Audit Findings 101 1

Unhandled return values of *transfer* and *transferFrom*: ERC20 implementations are not always consistent. Some implementations of *transfer* and *transferFrom* could return 'false' on failure instead of reverting. It is safer to wrap such calls into *require()* statements to these failures.

a. Recommendation: Check the return value and revert on 0/false or use OpenZeppelin's *SafeERC20* wrapper functions

b. Medium severity finding from Consensys Diligence Audit of Aave Protocol V2

Audit Findings 101 2

Random task execution: In a scenario where a user takes a flash loan, *_parseFLAndExecute()* gives the flash loan wrapper contract (*FLAaveV2*, *FLDyDx*) the permission to execute functions on behalf of the user's *DSProxy*. This execution permission is revoked only after the entire recipe execution is finished, which means that in case that any of the external calls along the recipe execution is malicious, it might call *executeAction()* back, i.e. Reentrancy Attack, and inject any task it wishes (e.g. take user's funds out, drain approved tokens, etc)

a. Recommendation: A reentrancy guard (mutex) should be used to prevent such attack

b. Critical severity finding from Consensys Diligence Audit of Defi Saver

Audit Findings 101 3

Tokens with more than 18 decimal points will cause issues: It is assumed that the maximum number of decimals for each token is 18. However uncommon, it is possible to have tokens with more than 18 decimals, as an example YAMv2 has 24 decimals. This can result in broken code flow and unpredictable outcomes

a. Recommendation: Make sure the code won't fail in case the token's decimals is more than 18

b. Major severity finding from Consensys Diligence Audit of Defi Saver

Audit Findings 101 4

Error codes of Compound's *Comptroller.enterMarket*, *Comptroller.exitMarket* are not checked: Compound's *enterMarket*/*exitMarket* functions return an error code instead of reverting in case of failure. DeFi Saver smart contracts never check for the error codes returned from Compound smart contracts.

a. Recommendation: Caller contract should revert in case the error code is not 0

b. Major severity finding from Consensys Diligence Audit of Defi Saver

Audit Findings 101 5

Reversed order of parameters in allowance function call: the parameters that are used for the allowance function call are not in the same order that is used later in the call to *safeTransferFrom*.

a. Recommendation: Reverse the order of parameters in allowance function call to fit the order that is in the safeTransferFrom function call.

b. Medium severity finding from Consensys Diligence Audit of Defi Saver

Audit Findings 101 6

Token approvals can be stolen in *DAOfiV1Router01.addLiquidity()*: *DAOfiV1Router01.addLiquidity()* creates the desired pair contract if it does not already exist, then transfers tokens into the pair and calls *DAOfiV1Pair.deposit()*. There is no validation of the address to transfer tokens from, so an attacker could pass in any address with nonzero token approvals to *DAOfiV1Router*. This could be used to add liquidity to a pair contract for which the attacker is the *pairOwner*, allowing the stolen funds to be retrieved using DAOfiV1Pair.withdraw().

a. Recommendation: Transfer tokens from msg.sender instead of lp.sender

b. Critical severity finding from Consensys Diligence Audit of DAOfi

Audit Findings 101 7

***swapExactTokensForETH* checks the wrong return value**: Instead of checking that the amount of tokens received from a swap is greater than the minimum amount expected from this swap, it calculates the difference between the initial receiver's balance and the balance of the router

a. Recommendation: Check the intended values

b. Major severity finding from Consensys Diligence Audit of DAOfi

Audit Findings 101 8

***DAOfiV1Pair.deposit()* accepts deposits of zero, blocking the pool**: *DAOfiV1Pair.deposit()* is used to deposit liquidity into the pool. Only a single deposit can be made, so no liquidity can ever be added to a pool where deposited == true. The deposit() function does not check for a nonzero deposit amount in either token, so a malicious user that does not hold any of the *baseToken* or *quoteToken* can lock the pool by calling deposit() without first transferring any funds to the pool.

a. Recommendation: Require a minimum deposit amount with non-zero checks

b. Medium severity finding from Consensys Diligence Audit of DAOfi

***GenesisGroup.commit* overwrites previously-committed values**: The amount stored in the recipient's *committedFGEN* balance overwrites any previously-committed value. Additionally, this also allows anyone to commit an amount of "0" to any account, deleting their commitment entirely.

a. Recommendation: Ensure the committed amount is added to the existing commitment.

b. Critical severity finding from Consensys Diligence Audit of Fei Protocol

Purchasing and committing still possible after launch: Even after *GenesisGroup.launch* has successfully been executed, it is still possible to invoke *GenesisGroup.purchase* and *GenesisGroup.commit*.

a. Recommendation: Consider adding validation in *GenesisGroup.purchase* and *GenesisGroup.commit* to make sure that these functions cannot be called after the launch.

b. Critical severity finding from Consensys Diligence Audit of Fei Protocol

***UniswapIncentive* overflow on pre-transfer hooks**: Before a token transfer is performed, Fei performs some combination of mint/burn operations via *UniswapIncentive.incentivize*. Both *incentivizeBuy* and *incentivizeSell* calculate buy/sell incentives using overflow-prone math, then mint / burn from the target according to the results. This may have unintended consequences, like allowing a caller to mint tokens before transferring them, or burn tokens from their recipient.

a. Recommendation: Ensure casts in *getBuyIncentive* and *getSellPenalty* do not overflow

b. Major severity finding from Consensys Diligence Audit of Fei Protocol

***BondingCurve* allows users to acquire FEI before launch**: allocate can be called before genesis launch, as long as the contract holds some nonzero PCV. By force-sending the contract 1 wei, anyone can bypass the majority of checks and actions in allocate, and mint themselves FEI each time the timer expires.

a. Recommendation: Prevent allocate from being called before genesis launch

b. Medium severity finding from Consensys Diligence Audit of Fei Protocol

***Timed.isTimeEnded* returns true if the timer has not been initialized**: *Timed* initialization is a 2-step process: 1) *Timed.duration* is set in the constructor 2) *Timed.startTime* is set when the method *_initTimed* is called. Before this second method is called, *isTimeEnded*() calculates remaining time using a *startTime* of 0, resulting in the method returning true for most values, even though the timer has not technically been started.

a. Recommendation: If Timed has not been initialized, *isTimeEnded*() should return false, or revert

b. Medium severity finding from Consensys Diligence Audit of Fei Protocol

Overflow/underflow protection: Having overflow/underflow vulnerabilities is very common for smart contracts. It is usually mitigated by using SafeMath or using solidity version ^0.8 (after solidity 0.8 arithmetical operations already have default overflow/underflow protection). In this code, many arithmetical operations are used without the 'safe' version. The reasoning behind it is that all the values are derived from the actual ETH values, so they can't overflow.

a. Recommendation: In our opinion, it is still safer to have these operations in a safe mode. So we recommend using SafeMath or solidity version ^0.8 compiler.

b. Medium severity finding from Consensys Diligence Audit of Fei Protocol

Unchecked return value for *IWETH.transfer* call: In *EthUniswapPCVController*, there is a call to *IWETH.transfer* that does not check the return value. It is usually good to add a require-statement that checks the return value or to use something like *safeTransfer*; unless one is sure the given token reverts in case of a failure.

a. Recommendation: Consider adding a require-statement or using *safeTransfer*

b. Medium severity finding from Consensys Diligence Audit of Fei Protocol

***GenesisGroup.emergencyExit* remains functional after launch**: *emergencyExit* is intended as an escape mechanism for users in the event the genesis launch method fails or is frozen. *emergencyExit* becomes callable 3 days after launch is callable. These two methods are intended to be mutually-exclusive, but are not: either method remains callable after a successful call to the other. This may result in accounting edge cases.

a. Recommendation: 1) Ensure launch cannot be called if *emergencyExit* has been called 2) Ensure *emergencyExit* cannot be called if launch has been called

b. Medium severity finding from Consensys Diligence Audit of Fei Protocol

Audit Findings 101 17

ERC20 tokens with no return value will fail to transfer: Although the ERC20 standard suggests that a transfer should return true on success, many tokens are non-compliant in this regard. In that case, the .transfer() call here will revert even if the transfer is successful, because solidity will check that the *RETURNDATASIZE* matches the ERC20 interface.

a. Recommendation: Consider using OpenZeppelin's SafeERC20

b. Major severity finding from Consensys Diligence Audit of bitbank

Audit Findings 101 18

Reentrancy vulnerability in *MetaSwap.swap()*: If an attacker is able to reenter *swap()*, they can execute their own trade using the same tokens and get all the tokens for themselves.

a. Recommendation: Use a simple reentrancy guard, such as OpenZeppelin's ReentrancyGuard to prevent reentrancy in *MetaSwap.swap()*

b. Major severity finding from Consensys Diligence Audit of MetaSwap

Audit Findings 101 19 (1/2)

A new malicious adapter can access users' tokens: The purpose of the *MetaSwap* contract is to save users gas costs when dealing with a number of different aggregators. They can just approve() their tokens to be spent by *MetaSwap* (or in a later architecture, the Spender contract). They can then perform trades with all supported aggregators without having to reapprove anything. A downside to this design is that a malicious (or buggy) adapter has access to a large collection of valuable assets. Even a user who has diligently checked all existing adapter code before interacting with *MetaSwap* runs the risk of having their funds intercepted by a new malicious adapter that's added later.

