and are always paid out in the underlying token.

Option NFTs can be freely transferred to other users while the option has not expired.

If the option has not been exercised by the option maturity, anyone can call `unlock` to unlock the vault collateral that the option had locked up.

For the option premium, the protocol and referral fees are not calculated by the vaults but are calculated by a fee calculator defined by the governance.

2.2.1 Privileges

The following functions can be called by the various privileged roles of the contract:

- `setProtocolFee [ DEFAULT_ADMIN_ROLE ]`
- `setProtocolFeeRecipient [ DEFAULT_ADMIN_ROLE ]`
- `setProtocolFeeCalc [ DEFAULT_ADMIN_ROLE ]`
- `setReferrals [ DEFAULT_ADMIN_ROLE ]`
- `setAutoExercisePeriod [ DEFAULT_ADMIN_ROLE ]`
- `grantRole [ DEFAULT_ADMIN_ROLE ]`
- `revokeRole [ DEFAULT_ADMIN_ROLE ]`
- `renounceRole [ role owner ]`

2.2.2 Issues & Recommendations

Issue #02	**Anyone can steal collateral tokens from any approved users' wallet as `create` allows an exploiter to specify and target the user**
Severity	HIGH SEVERITY

Location	Line 120-121
	`function create(`
	`address holder,`

Description	The `create` function which is responsible for buying options from a vault allows the caller to freely specify who to buy an option for. This holder is not validated in any form. However, the actual payment of the option is also taken from this holder: `factory.vaults(vaultId).collateralToken().safeTransferFrom(holder, address(factory.vaults(vaultId)), remain);` An exploiter can thus steal ALL approved tokens of users through the following exploit sequence: 1. For each approved collateral token: create a whitelisted vault with an extremely expensive pricer (eg. the options are priced ridiculously expensive). 2. For all users with an approval to the options contract with any of these collateral tokens, force these users to buy options within the vault with all of their approved tokens at the ridiculous price. 3. Finally, withdraw the vault shares and steal all premiums that the users have paid. An exploit like this has proven to have an absolutely catastrophic impact in the past as often people do not expect that exploiters can steal funds from their wallet. The raw monetary damage but also reputational damage of an exploit like this should not be underestimated.

Recommendation	Consider removing the `holder` parameter completely and always use `msg.sender`. To create an option position for someone else: 1. Create the option position for yourself 2. Transfer the NFT token to the desired recipient This is a much cleaner way compared to having a potentially and in fact risky `holder` parameter. All occurences of `holder` should therefore be replaced with `msg.sender`.
Resolution	RESOLVED The `holder` parameter was removed.

Issue #03	**Underlying NFT is never transferred when the `transfer` function is called, and transferring the underlying NFT does not actually allow the new owner to exercise the option**
Severity	HIGH SEVERITY
Location	Line 237 `function transfer(uint256 optionID, address newHolder) external {`
Description	The NFT can be transferred using the custom `transfer` function, however, this does not actually transfer the underlying NFT which is denoted via `ownerOf` and other ERC-721 functions. When the actual option is transferred via `transfer`, all ERC721 related functions still indicate that the sender is the owner. Similarly, when an ERC-721 transfer is done, eg. by using `safeTransferFrom`, the actual option is not moved as these ERC-721 balances are presently unused. This could severely mislead users that buy an option on an ERC-721 marketplace and makes the ERC-721 inheritance of the contract fundamentally futile.
Recommendation	Consider replacing the `transfer` function with an overwrite of the internal ERC-721 `transfer` function that is called on all internal transfers except mints and burns.
Resolution	RESOLVED `_transfer` was overwritten to transfer the NFT accordingly. However, `_msgSender()` is checked to be the `owner` of the options, meaning the `transferFrom` function will not work and renders approvals null and void.

Issue #04	**Lack of nuanced safeguards with regards to option creation increases edge-case risks in case of oracle failure, option mispricing, etc.**
Severity	MEDIUM SEVERITY
Description	Complex financial systems like lending protocols but also option vaults tend to be vulnerable to edge-case risks like oracle failure, token failure, chain failure, key compromise and more. Complex protocols being vulnerable to such edge-case risks is standard and we commend the Optix team for doing their part to mitigate a large portion of these risks. However, we recommend that further efforts are implemented to further safeguard individual vaults and option positions against infrequent and edge case risks, both known risks and unknown risks. The protocol already nicely limits the maximum amount of collateral a single vault is allowed to have in an effort to slowly ramp up this limit. We however want to recommend more limits to the protocol. The first set of limits should reduce the impact and freedom an exploiter has if they found a profitable way to create options. We recommend that a configurable hourly and/or daily limit is introduced on the number of options (more correctly, the amount of option collateral) that can be purchased. Once this limit is reached, users must wait before buying more options. In case a successful exploit is found, the impact would therefore be limited to this limit and the vault owner can disable the vault by marking it as "read only" to prevent further exploitation. The second set of recommendations we would like to make are related to oracle failure: we recommend that options are only allowed to be created if the oracle is considered "healthy". As health can be defined via a variety of metrics, it might make sense to abstract the oracle interface in a helper contract that returns `true/false` on an `isHealthy()` call. The basic checks that should likely be made are:

- Is the oracle response within a reasonable range of the on-chain spot rate?
- Is the oracle response up-to-date? If the oracle has not been updated for hours, it should likely be considered outdated.
- Is the oracle response strictly greater than zero?

Since blocking option creation should not directly put users at risk and does not prevent any existing options from maturing or vault stakers to withdraw their stake, the protocol can consider being quite elaborate in adding safeguards. The protocol can consider adding a modularized health check mechanism where the options or vault contract calls a vault-specific health check contract to ask if the vault is still healthy enough for more options to be created. To further improve on the quality of the health check, the "to be created" option specification should likely also be included. In case this health check function is not marked as `view`, the previous recommendation of hourly/daily limits could even be added within the health check.

For more argumentation why adding safeguards for extreme scenarios are desired, we recommend reading up on all of the lending protocol exploits that have occurred in the past years. The majority of the impact of these would have been reduced significantly if these protocols had better risk management procedures.

We would like to finally conclude that this issue is not necessarily raised because this protocol is risky by itself. The contract architecture is cleanly designed. We have however seen many codebases that were securely written, like Compound Finance, which still had to struggle through situations that could have been avoided given more up-front safeguards.

Recommendation	Consider adding an abstract, configurable, health-check contract that potentially takes the current vault state and the details of the current option as arguments. The health-check contract can then return true or false to indicate whether this option is allowed to be purchased. By not marking this check function as `view`, components like hourly/daily limits can even be modularized to this component. If the client decides to implement this recommendation, we should note that the actual health-check contracts would fall out-of-scope of the current audit. Apart from these specific recommendations, we would also recommend the team to start a bug bounty program with ImmuneFi and write detection tools that detect abnormal events like large portions of vault collateral suddenly leaving a vault. Ideally, these detection tools should automatically pause supplying and purchasing of options as well.
Resolution	PARTIALLY RESOLVED A `healthCheck` contract was added but by default, it always returns true. However the `DEFAULT_ADMIN` can set the `healthCheck` address to another contract that would do accurate health checks but this is not implemented yet.

Issue #05	**Governance risk: The protocol governance is free to block option purchases and can make them ridiculously expensive**
Severity	LOW SEVERITY
Description	The Optix governance is free to configure the protocol fee calculator contract and the referral contract. If either of these contracts revert during their calculation or return an excessive protocol fee amount, this would severely impact the user experience for people who are buying options. The most obvious impact is that governance can prevent create (purchasing an option) from being called. The more nuanced impact is that the governance can increase the cost of option purchases without limit at any point in time. This issue is rated as low as users are still allowed to mature their options and withdraw from underlying vaults without any problem. It is mostly included to inform contract systems that plan to build upon the Optix codebase about the fact that create might not always work as intended.
Recommendation	Consider using an Enumerable alternative to `AccessControl` to easily allow users to inspect who has admin privileges. Consider only granting admin privileges to a multisig set-up. Consider allowing the user to specify a `maxPremium` parameter during the create call and actually setting in with the necessarily UI communications.
Resolution	PARTIALLY RESOLVED A `maxPremium` parameter was added.

Issue #06	**Exercise check is insufficiently strict as it allows exercising while payoff is still zero**
Severity	LOW SEVERITY

Location	Line 217
	`require(option.strike <= currentPrice, "Options: Current price is too low");`
	
	Line 220
	`require(option.strike >= currentPrice, "Options: Current price is too high");`

Description	The `exercise` function allows for option exercise even if the payoff is zero. There should absolutely be no reason to exercise the option in this scenario except if the holder perhaps got bribed by the vault suppliers to remove their option contract (which would be an argument to remove the requirement in general so it is insufficient).
Recommendation	Consider making these requirements strictly greater and smaller than compared to allowing the variables to be equal to each other.
Resolution	RESOLVED The requirements are now strictly greater and smaller.

Issue #07	**Phishing risk: There is no way for the user to enforce the slippage/ price limits of their option purchase**
Severity	LOW SEVERITY
Description	The option price is calculated when the user requests to purchase an option. However, not a single parameter within the `create` function allows the user to specify the maximum price they are comfortable with paying for the option. This could put the user in a bad position where they end up paying a higher price than they are comfortable with, especially if front-run bots attempt to extract value from the transaction.
Recommendation	Consider adding a `maxPremium` parameter that is validated to be larger or equal to the total premium the user is about to pay.
Resolution	RESOLVED A `maxPremium` parameter was added.

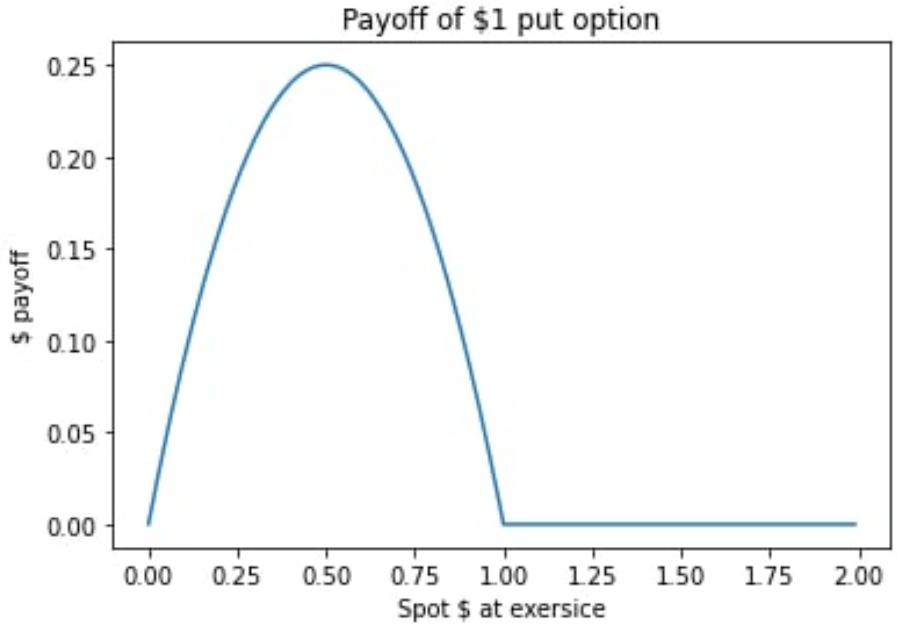
Issue #08	**DoS: create lacks a minimum option size allowing users to create minuscule positions that would be tedious and expensive to unlock**
Severity	LOW SEVERITY
Description	The `create` function allows users to create an option with a near zero `optionSize`. This would make it extremely tedious for vault suppliers to eventually unlock all of these minuscule positions to retrieve their collateral. This issue is marked as just low severity as this attack is equally or even more expensive for the attacking party.
Recommendation	Consider allowing the vault contract to specify a minimum contract size. This check could be isolated to the recommended health check contract to keep the options contract sufficiently simple.
Resolution	ACKNOWLEDGED

Issue #09	Lack of reentrancy guards
Severity	LOW SEVERITY
Description	The `create` function lacks a reentrancy guard and does not adhere to the checks-effects-interactions pattern. This could make it exploitable if the developer decides to upgrade the `OptionsERC721` contract to allow for the tokens to be burned again. Currently, the `_safeMint` function sort of acts as a reentrancy guard as it will revert if the `optionID` is already minted. Without this safeguard, various exploits and DoS vectors would become possible as we could overwrite a vault option that was created through a reentrancy call. Presently, such vectors are not possible due to the `mint` call causing a revert in reentrancy calls. However, in order to reassure code reviewers and as to safeguard the protocol against future additions, we strongly recommend adding at least a reentrancy guard.
Recommendation	Consider at least adding a reentrancy guard. The code quality can also be improved by moving the `collateralToken` transfers all the way down in the function to adhere to checks-effects-interactions. In this case, it is absolutely vital that all values (eg. `vaultFeeRecipient()`) are already calculated during the checks/effects section of the function, and not at the bottom of the function call.
Resolution	RESOLVED A non-reentrant was added and the transfers were moved at the end of the function to align more with checks-effects-interactions.

