Document potential edge cases for hook receiver contracts: The functions *withdrawTokenAndCall*() and *withdrawTokenAndCallOnBehalf*() make a call to a hook contract designated by the owner of the withdrawing stealth address. There are very few constraints on the parameters to these calls in the Umbra contract itself. Anyone can force a call to a hook contract by transferring a small amount of tokens to an address that they control and withdrawing these tokens, passing the target address as the hook receiver.

a. Recommendation: Developers of these *UmbraHookReceiver* contracts should be sure to validate both the caller of the *tokensWithdrawn*() function and the function parameters.

b. ConsenSys's Audit of Umbra

Document token behavior restrictions: As with any protocol that interacts with arbitrary ERC20 tokens, it is important to clearly document which tokens are supported. Often this is best done by providing a specification for the behavior of the expected ERC20 tokens and only relaxing this specification after careful review of a particular class of tokens and their interactions with the protocol.

a. ConsenSys's Audit of Umbra

b. Recommendation: Known deviations from "normal" ERC20 behavior should be explicitly noted as NOT supported by the Umbra Protocol: 1) Deflationary or fee-on-transfer tokens: These are tokens in which the balance of the recipient of a transfer may not be increased by the amount of the transfer. There may also be some alternative mechanism by which balances are unexpectedly decreased. While these tokens can be successfully sent via the *sendToken()* function, the internal accounting of the Umbra contract will be out of sync with the balance as recorded in the token contract, resulting in loss of funds. 2) Inflationary tokens: The opposite of deflationary tokens. The Umbra contract provides no mechanism for claiming positive balance adjustments. 3) Rebasing tokens: A combination of the above cases, these are tokens in which an account's balance increases or decreases along with expansions or contractions in supply. The contract provides no mechanism to update its internal accounting in response to these unexpected balance adjustments, and funds may be lost as a result.

Full test suite is recommended: The test suite at this stage is not complete and many of the tests fail to execute. For complicated systems such as DeFi Saver, which uses many different modules and interacts with different DeFi protocols, it is crucial to have a full test coverage that includes the edge cases and failed scenarios. Especially this helps with safer future development and upgrading each module. As we've seen in some smart contract incidents, a complete test suite can prevent issues that might be hard to find with manual reviews.

a. Recommendation: Add a full coverage test suite.

b. ConsenSys's Audit of DeFi Saver

Kyber getRates code is unclear: Function names don't reflect their true functionalities, and the code uses some undocumented assumptions.

a. ConsenSys's Audit of DeFi Saver

b. Recommendation: Refactor the code to separate getting rate functionality with getSellRate and getBuyRate. Explicitly document any assumptions in the code (slippage, etc).

Return value is not used for *TokenUtils.withdrawTokens*: The return value of *TokenUtils.withdrawTokens* which represents the actual amount of tokens that were transferred is never used throughout the repository. This might cause discrepancy in the case where the original value of *_amount* was *type(uint256).max*.

a. Recommendation: The return value can be used to validate the withdrawal or used in the event emitted

b. ConsenSys's Audit of DeFi Saver

Missing access control for *DefiSaverLogger.Log*: *DefiSaverLogger* is used as a logging aggregator within the entire dapp, but anyone can create logs.

a. Recommendation: Add access control to all functions appropriately

b. Consensys Audit of DeFi Saver

Remove stale comments: Remove inline comments that suggest the two uint256 values *DAOfiV1Pair.reserveBase* and *DAOfiV1Pair.reserveQuote* are stored in the same storage slot. This is likely a carryover from the *UniswapV2Pair* contract, in which *reserve0*, *reserve1*, and *blockTimestampLast* are packed into a single storage slot.

a. Recommendation: Remove stale comments

b. [ConsenSys's Audit of DAOfi](#)

Discrepancy between code and comments: There is a mismatch between what the code implements and what the corresponding comment describes that code implements.

a. **Recommendation:** Update the code or the comment to be consistent

b. [ConsenSys's Audit of mstable-1.1](#)

Remove unnecessary call to *DAOfiV1Factory.formula*(): The *DAOfiV1Pair* functions *initialize*(), *getBaseOut*(), and *getQuoteOut*() all use the private function *_getFormula*(), which makes a call to the factory to retrieve the address of the *BancorFormula* contract. The formula address in the factory is set in the constructor and cannot be changed, so these calls can be replaced with an immutable value in the pair contract that is set in its constructor.

a. Recommendation: Remove unnecessary calls

b. [ConsenSys's Audit of DAOfi](#)

Deeper validation of curve math: Increased testing of edge cases in complex mathematical operations could have identified at least one issue raised in this report. Additional unit tests are recommended, as well as fuzzing or property-based testing of curve-related operations. Improperly validated interactions with the *BancorFormula* contract are seen to fail in unanticipated and potentially dangerous ways, so care should be taken to validate inputs and prevent pathological curve parameters.

a. Recommendation: More validation of mathematical operations

b. [ConsenSys's Audit of DAOfi](#)

***GovernorAlpha* proposals may be canceled by the proposer, even after they have been accepted and queued**: *GovernorAlpha* allows proposals to be canceled via cancel. A proposer may cancel proposals in any of these states: Pending, Active, Canceled, Defeated, Succeeded, Queued, Expired.

a. Recommendation: Prevent proposals from being canceled unless they are in the Pending or Active states.

b. [ConsenSys's Audit of Fei Protocol](#)

Require a delay period before granting *KYC_ADMIN_ROLE*

Acknowledged: The KYC Admin has the ability to freeze the funds of any user at any time by revoking the *KYC_MEMBER_ROLE*. The trust requirements from users can be decreased slightly by implementing a delay on granting this ability to new addresses. While the management of private keys and admin access is outside the scope of this review, the addition of a time delay can also help protect the development team and the system itself in the event of private key compromise.

a. Recommendation: Use a *TimelockController* as the *KYC_DEFAULT_ADMIN* of the eRLC contract

b. [ConsenSys's Audit of eRLC](#)

Improve inline documentation and test coverage: The source-units hardly contain any inline documentation which makes it hard to reason about methods and how they are supposed to be used. Additionally, test-coverage seems to be limited. Especially for a public-facing exchange contract system test-coverage should be extensive, covering all methods and functions that can directly be accessed including potential security-relevant and edge-cases. This would have helped in detecting some of the findings raised with this report.

a. Recommendation: Consider adding natspec-format compliant inline code documentation, describe functions, what they are used for, and who is supposed to interact with them. Document function or source-unit specific assumptions. Increase test coverage.

b. [ConsenSys's Audit of 1inch Liquidity Protocol](#)

Unspecific compiler version pragma: For most source-units the compiler version pragma is very unspecific $\wedge 0.6.0$. While this often makes sense for libraries to allow them to be included with multiple different versions of an application, it may be a security risk for the actual application implementation itself. A known vulnerable compiler version may accidentally be selected or security tools might fall-back to an older compiler version ending up actually checking a different evm compilation that is ultimately deployed on the blockchain.