Audit Findings 101 19 (2/2)

a. Recommendation: Make *MetaSwap* contract the only contract that receives token approval. It then moves tokens to the Spender contract before that contract *DELEGATECALLs* to the appropriate adapter. In this model, newly added adapters shouldn't be able to access users' funds.

b. Medium severity finding from Consensys Diligence Audit of MetaSwap

Owner can front-run traders by updating adapters: *MetaSwap* owners can front-run users to swap an adapter implementation. This could be used by a malicious or compromised owner to steal from users. Because adapters are *DELEGATECALL'ed*, they can modify storage. This means any adapter can overwrite the logic of another adapter, regardless of what policies are put in place at the contract level. Users must fully trust every adapter because just one malicious adapter could change the logic of all other adapters.

a. Recommendation: At a minimum, disallow modification of existing adapters. Instead, simply add new adapters and disable the old ones.

b. Medium severity finding from Consensys Diligence Audit of MetaSwap

Users can collect interest from *SavingsContract* by only staking mTokens momentarily: The *SAVE* contract allows users to deposit *mAssets* in return for lending yield and swap fees. When depositing *mAsset*, users receive a "credit" tokens at the momentary credit/*mAsset* exchange rate which is updated at every deposit. However, the smart contract enforces a minimum timeframe of 30 minutes in which the interest rate will not be updated. A user who deposits shortly before the end of the timeframe will receive credits at the stale interest rate and can immediately trigger an update of the rate and withdraw at the updated (more favorable) rate after the 30 minutes window. As a result, it would be possible for users to benefit from interest payouts by only staking mAssets momentarily and using them for other purposes the rest of the time.

a. Recommendation: Remove the 30 minutes window such that every deposit also updates the exchange rate between credits and tokens.

b. Medium severity finding from Consensys Diligence Audit of mstable-1.1

Oracle updates can be manipulated to perform atomic front-running attack: It is possible to atomically arbitrage rate changes in a risk-free way by "sandwiching" the Oracle update between two transactions. The attacker would send the following 2 transactions at the moment the Oracle update appears in the mempool: 1) The first transaction, which is sent with a higher gas price than the Oracle update transaction, converts a very small amount. This "locks in" the conversion weights for the block since *handleExternalRateChange()* only updates weights once per block. By doing this, the arbitrageur ensures that the stale Oracle price is initially used when doing the first conversion in the following transaction. The second transaction, which is sent at a slightly lower gas price than the transaction that updates the Oracle, performs a large conversion at the old weight, adds a small amount of Liquidity to trigger rebalancing and converts back at the new rate. The attacker can obtain liquidity for step 2 using a flash loan. The attack will deplete the reserves of the pool.

a. Recommendation: Do not allow users to trade at a stale Oracle rate and trigger an Oracle price update in the same transaction.

b. Critical severity finding from Consensys Diligence Audit of Bancor v2 AMM

Certain functions lack input validation routines: The functions should first check if the passed arguments are valid first. These checks should include, but not be limited to: 1) uint should be larger than 0 when 0 is considered invalid 2) uint should be within constraints 3) int should be positive in some cases 4) length of arrays should match if more arrays are sent as arguments 5) addresses should not be 0x0

a. Recommendation: Add tests that check if all of the arguments have been validated. Consider checking arguments as an important part of writing code and developing the system.

b. Major severity finding from Consensys Diligence Audit of Shell Protocol

Remove *Loihi* methods that can be used as backdoors by the administrator: There are several functions in *Loihi* that give extreme powers to the shell administrator. The most dangerous set of those is the ones granting the capability to add assimilators. Since assimilators are essentially a proxy architecture to delegate code to several different implementations of the same interface, the administrator could, intentionally or unintentionally, deploy malicious or faulty code in the implementation of an assimilator. This means that the administrator is essentially totally trusted to not run code that, for example, drains the whole pool or locks up the users' and LPs' tokens. In addition to these, the function *safeApprove* allows the administrator to move any of the tokens the contract holds to any address regardless of the balances any of the users have. This can also be used by the owner as a backdoor to completely drain the contract.

a. Recommendation: Remove the *safeApprove* function and, instead, use a trustless escape-hatch mechanism. For the assimilator addition functions, our recommendation is that they are made completely internal, only callable in the constructor, at deploy time. Even though this is not a big structural change (in fact, it reduces the attack surface), it is, indeed, a feature loss. However, this is the only way to make each shell a time-invariant system. This would not only increase Shell's security but also would greatly improve the trust the users have in the protocol since, after deployment, the code is now static and auditable.

b. Major severity finding from Consensys Diligence Audit of Shell Protocol

A reverting fallback function will lock up all payouts: In *BoxExchange.sol*, the internal function *_transferEth*() reverts if the transfer does not succeed. The *_payment*() function processes a list of transfers to settle the transactions in an *ExchangeBox*. If any of the recipients of an ETH transfer is a smart contract that reverts, then the entire payout will fail and will be unrecoverable.

a. Recommendation: 1) Implement a queuing mechanism to allow buyers/sellers to initiate the withdrawal on their own using a 'pull-over-push pattern.' 2) Ignore a failed transfer and leave the responsibility up to users to receive them properly.

b. Critical severity finding from Consensys Diligence Audit of Lien Protocol

Audit Findings 101 26 (1/2)

***Saferagequit* makes you lose funds**: *safeRagequit* and *ragequit* functions are used for withdrawing funds from the LAO. The difference between them is that ragequit function tries to withdraw all the allowed tokens and *safeRagequit* function withdraws only some subset of these tokens, defined by the user. It's needed in case the user or *GuildBank* is blacklisted in some of the tokens and the transfer reverts. The problem is that even though you can quit in that case, you'll lose the tokens that you exclude from the list. To be precise, the tokens are not completely lost, they will belong to the LAO and can still potentially be transferred to the user who quit. But that requires a lot of trust, coordination, time and anyone can steal some part of these tokens.

Audit Findings 101 26 (2/2)

a. Recommendation: Implementing pull pattern for token withdrawals should solve the issue. Users will be able to quit the LAO and burn their shares but still keep their tokens in the LAO's contract for some time if they can't withdraw them right now.

b. Critical severity finding from Consensys Diligence Audit of The Lao

Audit Findings 101 27

Creating proposal is not trustless: Usually, if someone submits a proposal and transfers some amount of tribute tokens, these tokens are transferred back if the proposal is rejected. But if the proposal is not processed before the emergency processing, these tokens will not be transferred back to the proposer. This might happen if a tribute token or a deposit token transfers are blocked. Tokens are not completely lost in that case, they now belong to the LAO shareholders and they might try to return that money back. But that requires a lot of coordination and time and everyone who ragequits during that time will take a part of that tokens with them.

a. Recommendation: Pull pattern for token transfers would solve the issue

b. Critical severity finding from Consensys Diligence Audit of The Lao

Audit Findings 101 28

Emergency processing can be blocked: The main reason for the emergency processing mechanism is that there is a chance that some token transfers might be blocked. For example, a sender or a receiver is in the USDC blacklist. Emergency processing saves from this problem by not transferring tribute token back to the user (if there is some) and rejecting the proposal. The problem is that there is still a deposit transfer back to the sponsor and it could be potentially blocked too. If that happens, proposal can't be processed and the LAO is blocked.