Issue #10	Lack of validation
Severity	LOW SEVERITY

Description	The contract contains functions with parameters which are not properly validated. Having unvalidated parameters could allow the governance or users to provide variable values which are unexpected and incorrect. This could cause side-effects or worse exploits in other parts of the codebase.
	
	Consider validating the following function parameters:
	
	Line 26
	`address _protocolFeeRecipient,`
	
	Line 290
	`function setProtocolFeeRecipient(address value) external IsDefaultAdmin`
	
	A non-zero validation should occur on these addresses as tokens are being sent to it and such transfers tend to revert to the zero address.
	
	Line 87
	`_fees.protocolFee = protocolFeesCalcs.getProtocolFee(holder, period, optionSize, strike, latestPrice, optionType, vaultId, oracle);`
	
	The `protocolFee` of the `OptionsERC721` contract is validated to be at most 1% of the `optionSize`. However, this validation is no longer present if a different protocol fee calculator is used. In our opinion, this makes the 1% validation rather futile as it can be bypassed.
	
	It might be desired to validate certain other fees within the premium function.

It is likely desired to validate that the oracle in question is supposed to be used for puts or calls on the specific vault. Based on our understanding, Optix plans to use the put options with an inverted oracle compared to the call options. Using the same oracle on both sides is therefore likely undesired as is described in this graph that shows the PUT payoff in case it uses the call oracle of a vault:



Recommendation Consider validating the function parameters mentioned above.

For tokens that could potentially be transferred to the zero address, the client could also consider simply not executing the transfer if the recipient is the zero-address. This would allow the happy path to still continue.

Resolution PARTIALLY RESOLVED

The functions parameters of the premium are not validated and the transfers to `address(0)` can still occur as it is not checked.

Issue #11	The referrals contract does not receive the necessary inputs to properly calculate a referral percentage
Severity	LOW SEVERITY

Location	Line 108-110
	`function getReferFee(address holder,uint256 period, uint256 optionSize, uint256 strike, uint256 currentPrice, IStructs.OptionType optionType, uint vaultId, IOracle oracle) external view returns (uint256){`
	`    return referrals.referFee();`
	`}`

Description	The parameters of `getReferFee` are not passed onwards to the referrals contract. This gives the referrals contract very few options to do implement more advanced strategies and thus does not make sense for it to be a standalone contract which can be changed, as opposed to just being a fixed percentage. This issue is marked as low severity as the referral contract can still be changed indirectly by swapping out the whole protocol fees calculator.
Recommendation	Consider either moving to a fixed percentage for the referral fee within the default fee `calc` (this contract) or consider forwarding all parameters to the referrals implementation.
Resolution	RESOLVED The parameters are now forwarded to the referrals implementation.

Issue #12	**Returning fees as a percentage of `optionSize` is an unnecessary complication that might require extra care for fee calculation implementations with regards to rounding**
Severity	INFORMATIONAL
Description	Presently, fee calculators that calculate the premium a user must pay for an option return the fee in basis point percentage terms to the `optionSize`. This significantly reduces the accuracy of the premium fee possibilities and also complicates the process for the developers who need to write these pricers. The reason for this is that `optionSize` is only a very weak proxy for the position value. A deep in-the-money position with `optionSize = 1` would be much more valuable than a deep out-of-the-money position with `optionSize = 10`. Returning the fees as a percentage of `optionSize` therefore seems undesirable. It should be noted that basis-point precision is likely also relatively imprecise, and there might be risks related to rounding errors within the pricing functionality.
Recommendation	One should consider returning the fees in their real form, without unnecessary precision reductions.
Resolution	RESOLVED Fees are returned in amounts.

Issue #13	**EIP-165 `supportsInterface` does not return true for all relevant supported interfaces**
Severity	INFORMATIONAL

Location	Line 274-276
	`function supportsInterface(bytes4 interfaceId) public view virtual override(ERC721, AccessControl) returns (bool) {`
	`    return super.supportsInterface(interfaceId);`
	`}`

Description	EIP-165 defines an interface that allows third-party contracts and tools to query the contract and ask whether it supports a specific interface. Both ERC-721 and OpenZeppelin's `AccessControl` support this standard. However, the `supportsInterface` function presently only returns true for one of these interfaces.
Recommendation	Consider using the union of both interface queries: `return ERC721.supportsInterface(interfaceId) || AccessControl.supportsInterface(interfaceId);` Optionally, the interface of the actual `OptionsERC721` contract can be explicitly supported as well and be included in the union.
Resolution	RESOLVED Both interfaces return true.

Issue #14	Gas optimizations
Severity	INFORMATIONAL

Description	The contract contains multiple sections of code that could be further optimized for gas efficiency and we have consolidated these into a single issue to keep the report brief and readable.
	
	Line 15
	`OptionsVaultFactory public factory;`
	
	`factory` can be made `immutable`.
	
	Line 120
	`function create`
	
	The `create` function currently has extremely bad gas consumption properties. Constant properties like the `vaultId` and the vault's `collateralToken` are being fetched repeatedly. These variables should obviously be cached within the function as they are immutable. The number of contract calls could further be decreased by having the fee calculator return all fees within a single call to it. `factory.vaults(vaultId)` can also be cached in many locations throughout the contract.
	
	Line 190-193
	`if(oracle.latestAnswer() <= 0)`
	`    return 0;`
	`else`
	`    return uint256(oracle.latestAnswer());`
	
	The `oracle.latestAnswer()` can be cached to save gas and to make the contract more predictable in its behavior as theoretically these two calls could return different values.
	
	Line 200
	`function exercise(uint256 optionID) external`
	
	The option should be cached in a memory state as its values are queried many times during the function. However, the stored state of the option would need to be updated to Exercised.

	Lines 259 - 260
	`factory.vaults(option.vaultId).unlock(optionID);`
	`emit Expire(optionID, option.vaultId, option.premium.total);`
	
	The `vaultId` of the option can be cached to save gas as it currently queries the storage twice unnecessarily.

Recommendation	Consider implementing the gas optimizations mentioned above.
Resolution	RESOLVED Most recommendations have been implemented, but `option.vaultId` value is not cached and is still loaded twice.

Issue #15	Typographical errors
Severity	INFORMATIONAL

Description	We have consolidated several typographical errors into a single issue in an effort to keep the report size reasonable.
	
	Line 206
	`//only holder or approved can excercise within the auto exercise period`
	
	This comment should say `exercise`, not `excercise`.
	
	Line 267
	`function unlockAll(uint256[] calldata optionIDs) public {`
	
	`unlockAll` can be made external.
	
	Line 291
	`emit SetGlobalAddress(_msgSender(),SetVariableType.ProtocolFeeRecipient, address(protocolFeeRecipient), address(value));`
	
	The addresses are unnecessarily re-casted to `address()` within this event.
	
	Finally, the `natspec` comments of the `create` function do not define all arguments.
	
	Line 306
	`require(value<=30 minutes,"value<=30 mintues");`
	
	The revert message of this requirement should say minutes.

Recommendation	Consider fixing the typographical errors.
Resolution	RESOLVED Most recommendations have been implemented, but the addresses are still unnecessarily re-casted to `address()` within the `SetGlobalAddress` event.

2.3 OptionsVaultERC20

`OptionsVaultERC20` represents an individual options vault. Each option vault is backed by a single underlying ERC20 collateral token which is used to underwrite all option positions.

Regular users can supply collateral to the vault and receive transferable vault shares representing this collateral in return. Whenever people buy options from the vault, these shares will appreciate in value. Whenever people exercise these options, the shares depreciate.

The primary functions for suppliers to supply collateral to the vault is provide (and the more flexible `provideAndMint`). To withdraw this collateral again, the system introduces a safeguard to prevent users from withdrawing their position right before the vault incurs losses. This safeguard requires the users to initiate a 1 week withdrawal delay (using `initiateWithdraw`) period before they can withdraw within the next 2 days. Withdrawals are executed using the `withdraw` function. The one week withdrawal delay period can be adjusted by the vault owner.

Options can be bought from the vault via the `OptionsERC721` contract. An option is priced via the owner configurable vault fee calculator and the intrinsic and extrinsic premium from this fee calculator are collected into the vault collateral, increasing the vault share value and collateral balance.

Once the option gets bought, the required collateral to pay its potential payoff is locked and can no longer be withdrawn by users until the option is either exercised or expired via unlock.

Once options get exercised via the `OptionsERC721` contract, the payoff is transferred to the option holder and taken from the vault share value resulting in a loss to the vault shareholders.

The vaults should be deployed by either the protocol governance or by regular users via the `OptionsVaultFactory`.

It is important for users to realize that each vault is managed by a vault owner who can configure various properties of the vault, even if there are already deposits. The Optix team has added functionality for this owner to voluntarily but permanently abstain from configuring crucial parameters, to disable the mutability of those configuration parameters. Some of the important aspects that an owner can change for a vault include the whole methodology to price options, all oracles that are enabled on this vault, the vault fee, which is a percentage of each option contract sent to the owner, from the purchaser, the recipient of this fee, whether unwhitelisted users can supply collateral, whether unwhitelisted users can purchase options, who is in these whitelists, what the maximum amount of collateral is, whether option purchasing is enabled (read-only flag.).

Especially and most importantly are the enabled oracles and the vault fee calculator configurations as either of these can allow for options to be sold with an expected loss, which could lead to the full drainage of the vault.