---

a. Recommendation: Avoid floating pragmas. We highly recommend pinning a concrete compiler version (latest without security issues) in at least the top-level "deployed" contracts to make it unambiguous which compiler version is being used. Rule of thumb: a flattened source-unit should have at least one non-floating concrete solidity compiler version pragma.

b. [ConsenSys's Audit of 1inch Liquidity Protocol](#)

Use of hardcoded gas limits can be problematic: Hardcoded gas limits can be problematic as the past has shown that gas economics in ethereum have changed, and may change again potentially rendering the contract system unusable in the future.

a. Recommendation: Be conscious about this potential limitation and prepare for the case where gas prices might change in a way that negatively affects the contract system.

b. [ConsenSys's Audit of 1inch Liquidity Protocol](#)

Anyone can steal all the funds that belong to *ReferralFeeReceiver*: The *ReferralFeeReceiver* receives pool shares when users *swap()* tokens in the pool. A *ReferralFeeReceiver* may be used with multiple pools and, therefore, be a lucrative target as it is holding pool shares. Any token or ETH that belongs to the *ReferralFeeReceiver* is at risk and can be drained by any user by providing a custom mooniswap pool contract that references existing token holdings. It should be noted that none of the functions in *ReferralFeeReceiver* verify that the user-provided mooniswap pool address was actually deployed by the linked MooniswapFactory.

a. Recommendation: Enforce that the user-provided mooniswap contract was actually deployed by the linked factory. Other contracts cannot be trusted. Consider implementing token sorting and de-duplication (*tokenA!=tokenB*) in the pool contract constructor as well. Consider employing a reentrancy guard to safeguard the contract from reentrancy attacks. Improve testing. The methods mentioned here are not covered at all. Improve documentation and provide a specification that outlines how this contract is supposed to be used.

b. Critical finding in [ConsenSys's Audit of 1inch Liquidity Protocol](#)

Unpredictable behavior for users due to admin front running or general bad timing: In a number of cases, administrators of contracts can update or upgrade things in the system without warning. This has the potential to violate a security goal of the system. Specifically, privileged roles could use front running to make malicious changes just ahead of incoming transactions, or purely accidental negative effects could occur due to the unfortunate timing of changes. In general users of the system should have assurances about the behavior of the action they're about to take.

a. [ConsenSys's Audit of 1inch Liquidity Protocol](#)

b. Recommendation: We recommend giving the user advance notice of changes with a time lock. For example, make all system-parameter and upgrades require two steps with a mandatory time window between them. The first step merely broadcasts to users that a particular change is coming, and the second step commits that change after a suitable waiting period. This allows users that do not accept the change to withdraw immediately.

Improve system documentation and create a complete technical specification: A system's design specification and supporting documentation should be almost as important as the system's implementation itself. Users rely on high-level documentation to understand the big picture of how a system works. Without spending time and effort to create palatable documentation, a user's only resource is the code itself, something the vast majority of users cannot understand. Security assessments depend on a complete technical specification to understand the specifics of how a system works. When a behavior is not specified (or is specified incorrectly), security assessments must base their knowledge in assumptions, leading to less effective review. Maintaining and updating code relies on supporting documentation to know why the system is implemented in a specific way. If code maintainers cannot reference documentation, they must rely on memory or assistance to make high-quality changes. Currently, the only documentation for Growth DeFi is a single README file, as well as code comments.

a. Recommendation: Improve system documentation and create a complete technical specification

b. [ConsenSys's Audit of Growth DeFi](#)

Ensure system states, roles, and permissions are sufficiently restrictive: Smart contract code should strive to be strict. Strict code behaves predictably, is easier to maintain, and increases a system's ability to handle nonideal conditions. Our assessment of Growth DeFi found that many of its states, roles, and permissions are loosely defined.

  a. [ConsenSys's Audit of Growth DeFi](#)

b. Recommendation: Document the use of administrator permissions. Monitor the usage of administrator permissions. Specify strict operation requirements for each contract.

Evaluate all tokens prior to inclusion in the system: Review current and future tokens in the system for non-standard behavior. Particularly dangerous functionality to look for includes a callback (ie. ERC777) which would enable an attacker to execute potentially arbitrary code during the transaction, fees on transfers, or inflationary/deflationary tokens.

a. Recommendation: Evaluate all tokens prior to inclusion in the system

b. [ConsenSys's Audit of Growth DeFi](#)

Use descriptive names for contracts and libraries: The code base makes use of many different contracts, abstract contracts, interfaces, and libraries for inheritance and code reuse. In principle, this can be a good practice to avoid repeated use of similar code. However, with no descriptive naming conventions to signal which files would contain meaningful logic, codebase becomes difficult to navigate.

a. Recommendation: Use descriptive names for contracts and libraries

b. [ConsenSys's Audit of Growth DeFi](#)

Prevent contracts from being used before they are entirely initialized: Many contracts allow users to deposit / withdraw assets before the contracts are entirely initialized, or while they are in a semi-configured state. Because these contracts allow interaction on semi-configured states, the number of configurations possible when interacting with the system makes it incredibly difficult to determine whether the contracts behave as expected in every scenario, or even what behavior is expected in the first place.

a. Recommendation: Prevent contracts from being used before they are entirely initialized

b. [ConsenSys's Audit of Growth DeFi](#)

Potential resource exhaustion by external calls performed within an unbounded loop: *DydxFlashLoanAbstraction._requestFlashLoan* performs external calls in a potentially-unbounded loop. Depending on changes made to DyDx's *SoloMargin*, this may render this flash loan provider prohibitively expensive. In the worst case, changes to *SoloMargin* could make it impossible to execute this code due to the block gas limit.

a. Recommendation: Reconsider or bound the loop

b. [ConsenSys's Audit of Growth DeFi](#)

Owners can never be removed: The intention of *setOwners*() is to replace the current set of owners with a new set of owners. However, the *isOwner* mapping is never updated, which means any address that was ever considered an owner is permanently considered an owner for purposes of signing transactions.

a. Critical finding in [ConsenSys's Audit of Paxos](#)

b. Recommendation: In *setOwners_*(), before adding new owners, loop through the current set of owners and clear their *isOwner* booleans

Potential manipulation of stable interest rates using flash loans: Flash loans allow users to borrow large amounts of liquidity from the protocol. It is possible to adjust the stable rate up or down by momentarily removing or adding large amounts of liquidity to reserves.

a. [ConsenSys's Audit of Aave Protocol V2](#)

b. Recommendation: This type of manipulation is difficult to prevent especially when flash loans are available. Aave should monitor the protocol at all times to make sure that interest rates are being rebalanced to sane values.