a. Recommendation: Pull pattern for token transfers would solve the issue

b. Critical severity finding from Consensys Diligence Audit of The Lao

Token Overflow might result in system halt or loss of funds: If a token overflows, some functionality such as *processProposal*, *cancelProposal* will break due to SafeMath reverts. The overflow could happen because the supply of the token was artificially inflated to oblivion.

a. **Recommendation:** We recommend to allow overflow for broken or malicious tokens. This is to prevent system halt or loss of funds. It should be noted that in case an overflow occurs, the balance of the token will be incorrect for all token holders in the system

b. Major severity finding from Consensys Diligence Audit of The Lao

Whitelisted tokens limit: *_ragequit* function is iterating over all whitelisted tokens. If the number of tokens is too big, a transaction can run out of gas and all funds will be blocked forever.

a. **Recommendation:** A simple solution would be just limiting the number of whitelisted tokens. If the intention is to invest in many new tokens over time, and it's not an option to limit the number of whitelisted tokens, it's possible to add a function that removes tokens from the whitelist. For example, it's possible to add a new type of proposal that is used to vote on token removal if the balance of this token is zero. Before voting for that, shareholders should sell all the balance of that token.

b. Major severity finding from Consensys Diligence Audit of The Lao

Summoner can steal funds using bailout: The *bailout* function allows anyone to transfer kicked user's funds to the summoner if the user does not call safeRagequit (which forces the user to lose some funds). The intention is for the summoner to transfer these funds to the kicked member afterwards. The issue here is that it requires a lot of trust to the summoner on the one hand, and requires more time to kick the member out of the LAO.

a. **Recommendation:** By implementing pull pattern for token transfers, kicked member won't be able to block the ragekick and the LAO members would be able to kick anyone much quicker. There is no need to keep the *bailout* function.

b. Major severity finding from Consensys Diligence Audit of The Lao

Sponsorship front-running: If proposal submission and sponsorship are done in 2 different transactions, it's possible to front-run the *sponsorProposal* function by any member. The incentive to do that is to be able to block the proposal afterwards.

a. **Recommendation:** Pull pattern for token transfers will solve the issue. Front-running will still be possible but it doesn't affect anything.

b. Major severity finding from Consensys Diligence Audit of The Lao

Delegate assignment front-running: Any member can front-run another member's *delegateKey* assignment. If you try to submit an address as your *delegateKey*, someone else can try to assign your delegate address to themselves. While incentive of this action is unclear, it's possible to block some address from being a delegate forever.

a. Recommendation: Make it possible for a *delegateKey* to approve *delegateKey* assignment or cancel the current delegation. Commit-reveal methods can also be used to mitigate this attack.

b. Medium severity finding from Consensys Diligence Audit of The Lao

Queued transactions cannot be canceled: The Governor contract contains special functions to set it as the admin of the *Timelock*. Only the admin can call *Timelock.cancelTransaction*. There are no functions in Governor that call *Timelock.cancelTransaction*. This makes it impossible for *Timelock.cancelTransaction* to ever be called.

a. Recommendation: Short term, add a function to the Governor that calls *Timelock.cancelTransaction*. It is unclear who should be able to call it, and what other restrictions there should be around cancelling a transaction. Long term, consider letting Governor inherit from *Timelock*. This would allow a lot of functions and code to be removed and significantly lower the complexity of these two contracts.

b. High Risk severity finding from ToB's Audit of Origin Dollar

Proposal transactions can be executed separately and block *Proposal.execute* call: Missing access controls in the *Timelock.executeTransaction* function allow Proposal transactions to be executed separately, circumventing the *Governor.execute* function.

a. Recommendation: Short term, only allow the admin to call *Timelock.executeTransaction*

b. High Risk severity finding from ToB's Audit of Origin Dollar

Proposals could allow *Timelock.admin* takeover: The Governor contract contains special functions to let the guardian queue a transaction to change the *Timelock.admin*. However, a regular Proposal is also allowed to contain a transaction to change the *Timelock.admin*. This poses an unnecessary risk in that an attacker could create a Proposal to change the *Timelock.admin*.

a. Recommendation: Short term, add a check that prevents *setPendingAdmin* to be included in a Proposal

b. High Risk severity finding from ToB's Audit of Origin Dollar

Reentrancy and untrusted contract call in *mintMultiple*: Missing checks and no reentrancy prevention allow untrusted contracts to be called from *mintMultiple*. This could be used by an attacker to drain the contracts.

a. Recommendation: Short term, add checks that cause *mintMultiple* to revert if the amount is zero or the asset is not supported. Add a reentrancy guard to the *mint*, *mintMultiple*, *redeem*, and *redeemAll* functions. Long term, make use of Slither which will flag the reentrancy. Or even better, use Crytic and incorporate static analysis checks into your CI/CD pipeline. Add reentrancy guards to all non-view functions callable by anyone. Make sure to always revert a transaction if an input is incorrect. Disallow calling untrusted contracts.

b. High Risk severity finding from ToB's Audit of Origin Dollar

Lack of return value checks can lead to unexpected results: Several function calls do not check the return value. Without a return value check, the code is error-prone, which may lead to unexpected results.

a. Recommendation: Short term, check the return value of all calls mentioned above. Long term, subscribe to Crytic.io to catch missing return checks. Crytic identifies this bug type automatically.

b. High Risk severity finding from ToB's Audit of Origin Dollar

External calls in loop can lead to denial of service: Several function calls are made in unbounded loops. This pattern is error-prone as it can trap the contracts due to the gas limitations or failed transactions.

a. Recommendation: Short term, review all the loops mentioned above and either: 1) allow iteration over part of the loop, or 2) remove elements. Long term, subscribe to Crytic.io to review external calls in loops. Crytic catches bugs of this type.

b. High Risk severity finding from ToB's Audit of Origin Dollar

OUSD allows users to transfer more tokens than expected: Under certain circumstances, the OUSD contract allows users to transfer more tokens than the ones they have in their balance. This issue seems to be caused by a rounding issue when the *creditsDeducted* is calculated and subtracted.

a. Recommendation: Short term, make sure the balance is correctly checked before performing all the arithmetic operations. This will make sure it does not allow to transfer more than expected. Long term, use Echidna to write properties that ensure ERC20 transfers are transferring the expected amount.

b. High Risk severity finding from ToB's Audit of Origin Dollar

OUSD total supply can be arbitrary, even smaller than user balances: The OUSD token contract allows users to opt out of rebasing effects. At that point, their exchange rate is "fixed", and further rebases will not have an impact on token balances (until the user opts in).

a. Recommendation: Short term, we would advise making clear all common invariant violations for users and other stakeholders. Long term, we would recommend designing the system in such a way to preserve as many commonplace invariants as possible.

b. High Risk severity finding from ToB's Audit of Origin Dollar

Flash minting can be used to redeem *fyDAI*: The flash-minting feature from the *fyDAI* token can be used to redeem an arbitrary amount of funds from a mature token.

a. Recommendation: Short term, disallow calls to redeem in the *YDai* and Unwind contracts during flash minting. Long term, do not include operations that allow any user to manipulate an arbitrary amount of funds, even if it is in a single transaction. This will prevent attackers from gaining leverage to manipulate the market and break internal invariants.

b. Medium Risk severity finding from ToB's Audit of Yield Protocol

Lack of *chainID* validation allows signatures to be re-used across forks: *YDai* implements the draft ERC 2612 via the *ERC20Permit* contract it inherits from. This allows a third party to transmit a signature from a token holder that modifies the ERC20 allowance for a particular user. These signatures used in calls to permit in *ERC20Permit* do not account for chain splits. The *chainID* is included in the domain separator. However, it is not updatable and not included in the signed data as part of the permit call. As a result, if the chain forks after deployment, the signed message may be considered valid on both forks.

a. Recommendation: Short term, include the *chainID* opcode in the permit schema. This will make replay attacks impossible in the event of a post-deployment hard fork. Long term, document and carefully review any signature schemas, including their robustness to replay on different wallets, contracts, and blockchains. Make sure users are aware of signing best practices and the danger of signing messages from untrusted sources.

b. High Risk severity finding from ToB's Audit of Yield Protocol

Audit Findings 101 44 (1/2)

Lack of a contract existence check allows token theft: Since there's no existence check for contracts that interact with external tokens, an attacker can steal funds by registering a token that's not yet deployed. *_safeTransferFrom* will return success even if the token is not yet deployed, or was self-destructed. An attacker that knows the address of a future token can register the token in Hermez, and deposit any amount prior to the token deployment. Once the contract is deployed and tokens have been deposited in Hermez, the attacker can steal the funds. The address of a contract to be deployed can be determined by knowing the address of its deployer.