Disclaimer: Suppliers should carefully remember that supplying to an option vault incurs real risks. Up to the complete balance of the vault can be lost by unprofitable option underwriting and a badly configured fee calculator would significantly increase the likelihood of this. We strongly recommend that anyone who plans to supply larger sums of money to a vault carefully vet the vault configuration: Can any variables like the fee calculator still be changed? If so, is the owner reputable? Is the configured fee calculator written reasonably as to always price options with a reasonably high intrinsic and extrinsic price? Are the oracles used by the vault reputable and can the fee calculator be manipulated in any way by providing the wrong oracle to a put or a call? What is the probability of the other suppliers escaping the socialization of losses? What is the probability of other suppliers front-running the creation of profits by supplying "just-in-time"? Only once all these questions are answered, should a larger supplier consider supplying to a vault.

Optix is an open platform and not every vault will therefore be equally secure. The security of a vault fully and utterly depends on its configuration: It is expected that badly configured vaults will not be profitable and might lose their assets completely.

2.3.1 Issues & Recommendations

Issue #16	**An exploiter can drain any account by providing or withdrawing tokens for that account**
Severity	HIGH SEVERITY

Location	Line 89, 98, 133, 153 and 160
	`function provide(address _account, uint256 _collateralIn) public returns (uint256 mintTokens){`
	`function provideAndMint(address _account, uint256 _collateralIn, bool _mintVaultTokens, bool _collectedWithPremium) public returns (uint256 mintTokens){`
	`function initiateWithdraw(address _account) external`
	`function withdraw(address _account, uint256 _tokensToBurn) public returns (uint collateralOut) {`

Description	Many of the functions which allow users to deposit into or withdraw from a vault take an `_account` parameter which is never validated. This allows any exploiter to execute deposits and withdrawals for any user that has ever used the protocol. An exploiter can abuse this by withdrawing for everyone else when a profitable purchase comes in or supplying a bunch of tokens from other users when unprofitable losses are socialized. Even worse, it allows the exploiter to amplify the other attacks in the rest of this audit that drain the vault. This is because the exploiter can force all approved users to deposit more collateral before they drain it. Most critically and not relying on any other issues, the exploiter can drain all approvals to their own balance instantly using the following exploit pattern: 1. Flash loan and provide a large amount of collateral to own nearly 100% of the vault. 2. Deposit the whole collateral balance of all users with open approvals into the vault through `provideAndMint` with `_mintVaultTokens false` — this donates the deposit into your shares 3. Withdraw your shares, capturing nearly all of the profits from these value increases. 4. Pay back flashloan.

Recommendation	Consider removing the `_account` parameter from all functions and instead using `msg.sender`. Tokens are transferable, all zapping logic can therefore be orchestrated by transferring the vault token to the user indirectly.
Resolution	RESOLVED The `_account` parameter has been removed from all functions and `_msgSender()` is used.

Issue #17	Shares are vulnerable to frontrunning risk
Severity	HIGH SEVERITY
Description	Unlike traditional option vaults, Optix does not work with a periodic schedule. Users can join and leave a vault when they please (as long as departures are announced about a week beforehand), and options can be bought at any point in time with a free to choose maturity. This significantly improves the UX of both the yield farming suppliers of the vault and the option buyers, however, it comes with a significant trade-off and risk specific to anyone supplying to a vault with other users. These users run the risk that right before large incomes are created for the vault, other users front-run these transactions and deposit significant sums of extra collateral to capture larger portions of these profits. This design trade-off also mean that certain suppliers who took these profits might have already left the vault by the time that the actual costs of these options are incurred. This potentially puts the vault in a position where certain suppliers might be able to still escape the largest costs, leaving these costs to be completely borne by the unknowing passive users. An important example benchmark exploit is as follows, which can instantly draining a vault completely: 1. Exploiter is looking to steal all collateral of a 100 ETH collateralized vault 2. Exploiter initiates a withdrawal and waits 7 days for their withdrawal period to unlock 3. Exploiter flash loans 100k ETH (cost: free using balancer on ethereum) and deposits it all as collateral in the vault 4. Exploiter purchases 100 ETH call options which are extremely deep into the money (eg. strike is 5% of spot price). They might for example pay 96 ETH for this. They instantly receive this 90 ETH back as its almost completely allocated to their 100k ETH supply

5. Exploiter exercises their unlocked withdrawal and withdraws all flash loaned collateral, exploiter immediately pays this back and is now at a small loss due to protocol fees etc. of the purchase.

6. Exploiter exercises the in-the-money options within the same transaction and receives 95 ETH instantly. The exploiter made nearly pure profit excluding protocol fees

7. The original suppliers of the vault started off with owning 100 ETH before this transaction and ended up with owning just 5 ETH after it. They are completely ruined through an exploit without any collateral requirements or risk

Using this benchmark exploit as an example, the magnitude of this issue and why the withdrawal period is insufficient should be clearly demonstrated. Any unprivileged exploiter can drain the complete vault through a carefully constructed flashloan exploit.

It should also be noted that simply addressing this example is insufficient as it's just that- an example of the more fundamental issue.

We have looked at various other option protocols and all other protocols we looked at (eventually) address the issue fundamentally:

1. Some of them account for the value/expected obligation of outstanding options

2. Some of them only allow deposits/withdrawals during timeslots without any outstanding options (this is possible because the options these protocols write all expire at the same time, with a small gap between that expiry and the first underwriting of the next options)

3. Some of them only allow you to withdraw the unlocked portion of your vault supply, the locked portion of your supply is granted as the back asset of the outstanding options. This means that the locked portion is sent to your wallet, but you can only claim the non-exercised amount of that collateral after the related options are either exercised or expired.

	Implementing any one of these three fundamental resolutions that address this issue likely requires a complete redesign sadly enough, making this issue fundamental to the V1 Optix protocol design.
Recommendation	This issue is fundamental. The client has already introduced a mechanism for the most important portion by only allowing withdrawals for 2 days after waiting a 7 day period. However, this does not fully mitigate the issue. Other vaults traditionally address this issue by only allowing withdrawals once all options have matured, which means that everyone must have borne the option costs they received profits for before they can withdraw. Such a redesign would be tedious to implement within Optix. Finally, this issue does not present itself for vaults where there's only a single, trusted, supplier. Such vaults are safe. The client can therefore address the issue in various ways: 1. Acknowledge it and carefully evaluate the impact to ensure that exploiters will not be able to abuse this significantly. Unfortunately, given that we were already able to find proof-of-concept exploits that fully drain a vault this will be unlikely. Patching these individual exploits might be one approach but will likely only mitigate these examples as other exploit vectors might still exist as long as the fundamental issue exists. 2. Add further patches which reduce the impact: deposit delays, withdrawal delays, all deposits and withdrawals must occur on the same day of the week, deposit fees, withdrawal fees, KYC requirements, hourly global deposit/withdrawal limits, hourly exercise limits… All of these might still prove to be insufficient as they reduce impact but do not completely remove it. 3. Consider a careful redesign where the issue is completely mitigated, e.g. having a mechanism where users always bear the costs for the options they received the profit from. This would of course be a fundamental redesign of the protocol.

Resolution	RESOLVED This issue has been fundamentally addressed by enforcing the expiry of all options before people can withdraw and create LPs in a fixed intermediary period. We commend the Optix team for this fundamental and proper resolution, as opposed to simply patching impact.

Issue #18	**An exploiter can completely bypass existing share frontrunning protection through carefully pre-requesting withdrawals with dummy accounts**
Severity	HIGH SEVERITY
Description	As mentioned in the previous issue, the team has already taken steps to address this by only allowing withdrawals to occur for a period of 2 days after the user announced their withdrawal intentions 7 days earlier. This essentially enforces that users are forced to supply for at least 5 days for every 2 days of withdrawing power. Users can therefore also not escape any costs during these 5 day windows. However, the `OptionsVaultERC20` contract is transferable, and an exploiter can simply send any locked tokens to an account that is currently within the 2 day window. A meticulous exploiter will therefore use about 5 accounts and every 2 days initialize a withdrawal with the next account. The exploiter then always has an unlocked account after doing this for a week and continuing to do this forever. To withdraw locked tokens at any time, the exploiter then simply sends these tokens to the unlocked account and withdraws via that account. This exploit is completely costless and fully mitigates the withdrawal window protection. Using this exploit, it is therefore completely free for an exploiter to fully escape all costs of the Optix protocol. An exploiter with MEV power can go as far as to flash-supply to a vault when the vault generates profits, to capture most of these profits, and then instantly withdraw again with this exploit. This leaves the non-exploiting suppliers at a complete loss: they receive none of the profits but need to pay all option premiums. The most malicious exploiter will execute a flashloan exploit to fully drain a vault's collateral in a single transaction similar to the previous exploit:

	1. Write out deeply in-the-money options to themselves 2. Withdraw unlocked collateral 3. Exercise options 4. Profit
Recommendation	As discussed in the previous issue, this temporary unlock mechanism is insufficient to prevent severe exploits.
Resolution	RESOLVED This issue has been fundamentally addressed by enforcing the expiry of all options before people can withdraw and create LPs in a fixed intermediary period. There is therefore no longer a queue period for withdrawals. We commend the Optix team for this fundamental and proper resolution, compared to patching impact.

Issue #19	Option holders cannot exercise their options unless they have their permitted withdrawal period active
Severity	HIGH SEVERITY

Description	The exercise function within the `OptionsERC721` contract presently sends the payout to the user by calling the send function on this contract:
	
	`factory.vaults(option.vaultId).send(optionID, option.holder, profit);`
	
	The send function then calls `withdrawAndBurn` with `_burnVaultTokens` set as `false`:
	
	`withdrawAndBurn(_to, _amount*totalSupply() / collateralReserves, false);`
	
	However, due to a mistake within `withdrawAndBurn`, the fact that the user must have properly initiated a collateral withdrawal is always validated, regardless of this being a payout. This means that almost exactly 7 days ago, the option holder must have announced that they plan to exercise their option. Any options of a period shorter than 7 days are going to be extremely tedious to exercise. We expect most users to end up being surprised and severely disappointed when they realize near option maturity that they are unable to exercise their options and we strongly expect this to not be desired behavior.

Recommendation	Consider moving the collateral related logic to the `withdraw` function and using an internal function to only do the strictly mutual functionality between collateral withdrawals and token transfers. Overloading functions for these multiple very different purposes with boolean parameters to signal each purpose is not best practice and easily introduces mistakes.
Resolution	RESOLVED Withdrawal queuing has been removed.

Issue #20	**Centralization risk: Vault owners can adjust crucial parameters including the option pricing algorithm, the withdrawal delay for suppliers which allows the owner to prevent any supplier from ever withdrawing and price options too cheaply**
Severity	MEDIUM SEVERITY
Description	The owner of a vault can change the fee calculator for the options which returns the premium that must be levied for an option. If the fee is set too low, options will be sold at an expected loss. This is most dangerous for in-the-money options as a premium smaller than the intrinsic value would result in an immediate exercise profit to the purchaser and could be abused by the vault owner to drain the whole vault. The owner of a vault can also extend the withdrawal delay period to any timeframe, potentially thousands of years. This could allow them to prevent anyone from ever withdrawing or could allow them to strategically set it to still exit before losses occur. The owner of a vault can finally whitelist any oracle of their choosing which could also allow them to drain the vault collateral as they are able to write-out options to themselves using a malicious oracle that always instantly makes the options highly profitable to exercise. This issue is marked as medium risk compared to high as the client has already introduced safeguards that allow the owner to lock themselves out of adjusting most of these functionalities. However, we still need to include this as an issue to bring awareness to users to check these settings and also to ensure that some of the other settings are also allowed to be permanently locked, as this is not present on all settings right now. The withdrawal delay period can finally be set without bounds potentially preventing anyone from ever withdrawing their collateral. Finally, the SimpleSeller implementation allows the vault owner and operator to freely adjust option prices which can be used to drain a vault as well by underpricing options sold to the malicious owner or operator.