Only whitelist validated assets: Because some of the functionality relies on correct token behavior, any whitelisted token should be audited in the context of this system. Problems can arise if a malicious token is whitelisted because it can block people from voting with that specific token or gain unfair advantage if the balance can be manipulated.

a. Recommendation: Make sure to audit any new whitelisted asset.

b. [ConsenSys's Audit of Aave Governance DAO](#)

Underflow if *TOKEN_DECIMALS* are greater than 18: In *latestAnswer*(), the assumption is made that *TOKEN_DECIMALS* is less than 18.

a. Recommendation: Add a simple check to the constructor to ensure the added token has 18 decimals or less

b. [ConsenSys's Audit of Aave CPM Price Provider](#)

Chainlink's performance at times of price volatility: In order to understand the risk of the Chainlink oracle deviating significantly, it would be helpful to compare historical prices on Chainlink when prices are moving rapidly, and see what the largest historical delta is compared to the live price on a large exchange.

a. Recommendation: Review Chainlink's performance at times of price volatility

b. [ConsenSys's Audit of Aave CPM Price Provider](#)

Consider an iterative approach to launching. Be aware of and prepare for worst-case scenarios: The system has many components with complex functionality and no apparent upgrade path.

a. [ConsenSys's Audit of Lien Protocol](#)

b. Recommendation: We recommend identifying which components are crucial for a minimum viable system, then focusing efforts on ensuring the security of those components first, and then moving on to the others. During the early life of the system, have a method for pausing and upgrading the system.

Use of modifiers for repeated checks: It is recommended to use modifiers for common checks within different functions. This will result in less code duplication in the given smart contract and adds significant readability into the code base.

a. Recommendation: Use of modifiers for repeated checks

b. [ConsenSys's Audit of Balancer Finance](#)

Switch modifier order: *BPool* **functions often use modifiers in the following order:** *_logs_*, *_lock_*. Because *_lock_* is a reentrancy guard, it should take precedence over *_logs_*.

a. [ConsenSys's Audit of Balancer Finance](#)

b. Recommendation: Place *_lock_* before other modifiers; ensuring it is the very first and very last thing to run when a function is called.

Address codebase fragility: Software is considered "fragile" when issues or changes in one part of the system can have side-effects in conceptually unrelated parts of the codebase. Fragile software tends to break easily and may be challenging to maintain.

  a. [ConsenSys's Audit of MCDEX Mai Protocol V2](#)

b. Recommendation: Building an anti-fragile system requires careful thought and consideration outside of the scope of this review. In general, prioritize the following concepts: 1) Follow the single-responsibility principle of functions 2) Reduce reliance on external systems

Reentrancy could lead to incorrect order of emitted events: The order of operations in the *_moveTokensAndETHfromAdjustment* function in the *BorrowOperations* contract may allow an attacker to cause events to be emitted out of order. In the event that the borrower is a contract, this could trigger a callback into *BorrowerOperations*, executing the *_ adjustTrove* flow above again. As the *_moveTokensAndETHfromAdjustment* call is the final operation in the function the state of the system on-chain cannot be manipulated. However, there are events that are emitted after this call. In the event of a reentrant call, these events would be emitted in the incorrect order. The event for the second operation i s emitted first, followed by the event for the first operation. Any off-chain monitoring tools may now have an inconsistent view of on-chain state.

a. Recommendation: Apply the checks-effects-interactions pattern and move the event emissions above the call to _ *moveTokensAndETHfromAdjustment* to avoid the potential reentrancy.

b. [ToB's Audit of Liquidity](#)

Variable shadowing from OUSD to ERC20: OUSD inherits from ERC20, but redefines the _ *allowances* and _ *totalSupply* state variables. As a result, access to these variables can lead to returning different values.

a. [ToB's Audit of Origin Dollar](#)

---

b. Recommendation: Remove the shadowed variables (_ *allowances* and _ *totalSupply*) in OUSD.

VaultCore.rebase functions have no return statements: *VaultCore.rebase()* and *VaultCore.rebase(bool)* return a *uint* but lack a return statement. As a result these functions will always return the default value, and are likely to cause issues for their callers. Both *VaultCore.rebase()* and *VaultCore.rebase(bool)* are expected to return a *uint256*. *rebase()* does not have a return statement. *rebase(bool)* has one return statement in one branch (return 0), but lacks a return statement for the other paths. So both functions will always return zero. As a result, a third-party code relying on the return value might not work as intended.

a. Recommendation: Add the missing return statement(s) or remove the return type in *VaultCore.rebase()* and *VaultCore.rebase(bool)*. Properly adjust the documentation as necessary.

b. [ToB's Audit of Origin Dollar](#)

Multiple contracts are missing inheritances: Multiple contracts are the implementation of their interfaces, but do not inherit from them. This behavior is error-prone and might lead the implementation to not follow the interface if the code is updated.

a. Recommendation: Ensure contracts inherit from their interfaces

b. [ToB's Audit of Origin Dollar](#)

Solidity compiler optimizations can be dangerous: Yield Protocol has enabled optional compiler optimizations in Solidity. There have been several bugs with security implications related to optimizations. Moreover, optimizations are actively being developed . Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised. High-severity security issues due to optimization bugs have occurred in the past . A high-severity bug in the emscripten -generated solc-js compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6 .

a. Recommendation: Short term, measure the gas savings from optimizations, and carefully weigh them against the possibility of an optimization-related bug. Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

b. [ToB's Audit of Yield Protocol](#)

Permission-granting is too simplistic and not flexible enough: The Yield Protocol contracts implement an oversimplified permission system that can be abused by the administrator. The Yield Protocol implements several contracts that need to call privileged functions from each other. However, there is no way to specify which operation can be called for every privileged user. All the authorized addresses can call any restricted function, and the owner can add any number of them. Also, the privileged addresses are supposed to be smart contracts; however, there is no check for that. Moreover, once an address is added, it cannot be deleted.

a. Recommendation: Rewrite the authorization system to allow only certain addresses to access certain functions

b. [ToB's Audit of Yield Protocol](#)

Lack of validation when setting the maturity value: When a $fyDAI$ contract is deployed, one of the deployment parameters is a maturity date, passed as a Unix timestamp. This is the date at which point $fyDAI$ tokens can be redeemed for the underlying Dai. Currently, the contract constructor performs no validation on this timestamp to ensure it is within an acceptable range. As a result, it is possible to mistakenly deploy a $YDai$ contract that has a maturity date in the past or many years in the future, which may not be immediately noticed.

a. Recommendation: Short term, add checks to the *YDai* contract constructor to ensure maturity timestamps fall within an acceptable range. This will prevent maturity dates from being mistakenly set in the past or too far in the future. Long term, always perform validation of parameters passed to contract constructors. This will help detect and prevent errors during deployment.

b. [ToB's Audit of Yield Protocol](#)

Delegates can be added or removed repeatedly to bloat logs: Several contracts in the Yield Protocol system inherit the Delegable contract. This contract allows users to delegate the ability to perform certain operations on their behalf to other addresses. When a user adds or removes a delegate, a corresponding event is emitted to log this operation. However, there is no check to prevent a user from repeatedly adding or removing a delegation that is already enabled or revoked, which could allow redundant events to be emitted repeatedly.

a. Recommendation: Short term, add a *require* statement to check that the delegate address is not already enabled or disabled for the user. This will ensure log messages are only emitted when a delegate is activated or deactivated. Long term, review all operations and avoid emitting events in repeated calls to idempotent operations. This will help prevent bloated logs.

b. [ToB's Audit of Yield Protocol](#)

Lack of events for critical operations: Several critical operations do not trigger events, which will make it difficult to review the correct behavior of the contracts once deployed. Users and blockchain monitoring systems will not be able to easily detect suspicious behaviors without events.

a. [ToB's Audit of 0x Protocol](#)

b. Recommendation: Short term, add events where appropriate for all critical operations. Long term, consider using a blockchain monitoring system to track any suspicious behavior in the contracts.