Audit Findings 101 44 (2/2)

a. Recommendation: Short term, check for contract existence in *_safeTransferFrom*. Add a similar check for any low-level calls, including in *WithdrawalDelayer*. This will prevent an attacker from listing and depositing tokens in a contract that is not yet deployed. Long term, carefully review the Solidity documentation, especially the Warnings section. The Solidity documentation warns: The low-level call, *delegatecall* and *callcode* will return success if the called account is non-existent, as part of the design of EVM. Existence must be checked prior to calling if desired.

b. High Risk severity finding from ToB's Audit of Hermez

Audit Findings 101 45 (1/2)

No incentive for bidders to vote earlier: Hermez relies on a voting system that allows anyone to vote with any weight at the last minute. As a result, anyone with a large fund can manipulate the vote. Hermez's voting mechanism relies on bidding. There is no incentive for users to bid tokens well before the voting ends. Users can bid a large amount of tokens just before voting ends, and anyone with a large fund can decide the outcome of the vote. As all the votes are public, users bidding earlier will be penalized, because their bids will be known by the other participants. An attacker can know exactly how much currency will be necessary to change the outcome of the voting just before it ends.

Audit Findings 101 45 (2/2)

a. Recommendation: Short term, explore ways to incentivize users to vote earlier. Consider a weighted bid, with a weight decreasing over time. While it won't prevent users with unlimited resources from manipulating the vote at the last minute, it will make the attack more expensive and reduce the chance of vote manipulation. Long term, stay up to date with the latest research on blockchain-based online voting and bidding. Blockchain-based online voting is a known challenge. No perfect solution has been found yet.

b. Medium Risk severity finding from ToB's Audit of Hermez

Audit Findings 101 46

Lack of access control separation is risky: The system uses the same account to change both frequently updated parameters and those that require less frequent updates. This architecture is error-prone and increases the severity of any privileged account compromises.

a. Recommendation: Short term, use a separate account to handle updating the tokens/USD ratio. Using the same account for the critical operations and update the tokens/USD ratio increases underlying risks. Long term, document the access controls and set up a proper authorization architecture. Consider the risks associated with each access point and their frequency of usage to evaluate the proper design.

b. High Risk severity finding from ToB's Audit of Hermez

Audit Findings 101 47 (1/2)

Lack of two-step procedure for critical operations leaves them error-prone: Several critical operations are done in one function call. This schema is error-prone and can lead to irrevocable mistakes. For example, the setter for the whitehack group address sets the address to the provided argument. If the address is incorrect, the new address will take on the functionality of the new role immediately. However, a two-step process is similar to the approve-transferFrom functionality: The contract approves the new address for a new role, and the new address acquires the role by calling the contract.

Audit Findings 101 47 (2/2)

a. Recommendation: Short term, use a two-step procedure for all non-recoverable critical operations to prevent irrecoverable mistakes. Long term, identify and document all possible actions and their associated risks for privileged accounts. Identifying the risks will assist codebase review and prevent future mistakes.

b. High Risk severity finding from ToB's Audit of Hermez

Audit Findings 101 48 (1/2)

Initialization functions can be front-run: *Hermez*, *HermezAuctionProtocol*, and *WithdrawalDelayer* have initialization functions that can be front-run, allowing an attacker to incorrectly initialize the contracts. Due to the use of the *delegatecall* proxy pattern, *Hermez*, *HermezAuctionProtocol*, and *WithdrawalDelayer* cannot be initialized with a constructor, and have initializer functions. All these functions can be front-run by an attacker, allowing them to initialize the contracts with malicious values.

a. Recommendation: Short term, either: 1) Use a factory pattern that will prevent front-running of the initialization, or 2) Ensure the deployment scripts are robust in case of a front-running attack. Carefully review the Solidity documentation, especially the Warnings section. Carefully review the pitfalls of using delegatecall proxy pattern.

b. High Risk severity finding from ToB's Audit of Hermez

Missing validation of *_owner* argument could indefinitely lock owner role: A lack of input validation of the *_owner* argument in both the constructor and *setOwner* functions could permanently lock the owner role, requiring a costly redeploy. To resolve an incorrect owner issue, Uniswap would need to redeploy the factory contract and re-add pairs and liquidity. Users might not be happy to learn of these actions, which could lead to reputational damage. Certain users could also decide to continue using the original factory and pair contracts, in which owner functions cannot be called. This could lead to the concurrent use of two versions of Uniswap, one with the original factory contract and no valid owner and another in which the owner was set correctly. Trail of Bits identified four distinct cases in which an incorrect owner is set: 1) Passing address(0) to the constructor 2) Passing address(0) to the *setOwner* function 3) Passing an incorrect address to the constructor 4) Passing an incorrect address to the *setOwner* function.

a. Recommendation: Several improvements could prevent the four above mentioned cases: 1) Designate *msg.sender* as the initial owner, and transfer ownership to the chosen owner after deployment. 2) Implement a two-step ownership-change process through which the new owner needs to accept ownership. 3) If it needs to be possible to set the owner to address(0), implement a *renounceOwnership* function.

b. Medium Risk severity finding from ToB's Audit of Uniswap V3

Incorrect comparison enables swapping and token draining at no cost: An incorrect comparison in the swap function allows the swap to succeed even if no tokens are paid. This issue could be used to drain any pool of all of its tokens at no cost. The swap function calculates how many tokens the initiator (*msg.sender*) needs to pay (*amountIn*) to receive the requested amount of tokens (*amountOut*). It then calls the *uniswapV3SwapCallback* function on the initiator's account, passing in the amount of tokens to be paid. The callback function should then transfer at least the requested amount of tokens to the pool contract. Afterward, a require inside the swap function verifies that the correct amount of tokens (amountIn) has been transferred to the pool. However, the check inside the require is incorrect. The operand used is >= instead of <=.

a. Recommendation: Replace >= with <= in the require statement.

b. High Risk severity finding from ToB's Audit of Uniswap V3

Unbound loop enables denial of service: The swap function relies on an unbounded loop. An attacker could disrupt swap operations by forcing the loop to go through too many operations, potentially trapping the swap due to a lack of gas.

a. Recommendation: Bound the loops and document the bounds.

b. Medium Risk severity finding from ToB's Audit of Uniswap V3

Front-running pool's initialization can lead to draining of liquidity provider's initial deposits: A front-run on *UniswapV3Pool.initialize* allows an attacker to set an unfair price and to drain assets from the first deposits. There are no access controls on the initialize function, so anyone could call it on a deployed pool. Initializing a pool with an incorrect price allows an attacker to generate profits from the initial liquidity provider's deposits.

a. Recommendation: 1) moving the price operations from initialize to the constructor, 2) adding access controls to initialize, or 3) ensuring that the documentation clearly warns users about incorrect initialization.

b. Medium Risk severity finding from ToB's Audit of Uniswap V3

Swapping on zero liquidity allows for control of the pool's price: Swapping on a tick with zero liquidity enables a user to adjust the price of 1 wei of tokens in any direction. As a result, an attacker could set an arbitrary price at the pool's initialization or if the liquidity providers withdraw all of the liquidity for a short time.

a. Recommendation: No straightforward way to prevent the issue. Ensure pools don't end up in unexpected states. Warn users of potential risks.

b. Medium Risk severity finding from ToB's Audit of Uniswap V3

Failed transfer may be overlooked due to lack of contract existence check: Because the pool fails to check that a contract exists, the pool may assume that failed transactions involving destructed tokens are successful. *TransferHelper.safeTransfer* performs a transfer with a low-level call without confirming the contract's existence. As a result, if the tokens have not yet been deployed or have been destroyed, safeTransfer will return success even though no transfer was executed.

a. Recommendation: Short term, check the contract's existence prior to the low-level call in *TransferHelper.safeTransfer*. Long term, avoid low-level calls.

b. High Risk severity finding from ToB's Audit of Uniswap V3

Use of behavior in equality check: On the left-hand side of the equality check, there is an assignment of the variable *outputAmt_*. The right-hand side uses the same variable. The Solidity 0.7.3. documentation states that "The evaluation order of expressions is not specified (more formally, the order in which the children of one node in the expression tree are evaluated is not specified, but they are of course evaluated before the node itself). It is only guaranteed that statements are executed in order and short-circuiting for boolean expressions is done" which means that this check constitutes an instance of undefined behavior. As such, the behavior of this code is not specified and could change in a future release of Solidity.