Recommendation	Consider carefully designing the frontend to inform users about potential centralization risks of the various vaults. Consider adding a factory which automatically deploys governance-risk free vaults for normal vault owners which are not properly KYC-ed and might behave maliciously. Consider adding immutability booleans for all settings which can presently be changed at any time and might be relevant to lock down. We are unsure how someone can ever safeguard the `SimpleSeller` within the current design, as implied volatility will likely change over time.
Resolution	PARTIALLY RESOLVED The client has indicated that they will clearly communicate these risks on the frontend, and will promote safer vaults over unsafe vaults.

Issue #21	**`provideAndMint` does not adhere to checks-effects-interactions, allowing an exploiter to bypass certain checks through a reentrancy-exploit if the collateral token permits reentrancy**
Severity	MEDIUM SEVERITY
Description	The `provideAndMint` function does not adhere to checks-effects-interactions, allowing an exploiter to bypass certain checks through a reentrancy-exploit if the collateral token permits reentrancy. This is because the collateral transfer occurs before the reserves are actually incremented and any effects have occurred. The most obvious check that can be bypassed is therefore the `maxInvest` check. This check can therefore be exceeded through reentrancy if the collateral token permits it. This issue is only rated as medium compared to high severity as we were not able to exploit it further than bypassing the maxInvest check. We also expect that most vaults would not use reentrancy tokens.
Recommendation	Consider re-writing the function in checks-effects-interactions and adding reentrancy guards to all user-facing functions.
Resolution	RESOLVED Reentrancy guards have been added and the function was rewritten in checks-effects-interactions.

Issue #22	Options are no longer permitted to be sold once the collateral limit is reached
Severity	MEDIUM SEVERITY
Location	Line 104 `require(vaultCollateralTotal()+(_collateralIn*factory.getCollateralizationRatio(this)/1e4)<=maxInvest,"OptionsVaultERC20: Max invest limit reached");`
Description	The protocol caps the amount of collateral that can be supplied to a vault in an effort to slowly roll out vaults, increase these limits and manage potential exploitation risks. However, this cap is always validated. This is a problem as the `OptionsERC721` functionality also uses the `provideAndMint` function to deposit the option premium from option purchases. This means that once the `maxInvest` limit is reached, which is very likely during this roll-out period, no options can be purchased. Therefore it is likely that throughout the roll-out period, option purchasing will not be possible most of the time.
Recommendation	Consider only checking this requirement if the provision was not collected from a premium.
Resolution	RESOLVED This requirement is now only validated if the user instantiated the transaction as a liquidity addition.

Issue #23	**`vaultCollateralAvailable` can revert, causing various other functions to unnecessarily revert**
Severity	MEDIUM SEVERITY

Location	Line 293-295
	`function vaultCollateralAvailable() public view returns (uint256) {`
	`    return vaultCollateralTotal()-vaultCollateralLocked();`
	`}`

Description	The `vaultCollateralAvailable` function can potentially underflow and revert once the collateralization ratio is increased by governance. This causes a lot of functions that call this function to suddenly revert without a proper revert message.
Recommendation	Consider returning zero in case the locked amount is greater than the total amount.
Resolution	RESOLVED Zero is now returned by default.

Issue #24	The AccessControl from OpenZeppelin is not used as it should be
Severity	LOW SEVERITY

Location	Lines 75 - 77
	`_setupRole(VAULT_OPERATOR_ROLE, _owner);`
	`_setupRole(VAULT_LPWHITELIST_ROLE, _owner);`
	`_setupRole(VAULT_BUYERWHITELIST_ROLE, _owner);`

Description	The `DefaultAdmin` is not given to the owner, nor any account, thus meaning that no one can call the `grantRole` function. The team has exposed the function to grant and revoke the 3 differents roles, but it should be noted that the AccessControl's functions allows the `DEFAULT_ADMIN` to grant and revoke the different roles.
Recommendation	Consider removing the `vaultOwner` and use `DEFAULT_ADMIN` instead. Additionally, the function limited to the owner has to check if the sender has the `DEFAULT_ADMIN` role.
Resolution	RESOLVED

Issue #25	**`vault` does not explicitly handle the case where an undercollateralized vault becomes insolvent**
Severity	LOW SEVERITY
Description	The contract allows for vaults to potentially be undercollateralized which means that the vault does not need sufficient collateral to pay back all potential obligations. This however poses the threat that if the vault does not have enough collateral to pay out an obligation, that this obligation cannot be exercised for the remaining collateral that the vault does have.
Recommendation	Consider explicitly handling the case within send if the vault is insolvent: Should the remainder of tokens be sent in this scenario or should the send function simply revert as it currently does?
Resolution	RESOLVED When the transfer amount is now below the collateral reserves, the remaining tokens are transferred, an insolvency event is emitted and and the vault is made `readonly`.

Issue #26	Centralization risk: The operator can change any state to an immutable `boolean`
Severity	LOW SEVERITY
Description	The operator role allows another address than the owner to change certain boolean states that can allow, or disallow, certain parameters. These boolean can be set to their immutable state making them unchangeable. The issue is that any operator has the right to set them to the immutable boolean. As we expect operators to be EOA and owner to be a more trusted address (for example a multisig), we believe that the operator should not have the rights to set any of those parameters to their immutable state as it could lock bad params to make the vault worthless, for example with tremendously big fees.
Recommendation	Consider allowing only the owner to set booleans to their immutable state.
Resolution	RESOLVED The client immutability settings restricted to the owner.

Issue #27	Role management functions use AccessControl's public functions which already have authentication
Severity	LOW SEVERITY

Description	Line 384
	`grantRole(VAULT_LPWHITELIST_ROLE,_value);`
	
	Line 390
	`revokeRole(VAULT_LPWHITELIST_ROLE,_value);`
	
	Line 396
	`grantRole(VAULT_BUYERWHITELIST_ROLE,_value);`
	
	Line 402
	`revokeRole(VAULT_BUYERWHITELIST_ROLE,_value);`
	
	Various functions within the codebase internally use `grantRole` and `revokeRole` to grant and revoke wallet permissions of governance wallets. These functions are supposed to be callable by operator and default admin roles.
	
	However, as these function calls lack an underscore, the `AccessControl` public functions are used:
	
	`function grantRole(bytes32 role, address account) public virtual override onlyRole(getRoleAdmin(role))`

Recommendation	Consider using the internal `_grantRole` and `_revokeRole`.
Resolution	RESOLVED

Issue #28	**`provideAndMint` unnecessarily validates that `mintTokens` is larger than 0 and emits a mint event even if the function is called without requiring a share mint**
Severity	LOW SEVERITY

Description	Line 120
	`require(mintTokens > 0, "OptionsVaultERC20: Amount is too small");`
	
	The `provideAndMint` function must always virtually mint tokens, even if no tokens will be minted by setting the token minting to false.
	
	Line 124
	`emit Provide(_account, _vaultId, _collateralIn, mintTokens, _mintVaultTokens);`
	
	This event always emits a `mintVaultTokens` amount even if none are minted.

Recommendation	Consider only requiring this if tokens must be minted. Consider only emitting the mint amount if tokens are minted. This event issue should also be fixed on withdrawals as well.
Resolution	RESOLVED

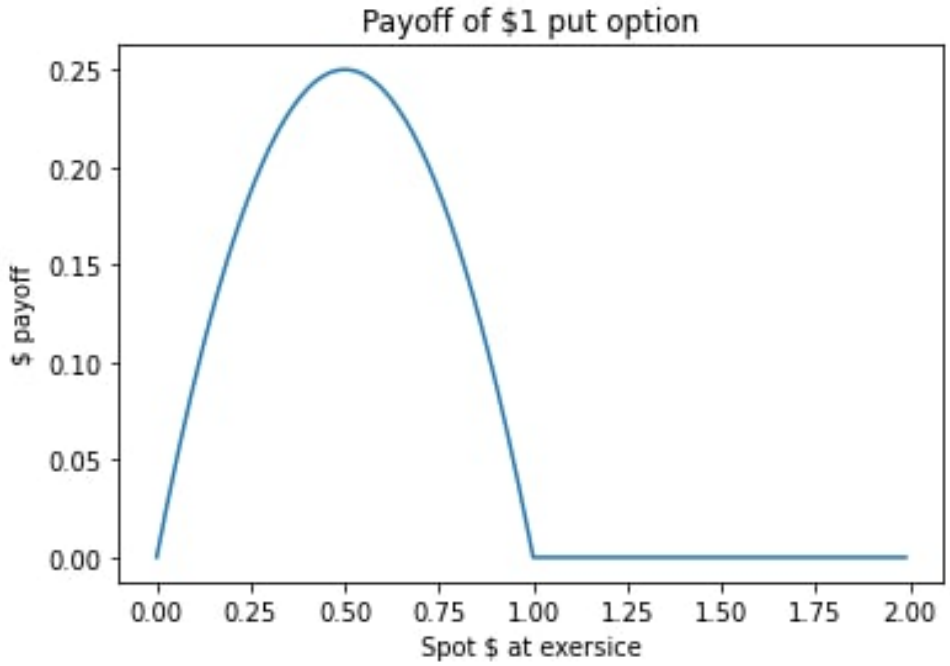
Issue #29	**The default value of the `vaultFee` is ignored completely due to the contract implementing the proxy pattern**
Severity	LOW SEVERITY
Location	Line 26 `uint256 public vaultFee = 100;`
Description	Proxy pattern contracts use the bytecode of the implementation address but the storage of the proxy address. In the case of Optix, each vault is deployed using the clones pattern where a minimal proxy is deployed that forwards calls to the immutable implementation address. However, any default storage variables like the `vaultFee` of 100 are only stored at the implementation address and do not exist within the proxy address. Therefore, all vaults are deployed with a default `vaultFee` of 0.
Recommendation	Consider moving the initialization of the `vaultFee` to the `initialize` function.
Resolution	RESOLVED

Issue #30	Collateralization ratio calculations are inverted compared to what they are supposed to be
Severity	LOW SEVERITY

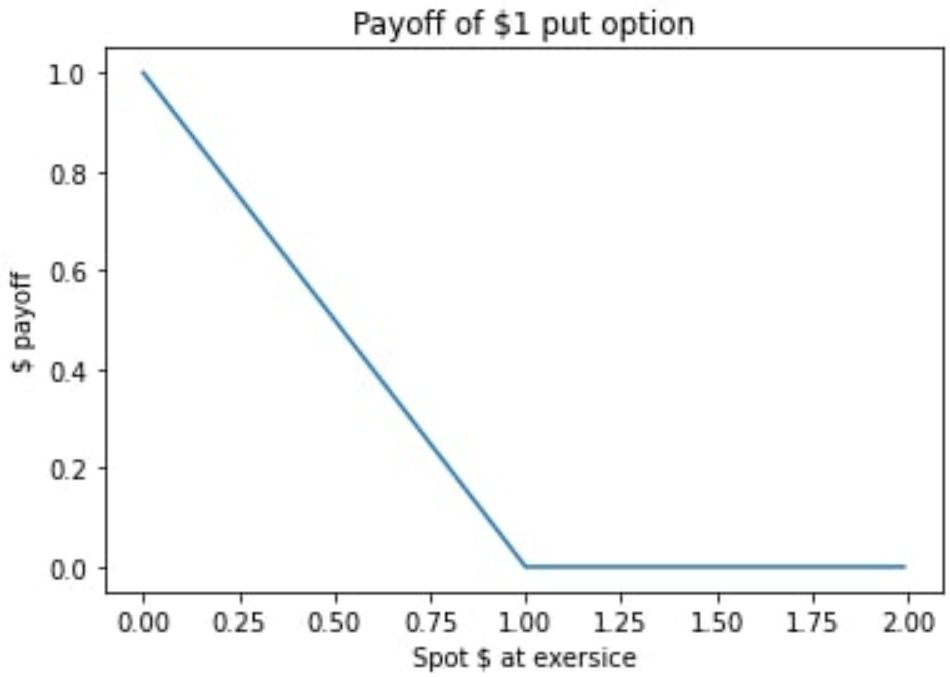
Location	Line 104
	`require(vaultCollateralTotal()+ (_collateralIn*factory.getCollateralizationRatio(this)/ 1e4)<=maxInvest,"OptionsVaultERC20: Max invest limit reached");`
	