***_assertStakingPoolExists* never returns true**: The *_assertStakingPoolExists* should return a bool to determine if the staking pool exists or not; however, it only returns false or reverts. The *_assertStakingPoolExists* function checks that a staking pool exists given a pool id parameter. However, this function does not use a return statement and therefore, it will always return false or revert.

  a. [ToB's Audit of 0x Protocol](#)

b. Recommendation: Add a return statement or remove the return type. Properly adjust the documentation, if needed.

_min* and _max* have unorthodox semantics: Throughout the Curve contract, *_minTargetAmount* and *_maxOriginAmount* are used as open ranges (i.e., ranges that exclude the value itself). This contravenes the standard meanings of the terms "minimum" and "maximum," which are generally used to describe closed ranges.

  a. [ToB's Audit of DFX Finance](#)

b. Recommendation: Short term, unless they are intended to be strict, make the inequalities in the require statements non-strict. Alternatively, consider refactoring the variables or providing additional documentation to convey that they are meant to be exclusive bounds. Long term, ensure that mathematical terms such as "minimum," "at least," and "at most" are used in the typical way—that is, to describe values inclusive of minimums or maximums (as relevant).

***CurveFactory.newCurve* returns existing curves without provided arguments**: *CurveFactory.newCurve* takes values and creates a Curve contract instance for each *_baseCurrency* and *_quoteCurrency* pair, populating the Curve with provided weights and assimilator contract references. However, if the pair already exists, the existing Curve will be returned without any indication that it is not a newly created Curve with the provided weights. If an operator attempts to create a new Curve for a base-and-quote-currency pair that already exists, *CurveFactory* will return the existing Curve instance regardless of whether other creation parameters differ. A naive operator may overlook this issue.

a. Recommendation: Consider rewriting *newCurve* such that it reverts in the event that a base-and-quote-currency pair already exists. A view function can be used to check for and retrieve existing Curves without any gas payment prior to an attempt at Curve creation.

b. [ToB's Audit of DFX Finance](#)

Missing zero-address checks in *Curve.transferOwnership* and *Router.constructor*: Like other similar functions, *Curve._transfer* and *Orchestrator.includeAsset* perform zero-address checks. However, *Curve.transferOwnership* and the Router constructor do not. This may make sense for *Curve.transferOwnership*, because without zero-address checks, the function may serve as a means of burning ownership. However, popular contracts that define similar functions often consider this case, such as OpenZeppelin's *Ownable* contracts. Conversely, a zero-address check should be added to the Router constructor to prevent the deployment of an invalid Router, which would revert upon a call to the zero address.

a. Recommendation: Short term, consider adding zero-address checks to the Router's constructor and Curve's *transferOwnership* function to prevent operator errors. Long term, review state variables which referencing contracts to ensure that the code that sets the state variables performs zero-address checks where necessary

b. [ToB's Audit of DFX Finance](#)

***safeApprove* does not check return values for approve call**: Although the Router contract uses OpenZeppelin's *SafeERC20* library to perform safe calls to ERC20's approve function, the Orchestrator library defines its own *safeApprove* function. This function checks that a call to approve was successful but does not check returndata to verify whether the call returned true. In contrast, OpenZeppelin's *safeApprove* function checks return values appropriately. This issue may result in uncaught approve errors in successful Curve deployments, causing behavior.

a. Recommendation: Short term, leverage OpenZeppelin's *safeApprove* function wherever possible. Long term, ensure that all low-level calls have accompanying contract existence checks and return value checks where appropriate.

b. [ToB's Audit of DFX Finance](#)

ERC20 token Curve does not implement symbol, name, or decimals: *Curve.sol* **is an ERC20 token and implements all six required ERC20 methods:** *balanceOf*, *totalSupply*, *allowance*, *transfer*, *approve*, and *transferFrom*. However, it does not implement the optional but extremely common view methods *symbol*, *name*, and *decimals*.

  a. [ToB's Audit of DFX Finance](#)

b. Recommendation: Short term, implement *symbol*, *name*, and *decimals* on Curve contracts. Long term, ensure that contracts conform to all required and recommended industry standards.

Insufficient use of *SafeMath*: *CurveMath.calculateTrade* is used to compute the output amount for a trade. However, although *SafeMath* is used throughout the codebase to prevent underflows/overflows, it is not used in this calculation. Although we could not prove that the lack of *SafeMath* would cause an arithmetic issue in practice, all such calculations would benefit from the use of *SafeMath*.

  a. [ToB's Audit of DFX Finance](#)

b. Recommendation: Review all critical arithmetic to ensure that it accounts for underflows, overflows, and the loss of precision. Consider using *SafeMath* and the safe functions of *ABDKMath64x64* where possible to prevent underflows and overflows.

***setFrozen* can be front-run to deny deposits/swaps**: Currently, a Curve contract owner can use the *setFrozen* function to set the contract into a state that will block swaps and deposits. A contract owner could leverage this process to front-run transactions and freeze contracts before certain deposits or swaps are made; the contract owner could then unfreeze them at a later time.

  a. [ToB's Audit of DFX Finance](#)

b. Recommendation: Short term, consider rewriting *setFrozen* such that any contract freeze will not last long enough for a malicious user to easily execute an attack. Alternatively, depending on the intended use of this function, consider implementing permanent freezes.