a. **Recommendation:** Short term, rewrite the if statement such that it does not use and assign the same variable in an equality check. Long term, ensure that the codebase does not contain undefined Solidity or EVM behavior.

b. High Risk severity finding from ToB's Audit of DFX Finance

Assimilators' balance functions return raw values: The system converts raw values to numeraire values for its internal arithmetic. However, in one instance it uses raw values alongside numeraire values. Interchanging raw and numeraire values will produce unwanted results and may result in loss of funds for liquidity provider.

a. **Recommendation:** Short term, change the semantics of the three functions listed above in the CADC, XSGD, and EURS assimilators to return the numeraire balance. Long term, use unit tests and fuzzing to ensure that all calculations return the expected values. Additionally, ensure that changes to the Shell Protocol do not introduce bugs such as this one.

b. High Risk severity finding from ToB's Audit of DFX Finance

System always assumes USDC is equivalent to USD: Throughout the system, assimilators are used to facilitate the processing of various stablecoins. However, the *UsdcToUsdAssimilator*'s implementation of the *getRate* method does not use the USDC-USD oracle provided by Chainlink; instead, it assumes 1 USDC is always worth 1 USD. A deviation in the exchange rate of 1 USDC = 1 USD could result in exchange errors.

a. **Recommendation:** Short term, replace the hard-coded integer literal in the *UsdcToUsdAssimilator*'s *getRate* method with a call to the relevant Chainlink oracle, as is done in other assimilator contracts. Long term, ensure that the system is robust against a decrease in the price of any stablecoin.

b. Medium Risk severity finding from ToB's Audit of DFX Finance

Assimilators use a deprecated Chainlink API: The old version of the Chainlink price feed API (*AggregatorInterface*) is used throughout the contracts and tests. For example, the deprecated function *latestAnswer* is used. This function is not present in the latest API reference (*AggregatorInterfaceV3*). However, it is present in the deprecated API reference. In the worst-case scenario, the deprecated contract could cease to report the latest values, which would very likely cause liquidity providers to incur losses.

a. **Recommendation:** Use the latest stable versions of any external libraries or contracts leveraged by the codebase

b. Undetermined Risk severity finding from ToB's Audit of DFX Finance

***cancelOrdersUpTo* can be used to permanently block future orders**: Users can cancel an arbitrary number of future orders, and this operation is not reversible. The *cancelOrdersUpTo* function (Figure 3.1) can cancel an arbitrary number of orders in a single, fixed-size transaction. This function uses a parameter to discard any order with salt less than the input value. However, *cancelOrdersUpTo* can cancel future orders if it is called with a very large value (e.g., *MAX_UINT256* - 1). This operation will cancel future orders, except for the one with salt equal to *MAX_UINT256*.

a. Recommendation: Properly document this behavior to warn users about the permanent effects of *cancelOrderUpTo* on future orders. Alternatively, disallow the cancelation of future orders.

b. High Risk severity finding from ToB's Audit of 0x Protocol

Specification-Code mismatch for *AssetProxyOwner* timelock period: The specification for *AssetProxyOwner* says: "The *AssetProxyOwner* is a time-locked multi-signature wallet that has permission to perform administrative functions within the protocol. Submitted transactions must pass a 2 week timelock before they are executed." The *MultiSigWalletWithTimeLock.sol* and *AssetProxyOwner.sol* contracts' timelock-period implementation/usage does not enforce the two-week period, but is instead configurable by the wallet owner without any range checks. Either the specification is outdated (most likely), or this is a serious flaw.

a. Recommendation: Short term, implement the necessary range checks to enforce the timelock described in the specification. Otherwise correct the specification to match the intended behavior. Long term, make sure implementation and specification are in sync. Use Echidna or Manticore to test that your code properly implements the specification.

b. High Risk severity finding from ToB's Audit of 0x Protocol

Unclear documentation on how order filling can fail: The 0x documentation is unclear about how to determine whether orders are fillable or not. Even some fillable orders cannot be completely filled. The 0x specification does not state clearly enough how fillable orders are determined.

a. Recommendation: Define a proper procedure to determine if an order is fillable and document it in the protocol specification. If necessary, warn the user about potential constraints on the orders.

b. High Risk severity finding from ToB's Audit of 0x Protocol

Market makers have a reduced cost for performing front-running attacks: Market makers receive a portion of the protocol fee for each order filled, and the protocol fee is based on the transaction gas price. Therefore market makers are able to specify a higher gas price for a reduced overall transaction rate, using the refund they will receive upon disbursement of protocol fee pools.

a. Recommendation: Short term, properly document this issue to make sure users are aware of this risk. Establish a reasonable cap for the protocolFeeMultiplier to mitigate this issue. Long term, consider using an alternative fee that does not depend on the tx.gasprice to avoid reducing the cost of performing front-running attacks.

b. Medium Risk severity finding from ToB's Audit of 0x Protocol

***setSignatureValidatorApproval* race condition may be exploitable**: If a validator is compromised, a race condition in the signature validator approval logic becomes exploitable. The *setSignatureValidatorApproval* function (Figure 4.1) allows users to delegate the signature validation to a contract. However, if the validator is compromised, a race condition in this function could allow an attacker to validate any amount of malicious transactions.

a. Recommendation: Short term, document this behavior to make sure users are aware of the inherent risks of using validators in case of a compromise. Long term, consider monitoring the blockchain using the *SignatureValidatorApproval* events to catch front-running attacks.

b. Medium Risk severity finding from ToB's Audit of 0x Protocol

Batch processing of transaction execution and order matching may lead to exchange griefing: Batch processing of transaction execution and order matching will iteratively process every transaction and order, which all involve filling. If the asset being filled does not have enough allowance, the asset's transferFrom will fail, causing *AssetProxyDispatcher* to revert. NoThrow variants of batch processing, which are available for filling orders, are not available for transaction execution and order matching. So if one transaction or order fails this way, the entire batch will revert and will have to be re-submitted after the reverting transaction is removed.

a. Recommendation: Short term, implement NoThrow variants for batch processing of transaction execution and order matching. Long term, take into consideration the effect of malicious inputs when implementing functions that perform a batch of operations.

b. Medium Risk severity finding from ToB's Audit of 0x Protocol

Zero fee orders are possible if a user performs transactions with a zero gas price: Users can submit valid orders and avoid paying fees if they use a zero gas price. The computation of fees for each transaction is performed in the *calculateFillResults* function. It uses the gas price selected by the user and the *protocolFeeMultiplier* coefficient. Since the user completely controls the gas price of their transaction and the price could even be zero, the user could feasibly avoid paying fees.

a. Recommendation: Short term, select a reasonable minimum value for the protocol fee for each order or transaction. Long term, consider not depending on the gas price for the computation of protocol fees. This will avoid giving miners an economic advantage in the system.

b. Medium Risk severity finding from ToB's Audit of 0x Protocol

Calls to *setParams* may set invalid values and produce unexpected behavior in the staking contracts: Certain parameters of the contracts can be configured to invalid values, causing a variety of issues and breaking expected interactions between contracts. *setParams* allows the owner of the staking contracts to reparameterize critical parameters. However, reparameterization lacks sanity/threshold/limit checks on all parameters.

a. Recommendation: Add proper validation checks on all parameters in *setParams*. If the validation procedure is unclear or too complex to implement on-chain, document the potential issues that could produce invalid values.

b. Medium Risk severity finding from ToB's Audit of 0x Protocol

Improper Supply Cap Limitation Enforcement: The *openLoan()* function does not check if the loan to be issued will result in the supply cap being exceeded. It only enforces that the supply cap is not reached before the loan is opened. As a result, any account can create a loan that exceeds the maximum amount of sETH that can be issued by the *EtherCollateral* contract.

a. Recommendation: Introduce a require statement in the *openLoan()* function to prevent the total cap from being exceeded by the loan to be opened.

b. High Risk severity finding from Sigma Prime's Audit of Synthetix EtherCollateral