	Line 280
	`return factory.getCollateralizationRatio(this)*collateralReserves/ 1e4;`

Description	The calculations which use the collateralization ratio multiply the reserves with the ratio to derive the actual allocatable balance. However, a lower collateralization ratio indicates that more balance should be allocatable so these calculations should divide by the collateralization ratio (with an optional zero check to handle division by zero explicitly).
Recommendation	Consider dividing by the collateralization ratio and handle the division by zero explicitly.
Resolution	RESOLVED

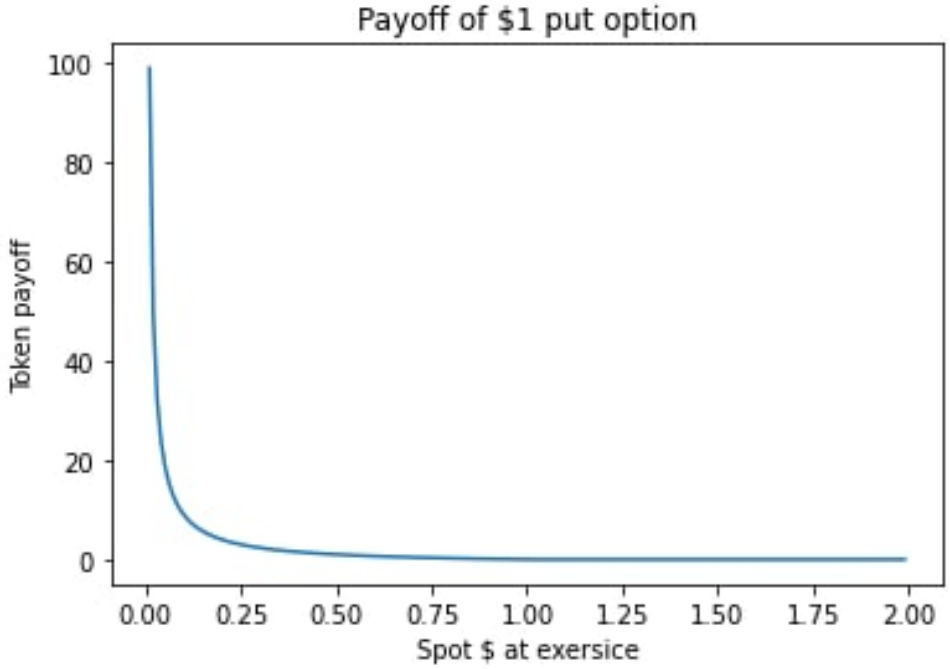
Issue #31	**`withdrawAndBurn` unnecessarily rounds the withdraw amount twice when it is called via send**
Severity	LOW SEVERITY
Description	The send function re-uses the `withdrawAndBurn` logic with a share input amount while it never actually burns these shares. This causes the functions to first convert the amount of tokens to withdraw into shares and then convert it back into tokens, unnecessarily incurring conversion rounding twice, further complicating the contract, and potentially introducing rounding risks.
Recommendation	Consider internalizing a function which does the necessary withdraw logic without the burn logic. This function can then be re-used within `unlock` and `withdraw`. `withdrawAndBurn` can likely be removed at this point.
Resolution	RESOLVED Payment logic is now handled in send.

Issue #32	Put options do not follow the traditional put option payoffs
Severity	LOW SEVERITY
Description	The protocol pays out put options in the currency of the collateral (the underlying asset of the option). This is a problem for put options on the underlying asset as it causes the payoff to be as follows:  This is because as the price of the asset drops, even though a user is eligible to receive more of that asset, their total value might decrease.

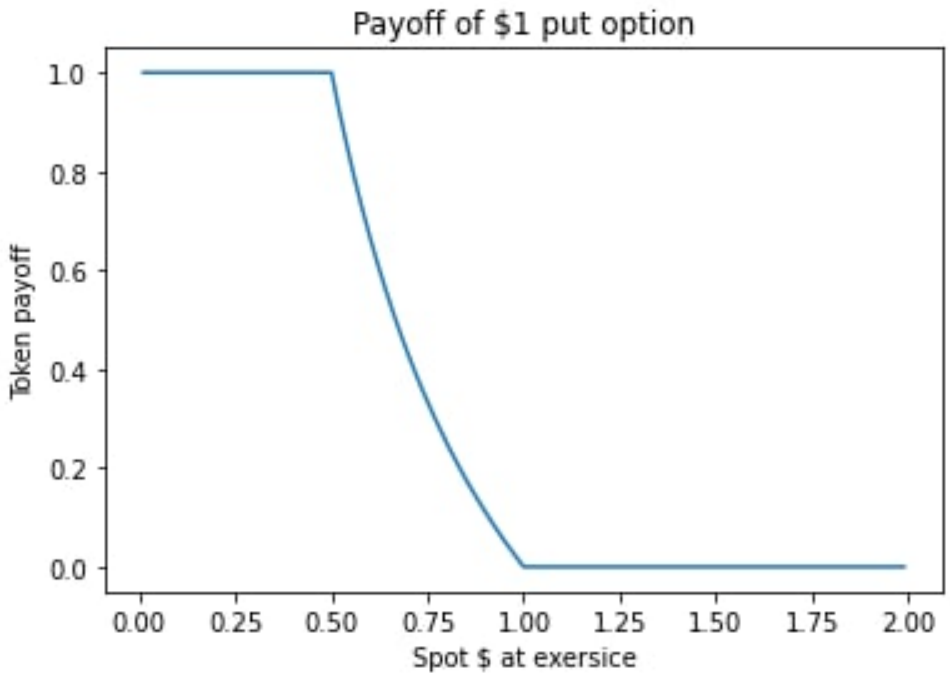
A correct put payoff should generally look as follows:



If the protocol wishes to pay such a payoff in the underlying tokens, they must follow a token payoff as follows to materialize it:



Recommendation Given that only one token is locked up as collateral, we recommend paying puts in the last pay-off graph, but capping token pay-off at 1 token:



Even better, we recommend doing away with puts altogether.

A BTC put option can be transformed into a USD call option at the same strike and expiry. The payoff of these two options should be identical. By only allowing for calls, we can still write out PUTs but these would be written out on the pricing currency's vault which ensures that the payout has enough collateral.

Resolution PARTIALLY RESOLVED

The client acknowledges that PUT payoffs do not work with the underlying as collateral; however, they plan to use an inverted oracle and the quotation unit as the vault for PUT options. We personally think put-call parity would have sufficed here and these could have been written with the normal oracle using call options.

Issue #33	**Silently checking for allowance on `initiateWithdraw` makes little sense**
Severity	LOW SEVERITY
Location	Line 134 `if (allowance(_account, address(this))>0){`
Description	The user must have explicitly allowed the contract for `initiateWithdraw` to be called. We do not see any benefit of this requirement, especially once this function is no longer callable for any user. Secondly, this should likely be a requirement and also check an explicit amount. Finally, if such a check remains, the allowance should likely also be decreased when the actual withdrawal occurs.
Recommendation	Consider simply removing the allowance check if it is not necessary.
Resolution	RESOLVED The section has been removed.

Issue #34	**`readonly` parameter still allows for collateral deposits**
Severity	LOW SEVERITY
Description	The contract contains functionality to disable option purchases in emergencies, which allows the vault owner to make the vault `readonly`. However, this functionality does not pause collateral deposits while this might also need to be paused in emergencies.
Recommendation	Consider also pausing collateral deposits if `readonly` is set.
Resolution	RESOLVED

Issue #35	The `setVaultFeeRecipient` function lacks an `address(0)` check
Severity	LOW SEVERITY
Description	As tokens are being sent to the `vaultFeeRecipient`, we recommend introducing a non-zero check as most token transfers fail to the zero address.
Recommendation	Consider checking that the address set as the `setVaultFeeRecipient` is not `address(0)`.
Resolution	RESOLVED

Issue #36	Vault token balances will be extremely high due to the configured decimals being too low
Severity	INFORMATIONAL
Description	When the first deposit occurs, the deposit amount is multiplied by `1e18` and this serves as the first ratio of vault tokens to deposit tokens. As the `OptionsVaultERC20` always has 18 decimals, this causes a vault deposit of 1 underlying token to be a huge deposit with a multiplier proportional to the underlying tokens decimals: - 1 USDC deposit = 1,000,000 vault tokens - 1 WETH deposit = 1,000,000,000,000,000,000 vault tokens
Recommendation	Consider overloading the `decimals()` function and returning `18 + underlying.decimals()` (this should be calculated once to avoid excessive gas usage of always calling the underlying token).
Resolution	RESOLVED Decimals are overloaded now to a larger value scaling with the underlying decimals.

Issue #37	Gas optimizations
Severity	INFORMATIONAL

Description	The contract contains multiple sections of code that could be further optimized for gas efficiency. We have consolidated these into a single issue in an effort to keep the report brief and readable.
	
	L133
	`function initiateWithdraw(address _account)  external`
	
	`withdrawInitiated[_account]` can be cached.
	
	L168 - 174
	`if (withdrawDelayPeriod>0){ // @audit gas - cache withdrawDelayPeriod and withdrawInitiated[_account]`
	`if ((withdrawInitiated[_account]==0) ||`
	`((block.timestamp < withdrawInitiated[_account] + withdrawDelayPeriod) ||`
	`(block.timestamp > withdrawInitiated[_account] + withdrawDelayPeriod + factory.withdrawWindow()))) {`
	`require(false, "OptionsVaultERC20: Invalid withdraw initiation date"); // @audit typo`
	`}`
	`}`
	
	`withdrawDelayPeriod` and `withdrawInitiated[_account]` can be cached to save gas.
	
	Line 180-181
	`collateralToken.approve(address(this),collateralOut);`
	`collateralToken.safeTransferFrom(address(this), _account, collateralOut);`
	
	Using `approve->transferFrom` flows to transfer out tokens is not done in general practice, even though it works. Consider replacing these two lines with the more appropriate `safeTransfer(address to, uint256 amount)` method.