Account creation spam: Hermez has a limit of $2^{**MAX\_NLEVELS}$ accounts. There is no fee on account creation, so an attacker can spam the network with account creation to fill the tree. If *MAX_NLEVELS* is below 32, an attacker can quickly reach the account limit. If *MAX_NLEVELS* is above or equal to 32, the time required to fill the tree will depend on the number of transactions accepted per second, but will take at least a couple of months. Ethereum miners do not have to pay for account creation. Therefore, an Ethereum miner can spam the network with account creation by sending L1 user transactions.

a. Recommendation: Short term, add a fee for account creation or ensure *MAX_NLEVELS* is at least 32. Also, monitor account creation and alert the community if a malicious coordinator spams the system. This will prevent an attacker from spamming the system to prevent new accounts from being created. Long term, when designing spam mitigation, consider that L1 gas cost can be avoided by Ethereum miners.

b. [ToB's Audit of Hermez Network](#)

Using empty functions instead of interfaces leaves contract error-prone: *WithdrawalDelayerInterface* is a contract meant to be an interface. It contains functions with empty bodies instead of function signatures, which might lead to unexpected behavior. A contract inheriting from *WithdrawalDelayerInterface* will not require an override of these functions and will not benefit from the compiler checks on its correct interface.

a. Recommendation: Short term, use an interface instead of a contract in *WithdrawalDelayerInterface*. This will make derived contracts follow the interface properly. Long term, properly document the inheritance schema of the contracts. Use Slither's inheritance-graph printer to review the inheritance.

b. [ToB's Audit of Hermez Network](#)

***cancelTransaction* can be called on non-queued transaction**: Without a transaction existence check in *cancelTransaction*, an attacker can confuse monitoring systems. *cancelTransaction* emits an event without checking that the transaction to be canceled exists. This allows a malicious admin to confuse monitoring systems by generating malicious events.

a. [ToB's Audit of Hermez Network](#)

b. Recommendation: Short term, check that the transaction to be canceled exists in *cancelTransaction*. This will ensure that monitoring tools can rely on emitted events. Long term, write a specification of each function and thoroughly test it with unit tests and fuzzing. Use symbolic execution for arithmetic invariants.

Contracts used as dependencies do not track upstream changes: Third-party contracts like *_concatStorage* are pasted into the Hermez repository. Moreover, the code documentation does not specify the exact revision used, or if it is modified. This makes updates and security fixes on these dependencies unreliable since they must be updated manually. *_concatStorage* is borrowed from the solidity-bytes-utils library, which provides helper functions for byte-related operations. Recently, a critical vulnerability was discovered in the library's slice function which allows arbitrary writes for user-supplied inputs.

a. Recommendation: Short term, review the codebase and document each dependency's source and version. Include the third-party sources as submodules in your Git repository so internal path consistency can be maintained and dependencies are updated periodically. Long term, identify the areas in the code that are relying on external libraries and use an Ethereum development environment and NPM to manage packages as part of your project.

b. [ToB's Audit of Hermez Network](#)

Expected behavior regarding authorization for adding tokens is unclear: *addToken* allows anyone to list a new token on Hermez. This contradicts the online documentation, which implies that only the governance should have this authorization. It is unclear whether the implementation or the documentation is correct.

a. [ToB's Audit of Hermez Network](#)

b. Recommendation: Short term, update either the implementation or the documentation to standardize the authorization specification for adding tokens. Long term, write a specification of each function and thoroughly test it with unit tests and fuzzing. Use symbolic execution for arithmetic invariants.

Contract name duplication leaves codebase error-prone: The codebase has multiple contracts that share the same name. This allows buidler-waffle to generate incorrect json artifacts, preventing third parties from using their tools. Buidler-waffle does not correctly support a codebase with duplicate contract names. The compilation overwrites compilation artifacts and prevents the use of third-party tools, such as Slither.

a. Recommendation: Short term, prevent the re-use of duplicate contract names or change the compilation framework. Long term, use Slither, which will help detect duplicate contract names.

b. ToB's Audit of Hermez Network

Use of hard-coded addresses may cause errors: Each contract needs contract addresses in order to be integrated into other protocols and systems. These addresses are currently hard-coded, which may cause errors and result in the codebase's deployment with an incorrect asset. Using hard-coded values instead of deployer-provided values makes these contracts incredibly difficult to test.

a. ToB's Audit of Advanced Blockchains

b. Recommendation: Short term, set addresses when contracts are created rather than using hard-coded values. This practice will facilitate testing. Long term, to ensure that contracts can be tested and reused across networks, avoid using hard-coded parameters.

Borrow rate depends on approximation of blocks per year: The borrow rate formula uses an approximation of the number of blocks mined annually. This number can change across different blockchains and years. The current value assumes that a new block is mined every 15 seconds, but on Ethereum mainnet, a new block is mined every ~13 seconds. To calculate the base rate, the formula determines the approximate borrow rate over the past year and divides that number by the estimated number of blocks mined per year. However, *blocksPerYear* is an estimated value and may change depending on transaction throughput. Additionally, different blockchains may have different block-settling times, which could also alter this number.

a. Recommendation: Short term, analyze the effects of a deviation from the actual number of blocks mined annually in borrow rate calculations and document the associated risks. Long term, identify all variables that are affected by external factors, and document the risks associated with deviations from their true values.

b. [ToB's Audit of Advanced Blockchains](#)

Flash loan rate lacks bounds and can be set arbitrarily: There are no lower or upper bounds on the flash loan rate implemented in the contract. The Blacksmith team could therefore set an arbitrarily high flash loan rate to secure higher fees. The Blacksmith team sets the *_flashLoanRate* when the Vault is first initialized. The *blackSmithTeam* address can then update this value by calling *updateFlashloanRate*. However, because there is no check on either setter function, the flash loan rate can be set arbitrarily. A very high rate could enable the Blacksmith team to steal vault deposits.

a. Recommendation: Short term, introduce lower and upper bounds for all configurable parameters in the system to limit privileged users' abilities. Long term, identify all incoming parameters in the system as well as the financial implications of large and small corner-case values. Additionally, use Echidna or Manticore to ensure that system invariants hold.

b. [ToB's Audit of Advanced Blockchains](#)

Logic duplicated across code: The logic in the repositories provided to Trail of Bits contains a significant amount of duplicated code. This development practice increases the risk that new bugs will be introduced into the system, as bug fixes must be copied and pasted into files across the system.

a. [ToB's Audit of Advanced Blockchains](#)

b. Recommendation: Short term, use inheritance to allow code to be reused across contracts. Changes to one inherited contract will be applied to all files without requiring developers to copy and paste them. Long term, minimize the amount of manual copying and pasting required to apply changes made to one file to other files.

Insufficient testing: The repositories under review lack appropriate testing, which increases the likelihood of errors in the development process and makes the code more difficult to review.

  a. [ToB's Audit of Advanced Blockchains](#)

b. Recommendation: Short term, ensure that the unit tests cover all public functions at least once, as well as all known corner cases. Long term, integrate coverage analysis tools into the development process and regularly review the coverage.

Project dependencies contain vulnerabilities: Although dependency scans did not yield a direct threat to the projects under review, yarn audit identified dependencies with known vulnerabilities. Due to the sensitivity of the deployment code and its environment, it is important to ensure dependencies are not malicious. Problems with dependencies in the JavaScript community could have a significant effect on the repositories under review.

  a. [ToB's Audit of Advanced Blockchains](#)

b. Recommendation: Short term, ensure dependencies are up to date. Several node modules have been documented as malicious because they execute malicious code when installing dependencies to projects. Keep modules current and verify their integrity after installation. Long term, consider integrating automated dependency auditing into the development workflow. If dependencies cannot be updated when a vulnerability is disclosed, ensure that the codebase does not use and is not affected by the vulnerable functionality of the dependency.