Improper Storage Management of Open Loan Accounts: When loans are open, the associated account address gets added to the *accountsWithOpenLoans* array regardless of whether the account already has a loan/is already included in the array. Additionally, it is possible for a malicious actor to create a denial of service condition exploiting the unbound storage array in *accountsSynthLoans*.

a. Recommendation: 1) Consider changing the *storeLoan* function to only push the account to the *accountsWithOpenLoans* array if the loan to be stored is the first one for that particular account ; 2) Introduce a limit to the number of loans each account can have.

b. High Risk severity finding from Sigma Prime's Audit of Synthetix EtherCollateral

Contract Owner Can Arbitrarily Change Minting Fees and Interest Rates: The *issueFeeRate* and *interestRate* variables can both be changed by the *EtherCollateral* contract owner after loans have been opened. As a result, the owner can control fees such as they equal/exceed the collateral for any given loan.

a. Recommendation: While "dynamic" interest rates are common, we recommend considering the minting fee (*issueFeeRate*) to be a constant that cannot be changed by the owner.

b. Medium Risk severity finding from Sigma Prime's Audit of Synthetix EtherCollateral

Inadequate Proxy Implementation Preventing Contract Upgrades: The *TokenImpl* smart contract requires Owner , name , symbol and decimals of *TokenImpl* to be set by the *TokenImpl* constructor. Consider two smart contracts, contract A and contract B . If contract A performs a delegatecall on contract B , the state/storage variables of contract B are not accessible by contract A . Therefore, when *TokenProxy* targets an implementation of *TokenImpl* and interacts with it via a *DELEGATECALL* , it will not be able to access any of the state variables of the *TokenImpl* contract. Instead, the *TokenProxy* will access its local storage, which does not contain the variables set in the constructor of the *TokenImpl* implementation. When the *TokenProxy* **contract is constructed it will only initialize and set two storage slots:** • The proxy admin address (*_setAdmin* internal function) • The token implementation address (*_setImplementation* private function) Hence when a proxy call to the implementation is made, variables such as *Owner* will be uninitialised (effectively set to their default value). This is equivalent to the owner being the 0x0 address. Without access to the implementation state variables, the proxy contract is rendered unusable.

a. Recommendation: 1) Set fixed constant parameters as Solidity constants. The solidity compiler replaces all occurrences of a constant in the code and thus does not reserve state for them. Thus if the correct getters exist for the ERC20 interface, the proxy contract doesn't need to initialise anything. 2) Create a constructor-like function that can only be called once within *TokenImpl* . This can be used to set the state variables as is currently done in the constructor, however if called by the proxy after deployment, the proxy will set its state variables. 3) Create getter and setter functions that can only be called by the owner . Note that this strategy allows the owner to change various parameters of the contract after deployment. 4) Predetermine the slots used by the required variables and set them in the constructor of the proxy. The storage slots used by a contract are deterministic and can be computed. Hence the variables Owner , name , symbol and decimals can be set directly by their slot in the proxy constructor.

b. Critical Risk severity finding from Sigma Prime's Audit of InfiniGold

Blacklisting Bypass via *transferFrom()* Function: The *transferFrom()* function in the *TokenImpl* contract does not verify that the sender (i.e. the from address) is not blacklisted. As such, it is possible for a user to allow an account to spend a certain allowance regardless of their blacklisting status.

a. Recommendation: At present the function *transferFrom()* uses the *notBlacklisted(address)* modifier twice, on the msg.sender and to addresses. The *notBlacklisted(address)* modifier should be used a third time against the from address.

b. High Risk severity finding from Sigma Prime's Audit of InfiniGold

Wrong Order of Operations Leads to Exponentiation of *rewardPerTokenStored*: *rewardPerTokenStored* is mistakenly used in the numerator of a fraction instead of being added to the fraction. The result is that *rewardPerTokenStored* will grow exponentially thereby severely overstating each individual's rewards earned. Individuals will therefore either be able to withdraw more funds than should be allocated to them or they will not be able to withdraw their funds at all as the contract has insufficient SNX balance. This vulnerability makes the Unipool contract unusable.

a. Recommendation: Adjust the function *rewardPerToken()* to represent the original functionality.

b. Critical Risk severity finding from Sigma Prime's Audit of Synthetix Unipool

Staking Before Initial notifyRewardAmount Can Lead to Disproportionate Rewards: If a user successfully stakes an amount of UNI tokens before the function *notifyRewardAmount()* is called for the first time, their initial *userRewardPerTokenPaid* will be set to zero. The staker would be paid out funds greater than their share of the SNX rewards.

a. Recommendation: We recommend preventing *stake()* from being called before *notifyRewardAmount()* is called for the first time.

b. High Risk severity finding from Sigma Prime's Audit of Synthetix Unipool

External Call Reverts if Period Has Not Elapsed: The function *notifyRewardAmount()* will revert if *block.timestamp >= periodFinish*. However this function is called indirectly via the *Synthetix.mint()* function. A revert here would cause the external call to fail and thereby halt the mint process. *Synthetix.mint()* cannot be successfully called until enough time has elapsed for the period to finish.

a. Recommendation: Consider handling the case where the reward period has not elapsed without reverting the call.

b. High Risk severity finding from Sigma Prime's Audit of Synthetix Unipool

Gap Between Periods Can Lead to Erroneous Rewards: The SNX rewards are earned each period based on reward and duration as specified in the *notifyRewardAmount()* function. The contract will output more rewards than it receives. Therefore if all stakers call *getReward()* the contract will not have enough SNX balance to transfer out all the rewards and some stakers may not receive any rewards.

a. Recommendation: We recommend enforcing each period start exactly at the end of the previous period.

b. Medium Risk severity finding from Sigma Prime's Audit of Synthetix Unipool

Malicious Users Can DOS/Hijack Requests From Chainlinked Contracts: Malicious users can hijack or perform Denial of Service (DOS) attacks on requests of Chainlinked contracts by replicating or front-running legitimate requests. The Chainlinked (*Chainlinked.sol*) contract contains the *checkChainlinkFulfillment*() modifier. This modifier is demonstrated in the examples that come with the repository. In these examples this modifier is used within the functions which contracts implement that will be called by the Oracle when fulfilling requests. It requires that the caller of the function be the Oracle that corresponds to the request that is being fulfilled. Thus, requests from Chainlinked contracts are expected to only be fulfilled by the Oracle that they have requested. However, because a request can specify an arbitrary callback address, a malicious user can also place a request where the callback address is a target Chainlinked contract. If this malicious request gets fulfilled first (which can ask for incorrect or malicious results), the Oracle will call the legitimate contract and fulfil it with incorrect or malicious results. Because the known requests of a Chainlinked contract gets deleted, the legitimate request will fail. It could be such that the Oracle fulfils requests in the order in which they are received. In such cases, the malicious user could simply front-run the requests to be higher in the queue.

a. Recommendation: This issue arises due to the fact that any request can specify its own arbitrary callback address. A restrictive solution would be where callback addresses are localised to the requester themselves.

b. High Risk severity finding from Sigma Prime's Audit of Chainlink

Lack of event emission after sensitive actions: The *_getLatestFundingRate* function of the *FundingRateApplier* contract does not emit relevant events after executing the sensitive actions of setting the *fundingRate*, *updateTime* and *proposalTime*, and transferring the rewards.

a. Recommendation: Consider emitting events after sensitive changes take place, to facilitate tracking and notify off-chain clients following the contract's activity.

b. Medium Risk severity finding from OpenZeppelin's Audit of UMA Phase 4

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. Medium Risk severity finding from OpenZeppelin's Audit of UMA Phase 4

Mooniswap pairs cannot be unpaused: The *MooniswapFactoryGovernance* contract has a shutdown function that can be used to pause the contract and prevent any future swaps. However there is no function to unpause the contract. There is also no way for the factory contract to redeploy a Mooniswap instance for a given pair of tokens. Therefore, if a Mooniswap contract is ever shutdown/paused, it will not be possible for that pair of tokens to ever be traded on the Mooniswap platform again, unless a new factory contract is deployed.

a. Recommendation: Consider providing a way for Mooniswap contracts to be unpaused.

b. Medium Risk severity finding from OpenZeppelin's Audit of 1inch Liquidity Protocol Audit