	L443 `function setIpfsHash(string memory _value) external` `_value` can be provided as `calldata` to save gas. The `getVaultId()` return value never changes. Consider providing it to the initializer and simply storing it in a storage variable to save gas.
Recommendation	Consider implementing the gas optimizations mentioned above.
Resolution	RESOLVED

Issue #38	**Contract does not use upgradeable OpenZeppelin dependencies**
Severity	INFORMATIONAL
Description	The contract is deployed as a minimal proxy clone. This means that the `constructor()` of the various dependencies will never be executed on the various vaults. OpenZeppelin has developed adjustments to their contracts that take this into account and instead expose initialization functions but the contract still uses the traditional dependencies with constructors. This issue is marked as informational as we could not find any serious risks of not calling the constructor within the OpenZeppelin dependency versions used within this contract.
Recommendation	Consider inheriting the upgradable alternatives to the OpenZeppelin dependencies and initializing them.
Resolution	ACKNOWLEDGED

Issue #39	Typographical errors
Severity	INFORMATIONAL

Description	We have consolidated several typographical errors into a single issue to keep the report more readable.
	
	The `initialize` function initializes various variables to their already set default value: `collateralReserves`, `lockedCollateralCall`, `lockedCollateralPut`, `ipfsHash` and `readOnly`. All of these variables do not need to be explicitly initialized to their default variable. Removing their initialization simplifies the contract for readers experienced with the accepted solidity practices.
	
	Lines 83, 94, 132, 152, 159, 192, 214, 233, 285, 292
	`* @nonce`
	
	The `nonce` natspec tag does not exist and should be `@notice`.
	
	Line 85
	`* @param vaultId Pool to provide to`
	
	This parameter is not defined.
	
	Line 86
	`* @param collateralIn Amount to deposit in the collatoral token`
	
	This comment should say `collateral` instead. All of these parameters actually start with an underscore as well.
	
	Line 144 & 172
	`require(false, "OptionsVaultERC20: Invalid withdraw initiation date");`
	`require(false, "OptionsVaultERC20: Invalid withdraw initiation date");`
	
	This structure can be simplified to explicitly require each of the individual cases or at the very least use a `revert` clause instead of `require(false, “”)`.

Line 237

```
require(_to != address(0));
```

This requirement lacks an explicit revert message.

Line 304

```
require(_msgSender()==vaultOwner, "OptionsVaultERC20: must have owner role");
```

The owner is not a role. This can remain unchanged however but we just wanted to inform the project in case they desire the owner to be a role.

Line 322

```
require(!OptionsLib.boolStateIsTrue(withdrawDelayPeriodLocked),"OptionsVaultERC20: setting is immutable");
```

Line 453

```
require(!OptionsLib.boolStateIsTrue(oracleEnabledLocked),"OptionsVaultERC20: setting is immutable");
```

The setting in the above two lines do not need to be `immutable`, they are just locked.

`provide`, `withdraw`, `lock`, `unlock`, `send`, `isOptionValid`, `vaultUtilization` can all be made `external`. Marking functions as external improves code readability as it communicates that the contract itself does not use them.

Recommendation Consider fixing the typographical errors.

Resolution RESOLVED

Issue #40	The protocol does not support tokens with a fee on transfer and tokens that rebase
Severity	INFORMATIONAL
Description	Due to the way the protocol handles token balances and especially deposits, collateral tokens with a fee on transfer are not supported. The counterparty token (the one in which the options are priced) can have such behaviors as this token is never transferred in/out of the protocol.
Recommendation	Carefully consider the behavior of each token when adding them to the protocol.
Resolution	RESOLVED The client does not plan to support such tokens in any way and understands this limitation.

2.4 OptionsVaultFactory

`OptionsVaultFactory` is the central contract for deploying and managing option collateral vaults. As discussed within the `OptionsVaultERC20` section of this report, collateral vaults are responsible for maintaining collateral to underwrite options, pricing these options and taking care of the relevant payouts. They are the central contracts within the Optix ecosystem and are completely independent from each other.

OptionsVaultFactory's most important function is the `createVault` function which allows anyone to deploy a new options vault. Each vault must have a configured collateral token which is the underlying asset of the options of that vault. The vault will then pay out option payoffs using these tokens.

The admin of the vault factory has various capabilities that might affect deployed vaults. The factory admin can whitelist permitted oracles, permitted collateral tokens and whether who is allowed to create vaults. All of these whitelist features can eventually be permanently disabled, opening the protocol up for anyone to do as they please.

2.4.1 Privileges

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

- `initialize [ DEFAULT_ADMIN_ROLE, callable once ]`
- `createVault [ CREATE_VAULT_ROLE or anyone after whitelist disabled ]`
- `setCreateVaultIsPermissionlessImmutable [ DEFAULT_ADMIN_ROLE ]`
- `setOracleIsPermissionlessImmutable [ DEFAULT_ADMIN_ROLE ]`
- `setCollateralTokenIsPermissionlessImmutable [ DEFAULT_ADMIN_ROLE ]`
- `setOracleWhitelisted [ DEFAULT_ADMIN_ROLE ]`
- `setCollateralTokenWhitelisted [ DEFAULT_ADMIN_ROLE ]`
- `setCollateralizationRatioBulk [ COLLATERAL_RATIO_ROLE ]`
- `setCollateralizationRatio [ COLLATERAL_RATIO_ROLE ]`
- `grantRole [ DEFAULT_ADMIN_ROLE ]`
- `revokeRole [ DEFAULT_ADMIN_ROLE ]`
- `renounceRole [ role owner ]`

2.4.2 Issues & Recommendations

Issue #41	**Governance risk: Governance can reduce the collateralization ratio of vaults to zero which can pose significant risks for people who have active options purchased**
Severity	HIGH SEVERITY
Description	At the time of this audit, the Optix governance can freely reduce the collateralization ratio of a vault which means it is able to sell more options than it could theoretically pay back if all are exercised at their maximum value. This poses a significant risk to users who have purchased options from a properly collateralized vault as governance can at any point in time step in, decrease the collateralization ratio to zero, and write out options with a shorter maturity that, if exercised, would take away all collateral and prevent the original option holders from exercising.
Recommendation	Consider handling the collateralization ratios at the vault contract level and by the contract owner. These should be similarly lockable to the other contract variables and we highly encourage this to be locked at 100% as under collateralization introduces significant risks.
Resolution	FAILED RESOLUTION The collateralisation ratio is only required to be strictly greater than 10%.

Issue #42	Lack of validation
Severity	LOW SEVERITY

Description	The contract contains functions with parameters which are not properly validated. Having unvalidated parameters could allow the governance or users to provide variable values which are unexpected and incorrect. This could cause side-effects or worse exploits in other parts of the codebase.
	
	Consider validating the following function parameters:
	
	Line 15
	`mapping(address => uint256) public vaultId;`
	
	The `public` function for this mapping is used within the codebase. However, this `public` function does not actually validate that the vault exists. We recommend making the mapping `internal` and adding a validated external function which reverts if the provided address does not exist. Otherwise, there is no way to differentiate between a vault with the vaultId being zero and a non-existent vault.
	
	A similar argument could be made for the `collateralizationRatio` ratio although this is less severe as there is a `public` function for this mapping already.
	
	Line 49
	`function createVault(address _owner, IOracle _oracle, IERC20 _collateralToken, IFeeCalcs _vaultFeeCalc) public returns (address){`
	
	`_vaultFeeCalc` should likely be whitelisted.
	
	The contract should validate that the `_oracle` relates to the `_collateralToken` and not to a different token pair.
	
	Line 132
	`function setCollateralizationRatio(OptionsVaultERC20 _address, uint256 _ratio) public {`
	
	The collateralization ratio is presently completely unvalidated and can take any value including 0. We do understand that this might be desired.

Recommendation	Consider validating the function parameters mentioned above.
Resolution	PARTIALLY RESOLVED This issue was partially resolved but the resolution is flawed — the first vault created cannot be used properly as it has the id 0.

Issue #43	**Typographical errors**
Severity	INFORMATIONAL

Description	We have consolidated the typographical errors into a single issue in an effort to keep the report size reasonable.
	
	Line 28
	`uint256 public withdrawWindow = 2 days;`
	
	Consider if it is really preferred to have the `withdrawWindow` constant be defined within the `OptionsVaultFactory` instead of simply within the `OptionsVaultERC20`.
	
	Line 33
	`bytes32 public constant CONTRACT_CALLER_ROLE = keccak256("CONTRACT_CALLER_ROLE");`
	
	The `CONTRACT_CALLER_ROLE` is unused.
	
	Line 44
	`if (optionsContract == address(0)){`
	
	Writing this `if`-statement as a requirement would be cleaner.
	
	Line 65
	`emit UpdateOracle(_oracle, vaults.length, true, _collateralToken, _oracle.decimals(), _oracle.description());`
	
	We do not understand the benefit of this event since this data is pretty much encoded within the other event.

Lines 76, 83, 90

```
function setCreateVaultIsPermissionlessImmutable(BoolState
_value) public {
function setOracleIsPermissionlessImmutable(BoolState
_value) public {
function
setCollateralTokenIsPermissionlessImmutable(BoolState
_value) public {
```

These functions are misnamed as they do not necessarily set the immutability of the boolean.

Line 133

```
require(hasRole(COLLATERAL_RATIO_ROLE, _msgSender()),
"OptionsVaultFactory: must have admin role");
```

The reversion message is erroneous as this function actually requires the collateral ratio role.

`createVault`, `vaultsLength`, `setCreateVaultIsPermissionlessImmutable`, `setOracleIsPermissionlessImmutable`, `setCollateralTokenIsPermissionlessImmutable`, `setOracleWhitelisted`, `setCollateralTokenWhitelisted`, `getCollateralizationRatio` and `setCollateralizationRatioBulk` can all be made `external`. Marking functions as `external` improves code readability as it communicates that the contract itself does not use them.

Recommendation Consider fixing the typographical errors.

Resolution RESOLVED

`vaultId` and `setCollateralizationRatio` functions can be made immutable.

Issue #44	Gas optimizations
Severity	INFORMATIONAL

Description	Throughout the contract, we have identified various sections that can be optimized for gas usage:
	
	Line 18
	`address public optionVaultERC20Implementation;`
	
	The `optionVaultERC20Implementation` variable can be made immutable.
	
	Line 28
	`uint256 public withdrawWindow = 2 days;`
	
	`withdrawWindow` can be made constant.
	
	Lines 64 - 67
	`emit CreateVault(vaults.length, _oracle, _collateralToken, vault);`
	`emit UpdateOracle(_oracle, vaults.length, true, _collateralToken, _oracle.decimals(), _oracle.description());`
	
	`vaultId[vault] = vaults.length;`
	
	`vaults.length` value should be cached to save gas.
	
	Lines 78 - 79
	`require(OptionsLib.boolStateIsMutable(createVaultIsPermissionless),"OptionsVaultFactory: setting is immutable");`
	`emit SetGlobalBoolState(_msgSender(),SetVariableType.CreateVaultIsPermissionless, createVaultIsPermissionless, _value);`
	
	`createVaultIsPermissionless` should be cached to save gas.

Lines 86 - 87

```
require(OptionsLib.boolStateIsMutable(oracleIsPermissionless
),"OptionsVaultFactory: setting is immutable");
emit
SetGlobalBoolState(_msgSender(),SetVariableType.OracleIsPerm
issionless, oracleIsPermissionless, _value);
```

`oracleIsPermissionless` should be cached to save gas.

Lines 92 - 93

```
require(OptionsLib.boolStateIsMutable(collateralTokenIsPermi
ssionless),"OptionsVaultFactory: setting is immutable");
emit
SetGlobalBoolState(_msgSender(),SetVariableType.CollateralTo
kenIsPermissionless, collateralTokenIsPermissionless,
_value);
```

`collateralTokenIsPermissionless` should be cached to save gas.