Lack of contract documentation makes codebase difficult to understand: The codebase lacks code documentation, high-level descriptions, and examples, making the contracts difficult to review and increasing the likelihood of user mistakes. The documentation would benefit from more detail.

a. [ToB's Audit of Advanced Blockchains](#)

b. Recommendation: Short term, review and properly document the above mentioned aspects of the codebase. Long term, consider writing a formal specification of the protocol.

ABIEncoderV2 is not production-ready: The contracts use the new Solidity ABI encoder, *ABIEncoderV2*. This experimental encoder is not ready for production. More than 3% of all GitHub issues for the Solidity compiler are related to experimental features, primarily *ABIEncoderV2*. Several issues and bug reports are still open and unresolved. *ABIEncoderV2* has been associated with more than [20 high-severity bugs](#), some of which are so recent that they have not yet been included in a Solidity release. For example, in March 2019 a [severe bug](#) introduced in Solidity 0.5.5 was found in the encoder.

a. Recommendation: Short term, use neither *ABIEncoderV2* nor any other experimental Solidity feature. Refactor the code such that structs do not need to be passed to or returned from functions. Long term, integrate static analysis tools like Slither into your CI pipeline to detect unsafe pragmas.

b. [ToB's Audit of Advanced Blockchains](#)

Contract owner has too many privileges: The owner of the contracts has too many privileges relative to standard users. Users can lose all of their assets if a contract owner private key is compromised. The contract owner can do the following: 1) Upgrade the system's implementation to steal funds 2) Upgrade the token's implementation to act maliciously 3) Increase the amount of *iTokens* for reward distribution to such an extent that rewards cannot be disbursed 4) Arbitrarily update the interest model contracts The concentration of these privileges creates a single point of failure. It increases the likelihood that the owner will be targeted by an attacker, especially given the insufficient protection on sensitive owner private keys. Additionally, it incentivizes the owner to act maliciously.

a. Recommendation: Short term: 1) Clearly document the functions and implementations the owner can change. 2) Split privileges to ensure that no one address has excessive ownership of the system. Long term, document the risks associated with privileged users and single points of failure. Ensure that users are aware of all the risks associated with the system.

b. [ToB's Audit of dForce Lending](#)

Poor error-handling practices in test suite: The test suite does not properly test expected behavior, as the contracts run in production. Additionally, certain components lack error-handling methods. These deficiencies can cause failed tests to be overlooked. In particular, the tests fail to properly check error messages. For example, errors are silenced with a try-catch statement. If this error is silenced, there will be no guarantee that a smart contract call has reverted for the right reason. As a result, if the test suite passes, it will provide no guarantee that the transaction call reverted correctly.

a. Recommendation: Short term, test these operations against a specific error message. Testing will ensure that errors are never silenced, and the test suite will check that a contract call has reverted for the right reason. Long term, follow standard testing practices for smart contracts to minimize the number of issues during development.

b. [ToB's Audit of dForce Lending](#)

Redundant and Unused Code: The *_recordLoanClosure()* function returns a boolean (*loanClosed*) which is never used by the calling function (see *_closeLoan*() , line [312]). Furthermore, since the *_recordLoanClosure*() function is only called via the *_closeLoan*() function, this means that *synthLoan.timeClosed* is always equal to zero (see *require* statement on line [305]). Therefore, the if statement on line [357] is redundant and unnecessary.

a. Recommendation: 1) Using the return value of the *_recordLoanClosure*() function or changing the function definition to stop returning *loanClosed* 2) Removing the if statement in line [357]

b. [Sigma Prime's Audit of Synthetix EtherCollateral](#)

Single Account Can Capture All Supply: The *EtherCollateral* smart contract does not rely on a *maxLoanSize* to limit the amount of ETH that can be locked for a loan. As a result, a single account can issue a loan that will reach the total minting supply.

a. Recommendation: Make sure this behaviour is understood and consider introducing and enforcing a cap (*maxLoanSize*) on the size of the loans allowed to be opened.

b. [Sigma Prime's Audit of Synthetix EtherCollateral](#)

Insufficient Input Validation: The constructor of the *EtherCollateral* smart contract does not check the validity of the addresses provided as input parameters. It is possible to deploy an instance of the *EtherCollateral* contract with the *synthProxy* , *sUSDProxy* and depot addresses set to zero. Similarly, the effective interest rate can be equal to zero if *interestRate* is set to any value lesser than 31536000 (*SECONDS_IN_A_YEAR*), as *interestPerSecond* will be null.

a. Recommendation: Consider introducing require statements to perform adequate input validation.

b. [Sigma Prime's Audit of Synthetix EtherCollateral](#)

Unused Event Logs: log events are declared but never emitted.

a. Recommendation: Remove these events from the EtherCollateral contract.

b. [Sigma Prime's Audit of Synthetix EtherCollateral](#)

Possible Unintended Token Burning in *transferFrom*() Function: *InfiniGold* allows users to convert/exchange their PMGT tokens to "gold certificates", which are digital artefacts effectively redeemable for actual gold. To do so, users are supposed to send their PMGT tokens to a specific burn address. The *transferFrom*() function does not check the to address against this burn address. Users may send tokens to the burn address, using the *transferFrom*() function, without triggering the emission of the *Burn(address indexed burner, uint256 value)* event, which dictates how the gold certificates are created and distributed.

a. Recommendation: Prevent sending tokens to the burn address in the *transferFrom*() function. This can be achieved by adding a *require* within *transferFrom*() which disallows the to address to be the *burnAddress* .

b. [Sigma Prime's Audit of InfiniGold](#)

Denial of Service Vector from Unbound List: The *reset*() internal function (called by the *replaceAll*() function) resets the role linked list by deleting all the elements (i.e. nodes) part of the bearer mapping. The caller is bound by the number of elements that are being removed for a particular role . Calling the *reset*() function will exceed the current block gas limit (i.e. 8,000,0000) for more than 371 total elements in a role linked list. Similarly, the *size*() and *toArray*() functions also loop through the linked list. This essentially means that listers, unlisters, minters, pausers, unpausers and owners can perform denial of service attacks on the lists they administer. In a scenario where the Roles library is leveraged by other smart contracts, calling these two functions will also result in a potential denial of service after a certain number of elements have been included in the linked list (this number would depend on the gas cost of the Opcodes implemented by the calling functions).

a. Recommendation: One way to ensure that the current block gas limit is not exceeded would be to introduce a condition in the *add*() function to check that the linked list size is strictly lesser than 371 elements before adding a new element. This additional condition would significantly increase the gas cost associated with calling the *add*() function, as a call to the *size*() function would be required to fetch the exact number of nodes in the linked list. Alternatively, the *gasleft*() Solidity special function could be used to make sure that going through the linked list does not exceed the block gas limit. Finally, the *reset*() could be changed to allow for removing an arbitrary number of nodes (by taking this number as a function parameter).