Attackers can prevent honest users from performing an instant withdraw from the Wallet contract: An attacker who sees an honest user's call to *MessageProcessor.instantWithdraw* in the mempool can grab the *oracleMessage* and *oracleSignature* parameters from the user's transaction, then submit their own transaction to *instantWithdraw* using the same parameters, a higher gas price (so as to frontrun the honest user's transaction), and carefully choosing the gas limit for their transactions such that the internal call to the *callInstantWithdraw* will fail on line 785 with an out-of-gas error, but will successfully execute the *if(!success)* block. The result is that the attacker's instant withdraw will fail (so the user will not receive their funds), but the *userInteractionNumber* will be successfully reserved by the *ReplayTracker*. As a result, the honest user's transaction will revert because it will be attempting to use a *userInteractionNumber* that is no longer valid.

a. Recommendation: Consider adding an access control mechanism to restrict who can submit *oracleMessages* on behalf of the user.

b. High Risk severity finding from OpenZeppelin's Audit of Futureswap V2

Not using upgrade safe contracts in *FsToken* inheritance: The *FsToken* contract is intended to be an upgradeable contract, used behind a proxy (namely, the *FsTokenProxy* contract). However, the contracts *ERC20Snapshot*, *ERC20Mintable* and *ERC20Burnable* in the inheritance chain of FsToken are not imported from the upgrade safe library *@openzeppelin/contracts-ethereum-package* but instead from *@openzeppelin/contracts*.

a. Recommendation: Use the upgrades safe library in this case will ensure the inheritance from Initializable and the other contracts is always linearized as expected by the compiler.

b. Medium Risk severity finding from OpenZeppelin's Audit of Futureswap V2

Unchecked output of the ECDSA recover function: The *ECDSA.recover* **function (in version 2.5.1) returns address(0) if the signature provided is invalid. This function is used twice in the Futureswap code:** Once to recover an *oracleAddress* from an *oracleSignature*, and again to recover the user's address from their signature. If the oracle signature was invalid, the *oracleAddress* is set to address(0). Similarly, if the user's signature is invalid, then the *userMessage.signer* or the withDrawer is set to address(0). This can result in unintended behavior.

a. Recommendation: Consider reverting if the output of the *ECDSA.recover* is ever address(0)

b. Medium Risk severity finding from OpenZeppelin's Audit of Futureswap V2

Adding new variables to multi-level inherited upgradeable contracts may break storage layout: The Notional protocol uses the OpenZeppelin/SDK contracts to manage upgradeability in the system, which follows the unstructured storage pattern. When using this upgradeability approach, and when working with multi-level inheritance, if a new variable is introduced in a parent contract, that addition can potentially overwrite the beginning of the storage layout of the child contract, causing critical misbehaviors in the system.

a. Recommendation: consider preventing these scenarios by defining a storage gap in each upgradeable parent contract at the end of all the storage variable definitions as follows: *uint256[50] __gap; // gap to reserve storage in the contract for future variable additions*. In such an implementation, the size of the gap would be intentionally decreased each time a new variable was introduced, thereby avoiding overwriting preexisting storage values.

b. Medium Risk severity finding from OpenZeppelin's Audit of Notional Protocol

Unsafe division in *rdivide* and *wdivide* functions: The function *rdivide* on line 227 and the function *wdivide* on line 230 of the *GlobalSettlement* contract, accept the divisor y as an input parameter. However, these functions do not check if the value of y is 0. If that is the case, the call will revert due to the division by zero error.

a. Recommendation: consider adding a *require* statement in the functions to ensure $y > 0$, or consider using the *div* functions provided in OpenZeppelin's SafeMath libraries

b. Medium Risk severity finding from OpenZeppelin's Audit of GEB Protocol

Incorrect *safeApprove* usage: The *safeApprove* function of the OpenZeppelin SafeERC20 library prevents changing an allowance between non-zero values to mitigate a possible front-running attack. Instead, the *safeIncreaseAllowance* and *safeDecreaseAllowance* functions should be used. However, the *UniERC20* library simply bypasses this restriction by first setting the allowance to zero. This reintroduces the front-running attack and undermines the value of the *safeApprove* function. Consider introducing an *increaseAllowance* function to handle this case.

a. Recommendation: *safeIncreaseAllowance* and *safeDecreaseAllowance* functions should be used

b. Medium Risk severity finding from OpenZeppelin's Audit of 1inch Exchange Audit

ETH could get trapped in the protocol: The Controller contract allows users to send arbitrary actions such as possible flash loans through the *_call* internal function. Among other features, it allows sending ETH with the action to then perform a call to a *CalleeInterface* type of contract. To do so, it saves the original *msg.value* sent with the operate function call in the *ethLeft* variable and it updates the remaining ETH left after each one of those calls to revert in case that it is not enough. Nevertheless, if the user sends more than the necessary ETH for the batch of actions, the remaining ETH (stored in the ethLeft variable after the last iteration) will not be returned to the user and will be locked in the contract due to the lack of a *withdrawEth* function.

a. Recommendation: Consider either returning all the remaining ETH to the user or creating a function that allows the user to collect the remaining ETH after performing a Call action type, taking into account that sending ETH with a push method may trigger the fallback function on the caller's address.

b. High Risk severity finding from OpenZeppelin's Audit of Opyn Gamma Protocol

Use of transfer might render ETH impossible to withdraw: When withdrawing ETH deposits, the *PayableProxyController* contract uses Solidity's *transfer* **function. This has some notable shortcomings when the withdrawer is a smart contract, which can render ETH deposits impossible to withdraw. Specifically, the withdrawal will inevitably fail when:** 1) The withdrawer smart contract does not implement a payable fallback function. 2) The withdrawer smart contract implements a payable fallback function which uses more than 2300 gas units. 3) The withdrawer smart contract implements a payable fallback function which needs less than 2300 gas units but is called through a proxy that raises the call's gas usage above 2300.

a. Recommendation: *sendValue* function available in OpenZeppelin Contract's Address library can be used to transfer the withdrawn Ether without being limited to 2300 gas units. Risks of reentrancy stemming from the use of this function can be mitigated by tightly following the "Check-effects-interactions" pattern and using OpenZeppelin Contract's *ReentrancyGuard* contract.

b. Medium Risk severity finding from OpenZeppelin's Audit of Opyn Gamma Protocol

Not following the Checks-Effects-Interactions pattern: The *finalizeGrant* function of the Fund contract is setting the *grant.complete* storage variable after a token transfer. Solidity recommends the usage of the Check-Effects-Interaction Pattern to avoid potential security issues, such as reentrancy. The *finalizeGrant* function can be used to conduct a reentrancy attack, where the token transfer in line 129 can call back again the same function, sending to the admin multiple times an amount of fee, before setting the grant as completed. In this way the grant.recipient can receive less than expected and the contract funds can be drained unexpectedly leading to an unwanted loss of funds.

a. Recommendation: Consider always following the "Check-Effects-Interactions" pattern, thus modifying the contract's state before making any external call to other contracts.

b. High Risk severity finding from OpenZeppelin's Audit of Endaoment

Updating the Governance registry and Guardian addresses emits no events: In the Governance contract the *registryAddress* and the *guardianAddress* are highly sensitive accounts. The first one holds the contracts that can be proposal targets, and the second one is a superuser account that can execute proposals without voting. These variables can be updated by calling *setRegistryAddress* and *transferGuardianship*, respectively. Note that these two functions update these sensitive addresses without logging any events. Stakers who monitor the Audius system would have to inspect all transactions to notice that one address they trust is replaced with an untrusted one.

a. Recommendation: Consider emitting events when these addresses are updated. This will be more transparent, and it will make it easier for clients to subscribe to the events when they want to keep track of the status of the system.

b. High Risk severity finding from OpenZeppelin's Audit of Audius

Audit Findings 101 90

The quorum requirement can be trivially bypassed with sybil accounts: While the final vote on a proposal is determined via a token-weighted vote, the quorum check in the *evaluateProposalOutcome* function can be trivially bypassed by splitting one's tokens over multiple accounts and voting with each of the accounts. Each of these sybil votes increases the *proposals[_proposalId].numVotes* variable. This means anyone can make the quorum check pass.

a. Recommendation: Consider measuring quorum size by the percentage of existing tokens that have voted, rather than the number of unique accounts that have voted.

b. High Risk severity finding from OpenZeppelin's Audit of Audius

Audit Findings 101 91 (1/2)

Inconsistently checking initialization: When a contract is initialized, its *isInitialized* state variable is set to true. Since interacting with uninitialized contracts would cause problems, the *_requireIsInitialized* function is available to make this check. However, this check is not used consistently. For example, it is used in the *getVotingQuorum* function of the Governance contract, but it is not used in the *getRegistryAddress* function of the same contract. There is no obvious difference between the functions to explain this difference, and it could be misleading and cause uninitialized contracts to be called.