Line 133

```
require(hasRole(COLLATERAL_RATIO_ROLE, _msgSender()),
"OptionsVaultFactory: must have admin role");
```

This check is done multiple times within the bulk setter. It might make sense to keep the role check on the `external` functions and call an internal function without a role check. We are fine with this remaining as-is as well.

Recommendation Consider implementing the above suggestions to optimize gas usage. Given that most of them are just governance functions and not targets for extreme gas optimization, acknowledgement is also more than acceptable.

Resolution PARTIALLY RESOLVED

`createVaultIsPermissionless`, `oracleIsPermissionless` and `collateralTokenIsPermissionless` are not cached.

Issue #45	`createVault` does not strictly adhere to checks-effects-interactions
Severity	INFORMATIONAL
Description	The `createVault` function does not adhere to checks-effects-interactions: this increases reentrancy risks. This issue is marked as informational because the current code-layout is desired in our opinion and because there's presently no way to exploit this with the current implementation of the vault. However, we still want to draw attention to this fact and recommend an additional safeguard to future-proof this function.
Recommendation	Consider adding a `nonReentrant` modifier using OpenZeppelin's `ReentrancyGuard` dependency as an additional safeguard.
Resolution	RESOLVED A reentrancy guard was added.

Issue #46	Lack of events for and `initialize` and `setCollateralizationRatio`
Severity	INFORMATIONAL
Description	Functions that affect the status of sensitive variables should emit events as notifications.
Recommendation	Add events for the above functions.
Resolution	PARTIALLY RESOLVED `initialize` still emits no events.

2.5 SimpleSeller

`SimpleSeller` is used to calculate and return the price of an option that will be purchased by users. Any vault owner or operator can update the different parameters to change the price of the options of the given vault which means that any vault owner and operator can potentially reduce this price to near zero to write options for free (which is a governance risk).

`OptionERC721` will use the `SimpleSeller` to return the intrinsic, extrinsic, and vault fees.

The intrinsic fee is calculated using the option type, the strike price, the current price, and the option size. Any option out of the money will not have any intrinsic fee as they have no intrinsic value. The more in the money an option is, the more intrinsic value it has and the bigger the intrinsic fee which is directly proportional and equal to the depth of that option being in-the-money at the time of underwriting.

The extrinsic fee represents the portion of the option worth that has been assigned by factors other than the underlying asset's price. The primary factors considered with extrinsic value are the time remaining and the implied volatility. It also applies a factor that adjusts the fee based on the utilization of the vault.

The function uses the current price and period and performs a sequential search to find the first matching period/strike cell in the price point matrix and then returns the fee for that applying a vault utilization factor to it.

The behavior of the extrinsic fee function for calls is as follows:

- `findStrikeAndMatchPeriod`
  - If the strike is less than or equal to the current price, use the 0 strike; otherwise calculate the strike percentage away from the current price
  - Find the first registered period that is close to the period we are searching for

- `findMatchFee`
  - Iterate through the price points array for the period located
  - Find the first strike that is that close to the strike we are searching for
  - Return the matched fee multiplied by the factor
- `getFactor`
  - calculate what the vault utilization is now and what it will be including if this option size is sold
  - calculate the average factor to apply by adding the factors up between the start utilization and the end and divide by the count (the factor represents the penalty or discount on the fee based on the utilization)

The vault fee is defined in the OptionsVaultERC20 itself.

All the different fees are returned as percentages in basis points, meaning that 1 is 0.01% and 10 000 is 100%.

2.5.1 Privileges

The following functions can be called by the following priviledged roles of the contract:

- `setFactor [ vault owner or operator ]`
- `deletePricePoints [ vault owner or operator ]`
- `pushPricePoints [ vault owner or operator ]`

2.5.2 Issues & Recommendations

Issue #47	The `getFactor` function is severely flawed
Severity	HIGH SEVERITY

Description	Several inputs will make the `getFactor` function revert.
	
	If the `startUtil` and `endUtil` are between `]factorRange * i; factorRange * (i+1)[`, the `factorSum` would be 0. Hopefully, the function would revert as the `factorCnt` would also be 0, but this is not the expected behavior.
	
	When the utilization of a vault would be close to the maximum, or to be precise, greater than `10_000 - factorRange`, the function would systematically revert.
	
	L209 - 218
	`uint factorRange = 1e4/factorArray.length;`
	`uint factorSum;`
	`uint factorCnt;`
	`for(uint i=0; i<factorArray.length; i++){`
	`if((factorRange*i>=startUtil)&& (factorRange*i<=endUtil))`
	`{`
	`factorSum += factorArray[i];`
	`factorCnt += 1;`
	`}`
	`}`
	`return factorSum/factorCnt;`
	
	Lastly, the `factorRange` is rounded down unnecessarily.
	
	`getFactor` is flawed in the following ways in summary:
	- The `factorRanges` do not span the whole 100% utilization range and this causes the vault to never allow to be fully allocated.
	- The `factorRanges` are unnecessarily rounded down, causing the function to return imprecise results, especially for larger utilizations.
	- The average factor often rounds in favor of the user which is bad practice.

Recommendation Consider fixing the `getFactor` function:

```
for(uint256 i; i<factorArray.length; i++){
    if((i*1e4>=startUtil*(factorArray.length-1))){
                factorSum += factorArray[i];
                factorCnt += 1;
               if (i*1e4>endUtil*(factorArray.length-1))
break;
    }
}
```

Notice the following changes:

- `factorArray.length-1`: This is done to span the whole 100% utilization range, otherwise there is one element too few to span the utilization range.
- In-line multiplication instead of division beforehand: We moved the division factor in-line first to reduce the rounding error and then move it to the other side of the equation to fully mitigate the rounding error.
- The first factor outside of the utilization window is still included: As long as factors strictly increase with greater utilization (this should be enforced), this new logic ensures that the factor will always be rounded against the favor of the user by including the first factor outside of the utilization range. Otherwise, the user might be able to game the system by picking an option size just before the next utilization factor.

Resolution RESOLVED

Issue #48	`callPrices` or `putPrices` cannot be reset
Severity	HIGH SEVERITY
Description	The `callPrices` or `putPrices` arrays cannot be reset and are always increasing.
Recommendation	Consider deleting the `callPrices` and `putPrices` arrays within the `pushPricePoints` function. Note that deletion can consume `O(N)` gas in certain cases with `N` being the number of elements within the array. The client should be mindful of gas usage with many elements.
Resolution	RESOLVED

Issue #49	The different periods may not be in decreasing order while strikes and factors might not be in increasing order
Severity	MEDIUM SEVERITY

Location	L178 - 184
	`for(uint i=0; i<periods.length; i++){`
	`  if (lastPeriod != periods[i]){`
	`  callPeriods[vaultId][oracle].push(periods[i]);`
	`  lastPeriod = periods[i];`
	`  }`
	`  callPrices[vaultId][oracle]`
	`[lastPeriod].push(PricePoint(strikes[i],fees[i]));`
	`}`

Description	The `pushPricePoints` function does not validate that the different inputs are in decreasing order. If these parameters are set in any other order, the functions used to find parameters inside these arrays will misbehave and may return wrong values. `periods`, `strikes` and `factors` must all be in an either non-decreasing or non-increasing order for the contract to behave as expected. Strictly speaking, `factors` is the only element which does not need to adhere to ordering but we recommend to enforce this as well as it does not make sense to price options more cheaply as utilization increases.
Recommendation	Consider asserting that the `periods` and `strikes` are added in the correct order. One could use the `lastPeriod` value and add a `lastStrike` to check and revert on values that are not decreasing or increasing. A similar methodology should be used for the factors.
Resolution	RESOLVED

Issue #50	The pricing often rounds in the user's favor
Severity	MEDIUM SEVERITY
Description	Rounding in the favor of the user is a strict anti-pattern within Solidity as it might allow for exploitation in extreme cases where a user is able to receive something with a higher expected value than what they paid for. Within the `SimpleSeller`, the match that is found is always the one which is slightly cheaper than the actual option parameters which means that the user is incentivized to put their parameters right before the next price "quadrant" and pay too little for their option. Finally, rounding with basis-point precision might be too imprecise for near free options as the price might round to zero (this should be validated to not be the case in our opinion) or from 2 to 1 basis points which would be a 50% price reduction. This is only added informationally since the vaultFee is a large constant and trumps this rounding error.
Recommendation	Consider adjusting the fee match to find the strike closest but smaller than the provided strike (as this has a greater fee) and the period closest but larger than the provided period (as this has a greater fee). Consider adjusting the `getFactor` logic to strictly round against the favor of the user, as is explained in another issue.
Resolution	PARTIALLY RESOLVED The fee is now the weighted average between the previous and the next one. In our opinion, the fee should always be the bigger one. This would make the whole contract more efficient and easier to understand.

Issue #51	**The `findStrikeAndMatchPeriod` function does not always revert on a currentPrice of 0**
Severity	LOW SEVERITY
Description	A current price of 0 would mean that the oracle is broken and cannot be trusted. The `findStrikeAndMatchPeriod` function will not always revert if the `currentPrice` is 0.
Recommendation	Consider reverting with a proper message if the `currentPrice` is 0.
Resolution	RESOLVED

Issue #52	**The `findMatchFee` function does not revert if an incorrect `optionType` is provided**
Severity	LOW SEVERITY
Location	L97 `require (true,"No fee calculated");`
Description	The `findMatchFee` function does not revert if the `optionType` is not a Put or a Call. This is due to the requirement being incorrect. This issue is marked as low severity as the validation is however done within `getExtrinsicFee` — the requirement is simply incorrect.

Recommendation	Consider fixing the requirement statement to:
	
	`require (false,"No fee calculated");`
	
	Additionally, using a revert statement would be much cleaner:
	
	`revert("No fee calculated");`

Resolution	RESOLVED

Issue #53	The different stored arrays lack a getter for their length
Severity	LOW SEVERITY
Description	To query the different arrays, the length of an array needs to be known by users or any protocol. Without it, they need to query the array until it reverts blindly.
Recommendation	Consider adding a length function for the `callPeriods`, `callPrices`, `callFactor`, `putPeriods`, `putPrices` and `putFactor` arrays.
Resolution	RESOLVED

Issue #54	Lack of events for the `setFactor`, `deletePricePoints` and `pushPricePoints` functions
Severity	INFORMATIONAL
Description	Functions that affect the status of sensitive variables should emit events as notifications.
Recommendation	Add events for the above functions.
Resolution	RESOLVED

Issue #55	**`setFactor`, `deletePricePoints` and `pushPricePoints` can be made `external`**
Severity	INFORMATIONAL
Description	Functions that are not used within the contract but only externally can be marked as such with the external keyword. 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 marking the functions mentioned above as `external`.
Resolution	PARTIALLY RESOLVED The `setFactor` function is still public.

Issue #56	Lack of validation
Severity	INFORMATIONAL
Description	The contract contains functions with parameters which are not properly validated. Having unvalidated parameters could allow the governance or users to provide variable values which are unexpected and incorrect. This could cause side-effects or worse exploits in other parts of the codebase. The different fees returned by the different getters can be tremendously huge. Consider validating the parameters of the `setFactor` and `pushPricePoints` functions so the maximum fee is still reasonable. As mentioned before, `order` should be validated. `setFactor`, `deletePricePoints` and `pushPricePoints` do not check the `optionType`. Line 59 `require (callPeriods[vaultId][oracle].length>0,"No periods for this vault->oracle");` This check should check the `putPeriods` in case the provided type is the `Put` type. Factors should be strictly greater than zero when set, just like the match fees.
Recommendation	Consider validating the function parameters mentioned above.
Resolution	PARTIALLY RESOLVED Only the `length` check was added.