b. [Sigma Prime's Audit of InfiniGold](#)

ERC20 Implementation Vulnerable to Front-Running: Front-running attacks involve users watching the blockchain for particular transactions and, upon observing such a transaction, submitting their own transactions with a greater gas price. This incentivises miners to prioritise the later transaction. The ERC20 implementation is known to be affected by a front-running vulnerability, in its *approve*() function.

a. Recommendation: Be aware of the front-running issues in *approve*() , potentially add extended approve functions which are not vulnerable to the front-running vulnerability for future third-party-applications. See the Open-Zeppelin [8] solution for an example. We note that modifying the ERC20 standard to address this issue may lead to backward incompatibilities with external third-party software.

b. Sigma Prime's Audit of InfiniGold

Unnecessary *require* Statement: The following *require* statement in *Blacklistable.sol* can be removed: *require(to != address(0));* Indeed, this check is implemented in the *_transfer*() function in the *ERC20.sol* smart contract.

a. Sigma Prime's Audit of InfiniGold

b. Recommendation: Consider removing the require statement for gas saving purposes.

Rounding to Zero if Duration is Greater Than Reward: The *rewardRate* **value is calculated as follows:** *rewardRate = reward/duration*. Due to the integer representation of these variables, if duration is larger than reward the value of *rewardRate* will round to zero. Thus, stakers will not receive any of the reward for their stakes. Furthermore, due to the integer rounding, the total rewards distributed may be rounded down by up to one less than duration . As a result, the Unipool contract may slowly accumulate SNX.

a. Recommendation: Beware of the rounding issues when calling the *notifyRewardAmount*() function. We also recommend some way of allowing the excess SNX reward from rounding to be claimed or withdrawn from the Unipool contract.

b. Sigma Prime's Audit of Synthetix Unipool

Withdrawn Event Log Poisoning: Calling the *withdraw*() function will emit the Withdrawn event. No UNI tokens are required as this function can be called with *amount* = 0 . As a result a user could continually call this function, creating a potentially infinite amount of events. This can lead to an event log poisoning situation where malicious external users spam the Unipool contract to generate arbitrary Withdrawn events.

a. Recommendation: Consider adding a *require* or *if* statement preventing the *withdraw*() function from emitting the Withdrawn event when the amount variable is zero.

b. Sigma Prime's Audit of Synthetix Unipool

Insufficient incentives to liquidator: The liquidation process is a very important part of every DeFi project because it allows to extinguish the problem of having the whole system under-collateralized under critical conditions of the market, and it needs a design that incentivizes its speed of execution. The Holdefi contract implements the liquidation process for those accounts that may have an under-collateralized balance or that may have been inactive for a whole year without interacting with the project. The liquidator would end up paying for the expensive liquidation process, without receiving any benefit. Buying discounted collateral assets could be considered as an incentive to the liquidators

a. Recommendation: Consider improving the incentive design to give the liquidators higher incentives to execute the liquidation process

b. OpenZeppelin's Audit of HoldeFi

Markets can become insolvent: When the value of all collateral is worth less than the value of all borrowed assets, we say a market is insolvent. The Holdefi codebase can do many things to reduce the risk of market insolvency, including: prudent selection of collateral-ratios, incentivizing third-party collateral liquidation, careful selection of which tokens are listed on the platform, etc. However, the risk of insolvency cannot be entirely eliminated, and there are numerous ways a market can become insolvent.

a. OpenZeppelin's Audit of HoldeFi

b. Recommendation: This risk is not unique to the Holdefi project. All collateralized loans (even non-blockchain loans) have a risk of insolvency. However, it is important to know that this risk does exist, and that it can be difficult to recover from even a small dip into insolvency. Consider adding more targeted tests for these scenarios to better understand the behavior of the protocol, and designing relevant mechanics to make sure the platform operates properly. Also consider communicating the potential risks to the users if needed.

Not using OpenZeppelin contracts: OpenZeppelin maintains a library of standard, audited, community-reviewed, and battle-tested smart contracts. Instead of always importing these contracts, the Holdefi project reimplements them in some cases, while in other cases it just copies them. This increases the amount of code that the Holdefi team will have to maintain and misses all the improvements and bug fixes that the OpenZeppelin team is constantly implementing with the help of the community.

a. Recommendation: Consider importing the OpenZeppelin contracts instead of reimplementing or copying them. These contracts can be extended to add the extra functionalities required by Holdefi.

b. OpenZeppelin's Audit of HoldeFi

Lack of indexed parameters in events: Throughout the Holdefi's codebase, none of the parameters in the events defined in the contracts are indexed.

a. OpenZeppelin's Audit of HoldeFi

b. Recommendation: Consider indexing event parameters to avoid hindering the task of off-chain services searching and filtering for specific events.

Named return variables: There is an inconsistent use of named return variables across the entire codebase.

a. [OpenZeppelin's Audit of HoldeFi](#)

b. Recommendation: Consider removing all named return variables, explicitly declaring them as local variables in the body of the function, and adding the necessary explicit return statements where appropriate. This should favor both explicitness and readability of the project.

block.timestamp Unreliable: Code uses the *block.timestamp* as part of the calculations and time checks. Nevertheless, timestamps can be slightly altered by miners to favor them in contracts that have logics that depend strongly on them.

a. [OpenZeppelin's Audit of HoldeFi](#)

b. Recommendation: Consider taking into account this issue and warning the users that such a scenario could happen. If the alteration of timestamps cannot affect the protocol in any way, consider documenting the reasoning and writing tests enforcing that these guarantees will be preserved even if the code changes in the future.

Assignment in *require* statement: In the *YieldOracle* contract, there is a *require* statement that makes an assignment. This deviates from the standard usage and intention of *require* statements and can easily lead to confusion.

a. Recommendation: Consider moving the assignment to its own line before the *require* statement and then using the *require* statement solely for condition checking.

b. [OpenZeppelin's Audit of BarnBrige Smart Yield Bonds](#)

Commented code: Throughout the codebase there are lines of code that have been commented out with //. This can lead to confusion and is detrimental to overall code readability.

a. **Recommendation:** Consider removing commented out lines of code that are no longer needed.

b. [OpenZeppelin's Audit of BarnBrige Smart Yield Bonds](#)

Misleading *revert* messages: Error messages are intended to notify users about failing conditions, and should provide enough information so that the appropriate corrections needed to interact with the system can be applied. Uninformative error messages greatly damage the overall user experience, thus lowering the system's quality.

a. OpenZeppelin's Audit of Compound Governor Bravo

b. Recommendation: Consider not only fixing the specific issues mentioned, but also reviewing the entire codebase to make sure every error message is informative and user-friendly enough. Furthermore, for consistency, consider reusing error messages when extremely similar conditions are checked.