Audit Findings 101 91 (2/2)

a. Recommendation: Consider calling *_requireIsInitialized* consistently in all the functions of the *InitializableV2* contracts. If there is a reason to not call it in some functions, consider documenting it. Alternatively, consider removing this check altogether and preparing a good deployment script that will ensure that all contracts are initialized in the same transaction that they are deployed. In this alternative, it would be required to check that contracts resulting from new proposals are also initialized before they are put in production.

b. Medium Risk severity finding from OpenZeppelin's Audit of Audius

Audit Findings 101 92

Voting period and quorum can be set to zero: When the Governance contract is initialized, the values of *votingPeriod* and *votingQuorum* are checked to make sure that they are greater than 0. However, the corresponding setter functions *setVotingPeriod* and setVotingQuorum allow these variables to be reset to 0. Setting the *votingPeriod* to zero would cause spurious proposals that cannot be voted. Setting the quorum to zero is worse because it would allow proposals with 0 votes to be executed.

a. Recommendation: Consider adding the validation to the setter functions

b. Medium Risk severity finding from OpenZeppelin's Audit of Audius

Some state variables are not set during initialize: The Audius contracts can be upgraded using the unstructured storage proxy pattern. This pattern requires the use of an initializer instead of the constructor to set the initial values of the state variables. In some of the contracts, the initializer is not initializing all of the state variables.

a. Recommendation: Consider setting all the required variables in the initializer. If there is a reason for leaving them uninitialized, consider documenting it, and adding checks on the functions that use those variables to ensure that they are not called before initialization.

b. Medium Risk severity finding from OpenZeppelin's Audit of Audius

Expired and/or paused options can still be traded: Option tokens can still be freely transferred when the Option contract is either paused or expired (or both). This would allow malicious option holders to sell paused / expired options that cannot be exercised in the open market to exchanges and users who do not take the necessary precautions before buying an option minted by the Primitive protocol.

a. Recommendation: Should this be the system's expected behavior, consider clearly documenting it in user-friendly documentation so as to raise awareness in option sellers and buyers. Alternatively, if the described behavior is not intended, consider implementing the necessary logic in the Option contract to prevent transfers of tokens during pause and after expiration.

b. Medium Risk severity finding from OpenZeppelin's Audit of Primitive

ERC20 transfers can misbehave: The *_transferFromERC20* function is used throughout *ACOToken.sol* to handle transferring funds into the contract from a user. It is called within mint, within *mintTo*, and within *_validateAndBurn*. In each case, the destination is the ACOToken contract. Such transfers may behave unexpectedly if the token contract charges fees. As an example, the popular USDT token does not presently charge any fees upon transfer, but it has the potential to do so. In this case the amount received would be less than the amount sent. Such tokens have the potential to lead to protocol insolvency when they are used to mint new ACOTokens. In the case of *_transferERC20*, similar issues can occur, and could cause users to receive less than expected when collateral is transferred or when exercise assets are transferred.

a. Recommendation: Consider thoroughly vetting each token used within an ACO options pair, ensuring that failing *transferFrom* and *transfer* calls will cause reverts within *ACOToken.sol*. Additionally, consider implementing some sort of sanity check which enforces that the balance of the ACOToken contract increases by the desired amount when calling *_transferFromERC20*.

b. Medium Risk severity finding from OpenZeppelin's Audit of ACO Protocol

Incorrect event emission: The *UniswapWindowUpdate* event of the *UniswapAnchoredView* contract is currently being emitted in the *pokeWindowValues* function using incorrect values. In particular, as it is being emitted before relevant state changes are applied to the *oldObservation* and *newObservation* variables, the data logged by the event will be outdated.

a. Recommendation: Consider emitting the *UniswapWindowUpdate* event after changes are applied so that all logged data is up-to-date.

b. Medium Risk severity finding from OpenZeppelin's Audit of Compound Open Price Feed – Uniswap Integration

Anyone can liquidate on behalf of another account: The Perpetual contract has a public liquidateFrom function that bypasses the checks in the liquidate function. This means that it can be called to liquidate a position when the contract is in the *SETTLED* state. Additionally, any user can set an arbitrary from address, causing a third-party user to confiscate the under-collateralized trader's position. This means that any trader can unilaterally rearrange another account's position. They could also liquidate on behalf of the Perpetual Proxy, which could break some of the Automated Market Maker invariants, such as the condition that it only holds LONG positions.

a. Recommendation: Consider restricting *liquidateFrom* to internal visibility

b. Critical Risk severity finding from OpenZeppelin's Audit of MCDEX Mai Protocol

Orders cannot be cancelled: When a user or broker calls *cancelOrder*, the cancelled mapping is updated, but this has no subsequent effects. In particular, *validateOrderParam* does not check if the order has been cancelled.

a. Recommendation: Consider adding this check to the order validation to ensure cancelled orders cannot be filled.

b. Critical Risk severity finding from OpenZeppelin's Audit of MCDEX Mai Protocol

Re-entrancy possibilities: There are several examples of interactions preceding effects: 1) In the deposit function of the Collateral contract, collateral is retrieved before the user balance is updated and an event is emitted. 2) In the *_withdraw* function of the Collateral contract, collateral is sent before the event is emitted 3) The same pattern occurs in the *depositToInsuranceFund*, *depositEtherToInsuranceFund* and *withdrawFromInsuranceFund* functions of the Perpetual contract. It should be noted that even when a correctly implemented ERC20 contract is used for collateral, incoming and outgoing transfers could execute arbitrary code if the contract is also ERC777 compliant. These re-entrancy opportunities are unlikely to corrupt the internal state of the system, but they would affect the order and contents of emitted events, which could confuse external clients about the state of the system.

Audit Findings 101 99 (2/2)

a. Recommendation: Consider always following the "Check-Effects-Interactions" pattern or use *ReentrancyGuard* contract is now used to protect those functions

b. Medium Risk severity finding from OpenZeppelin's Audit of MCDEX Mai Protocol

Audit Findings 101 100 (1/2)

Governance parameter changes should not be instant: Many sensitive changes can be made by any account with the *WhitelistAdmin* role via the functions *setGovernanceParameter* within the *AMMGovernance* and *PerpetualGovernance* contracts. For example, the *WhitelistAdmin* can change the fee schedule, the initial and maintenance margin rates, or the lot size parameters, and these new parameters instantly take effect in the protocol with important effects. For example, raising the maintenance margin rate could cause *isSafe* to return False when it would have previously returned True. This would allow the user's position to be liquidated. By changing *tradingLotSize*, trades may revert when being matched, where they would not have before the change. These are only examples; the complexity of the protocol, combined with unpredictable market conditions and user actions means that many other negative effects likely exist as well.

Audit Findings 101 100 (2/2)

a. Recommendation: Since these changes are occasionally needed, but can create risk for the users of the protocol, consider implementing a time-lock mechanism for such changes to take place. By having a delay between the signal of intent and the actual change, users will have time to remove their funds or close trades that would otherwise be at risk if the change happened instantly.

b. Medium Risk severity finding from OpenZeppelin's Audit of MCDEX Mai Protocol

Audit Findings 101 101 (1/2)

Votes can be duplicated: The Data Verification Mechanism uses a commit-reveal scheme to hide votes during the voting period. The intention is to prevent voters from simply voting with the majority. However, the current design allows voters to blindly copy each other's submissions, which undermines this goal. In particular, each commitment is a masked hash of the claimed price, but is not cryptographically tied to the voter. This means that anyone can copy the commitment of a target voter (for instance, someone with a large balance) and submit it as their own. When the target voter reveals their salt and price, the copycat can "reveal" the same values. Moreover, if another voter recognizes this has occurred during the commitment phase, they can also change their commitment to the same value, which may become an alternate Schelling point.

Audit Findings 101 101 (2/2)

a. Recommendation: Consider including the voter address within the commitment to prevent votes from being duplicated. Additionally, as a matter of good practice, consider including the relevant timestamp, price identifier and round ID as well to limit the applicability (and reusability) of a commitment.

b. High Risk severity finding from OpenZeppelin's Audit of UMA Phase 1