Issue #57	Typographical errors
Severity	INFORMATIONAL

Description	We have consolidated the typographical errors into a single issue to keep the report brief.
	
	L65
	`function findMatchFee(uint256 findStrike, uint256 matchPeriod, uint256 period, uint256 optionSize, uint256 strike, uint256 currentPrice, IStructs.OptionType optionType, uint vaultId, IOracle oracle) internal view returns(uint256)`
	
	`findMatchFee` does not use its period and `currentPrice` parameters. Consider removing them for clarity and better UX.
	
	L154 - 159
	`if (optionType == OptionType.Call){ callFactor[vaultId][oracle] = factors; } else{ putFactor[vaultId][oracle] = factors; }`
	
	Any other `OptionType` will pass the second check. For example, the invalid option type would change the `putFactors`.
	
	L162
	`function deletePricePoints(uint vaultId, IOracle oracle, OptionType optionType) public`
	
	The `optionType` parameter is not used.
	
	The invalid option type would still be valid for `deletePricePoints`.
	
	L66
	`uint matchStrike = 0;`
	
	This parameter is not used.

	`msg.sender` is used while the rest of the codebase uses `_msgSender()`. This is inconsistent and would prevent the usage of meta-transaction platforms.
Recommendation	Consider fixing the typographical errors.
Resolution	RESOLVED

Issue #58	**Gas optimizations**
Severity	INFORMATIONAL

Description	The contract contains multiple sections of code that could be further optimized for gas efficiency. We have consolidated these into a single issue in an effort to keep the report brief and readable.
	
	L12
	`OptionsVaultFactory public factory;`
	
	The factory variable can be casted as immutable in order to save gas.
	
	L71 - 78
	`for(uint i=0; i<callPrices[vaultId][oracle][matchPeriod].length; i++){ if (callPrices[vaultId][oracle][matchPeriod][i].strike >= findStrike){ matchStrike = callPrices[vaultId][oracle][matchPeriod][i].strike; matchFee = callPrices[vaultId][oracle][matchPeriod][i].fee; foundMatch = true; break; } }`
	
	The `callPrices[vaultId][oracle][matchPeriod].length` and `callPrices[vaultId][oracle][matchPeriod][i].strike` can be cached to save gas.

L152, 168, 201

```
function setFactor(uint vaultId, IOracle oracle, OptionType
optionType, uint256[] memory factors) public

function pushPricePoints(uint vaultId, IOracle oracle,
OptionType optionType, uint[] memory periods, uint[] memory
strikes, uint[] memory fees) public

function getFactor(uint vaultId, uint256[] memory
factorArray, IOracle oracle, uint256 optionSize) public view
returns(uint)
```

The different arrays are provided as `memory`. In order to save gas, they can be marked as calldata.

The `deletePricePoints` function does not use the `optionType` parameter.

Recommendation	Consider implementing the gas optimizations mentioned above.
Resolution	PARTIALLY RESOLVED

2.6 Referrals

`Referrals` is used by the `OptionsERC721` contract to keep track of a user's referrals. During any option purchase, the purchaser might pay a small referral fee to their referral.

The Referrals contract is rather advanced. The protocol can specify a default referral address which is likely the protocol itself. All referrals revert back to the default referral address after 26 weeks. This means that eventually, the protocol will earn the referral income of everyone.

The contract also allows for the contract admin to blacklist specific users from being the referral of others. Once a user is blacklisted, existing referees will unset back to the default referrer once they purchase another option. This can be avoided in case the user sets a new referrer during this purchase, which would re-lock the new referrer for 26 weeks.

Any wallet or contract with the `CONTRACT_CALLER_ROLE` privilege can set referrals for users.

The client has indicated that they plan to potentially move to a more elaborate referral mechanism in the future, so users should note that this description might eventually become outdated compared to the referral contract that is used in production.

2.6.1 Privileges

The following functions can be called by the owner and other privileged roles of the contract:

- `setReferralFeeRecipient [ DEFAULT_ADMIN_ROLE ]`
- `setBlacklisted [ DEFAULT_ADMIN_ROLE ]`
- `setBlacklistedAll [ DEFAULT_ADMIN_ROLE ]`
- `grantRole [ DEFAULT_ADMIN_ROLE ]`
- `revokeRole [ DEFAULT_ADMIN_ROLE ]`
- `renounceRole [ role owner ]`
- `initialize`
- `setReferralPeriod [ DEFAULT_ADMIN_ROLE ]`

2.6.2 Issues & Recommendations

Issue #59	**Governance risk: Governance can override referrals at any point in time and gives themselves this privilege by default**
Severity	LOW SEVERITY
Location	Line 27 `_setupRole(CONTRACT_CALLER_ROLE, _msgSender());`
Description	The governance can grant any contract or wallet privileges to set referrals for users. This means that they could easily grant themselves privileges to override anyone's referral to their own wallet and steal referral fees. Additionally, the `CONTRACT_CALLER_ROLE` is granted to the deployer. This allows the deployer to, by default, set the referrals of anyone. We do not see the point or value of this as this should be done by the OptionsERC721 contract. This issue is marked as low severity as this only affects the price of options during creation. Existing contracts or vault suppliers remain unaffected.
Recommendation	Consider using an Enumerable alternative to `AccessControl` to easily allow users to inspect who has admin privileges. Consider only granting admin privileges to a multisig. Consider not granting the deployer the `CONTRACT_CALLER_ROLE` by default.
Resolution	RESOLVED The options contract must now be initialized once.

Issue #60	First address can be added multiple times to the referrers array
Severity	LOW SEVERITY
Description	The contract keeps track of a list of referrer ids. Each referrer (either active or inactive) has an `id` which can be used for frontend purposes. To ensure that a user is not added twice to the referrer array, the `id` in the `address=>id` mapping must still be zero. However, as ids start from `id` zero, this would allow for the first referrer to be added twice.
Recommendation	Consider whether number indexation is really necessary. Indexing by address might be sufficient. In this case, an `EnumerableSet` would suffice and would be able to list all unique referrer addresses. Alternatively, referrer ids could start from `id` 1 (by executing the push before the set) or the deployer can be added as the first referrer to circumvent the issue.
Resolution	RESOLVED The deployer is immediately added as a referrer.

Issue #61	The `setReferralFeeRecipient` function lacks an `address(0)` check
Severity	LOW SEVERITY
Description	Currently, only the constructor checks that the `referralFeeRecipient` is not `address(0)`. This should also be checked inside the `setReferralFeeRecipient` function as it could prevent users from calling `create` on the `OptionsERC721` contracts.
Recommendation	Consider checking that the address set as the `referralFeeRecipient` is not `address(0)`.
Resolution	RESOLVED

Issue #62	Gas optimizations
Severity	INFORMATIONAL
Description	The contract contains multiple sections of code that could be further optimized for gas efficiency. We have consolidated these in a single issue in an effort to keep the report brief and readable. `referFee` and `referralPeriod` can be made `constant`. The `referredBy[holder]` value is queried multiple times inside the `captureReferral` function. This value can be cached to save some gas. The `setBlacklistedAll` function checks that the `msg.sender` has the `DEFAULT_ADMIN_ROLE` in each iteration. Consider using an `internal` function instead so the role is checked only once to save gas. Line 72 `emit AddReferrer(value, referrerId[value]);` The `referrerId` can be cached as `referrals.length` to avoid fetching it from memory again. The `captureReferral` function appears to always update `referredDate[holder]` twice to `block.timestamp` in case the holder had blacklisted the `referredByIn`.
Recommendation	Consider implementing the gas optimizations mentioned above.
Resolution	PARTIALLY RESOLVED `setBlacklistedAll` and the event on line 72 have been optimized.

Issue #63	Typographical errors
Severity	INFORMATIONAL
Description	- `captureReferral` can be made `external`. - Consider whether it is desirable for `addReferrer` to be `public` and be callable on any wallet.
Recommendation	Consider fixing the typographical errors.
Resolution	RESOLVED

2.7 OptionsLib

The `OptionsLib` dependency is a library used that defines a special boolean type which can take on four values.

- `FalseMutable`
- `TrueMutable`
- `FalseImmutable`
- `TrueImmutable`

These booleans are used throughout the system to define temporarily configurable functionality. For example: the governance can configure whether anyone can create an Options vault with one of these booleans within the `OptionsVaultFactory`. Once the boolean is made immutable, it can never be changed. This means that the governance can permanently make the vaults accessible to be created by everyone.

This mechanism aligns with Optix's ambitions to eventually become a fully decentralized platform.

2.7.1 Issues & Recommendations

Issue #64	**Code style improvement: Attaching the library to the `BoolState` struct**
Severity	INFORMATIONAL

Description	As the `OptionsLib` defines various functions which start with the `IStructs.BoolState` type, the client is free to simplify the codebase by attaching the `OptionsLib` to the type.
	
	`using OptionsLib for IStructs.BoolState;`
	
	By doing so, all functions become accessible within the `BoolState` types:
	
	`myBool.boolStateIsTrue();`

Recommendation	Consider implementing the recommendation above. We also recommend shortening the function names as they contain redundant information. The `boolStateIsTrue` name can be shortened to `isTrue` or `value` while `boolStateIsMutable` can be shortened to `isMutable` or `mutable`.
Resolution	RESOLVED

2.8 Interfaces

The Interfaces file serves as a dependency including all definitions and base OpenZeppelin dependencies used throughout the codebase. It is imported by most of the contracts within the Optix codebase.

2.8.2 Issues & Recommendations

Issue #65	Unused portions of code
Severity	INFORMATIONAL

Location	Lines 15-18
	`// import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol"; //`
	`// import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/utils/SafeERC20.sol";`
	`// import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/ERC721.sol";`
	`// import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/AccessControl.sol";`

Description	The code contains sections of code which are not in use. This unnecessarily makes the contract more verbose.
Recommendation	Consider removing the unused portions of code.
Resolution	RESOLVED

Issue #66	`IStructs` interface is too broad which might cause unused events to show up in the contracts their ABI specification
Severity	INFORMATIONAL
Description	Presently the `IStructs` interface is inherited by nearly all contracts. However, it contains events which are not used by all contracts and therefore unnecessarily appear within the ABI of these contracts.
Recommendation	Consider avoiding having such a broad interface and making an interface folder which defines the narrow interfaces of each contract. There can still be inheritance of common structs and components of these interfaces.
Resolution	ACKNOWLEDGED

Issue #67	The different interfaces do not define the functions of their contracts
Severity	INFORMATIONAL
Description	Currently the different interfaces of the contracts do not define the function of the contracts. As the best practice is to import the interface of a contract instead of the whole contract when interacting with it, the interface needs to define the different functions with their parameters and their returned values.
Recommendation	Consider adding the different functions of their contract without their implementation on the interface to allow people to interact with those contracts by only importing their interfaces. This would also allow for resolving the global issue where oftentimes the whole contracts are imported within the codebase themselves.
Resolution	RESOLVED


PALADIN
BLOCKCHAIN SECURITY