Multiple outdated Solidity versions in use: Outdated versions of Solidity are being used in all contracts. The compiler options in the truffle-config file specifies version 0.6.6, which was released on April 6, 2020. Throughout the codebase there are also different versions of Solidity being used.

a. OpenZeppelin's Audit of Fei Protocol

b. Recommendation: As Solidity is now under a fast release cycle, consider using a more recent version of the compiler, such as version 0.7.6. In addition, to avoid unexpected behavior, consider specifying explicit Solidity versions in pragma statements.

Test and production constants in the same codebase: The *CoreOrchestrator* contract defines the *TEST_MODE* boolean variable which is used to define several constants in the system. This decreases legibility of production code, and makes the system's integral values more error-prone.

a. Recommendation: Consider having different environments for production and testing, with different contracts.

b. [OpenZeppelin's Audit of Fei Protocol](#)

Unnecessarily small integer sizes: In Solidity, using integers smaller than 256 bits tends to increase gas costs because the Ethereum Virtual Machine must perform additional operations to zero out the unused bits. This can be justified by savings in storage costs in some scenarios, however, that is not generally the case in this codebase.

a. [OpenZeppelin's Audit of Fei Protocol](#)

b. Recommendation: Consider using integers of size 256 bits to improve gas efficiency and mitigate function reverts.

Use of *uint* instead of *uint256*: Across the codebase, there are hundreds of instances of *uint*, as opposed to *uint256*.

a. Recommendation: In favor of explicitness, consider replacing all instances of *uint* with *uint256*.

b. [OpenZeppelin's Audit of Fei Protocol](#)

Functions with unexpected side-effects: Some functions have side-effects. For example, the *_getLatestFundingRate* function of the *FundingRateApplier* contract might also update the funding rate and send rewards. The getPrice function of the *OptimisticOracle* contract might also settle a price request. These side-effect actions are not clear in the name of the functions and are thus unexpected, which could lead to mistakes when the code is modified by new developers not experienced in all the implementation details of the project.

a. Recommendation: Consider splitting these functions in separate getters and setters. Alternatively, consider renaming the functions to describe all the actions that they perform.

b. [OpenZeppelin's Audit of Uma Phase 4](#)

Unsafe casting: In line 554 of the *TaxCollector* contract, the value of *coinBalance(receiver)* is an *uint*. This is cast to an *int* and then negated. However, since *uint* can store higher values than *int*, it is possible that casting from *uint* to *int* may create an overflow.

a. [OpenZeppelin's Audit of GEB Protocol](#)

b. Recommendation: Consider verifying that the value of *coinBalance(receiver)* is within the acceptable range for negative *int* values before casting and negating. Consider using OpenZeppelin's *SafeCast* contract, which provides functions for safely casting between types.

Missing error messages in *require* statements: There are many places where *require* statements are correctly followed by their error messages, clarifying what was the triggered exception. However, there are places where *require* statements are not followed by the corresponding error messages. If any of those *require* statements were to fail the checked condition, the transaction
would revert silently without an informative error message.

a. Recommendation: Consider including specific and informative error messages in all *require* statements.

b. [OpenZeppelin's Audit of GEB Protocol](#)

Uncommented assembly block: The *OracleRelayer* contract includes an assembly block in the *rpower*() function. The same assembly block is repeated in the *TaxCollector* and *CoinSavingsAccount* contracts. While this does not pose a security risk per se, it is at the same time a complicated and critical part of the system. Moreover, as this is a low-level language that is harder to parse by readers, consider including extensive documentation regarding the rationale behind its use, clearly explaining what every single assembly instruction does. This will make it easier for users to trust the code, for reviewers to verify it, and for developers to build on top of it or update it. Note that the use of assembly discards several important safety features of Solidity, which may render the code unsafer and more error-prone.

a. Recommendation: Consider implementing thorough tests to cover all potential use cases of these functions to ensure they behave as expected.

b. OpenZeppelin's Audit of GEB Protocol

Unnecessary *require* statements: There are several instances in the code base where the *require* statements or conditional checks are unnecessary. For instance: In the *OracleRelayer* contract, the *require* statement in the *modifyParameters* function at line 189 checks if the input parameter data > 0. This is unnecessary since the same condition is already checked in the *require* statement at line 187.

a. Recommendation: To simplify the code and prevent wastage of gas, consider removing the unnecessary checks.

b. OpenZeppelin's Audit of GEB Protocol

Unnecessary event emission: The *popDebtFromQueue* function of the *AccountingEngine* contract is emitting a useless event whenever someone tries to call it with a *debtBlockTimestamp* that has not been saved before.

a. OpenZeppelin's Audit of GEB Protocol

b. Recommendation: To simplify the code and prevent wastage of gas, avoid emitting unnecessary events.

***oToken* can be created with a non-whitelisted collateral asset**: A product consists of a set of assets and an option type. Each product has to be whitelisted by the admin using the *whitelistProduct* function from the Whitelist contract.

a. Recommendation: Consider validating if the assets involved in a product have been already whitelisted before allowing the creation of *oTokens*.

b. OpenZeppelin's Audit of Opyn Gamma Protocol

Mismatches between contracts and interfaces: Interfaces define the exposed functionality of the implemented contracts. However, in several interfaces there are functions from the counterpart contracts that are not defined.

a. [OpenZeppelin's Audit of Opyn Gamma Protocol](#)

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

Actions not executed atomically might lead to inconsistent state: The *setAssetPricer*, *setLockingPeriod,* and *setDisputePeriod* functions of the Oracle contract execute actions that are always expected to be performed atomically. Failing to do so can lead to inconsistent states in the system.

a. [OpenZeppelin's Audit of Opyn Gamma Protocol](#)

b. Recommendation: Consider implementing an additional function that calls the *setAssetPricer*, *setLockingPeriod*, and *setDisputePeriod* functions, so that these actions can be executed atomically in a single transaction.

Chainlink pricer is using a deprecated API: The Chainlink Pricer is currently using multiple functions from a deprecated Chainlink API such as *latestAnswer*() in L61, *getTimestamp*() in L74. These functions might suddenly stop working if Chainlink stopped supporting deprecated APIs.

a. Recommendation: Consider refactoring these to use the latest Chainlink API.

b. [OpenZeppelin's Audit of Opyn Gamma Protocol](#)

Funds can be lost: The *sweepTimelockBalances* function accepts a list of users with unlocked balances to distribute. However, if there are duplicate users in the list, their balances will be counted multiple times when calculating the total amount to withdraw from the yield service.

a. Recommendation: Consider checking for duplicate users when calculating the amount to withdraw.

b. [OpenZeppelin's Audit of PoolTogether V3](#)

Use *delete* to clear variables: The Controller contract sets a variable to the zero address in order to clear it. Similarly, the *SetToken* clears the locker by assigning the zero address.

a. OpenZeppelin's Audit of Set Protocol

b. Recommendation: The *delete* key better conveys the intention and is also more idiomatic. Consider replacing assignments of zero with *delete* statements.